ROUGH CONSENSUS

How Engineers, Hackers, and Spies Built the Internet

MICHAEL J. BOMMARITO II

Rough Consensus

Rough Consensus

How Engineers, Hackers, and Spies Built the Internet

Michael J Bommarito II

First Edition

2026

Rough Consensus:
How Engineers, Hackers, and Spies Built the Internet

Cover design and typesetting by the author. Printed in the United States of America.

Book Website: `rfcbook.com`

Author Website: `michaelbommarito.com`

ISBN 979-8-9943457-7-1 (paperback)

ISBN 979-8-9943457-6-4 (ebook)

Publisher's Cataloging-in-Publication Data
Names: Bommarito, Michael J., II, author.
Title: Rough consensus :
how engineers, hackers, and spies built—and broke—the internet / Michael J Bommarito II.
Description: First edition. | Michigan : 2026. | Includes bibliographical references.
Identifiers: ISBN 979-8-9943457-7-1 (paperback) | ISBN 979-8-9943457-6-4 (ebook)
Subjects: LCSH:
Internet—History.
Computer network protocols—History.
Internet—Security measures.
Computer security.
Data protection.

Classification: LCC TK5105.875.I57 B66 2026 | DDC 004.678—dc23

Pre-Publication Preview

Not for Resale or Redistribution

This is an advance copy of Rough Consensus: How Engineers, Hackers, and Spies Built the Internet*, made available by the author ahead of general publication.*

Please note:

- *This preview is provided for review, comment, and personal use only.*
- *Final copy edits, page proofs, and index may differ from the published edition.*
- *Please do not redistribute, repost, or extract substantial portions of this PDF.*
- *Quotations for review or commentary are welcome with attribution to the forthcoming print/ebook edition.*

Comments and corrections are welcome: mike@273ventures.com.

Pre-publication version: May 2026.

For the engineers who built a network on trust,
and for those who keep it running despite everything.

For Steve Crocker, who asked the first question.

But most especially,

in memory of Jon Postel (1943–1998) and Dan Kaminsky (1979–2021),
who trusted the network and gave it their lives.

CONTENTS

==========

PROLOGUE

==========

I grew up on the internet in the 1990s, when logging on felt like heading west into an unknown horizon on a trail full of new friends and strange stories.

Back then, the internet felt like it belonged to us — the ones who showed up, learned the protocols, contributed. No algorithms decided what we should see. No platforms constrained what we could build. No advertisements followed us across sites. No paid influencers drowned out the genuine chatter on comp.lang.lisp. And nothing stopped you from port scanning half the internet.

The system wasn't watching back. For better or worse, neither was anyone else.

In hindsight, we didn't know that we were living in the last years of something magical. We didn't appreciate that the network we loved ran on fragile assumptions; assumptions, as we'll see, that wouldn't survive contact with the world it was about to swallow.

This book is about those assumptions we took for granted.

The engineers who built the internet assumed trust. They were colleagues, a few hundred people who often knew each other by first name. When they wrote protocols for sending mail or resolving ad-

dresses or routing packets, they didn't add authentication. Who needs authentication when you work with the person, when you bike past their house every day, or when you'll see them next month? The authentication layer was external, because the entire network fit within a relatively small social network.

Today, it's different. It's been decades since the internet fell within two degrees of social connection. Now, it carries the banking system, the power grid, the medical system, the elections, and more. Over five billion people, as of 2025, many adversarial, now use infrastructure originally designed for a few thousand friends and close acquaintances.

But despite how much has changed, the protocols still trust.

I wrote this book so we don't forget the people and assumptions that got us here. The evidence they left behindf is still here — thousands of Requests for Comments, starting with a document that apologized for existing, that capture 50 years of decisions made by people who couldn't see what was coming, who built anyway, who hoped the next person on the network would be a friend.

Most of the time, they were right. Some of the time, they were wrong.

This is the story of what they built, what broke, and what still works. It begins, prophetically, with a crash and an apology.

==========

PART I

THE NETWORK

==========

CHAPTER 1

THE REQUEST FOR COMMENTS

Very little of what is here is firm
and reactions are expected.

— Steve Crocker, RFC 1, April 7, 1969

It's October 29, 1969. The night is getting late in Los Angeles. Room 3420 of Boelter Hall is lit by fluorescent tubes and the glow of a teletype terminal. Charley Kline, a twenty-one-year-old graduate student at UCLA, sits at the keyboard of an SDS Sigma 7—a hulking mainframe that costs more than most houses. Beside him, a telephone receiver rests off its hook, the line open to Stanford Research Institute three hundred fifty miles north.

On the other end, Bill Duvall waits at his own terminal, an SDS 940. Between them runs something new: a fifty-kilobit leased line connecting two Interface Message Processors—refrigerator-sized computers that will route messages across this experimental network. Four machines in total. Two universities. One idea about to be tested.

Kline types "L."

"Got the L," Duvall confirms through the telephone handset.

Kline types "O."

"Got the O."

Kline types "G"—and the system crashes. The host-to-IMP software crashes. The connection dies. The first message ever sent across the network that will become the internet is two letters long: `LO`.

Leonard Kleinrock, the professor supervising from across the room, would later call it "a more succinct, more powerful, more prophetic message than we could have wished for."[1] Lo and behold. As if the network knew what it was becoming and announced itself with accidental poetry. Decades later, the filmmaker Werner Herzog named his documentary about the internet *Lo and Behold* in tribute to those two letters—an accident transformed into prophecy.

Within an hour, Kline and Duvall got the full `LOGIN` working. The bug was fixed; the connection held. But the crash came first. Before the network carried its first complete word, it had already failed and recovered. That pattern—break, fix, keep running—would repeat for the next fifty years.

No one in Room 3420 imagined five billion users. They couldn't imagine email, the web, ransomware, online banking. They imagined researchers at four universities sharing files. The world they built for was the world they knew—a few hundred people, a handful of nodes, problems measured in kilobytes.

Six months earlier, another graduate student had set this moment in motion—though he had no idea what he was starting.

1.1. THE FIRST NOTE

Steve Crocker was twenty-four years old, a graduate student at UCLA, and he had a problem no one was solving.

The Advanced Research Projects Agency had funded four universities to connect their computers. UCLA would be the first node, followed

by Stanford Research Institute, UC Santa Barbara, and the University of Utah. Bolt, Beranek and Newman was building the IMPs in Cambridge. The hardware would handle routing. But no one had specified what the computers themselves should say to each other once connected.

Crocker and his fellow graduate students had been waiting for the protocol documents to arrive from the East Coast. MIT, Harvard, BBN—someone there would surely take the lead. Weeks passed. The first IMP was scheduled for delivery in September. Nothing came from the East.[2]

So Crocker decided to write something himself.

The problem was authority. He had none. "We were just a bunch of graduate students," Crocker would later explain, "and we didn't have the authority to issue specifications or standards."[3] If he published something that sounded too official, the established researchers might dismiss it. Or worse, take offense that a student had presumed to dictate terms.

Crocker wrote late at night, in his apartment, away from the computer lab and its distractions. He was drafting a document that would shape how the internet developed, and he was terrified of getting it wrong.

His solution was linguistic. Rather than call his document a specification or a standard, Crocker chose a title that disclaimed authority: Request for Comments. Not a proclamation but an invitation. The title signaled he expected pushback, that nothing was settled, that the document was provisional by design.[4]

The opening words reinforced this humility:

> ```
> RFC 1 --- April 7, 1969 --- Steve Crocker
>
> Very little of what is here is firm and reactions are
> expected.
> ```

The message was clear: argue with me. Tell me where I'm wrong. We'll figure this out together.

He numbered the document "1" and sent it to the other sites on April 7, 1969.

Two weeks later, Crocker published RFC 3, which codified the humility into formal policy. "The content of a NWG note may be any thought, suggestion, etc. related to the HOST software or other aspect of the network," he wrote. "Notes are encouraged to be timely rather than polished." The minimum length for a contribution was one sentence. There would be no gatekeeping, no peer review, no committee approval. If you had an idea, you wrote it down and sent it out.[5]

The humility was strategic. Crocker had created a system where disagreement didn't require confrontation. You didn't tell someone they were wrong—you just published an alternative. RFC 4 could contradict RFC 3, and both would stay in the archive. The process was iterative: ideas competed, implementations tested them, the community converged on what worked.

But there was an assumption buried in the process, so fundamental it went unstated. Crocker was writing for colleagues. The Network Working Group that would review his documents numbered perhaps two dozen people scattered across four universities. Trust was built into the system because trust already existed in the room.

No one asked what would happen when the room got bigger.

How do rules and norms emerge where no institutions or social structures yet exist?

The Network Working Group had no charter, no budget, no formal authority. It was a frontier settlement—a handful of people in unmapped territory, making up the rules as they went. Crocker led

it because he was willing to do the work. Meetings happened wherever people could gather—a conference room at UCLA, a borrowed office at Stanford, the margins of academic conferences. No one issued invitations. If you cared about the problem, you showed up.

At the time, Jon Postel was a quiet, bearded man surrounded by creeping stacks of papers. These stacks, which would later migrate from his UCLA office to USC's Information Sciences Institute, weren't just research print outs or papers to read; they were the original RFC organization system. Jon assigned numbers, caught typos, and ensured that anyone who wanted to contribute could find what had come before. As usual, it wasn't a formal position. No one had appointed him; he just did it because it had to be done.

For the next thirty years, Postel would remain the RFC Editor, instilling the system with an institutional memory that would prove critical to its success.

Whereas Jon was a quiet man, Vint Cerf had the opposite problem: he was practically deaf. Vint attended when his schedule allowed. He was a graduate student like the others, working on network protocols that would shape his future `TCP/IP` protocol while reading lips in meetings. Years later, he and his wife Sigrid – who was also deaf – would become the first family of email, advocating publicly for the technology they relied on for communication at home.

This cast of characters, largely graduate students in their mid-twenties, were betting on an idea that most telephone engineers considered absurd. The entire Bell System ran on circuit switching: when you dialed a number, the network reserved a dedicated wire between your phone and the other person's. The line was yours for the duration of the call. If you paused to think, the circuit sat idle. If the wire was cut, the call dropped. Wasteful, but predictable—and it had worked for a century.

Packet switching dispensed with the dedicated line. Messages were broken into small pieces—packets—each stamped with a destination and released into the network to find their own way. Different packets might take different routes. At the far end, the pieces reassembled. No reserved circuits, no wasted silence. If one path failed, packets rerouted around the damage. The network didn't need a plan. It needed cooperation between the machines at each end—and that cooperation was exactly what Crocker's group had to define.

So they worked the way the network did: in small pieces, routed by need. Crocker published an RFC proposing how hosts should begin a conversation. Someone at SRI found a flaw. Crocker published a revision. Postel assigned it a number and filed the original alongside it—nothing was deleted, nothing overwritten. The cycle repeated, each round producing something slightly more refined. Meetings happened wherever people could gather: a borrowed conference room at UCLA, someone's apartment near campus, the hallway outside a talk at a computing conference. Someone would sketch an idea on a chalkboard. Then someone else would try to implement it. If the code worked, the idea stuck. If it broke, someone wrote another RFC proposing a fix.

By the end of 1969, the network Kline and Duvall had tested that October night connected four nodes. By 1971, fifteen.[6] By 1973, thirty-seven. Each new node made the network more useful and the community slightly larger—but still small enough that everyone knew everyone.

That was how the norms emerged—not from a charter or a committee, but from shared work and mutual dependence. "Rough consensus and running code" became the informal motto—a phrase Dave Clark would later articulate at the 1992 IETF meeting, but which described a culture already in practice.[7] Reputation was earned by contributing. Disagreement meant publishing an alternative. Shame, not enforce-

ment, kept people honest—because the community was small enough that everyone could see everyone.

These are frontier norms. They work beautifully when the group fits in a room. And still the protocols assumed it always would.

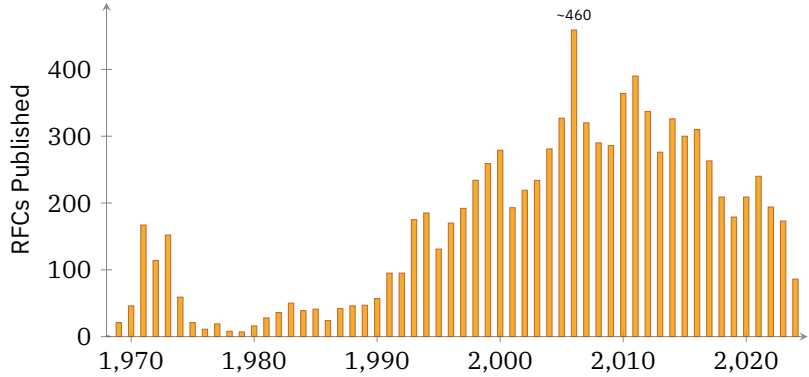


Figure 1.1: RFC publications by year, 1968–2024. From a single document in 1968 to nearly ten thousand today, the RFC archive traces the internet's growth. Peak production occurred in 2006 with approximately 460 RFCs.

1.2. THE BET

RFC 1 contains an assumption so fundamental that Crocker never wrote it down. He didn't need to. Everyone reading the document already knew.

The assumption is trust.

Crocker was writing for a club with fewer than a hundred members. They met in conference rooms and shared hotel bars at workshops. If someone misbehaved—sent junk traffic, ignored a specification, acted in bad faith—everyone would know by the next meeting. Social consequences enforced cooperation more effectively than any authentication mechanism could.

So the protocols inherited this DNA. Messages came from who they claimed to come from. Software would be used as its authors intended. Mistakes were honest. Malice was unthinkable—not globally, not in the world at large, but here, in this room, among these colleagues.

Some might call this naive, but the truth is that it was just engineers and scientists doing what they do best: building an efficient solution to the problem *they* faced.

Consider the alternative. A network designed for adversaries would have required cryptographic signatures on every packet, authentication handshakes before every connection, verification of identity at every hop. The computational overhead alone would have been staggering in 1969, when the entire internet ran on machines with less processing power than a modern thermostat.

But more than that: a paranoid network would have been a slow network. A complex network. A network that graduate students couldn't build in their spare time, that couldn't grow through informal collaboration, that couldn't accumulate protocols through rough consensus and running code.

The engineers made a choice. Simplicity over paranoia. Speed over verification. They bet that the frontier would stay a frontier—that the people using the network would remain the kind of people building it.

The bet paid off for nineteen years. The network grew from four nodes to sixty thousand hosts. Email spread. File transfer worked. Remote login connected researchers across continents. The protocols did exactly what they were designed to do, in exactly the environment they were designed for.

The cost of the bet came due on a single evening in November 1988.

1.3. NOVEMBER 2, 1988

1.3.1. THE RELEASE

How big was the internet in 1988?

No one actually knew. The network that had connected four universities in 1969 now linked roughly sixty thousand hosts—universities, government labs, defense contractors, research hospitals.[8] But there was no central registry, no master list. The network had grown the way the protocols had: organically, node by node, with nobody keeping count. The number sixty thousand was an estimate, and not a confident one.

Robert Tappan Morris wanted a real answer. He was twenty-two, a first-year graduate student at Cornell, and the son of a cryptographer.[9] His father had helped build Unix's password encryption at Bell Labs with Ken Thompson and would later become chief scientist at the NSA's National Computer Security Center. The younger Morris had grown up debugging code with his father. He understood how computers worked, and how they failed.

How would you count every machine on a network with no central directory? You could ask system administrators, but there were thousands of them and no list that reached them all. You could scan for addresses, but the address space was vast and mostly empty. Or you could write a program that counted for you—one that copied itself from machine to machine, hopping along the connections the network was built on, tallying hosts as it went. A census taker that traveled the network on its own.

Morris wrote the program. It would slip quietly from host to host, check whether it had already visited, and move on. When it finished, he'd have his number.

On the evening of November 2, he logged into a machine at MIT.[10] Not Cornell—he didn't want the program traced back to his home institution. At some point after 8:30 PM Eastern, he released it.

It worked. It worked too well.

1.3.2. THE SPREAD

The worm exploited four vulnerabilities, and each one reflected the trust assumptions that had shaped the ARPANET from the beginning.[11]

To understand how, you need to know what these machines were. The computers on the 1988 internet weren't personal—they were shared. A university might have one or two big Unix systems, and dozens of researchers would log into the same machine at the same time, each running their own terminal session. Behind the scenes, programs called **daemons** ran constantly in the background, handling requests. The name comes from Maxwell's demon—the thought experiment about a creature that sorts molecules without being told to. A mail daemon accepted incoming messages. A print daemon managed the queue for the printer down the hall. Each daemon listened and waited, trusting that the requests it received were legitimate.

`sendmail` was the mail daemon on most Unix systems. It had a debug mode—a testing backdoor that allowed remote administration. The mode should have been disabled on production machines. Often it wasn't. Morris's worm sent commands through the debug interface that made `sendmail` download and execute the worm code. The program assumed administrators would configure it safely. They hadn't.

`fingerd`—the finger daemon—answered a simpler question: who's logged in right now? On a shared machine with thirty users, this mattered. If you wanted to know whether a colleague was at her desk before walking across campus, you could type `finger sue@stanford` and find out. The `fingerd` daemon would accept the query, look up the

answer, and send it back. But the program trusted its input without checking how much data was arriving. Morris sent far more data than `fingerd` expected, and the excess spilled past the program's memory boundaries, overwriting the machine's own instructions with new ones—instructions that told the computer to execute the worm. This technique, stuffing more data into a program than it has room for, would later be named a **buffer overflow**. It remains one of the most common vulnerabilities in software today.

The third vulnerability was a feature, not a bug. Unix machines could trust one another through files called `.rhosts`—if you'd authenticated on machine A, machine B would let you in without a password. Researchers used it constantly. Who wants to type credentials every time they switch between the machines in their own lab? Morris's worm exploited this chain: once it compromised one machine, it could walk through the network as that machine's trusted guest.

The fourth vector was the most elemental. The worm carried a dictionary of 432 common words and tried them as passwords against the Unix `/etc/passwd` file. It also tried the username itself, the username reversed, and the username doubled. The system trusted users to choose strong passwords. Many hadn't. A researcher who chose "password" or their own name handed Morris's worm the keys.

Morris had intended a slow, invisible propagation. His worm would check whether a machine was already infected before copying itself, but he worried administrators might fake infection to inoculate their systems. So he added an override: one time in seven, the worm would reinstall anyway.

The one-in-seven ratio was wrong. Machines weren't just infected—they were reinfected, again and again. Each copy of the worm spawned more copies, each consuming memory and processing power. Machines that should have been running a few dozen programs were now running hundreds. Systems froze, rebooted, reinfected immediately.

By morning, roughly six thousand machines were compromised—about ten percent of the internet.[8]

1.3.3. THE RESPONSE

There was no playbook. No incident response teams. No security mailing lists. The network had no procedure for surviving a graduate student.

System administrators discovered the problem when their machines slowed, then froze. They called each other on telephones—the same telephones the network was supposed to replace. Comparing symptoms, pulling cables from walls, powering down servers. The most primitive defense imaginable: if the worm spreads through the network, cut the network.

Morris realized his mistake that night. He called Andrew Sudduth, a computer manager at Harvard, and asked him to distribute instructions for killing the worm. Sudduth sent the message anonymously—but the network was already too clogged to carry it. The cure couldn't reach the patients.[12]

Morris also called a friend—Paul Graham, a graduate student at Harvard. Graham suggested writing a second program to kill the first. Morris refused. "I had messed up with the first one," he later testified, "and it didn't appear that I would be able to do any better the second time."[12]

Graham, testifying as a prosecution witness, would later defend Morris's intentions while faulting his execution. "I said, 'You idiot,' " Graham told the jury. "It was such a good idea and he just blew it. I was so mad."[12]

Within days, DARPA funded a Computer Emergency Response Team at Carnegie Mellon.[13] CERT/CC would coordinate responses to future

incidents, publish advisories, serve as a clearinghouse. The internet needed a fire department. Now it had one.

Within a week, an anonymous tip to the *New York Times* revealed Morris's identity. Investigators traced the worm back to him.

In 1990, Morris became the first person convicted under the Computer Fraud and Abuse Act of 1986.[14] The sentence: three years probation, four hundred hours of community service, $10,050 in fines.

1.3.4. THE AFTERMATH

Morris did not go to prison. He became a tenured professor at MIT. And in 2005, he co-founded a company with the friend he had called that November night. The company was Y Combinator.

Y Combinator became Silicon Valley's most influential startup accelerator, funding Airbnb, Dropbox, Stripe, Reddit, and thousands of other companies. Its philosophy would have been familiar to anyone who had written early RFCs: launch fast, grow fast, fix problems when they find you. Ship now, patch later, trust that your users will behave. The pattern Morris had accidentally stress-tested in 1988 became the gospel his own institution preached to a generation of founders—many of whom would repeat the original bet, collecting user data at enormous scale with the same breezy confidence in good faith that had shaped the protocols themselves.

Graham had called it "such a good idea." Y Combinator would make a business of such good ideas.

The conviction didn't end Morris's career. It barely interrupted it. But it did end something else.

None of the vulnerabilities the worm exploited were new. `sendmail`'s debug mode was in the manual. Buffer overflows were a known class of bug. The `.rhosts` trust chain was a feature that administrators relied on every day. Everyone knew these things were

there. Nobody had thought to ask what happened when someone used them all at once.

After the worm, they started asking. System administrators disabled debug modes, tightened password policies, reconsidered which machines should trust which. CERT published advisories. Firewalls went up. Intrusion detection systems appeared on networks that had never needed monitoring.

The protocols underneath didn't change. IP still delivered packets without checking who sent them. TCP still opened connections on faith. The trust was still there, woven into every layer. The new defenses went on top, and they have been going on top ever since.

Crocker is still at it. More than fifty years after he numbered that first document and sent it to three other universities, he still shows up to working group meetings, still contributes to standards. When interviewers ask whether he regrets building protocols without security, he says what he has always said: the network would never have grown if they had designed it for adversaries.[3]

He's right, and that's the hard part. The trust was never a mistake. It was the reason any of this worked—the reason four nodes could become four million, the reason a handful of graduate students could build something that five billion people would one day depend on. A system that demanded cryptographic proof before every connection would have been a smaller system, a slower one, and almost certainly not the one that ended up carrying the world's banking, medical records, and elections. The vulnerability and the miracle were the same thing.

The RFCs kept accumulating—more than nine thousand by 2025, each one a quiet act of faith that strangers would implement it honestly.

The trust was broken on a November night in 1988, and the trust held anyway. The packets kept flowing. They still are.

But the worm was only the beginning. The protocols that carried passwords in plain text, that let any machine claim any address, that routed packets on faith alone—those protocols were still running, still trusting, still assuming the next connection came from a friend.

The next test wouldn't come from a graduate student with a census program. It would come from a spy.

CHAPTER 2

THE REMOTE LOGIN

The function of the Telnet process
is to make a terminal at a user site
appear over the network as logically equivalent
to a terminal "directly" connected to the server site.

— RFC 97, February 1971[15]

2.1. THE VERY LONG CORD

John Melvin sat in front of a Teletype at Stanford Research Institute in early 1971, typing commands that would travel three thousand miles. The machine in front of him was a hulking beige rectangle—its keyboard mechanical, its printhead clacking with each character. Anyone walking past would think he was using a local computer.

He wasn't. The machine he was talking to was at UCLA. And that was the whole point—making the network disappear, making the terminal lie so convincingly that the distance between two cities collapsed into the click of a key.

In 1971, computing resources were precious. A single mainframe might cost a million dollars. If you needed software that existed only at another institution, your options were limited: fly there, ship data on tape, or beg for an account. The ARPANET promised something better. Connect the computers, and researchers could share them.

Terminals were the obstacle.

Every computer manufacturer had invented their own way of talking to humans. An IBM terminal sent different codes than a DEC terminal. The control characters that moved a cursor, cleared a screen, or signaled the end of a line varied from system to system. If you connected a UCLA terminal to a machine at MIT, nothing worked right. The cursor would jump. The backspace key might print garbage. Even the concept of a "line" meant different things to different machines.

Melvin and his colleague Richard Watson were trying to solve this at SRI. Their solution was published in February 1971 as RFC 97: "A First Cut at a Proposed Telnet Protocol."[15]

The core idea was elegant. Instead of requiring every terminal type to understand every computer type, they invented a fictional terminal: the Network Virtual Terminal, the `NVT`. Your terminal speaks its own dialect; the `NVT` speaks a common language; the remote computer speaks yet another. The `NVT` sits in the middle, converting between them. It had a keyboard that could generate all 128 ASCII character codes and a printer that understood a standard set of control characters. The `NVT` was deliberately simple: no graphics, no color, just text.

The protocol worked like this: the user's `Telnet` client translated their local terminal's quirks into `NVT` format. The `NVT` data traveled over the wire. At the server side, another `Telnet` process translated from `NVT` into whatever the local machine expected. The user saw their familiar terminal; the server saw its familiar input.

"The user should be able to cause generation of all codes which a server system terminal can generate," Melvin and Watson wrote.

Remote access was useless if it felt foreign. The point was to create an illusion: you were there, even though you weren't.

The protocol evolved over the next twelve years, and by May 1983, when Postel and Joyce Reynolds published RFC 854, `Telnet` had matured into a proper standard.[16] The `NVT` was carefully specified, the option negotiation system—DO, DON'T, WILL, and WON'T commands—was formalized, and remote login finally had a complete specification.

Reynolds was Postel's closest collaborator at ISI, and together they authored dozens of RFCs. RFC 854 was one of many—methodical, thorough, precise.

The specification Postel and Reynolds wrote assumed something that seemed obvious in 1983: the network was private. They focused on mechanics: how characters were transmitted, how options were negotiated, how the `NVT` translated between systems. They never discussed encryption. Authentication beyond a login prompt went unmentioned.

In their world, it was a trusted wire between friends.

2.2. THE ILLUSION

2.2.1. HOW TELNET WORKS

Think of `Telnet` as a very long cord plugging your keyboard into a distant computer.

When you type, your keystrokes travel over the wire. When the computer responds, its output travels back. If the connection is fast enough, you feel like you're sitting at the remote machine. The network becomes invisible.

That invisibility cuts both ways—if the network disappears for you, it might also disappear for everyone watching.

A `Telnet` client opens a `TCP` connection to port 23 on the server. Ports work like apartment numbers: the `IP` address gets you to the right machine; the port number tells you which service. Port 23 is `Telnet`; port 80 is web servers; port 25 is email. `TCP` handles reliability by ensuring packets arrive in order. `Telnet` sits on top, carrying your keystrokes in the same form you typed them.

The connection starts with option negotiation. Both sides agree on ground rules: echo, binary mode, terminal type. The server sends "DO ECHO"; the client responds "WILL" or "WON'T." None of this is encrypted, so an eavesdropper sees the setup and watches everything that follows.

RFC 854 describes this negotiation:

> RFC 854 --- May 1983 --- Postel & Reynolds
>
> The basic strategy for setting up the use of options is to have either party (or both) initiate a request that some option take effect. The other party may then either accept or reject the request. If the request is accepted the option immediately takes effect; if it is rejected the associated aspect of the connection remains as specified for an NVT.

Once the options are settled, the connection becomes a bidirectional stream of characters. You type; the characters go to the server. The server runs your commands; the output comes back. Every keystroke crosses the network. The illusion depends on low latency.

The Network Virtual Terminal sits at the heart of this. Both ends pretend they're talking to an `NVT`, translating to and from their local conventions. The `NVT` uses seven-bit ASCII. Carriage return followed by

line feed means "new line." Specific control codes signal interrupts and aborts. A VT100 terminal in California can talk to an IBM mainframe in Massachusetts because both sides speak NVT.

That universality is the problem. Any terminal can talk to any machine—and anyone watching the wire can read what they're saying. The characters that cross the network are human-readable. Your password looks like your password.

2.2.2. THE INVISIBLE WIRE

Telnet's vulnerability is different from IP's routing weakness—not about who belongs, but about what they can see.

Every keystroke travels in plain text: your password, your commands, your data. The protocol provides no encryption because Postel and Reynolds assumed the wire was safe—the ARPANET ran over dedicated leased lines, terminals sat in locked computer rooms, and eavesdropping required physical access to cables in secure facilities.

RFC 854 devotes pages to option negotiation and terminal emulation, but encryption never comes up and the word "security" appears only once. The premise was environmental: controlled infrastructure meant controlled access.

This wasn't written in any RFC. It didn't need to be. It was the water the fish swam in.

Telnet cannot verify that the server is who it claims to be. No certificates, no cryptographic proofs. If someone redirects your connection to their machine instead of the real server, you type your password into their login prompt and never know the difference.

The danger extended beyond Telnet. A related set of protocols—the Berkeley "r-commands" including rlogin and rsh—allowed administrators to configure trust files like `.rhosts`: "If someone connects from

this machine, trust them." The idea was convenience. If you had an account on multiple machines, why type your password over and over?

But trust propagates: compromise one machine in a trusted relationship, and you unlock all its partners. Convenience becomes a highway.

Trust the machine, trust the network, trust that everyone with access belongs—this made sense in a world of locked computer rooms and leased lines. It became dangerous when that world changed.

2.3. SEVENTY-FIVE CENTS

The discrepancy was 75 cents.

Clifford Stoll didn't care about 75 cents. He was an astronomer, not an accountant. But this was August 1986, he had just taken a job as a systems administrator at Lawrence Berkeley Laboratory, and someone had asked him to figure out why the computer accounting system didn't balance.[17] Nine seconds of computer time—worth 75 cents—had no matching account.

Stoll was curious, because nine seconds wasn't a rounding error. Someone had used the system without being charged, which meant an unauthorized account. He started digging.

The FBI wasn't interested. "How much money was lost?" the agent asked. "Um ...75 cents," Stoll replied. "Don't bother us with such trifles."[18]

But Stoll didn't shut it down. He kept digging. What he found was an intruder—someone breaking into the Lawrence Berkeley systems and using them to attack other computers. The attacker was methodical, patient, skilled. Over the following months, Stoll tracked him

across dozens of networks, through military systems and university computers, watching in real time as he pillaged sensitive databases.

The intruder's name was Markus Hess. He was a young German hacker operating from Hannover, selling what he found to the KGB.[19]

2.3.1. THE HUNT

Hess's technique relied on the trust that `Telnet` took for granted.

He would connect to a system over `Telnet`, log in with stolen credentials, and explore. Passwords traveled in cleartext; Hess captured them by monitoring network traffic. Once he had access to one machine, he would use `Telnet` to hop to another, collecting more passwords, building a chain of access that stretched from machine to machine.

The Unix trusted hosts mechanism was particularly useful. Many systems allowed passwordless login from certain machines. If Hess compromised one system in a trusted relationship, he automatically gained access to its partners. The network trusted him because he was coming from a trusted machine.

Stoll watched the intruder work. He hooked up printers to log every keystroke. He set traps, creating fake files with tempting names. He tracked the attacker's movements across a maze of systems, through military contractors and research labs, watching the commands scroll by.

Hess would connect to Lawrence Berkeley, then hop to a military contractor, then to a Navy system, then to an Army database. Each connection was plaintext. Each system trusted the one before it. The chain of trust became a highway for espionage.

No one wanted responsibility: the FBI wasn't interested in a 75-cent theft, the CIA said the intrusions were domestic, and the NSA said the intruder was foreign but the crimes were American.

Stoll kept investigating and built a honeypot: a fake "SDI" database filled with invented documents about Reagan's missile defense program. The file names were irresistible, and when Hess downloaded them, Stoll had proof that the intrusions were espionage, not idle curiosity.

The investigation took ten months. Stoll traced the connections through a satellite link to Germany, worked with German authorities, and finally saw Hess arrested on June 29, 1987. The hacker was convicted of espionage on February 15, 1990.[19,20] He had sold secrets to the KGB for cash and cocaine.

2.3.2. WHAT TELNET REVEALED

The Cuckoo's Egg—Stoll's 1989 book about the investigation—became the first popular account of computer espionage.[18] It demonstrated something the protocol designers had never imagined: their private network had become public.

Hess didn't break `Telnet`; he used it exactly as designed. Passwords in cleartext weren't a bug—they were a feature of a system designed for a different world.

But the community had changed. The ARPANET had been absorbed into the larger internet, the leased lines had become a global mesh, and the locked computer room was now anyone's home.

Stoll's 75-cent discrepancy revealed a world where strangers could watch your passwords flow by.

After Stoll's book was published, security researchers began to take password protection more seriously, but `Telnet` remained in use for years because the convenience was hard to give up. Encryption was expensive, complicated, and often subject to export controls.

It would take another decade before `Telnet` began to fade. In 1995, Tatu Ylönen at Helsinki University of Technology released `SSH`, the

Secure Shell, after his university's network was compromised by a password-sniffing attack.[21] `SSH` looked like `Telnet` to users but encrypted everything—passwords were protected and sessions were authenticated.

The IETF published `SSH` as RFC 4253 in January 2006.[22] By then, most security-conscious organizations had already switched, and `Telnet` retreated to legacy systems.

Hess didn't invent a new attack—he exploited the gap between what the protocol expected and what the network had become. Protocols outlive their creators, and the network they built for friends had become a network for the world.

But `Telnet` never died. It just faded from view.

In December 2025, network sensors recorded approximately 914,000 `Telnet` sessions per day traversing the internet's backbone.[23] The protocol that Postel and Reynolds had designed for trusted networks—the one that sent every password in cleartext—was still running, forty years later, on routers, industrial controllers, and legacy systems that no one had bothered to upgrade.

Then, on January 14, 2026, at 21:00 UTC, the traffic collapsed. In a single hour, `Telnet` sessions dropped 65 percent. By the next hour, they were down 83 percent from baseline. Eighteen autonomous systems went completely silent on port 23—major carriers like Vultr, Cox, Charter, and British Telecom simply stopped passing `Telnet` traffic.

Six days later, security researchers disclosed CVE-2026-24061: a critical authentication bypass in GNU Inetutils `telnetd`.[24] An attacker could inject arguments through the USER environment variable during option negotiation—the same negotiation system that RFC 854 had specified in 1983. By sending `-f root` as a username, an attacker received an unauthenticated root shell. The vulnerability was eleven years old.

Someone had known before the disclosure. The timing suggested that backbone providers had received advance warning and filtered port 23 before the announcement went public. The borrowed time had almost run out.

And yet `Telnet` persists. As of February 2026, roughly 370,000 sessions still cross the internet daily—one-third the pre-collapse baseline, but not zero.[23] The protocol designed for a network of colleagues continues to run, passwords exposed, on a network of strangers.

`Telnet` gave you a window into a distant machine. But sometimes you didn't want to sit at the keyboard and type—you wanted to reach through and grab something. A file. A dataset. A program someone wrote three thousand miles away. That required a different protocol, built in the same era, carrying the same assumptions.

CHAPTER 3

THE FILE TRANSFER

The objectives of FTP are
1) to promote sharing of files
(computer programs and/or data),
2) to encourage indirect or implicit
(via programs) use of remote computers,
3) to shield a user from variations
in file storage systems among hosts.

— RFC 959, October 1985

3.1. THE YELLOW LEGAL PAD

April 1971. Abhay Bhushan needed to give his colleague a file, but the colleague worked at Stanford while Bhushan worked at MIT. Between them lay the ARPANET and three thousand miles of telephone cables, linking a few dozen computers at universities and research labs. The network worked, messages traveled, but the computers spoke different languages, stored data differently, organized filesystems according to competing philosophies. Bhushan's Honeywell 645 ran Multics,

an operating system treating files like library books—organized, cataloged, protected. His colleague's machine ran ITS, the Incompatible Time-sharing System, which provided no access controls at all. The file Bhushan wanted to send was just text, but it might as well have been written in cuneiform.

The obvious solution: ship a tape by mail. Researchers did this constantly—write data to magnetic tape, package it, hand it to the postal service. Days passed before the tape arrived. Then someone mounted it on a compatible drive, hoping the encoding matched. The joke in computer science departments: a station wagon full of tapes had better bandwidth than any network—just terrible latency.

For collaborative work, shipping tapes was absurd. There's a network connecting these machines. Why can't they just send files?

The ARPANET in 1971 connected machines with almost nothing in common. A PDP-10 at UCLA, an IBM 360 at MIT, a Honeywell at Harvard—each stored text differently. Seven bits per character, eight bits, nine bits packed into larger words. Each had its own line endings, directory structure, file concept. Some treated files as byte streams, others used fixed-length records. An IBM mainframe's text file would be incomprehensible to a DEC minicomputer.

Bhushan was twenty-six, a graduate student at MIT's Project MAC. He had arrived from India three years earlier. Project MAC—"Machine-Aided Cognition," though some joked "Man Against Computer"—was a birthplace of time-sharing. The Multics system was experimental, ambitious, bristling with security features. Bhushan sat in perhaps the most advanced computing environment on Earth and couldn't share a document across the country.

The problem wasn't the network—the Network Control Protocol shipped data between machines, `Telnet` let users log in remotely. But files were different. A file isn't just bytes; it's bytes with structure. Text files have line endings, programs have sections and entry points.

Send a Multics text file to an ITS machine, and you'd get gibberish: wrong encoding, wrong line endings, uninterpretable structure. The machines could talk but couldn't understand each other.

The ARPANET community had been discussing file transfer since the network's earliest days. RFC 80, published in December 1970, proposed mechanisms for exchanging files. But those early proposals required users to understand the remote system's conventions. Bhushan wanted to hide the differences entirely.

He sketched a solution on a yellow legal pad, the same kind Steve Crocker had used drafting RFC 1. He called it a File Transfer Protocol.

The key insight: don't preserve the sender's format—translate instead. The protocol would define neutral representations, a diplomatic language for data. When a Multics system sent a text file, it would translate to this common tongue. When an ITS system received it, it would translate back. The actual bytes on the wire would be neither Multics nor ITS, but network ASCII, a standard representation any machine could speak.

The ARPANET itself used this logic. The Interface Message Processors translated at the boundaries, converting host-specific packets into network-standard messages and back. Bhushan applied the same principle one layer up, to files instead of packets. Let each system keep its internal representation. Just agree on what goes over the wire.

RFC 114 hit the ARPANET on April 16, 1971.[25] "A principal objective of the protocol," Bhushan wrote, "is to promote the indirect use of computers on the network." You could transfer a file without logging in, without learning a foreign operating system, without understanding how the receiving machine organized its storage. The protocol would handle the translation.

Bhushan tested on the hardest possible pair: "Since our implementation involves two dissimilar systems (Multics is a 'service' system, ITS is not) with different file systems (Multics provides elaborate access

controls, ITS provides none), we feel that the file transfer mechanisms proposed are generalizable." If it worked between Multics and ITS, it would work anywhere.

Any computer on the ARPANET could now send files to any other, regardless of manufacturer, operating system, or internal storage format. The protocol defined transactions: retrieve, store, append, rename, delete. It defined data types: ASCII text, binary, EBCDIC for IBM mainframes. Two alien machines could negotiate a file transfer without knowing each other's internal architecture.

RFC 114 even anticipated remote program execution. Bhushan included an "execute" request. The idea anticipated what we now call remote procedure calls—but it never gained traction. File transfer was complicated enough.

The protocol evolved quickly as engineers added features, fixed bugs, and argued about edge cases. By 1972, RFC 172 expanded the protocol with restart capabilities,[26] and by 1973 another revision followed—the file transfer problem had more wrinkles than anticipated.

Some problems only surfaced once the protocol ran against real machines: what if the receiving machine runs out of disk space, or the connection drops mid-transfer?

The RFCs multiplied: 172, 265, 354, 542, 765. Engineers debated one connection or two, data compression, line-ending conventions. RFC 686, published in 1975, was titled "Leaving Well Enough Alone"—a plea from engineers tired of constant changes. The simple idea of "send a file" accumulated fourteen years of refinements.

In October 1985, Postel and Joyce Reynolds published RFC 959, the definitive specification.[27] Sixty pages describing data types, transfer modes, access controls, error handling. The objectives echoed Bhushan's original vision: promote file sharing, encourage indirect use of remote computers, shield users from filesystem variations, transfer data reliably and efficiently.

RFC 959 codified the protocol's most distinctive feature: the separation of control and data. Unlike `Telnet`, which used a single connection, `FTP` used two—the control connection handled commands and responses, while the data connection handled file content. You could issue commands while data flowed separately. The design solved a real problem: it kept the session interactive even during large transfers. It was also, as we will see, a security nightmare.

In 1985, security meant access control—could the user read this file, or write to this directory? `FTP` handled authentication with username and password. The access control commands came first: `USER` to specify your identity, `PASS` for your password, optionally `ACCT` for billing. Only after authentication could you move files.

The protocol also permitted anonymous `FTP`, where a server could accept the username "anonymous" and use the user's email address as a password. The convention arose organically: administrators who wanted to share files publicly configured their servers to accept this special username, and the email address served as a logbook entry rather than a credential. This wasn't authentication—it was a social convention, and the email address was never verified.

Anonymous `FTP` embodied the culture that built the ARPANET: research should be shared, software should be distributed, knowledge should flow freely. Major universities maintained public archives, government agencies published data, and software developers distributed code. This was trust by default, scaled globally. The protocol assumed anonymous users would behave responsibly. They were colleagues, after all.

The system worked—until it didn't.

3.2. TWO PHONE LINES

3.2.1. HOW FTP WORKS

Imagine a warehouse with two telephone lines. One line connects you to the warehouse manager. You call that line to give instructions: "Show me what's in aisle four. Ship the blue boxes to this address." The manager responds, confirms your requests, coordinates the work. The second line connects you to the loading dock. When the manager approves a shipment, workers use this line to describe what they're sending. Then the actual boxes appear at your door. You can keep talking to the manager even while shipments are in transit.

FTP works identically. One connection handles commands. Another handles data.

3.2.2. THE CONTROL CHANNEL

When you connect to an FTP server, your client opens a TCP connection to port 21. This control channel stays open for the entire session. Everything you type travels over this connection—and the remote host believes what you type without verification. You claim a username; it accepts. You claim an email address; it records it. The protocol believes you.

The conversation follows a simple pattern. You send `USER anonymous`, and the reply comes back `331 Please specify the password`. You send `PASS someone@example.com`, and the response is `230 Login successful`. Three-digit codes tell you what happened—2xx for success, 5xx for failure, others in between. But that email address was never checked. The protocol just took your word.

```
$ ftp files.example.edu
Connected to files.example.edu.
220 Example \proto{FTP} Server ready.
Name: anonymous
331 Please specify the password.
Password: user@domain.com
230 Login successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection.
pub/
incoming/
README
226 Transfer complete.
```

The server asked for a password and received an email address. It never verified that address, never sent a confirmation, never checked against any list—it just said "Login successful." The command vocabulary is minimal: authenticate, navigate, transfer, quit. The design expected humans might type commands by hand. Modern clients hide these behind graphical interfaces, but the underlying conversation remains the same—and every command is still accepted at face value.

3.2.3. THE DATA CHANNEL

File content travels over a separate connection. When you request a file, the server opens a new TCP connection for that data. When the transfer completes, this connection closes.

The separation solved a real problem. Bhushan's original 1971 design used one, which created problems: if you were sending a large file and needed to abort, you couldn't send an abort command until the

data stopped. By separating command and data streams, the protocol remained interactive, letting you issue an abort at any moment.

A sensible solution for 1972—but the second connection introduced a question that would haunt the protocol: where should the server send the data?

In **active mode**, the client tells the server where to send data. The client issues a `PORT` command with an `IP` address and port number. The server opens a connection to that address.

Here's what the server doesn't do: verify that the address belongs to the client. You could specify your own machine, or you could specify someone else's entirely—the server connects wherever you tell it.

In 1971, this seemed reasonable. Everyone connected was a colleague. No one would lie about their address.

In **passive mode**, the roles reverse. The client sends a `PASV` command. The server responds with its own `IP` address and port. The client connects to that address. This solved the firewall problem—both connections are outgoing from the client's perspective—but the underlying trust remained untouched. The server still believed everything on the control channel: the username, the email address, the client's stated intentions. Passive mode changed who initiated the data connection. It changed nothing about what the server would believe.

3.2.4. THE HONOR OF STRANGERS

`FTP` trusts clients to tell the truth about themselves. The pattern repeats: the unverified email address, the PORT command pointing anywhere the client chooses. The protocol asks questions and believes the answers.

Anonymous `FTP` made this trust explicit. What did the "password" actually prove? Nothing—it was your email address, not verified, not checked, just recorded for courtesy. RFC 959 calls it "identification for

access control" rather than authentication. You weren't proving who you were. You were promising to behave.

Telnet trusted the network path, and FTP trusted the client's self-declarations—both forms of trust emerged from the same source: a network where reputation preceded you. In 1971, the ARPANET's researchers could fit in a conference room. Face-to-face meetings made deception pointless. The protocols reflected that physical proximity.

> RFC 959 --- October 1985 --- Postel & Reynolds
>
> Access controls define users' access privileges to the use of a system, and to the files in that system. Access controls are necessary to prevent unauthorized or accidental use of files. It is the prerogative of a server-FTP process to invoke access controls.

Read that last sentence again: "the prerogative of a server-FTP process." Security was optional—the protocol defined mechanisms but imposed no requirements, leaving decisions about checking credentials, restricting uploads, or verifying addresses to local administrators. The specification shrugged.

The open culture flourished: the GNU project distributed free software via anonymous FTP, the IETF published RFCs, and universities maintained public archives of BSD source code, Emacs, and research papers. The servers embodied an ideal—knowledge should flow freely.

Then the internet opened to the world, and everything changed.

3.3. THE HIDDEN DIRECTORIES

In 1995, an underground network called "the Scene" had been growing in the internet's shadows, where stolen software was the currency. Release groups—Razor 1911, Fairlight, Drink or Die—competed to crack new commercial software and distribute it within hours of retail release. The hierarchy was rigid: crackers who defeated copy protection at the top, couriers who distributed releases below, and leeches who downloaded but contributed nothing.

Their distribution mechanism was anonymous FTP servers—thousands of them, and not their own. Scene participants would find misconfigured FTP servers at universities and corporations, then create hidden directories with names like `...` (three dots) or `..` (dot-dot-space). The trick exploited how Unix displayed directories. A listing showing only `.` and `..` would seem empty, but a third entry with three dots or trailing whitespace would be invisible unless you knew to look. A system administrator might discover forty gigabytes of pirated software buried in the physics department's file share. Universities were prized targets: fast connections, minimal oversight.

Anonymous FTP made this trivial because many servers permitted anonymous uploads to `incoming` or `pub` directories. Couriers scanned networks for vulnerable servers—anonymous access enabled, write permissions available, fast connections, naive administrators—and called these "pubs," maintaining carefully guarded lists. The competition to find new pubs was as fierce as the competition to release software first.

Operation Buccaneer in 2001 led to prosecutions across six countries.[28] Over 100 search warrants executed in a single day. Defendants included MIT students, corporate employees, government contractors. But for years, the protocol itself did nothing to stop them. FTP assumed that anonymous access meant public access. The Scene exploited that expectation without breaking the protocol.

The Scene's use of anonymous FTP was brazen, illegal, and eventually prosecuted, but a more subtle vulnerability lurked in the protocol itself. The Scene had abused the culture of anonymous access; the vulnerability Hobbit found abused the protocol's fundamental architecture.

The PORT command let clients tell servers where to send data, and the connection opened without verification. In 1995, a security researcher who went by "Hobbit" realized what this meant.[29] The PORT command accepts any address, so you can direct traffic somewhere else entirely, and the request will be honored.

This became the FTP bounce attack. An attacker connects to an FTP server and logs in as anonymous, but instead of requesting a file for themselves, they issue a PORT command specifying a target machine's address. The server, following the protocol without question, opens a connection to the target—and the target sees an incoming connection from the FTP server, not the attacker. The attacker has laundered their connection through a trusted intermediary.

```
220 ftp.trusted-university.edu ready
USER anonymous
331 Please specify the password.
PASS attacker@fake.email
230 Login successful.
PORT 10,0,0,5,0,25
200 PORT command successful.
LIST
```

That PORT command told the server to connect to 10.0.0.5 on port 25—the SMTP mail port. The server dutifully opens a connection to an internal mail server and sends a directory listing. The listing isn't useful, but the connection is. The attacker has made the FTP server

initiate a connection to an internal machine that might not be reachable from outside. If a firewall protects the target, it might trust connections from the FTP server's IP address. The attacker has bounced their probe through a legitimate server.

FTP bounce was particularly useful for reconnaissance, since port scanning normally left traces—the scanner's IP address appeared in every log. With the FTP bounce attack, those connections came from a legitimate university server instead.

The technique enabled port scanning through firewalls: an attacker could probe a target network by issuing PORT commands with different port numbers and observing the server's response. The attacker never made a direct connection—the FTP server did all the work.

More creatively, attackers could bypass access controls entirely. A firewall configured to trust a university's network would accept connections tunneled through that university's FTP server. The bounce attack turned trusted servers into unwitting proxies.

The FTP bounce attack wasn't a bug—the server was following the protocol correctly. The specification simply expected something no longer true: that users would only request connections to their own machines. RFC 959 had even anticipated third-party transfers. The specification explicitly described how "a user might wish to transfer files between two hosts, neither of which is a local host." This was a feature—in a world where all users were trustworthy. Hobbit had simply used the feature for purposes its designers never imagined.

The fix seemed obvious: restrict where the PORT command could point, so most server implementations added this check within months. They would only accept PORT commands specifying the same IP address that the control connection came from.

But the fix broke legitimate use cases. Organizations with complex network topologies sometimes wanted data to flow between servers rather than through a client. The three-way transfer that RFC 959

explicitly documented became impossible. Passive mode offered a partial solution, but had its own complications with firewalls and NAT devices. Organizations gradually locked down their servers, requiring authentication, restricting uploads, limiting what anonymous users could do. The open culture contracted behind access controls.

By the late 1990s, FTP's security model looked untenable: passwords traveled in cleartext, the PORT command was a vector for abuse, and anonymous access had become synonymous with liability. The Web was making FTP less necessary anyway—HTTP could serve files without two-channel complexity, and a web server on port 80 worked through firewalls without special handling. FTP, by contrast, required NAT helpers and stateful packet inspection.

Security-conscious organizations migrated to alternatives: SFTP encrypted everything and used a single connection, solving both the confidentiality problem and the firewall problem, while SCP provided secure copying over SSH. FTPS added TLS encryption to FTP, though the two-channel design made integration awkward.

The FTP bounce attack required no software bugs. Every command was valid. Every response was correct. The vulnerability was in the assumptions, not the code.

Bhushan, writing in 1971, wasn't naive—he explicitly noted the tension between Multics (which "provides elaborate access controls") and ITS (which "provides none"). But the ARPANET was small, the users were colleagues, and the risk of someone lying about their destination address was approximately zero.

No one in 1971 could imagine the warez scene. The protocol kept working, exactly as specified, in a world it was never designed to serve.

The archives are closed.

The RFC archive moved from anonymous FTP to HTTP long ago. The university servers that once held gigabytes of freely shared software have shut their public directories. The warez servers are gone. What remains is invisible: FTP running inside corporate networks, inside batch jobs scheduled years ago, inside mainframe workflows that nobody has touched since the Clinton administration. The protocol moves files between servers on a schedule nobody remembers setting, in directories nobody monitors, carrying data nobody watches arrive.

In 2021, Chrome and Firefox dropped native FTP support.[30] Browsers had treated FTP as a curiosity for years—something you could type into an address bar but almost never did. The browsers moved on. FTP didn't. As of 2025, it runs in the background of enterprise computing the way old pipes run behind new drywall: out of sight, past its design life, still carrying the load.

The open culture that anonymous FTP embodied didn't survive the internet's growth. Knowledge stopped flowing freely when the internet stopped being a collegial network. The protocol survived anyway—not as a statement of values, but as a ghost. The idealism is gone. The cron jobs remain.

CHAPTER 4

THE LANGUAGE OF PACKETS

TCP implementations will follow
a general principle of robustness:
be conservative in what you do,
be liberal in what you accept from others.

— RFC 793, Jon Postel, 1981

4.1. THE CABANA HYATT

The envelope was from a Palo Alto hotel.[31] Late 1973. Vint Cerf sat in the lobby of the Cabana Hyatt, sketching diagrams with Bob Kahn. They were trying to solve an impossible problem: how do you make networks that speak different languages communicate?

Cerf had been working on the ARPANET for years. He knew its protocol, NCP, intimately. Kahn had built the software that ran the IMPs. Between them, they understood the network better than anyone.

NCP was brilliant for what it did. It let ARPANET computers exchange messages reliably, efficiently, predictably. But NCP was married to the ARPANET's particular hardware. Try to connect to a satellite

network or the packet radio systems the military was testing, and NCP fell apart.

DARPA wanted more: a network of networks—an "internetwork" tying together satellite links, radio connections, and whatever else the future might bring. The system had to survive pieces being blown up. The Cold War was not an abstraction.

So Cerf and Kahn sat in that hotel lobby and asked a question that would reshape the world: what if the network didn't care what it was carrying?

The insight arrived in pieces over months of meetings, and the hotel envelope was just one artifact of a longer conversation. But the core idea was simple.

Split the problem in two.

One protocol would handle addressing and routing. It would wrap data in an envelope with source and destination addresses and trust the network to deliver it. This was the Internet Protocol—IP. It made no promises about reliability. Packets might arrive out of order or not at all. IP's job was just to try.

The other protocol would handle reliability. It would sit on top of IP and provide everything applications needed: guaranteed delivery, correct ordering, flow control. This was the Transmission Control Protocol—TCP.

Together, they became TCP/IP. Cerf and Kahn published their design in IEEE Transactions on Communications in May 1974.[32] The paper would be cited thousands of times. At the moment, it was just an idea. Rough consensus in a hotel lobby. Running code within months.

Cerf had a personal stake in network communication. His wife Sigrid was deaf. They had met at Stanford in the late 1960s, when communication for the deaf meant handwritten notes or face-to-face conversation. The teletype machines that preceded email were trans-

forming her world—text over wires, messages that didn't require hearing.

Cerf saw what networks could mean for people who couldn't pick up a phone. Email wasn't just a convenience; it was access.

He pushed the implementation forward. By 1978, multiple versions of TCP/IP were running at research sites. Postel was turning the informal designs into formal specifications. RFC 791 would define IP.[33] RFC 793 would define TCP.[34] September 1981.

But specifications are not deployment. The ARPANET still ran on NCP. Switching to TCP/IP meant rewriting everything, debugging everything, coordinating hundreds of sites across the country. The transition would require something unprecedented: a day when everyone switched at once.

They called it Flag Day.[35] January 1, 1983—New Year's Day—the entire ARPANET would cut over from NCP to TCP/IP in a single coordinated moment. Machines that weren't ready would simply stop working.

The planning took months. RFC 801, the transition plan, laid out the stakes in capital letters: "There are some very tempting shortcuts in the implementation of IP and TCP. DO NOT BE TEMPTED!"[35] The warning was prescient. Shortcuts taken then—ignoring checksums, trusting inputs—would become vulnerabilities for decades.

When the day came, Cerf had buttons made.[36] "I survived the TCP/IP transition," they read. Cerf meant it as a joke, mostly.

The transition worked. Not perfectly—there were bugs, panics, sites that discovered their software wasn't quite ready. But it worked. The ARPANET had a new language.

More than that: a language designed to reach beyond itself. Within months, other networks began connecting. CSNET, linking computer science departments. BITNET, linking universities. Each could speak

to the others through TCP/IP. The "internet" was no longer a theory—it had become a network of networks, growing month by month.

Flag Day was the last time anyone could break everything at once. After January 1, 1983, backward compatibility became a permanent constraint. The internet had grown too large to reboot.

The protocols that Cerf and Kahn designed on envelopes and legal pads in 1973 became frozen in place. Their assumptions, their trade-offs, their blind spots—all of it locked in by success.

And one of those assumptions would haunt the internet for decades.

4.2. THE HANDSHAKE

4.2.1. HOW TCP/IP WORKS

Think of the postal system. You write a letter, put it in an envelope, write an address on the front and a return address in the corner. You drop it in a mailbox. Postal workers sort it, route it, hand it from truck to truck until it reaches its destination.

IP works similarly.

Every piece of data gets wrapped in a packet with source and destination addresses. Routers examine the destination address and forward the packet toward its goal, one hop at a time. The packet might take different routes on different days. IP doesn't guarantee any particular path. It just tries to deliver.

The source address serves the same purpose as a return address. If something goes wrong, the source address tells routers where to send error messages.

Here is the first assumption: the source address is honest. When a packet arrives with source address 192.168.1.1, IP believes it. No verification. No proof. The address says what it says, and the network accepts it.

This made perfect sense in 1974. The ARPANET connected a few dozen sites. The researchers at each had met at conferences, collaborated on papers, shared the same professional circles. Lying about your address was pointless. Why would a Stanford researcher pretend to be at MIT?

This expectation was encoded into the protocol. RFC 791 describes how IP works.[33] Nowhere in its 45 pages does it mention authentication. The concept simply did not arise.

4.2.2. THE IP HEADER

Every IP packet begins with a header—metadata that routers use to move the packet through the network. Three fields do the real work: Source Address, Destination Address, and Time to Live. Notice what's conspicuously absent—no field that says "verified by" or "authenticated at." The addresses are claims, not proofs.

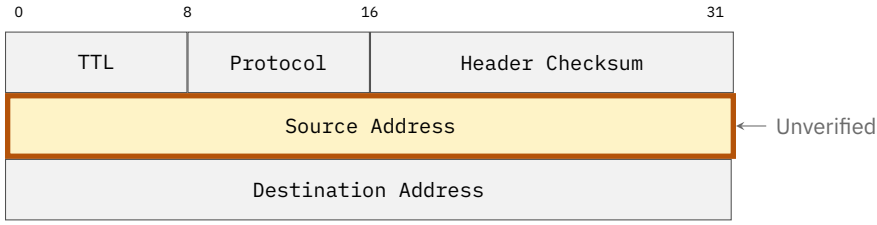


Figure 4.1: IPv4 header fields (simplified). The source address is accepted without verification.

The Time to Live (TTL) starts at a number—typically 64 or 128. Each router that handles the packet subtracts one. When the counter reaches zero, the packet is discarded.

The Source and Destination fields are each 32 bits long. This is why IPv4 addresses look like four numbers separated by dots: 192.168.1.1. Four billion addresses seemed infinite when the specification was finalized in 1981. That estimate proved insufficient.

The header has a checksum protecting against transmission errors. But a checksum only verifies the header wasn't damaged in transit. It says nothing about whether the source address is true.

4.2.3. BEST-EFFORT DELIVERY

What does IP actually promise? Almost nothing.

"Best-effort delivery" means the network tries to deliver packets but makes no guarantees. Packets might arrive out of order, might be duplicated, might vanish entirely—IP does not care. It just tries.

Why strip out all these guarantees? Because simplicity is power. By keeping IP minimal, Cerf and Kahn made it universal. Satellite links could carry IP packets. Radio networks could carry IP packets. Anything could, because IP asked so little.

The consequence: reliability had to come from somewhere else—enter TCP.

4.2.4. THE THREE-WAY HANDSHAKE

TCP provides everything IP lacks: reliable delivery, correct ordering, flow control. It accomplishes this through the three-way handshake—a brief exchange establishing the connection, like two people shaking hands before a conversation.

The client reaches out: "I want to connect." It sends a SYN packet with a sequence number—a 32-bit value that will track every byte in this conversation. The response comes back: "I hear you, and I want to connect too." SYN-ACK—two flags in one packet, with a fresh sequence

number for the return direction. The client confirms: "Got it." ACK. Three packets, connection established.

Notice what just happened. The server believed the client's address. The client believed the server's response came from the right place. Neither side proved who they were. The handshake completes based on an assumption: if you can finish it, you must be who you claim to be.

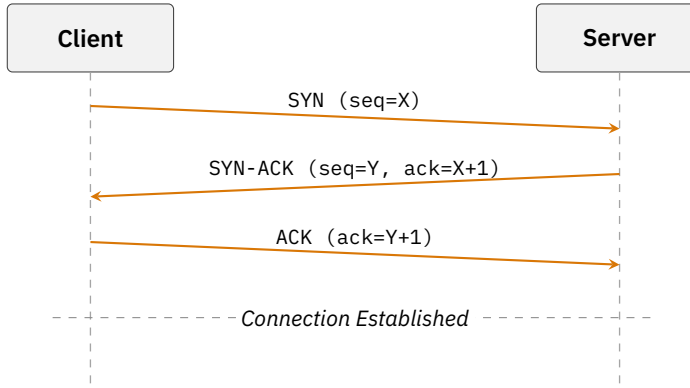


Figure 4.2: TCP three-way handshake. Both sides exchange sequence numbers to establish a connection.

Sequence numbers have two design purposes: reassembling out-of-order data and tracking what's been received. But systems came to rely on a third property: completing the handshake implied identity.

Here is the assumption that emerged: if someone sends a SYN from address 192.168.1.1, receives the SYN-ACK, and successfully sends the ACK, then they must actually be at 192.168.1.1. After all, how else could they receive the SYN-ACK?

This reasoning was sound in 1974. It remained sound for two decades. Then someone realized it had a flaw.

4.2.5. SEQUENCE NUMBER PREDICTABILITY

Sequence numbers track the bytes in a conversation—the receiver uses them to put bytes back in order, and they also prove both sides are genuinely connected. Only someone who received the SYN-ACK could know the server's sequence number.

This logic has a flaw: it depends on the sequence number being impossible to guess. The handshake's security depends on unpredictable sequence numbers. If someone can guess the number, they don't need to receive the SYN-ACK. They can predict it and complete the handshake blind.

Early TCP implementations chose sequence numbers poorly. Some systems incremented a global counter for each new connection. If you knew the counter's current value—perhaps from a legitimate connection you had just opened—you could predict the next value with high accuracy.

RFC 793 addressed sequence number selection, but as a reliability concern, not a security one.[34] The specification noted: "To avoid confusion we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation." The worry was stale data corrupting new connections, not attackers predicting values.

The specification suggested using a clock-based scheme that would change sequence numbers over time. Four years later, in 1985, Robert Morris Sr.—the worm author's father, then at the NSA—published the first analysis showing that predictable sequence numbers were a security risk.[37] "Hard to predict" was not the same as "cryptographically random." Implementations varied. Some were weak.

4.2.6. POSTEL'S LAW

TCP embedded another principle in its design, one that would prove double-edged.

The principle later became known as the Robustness Principle: send perfectly formatted packets, but gracefully accept packets that are slightly malformed. If the intent is clear, accept it.

The principle helped the early internet survive—implementations had bugs, networks were noisy, and forgiveness kept connections alive.

But think about what "liberal in what you accept" means for an attacker. Every variation a protocol tolerates is a variation someone can probe. Every edge case is a potential way in. Postel's Law made the early internet work. It also made it vulnerable.

4.2.7. THE END-TO-END PRINCIPLE

One more design choice shaped TCP/IP's security profile: the end-to-end principle.[38]

Where should the intelligence live—in the network, or at the edges? Cerf and Kahn chose the edges. Routers should be simple. They should forward packets and nothing more. All sophisticated work—reliability, security, application logic—should happen at the endpoints.

When you send an email, your computer and the recipient's server handle encryption. The routers in between see only scrambled bytes. They don't decrypt. They don't authenticate. They don't ask whether this packet should exist at all. That's not their job.

This made the network fast and flexible. New applications could be invented without changing infrastructure. HTTP didn't require router upgrades. Neither did email, file transfer, or video streaming.

But it also meant security was your problem. If you wanted encryption, you added it. If you wanted authentication, you added it. The network itself was just pipes.

The end-to-end principle made TCP/IP scalable. It also made TCP/IP trust-dependent. Every packet that reached your machine had passed through routers that neither knew nor cared whether that packet was legitimate. If your endpoint wasn't thinking about security, no one was.

4.2.8. THE UNSIGNED ENVELOPE

IP source addresses are claims, not proofs. TCP's handshake depends on sequence numbers, not identity. The entire design assumed that the machines completing a handshake were who they said they were, because in 1974 that assumption was nearly always correct.

The ARPANET connected a few dozen institutions. Everyone involved had met at conferences. The social layer—reputation, accountability, professional consequences—did the work that cryptographic verification would have done. That social layer didn't scale.

4.3. CHRISTMAS IN SAN DIEGO

Christmas Day, 1994. Tsutomu Shimomura was not at home.[39] He was in the Sierra Nevada, skiing, celebrating the holiday away from his computers. At 2:09 PM Pacific time, his computers were under attack.

Shimomura worked at the San Diego Supercomputer Center and was one of the best security researchers in the field. His machines contained tools for analyzing cellular phone networks, among other sensitive capabilities, and someone wanted those tools. The attacker knew what he was doing.

The attack required combining three techniques. Each exploited a different weakness in TCP/IP.

First: the SYN flood. Shimomura's network included a machine called SDSC-X, which trusted another machine called SDSC-S. The trust was expressed through a Unix file called .rhosts—a relic of simpler times that said, in effect, "if someone logs in from this machine, believe them."

The attacker started by crippling SDSC-S. He sent it hundreds of SYN packets—the opening moves of TCP handshakes—but never completed the connections. Each half-open connection consumed resources, filling buffers, exhausting the machine's capacity to respond. It was not destroyed; it was silenced.

Why silence the machine? Because the next step required pretending to be that machine. If SDSC-S was awake, it would receive unexpected packets and send back error messages exposing the deception.

Second: IP spoofing. With SDSC-S incapacitated, the attacker sent a SYN packet to SDSC-X, but the source address was not his own. It claimed to be from SDSC-S—the machine SDSC-X trusted.

IP had no way to verify source addresses. The protocol accepts whatever address the packet claims. SDSC-X believed the connection request came from its trusted neighbor.

SDSC-X received the SYN and responded with a SYN-ACK. That SYN-ACK traveled to SDSC-S, the address listed on the incoming packet. But SDSC-S was drowning in connection requests. It could not respond.

Third: sequence number prediction. Now the attacker faced the hard part. He needed to complete the handshake by sending an ACK with the correct sequence number, but he had never received the SYN-ACK—it had gone to SDSC-S. He was blind, so he guessed.

The attacker had already probed the network, opening legitimate connections to learn how the target machine chose its sequence num-

bers. He had identified the pattern, calculated what the sequence number should be, and sent an ACK with that value. He was right.

The connection was established. As far as SDSC-X could tell, SDSC-S had just logged in. The trust relationship activated. The attacker sent a command:

```
echo '+ +' >> /.rhosts
```

Two characters and a space, appended to the root user's trust file. Those characters meant "trust everyone, from everywhere." The machine was open.

The attacker explored at leisure. He copied files, including the cellular monitoring software Shimomura had developed. He left a voicemail taunting Shimomura. He vanished.

The attacker was Kevin Mitnick. Mitnick was already notorious—he had been arrested twice for computer crimes, had served time, had been released with strict conditions against using computers. None of it stopped him. He was compulsive, brilliant, and now in possession of tools that could intercept cellular phone calls.

Shimomura learned about the attack two days later, when a colleague noticed anomalies in the logs. What he found enraged him: someone had walked through his defenses using techniques he himself had contemplated, and the taunt on his voicemail turned professional curiosity into pursuit. He began hunting.

The chase lasted weeks. Shimomura worked with the FBI, with phone company technicians, tracing Mitnick's electronic footprints. The irony was exquisite: Shimomura used the same cellular monitoring capabilities Mitnick had stolen to track him down.

On February 15, 1995, at two in the morning, FBI agents arrested Kevin Mitnick at his apartment in Raleigh, North Carolina.[39] Shimomura was there. According to some accounts, Mitnick greeted him from the doorway: "Hi, Tsutomu. Congratulations." The hunt was over.

The attack revealed something troubling about TCP/IP. IP forwarded packets without verifying their source addresses, TCP completed handshakes based on sequence numbers, and the trust relationships in .rhosts files granted access based on IP addresses. Nothing broke. Nothing needed to break. The flaw lay in what the protocols assumed.

IP assumed source addresses were honest. TCP assumed sequence numbers were unpredictable. The trust model assumed IP addresses represented identity. Mitnick exploited all three assumptions simultaneously. IP spoofing gave him the false identity. Sequence number prediction let him complete the blind handshake. The trust relationship gave him root access. Three weaknesses, one attack, no bugs required.

The response came quickly, at least by internet standards.

Researchers had documented TCP sequence number weaknesses years before the attack. Robert Morris described the vulnerability in 1985[37], and Steve Bellovin published a detailed analysis in 1989.[40] The Mitnick attack proved these theoretical concerns were practically exploitable. RFC 1948, published in May 1996, proposed cryptographically random initial sequence numbers.[41] Operating systems began implementing the fix.

SYN cookies—a clever technique for handling SYN floods without exhausting server resources—were invented and deployed. Network administrators learned to filter packets with spoofed source addresses at network borders. The .rhosts mechanism fell out of favor, replaced

by SSH and other protocols that didn't trust IP addresses for authentication.

The protocols themselves did not change. RFC 791 and RFC 793 still describe IP and TCP as they did in 1981. But the implementations evolved. The design choices Cerf and Kahn made in 1973 were patched over, worked around, compensated for.

The patches are imperfect. IP spoofing remains possible on many networks. DDoS attacks—the descendant of Mitnick's SYN flood—still afflict the internet. Source address verification exists but is not universally deployed. The trust model that made the internet possible remains embedded in its foundations.

We cannot start over. Flag Day was the last chance to break everything at once, and that was 1983. Every computer on Earth now speaks TCP/IP. The protocols are frozen by their own success. So the fixes came as layers: TLS to encrypt, firewalls to filter, authentication protocols to verify what the handshake never could.

When the network grew into something larger, stranger, more adversarial, the protocols did not change. Only the world around them did.

The sketch on the envelope still describes the handshake.

SYN, SYN-ACK, ACK. Three packets, connection established. Cerf and Kahn worked it out in a hotel lobby in 1973, and the notation they used then is the notation running now—on every smartphone, every server, every router between them. As of 2025, the internet handles billions of TCP handshakes per second. The envelope from the Cabana Hyatt is long gone. The diagram on it is not.

What they could not sketch was the scale. The ARPANET had 37 hosts when they started writing. The internet now connects five billion

people. The assumptions built into those three packets—that source addresses are honest, that sequence numbers are unpredictable, that the machines completing a handshake are who they claim to be—were reasonable for 37 hosts. They are still running on five billion. The gap between what the sketch imagined and what it became is not a design flaw. It is the measure of what two engineers in a hotel lobby accidentally built.

`TCP/IP` carried packets across the globe. But a packet arriving at its destination network still faced one more problem: how to find the specific machine? `IP` addresses were logical, abstract—Ethernet cables connected physical hardware. Another protocol, written by an MIT programmer who described the network as a "jungle," would solve that problem. Its trust model would prove just as fragile.

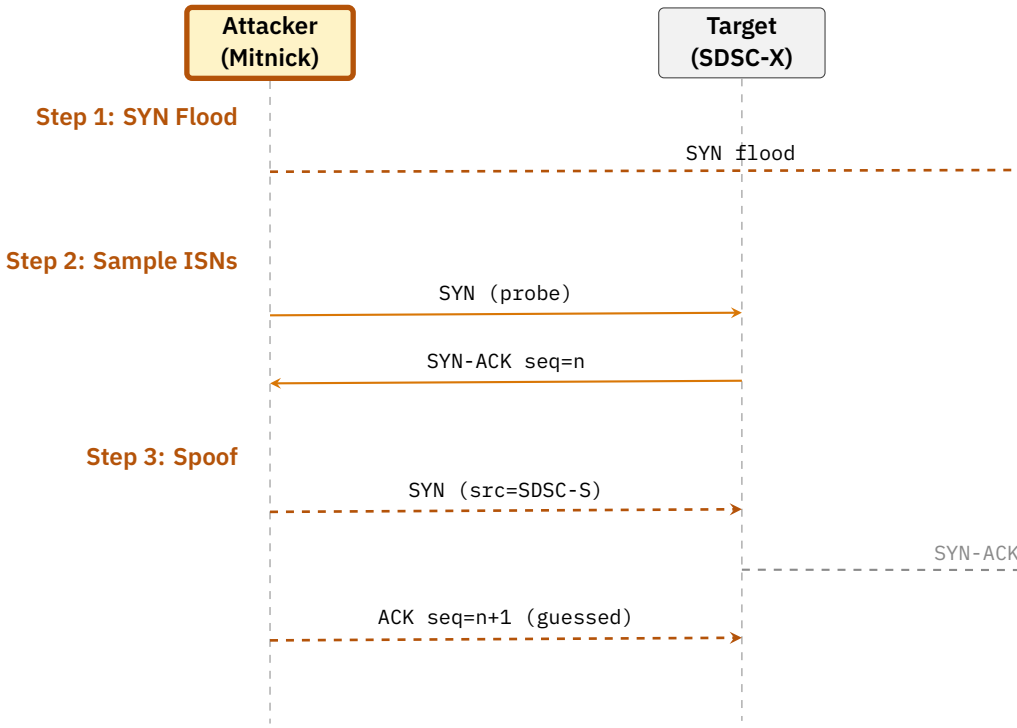


Figure 4.3: The Mitnick attack chain. Mitnick first silences the trusted machine with a SYN flood (Step 1), samples sequence numbers from the target (Step 2), then spoofs a connection from the trusted machine using predicted sequence numbers (Step 3). The SYN-ACK goes to the silenced machine, which cannot respond.

CHAPTER 5

THE FORGOTTEN PROTOCOLS

The world is a jungle in general,
and the networking game contributes many animals.

— David C. Plummer, RFC 826, November 1982

5.1. THE EMBARRASSING PROBLEM

David Plummer had an embarrassingly simple problem.

November 1982. Plummer was working at MIT with Ethernet cables snaking across the computer lab floor. Ethernet—the physical network standard that would connect most of the world's computers—had just arrived. The DEC, Intel, and Xerox consortium had published their specification two years earlier. Now 10-megabit coaxial networks were spreading through universities and research labs.[42] Plummer could see the future of local networking laid out before him. What he couldn't see was how to deliver a packet to the right machine.

The difficulty: the Internet Protocol gave every machine a logical address, a 32-bit number like 10.0.0.1. Ethernet gave every machine a physical address, a 48-bit number assigned to the network card at the

factory—though it can be changed in software, a fact that would matter later. IP knew where a packet should go, Ethernet knew how to get it there—but nothing connected the two.

Consider a packet arriving at a router that determined the destination, 10.0.0.5, was on the local network. The router knew the IP address, but the Ethernet hardware needed a MAC address—the 48-bit hardware identifier like `00:1A:2B:3C:4D:5E`. Since the router had no idea which MAC address corresponded to 10.0.0.5, the packet could not be delivered.

The obvious approach was a table. An administrator could maintain a file listing every IP address and its corresponding MAC address, updated by hand whenever machines were added or moved. But tables didn't scale. By 1982, networks were already too large and too dynamic for manual mapping.

Plummer's solution was to ask.

The Address Resolution Protocol, published as RFC 826[43], worked by asking. When a machine needed to find the hardware address for an IP address, it simply broadcast the question to the local network. Some machine would answer. No central registry, no configuration files, no administrator required.

Plummer acknowledged the collaborative development. "This protocol is the result of a great deal of discussion with several other people," he wrote in RFC 826, "most notably J. Noel Chiappa, Yogen Dalal, and James E. Kulp." This was how the early internet worked: small groups hashing out solutions over whiteboards and email, crediting each other in the RFCs.

One assumption Plummer didn't question: the machines telling you their addresses were telling the truth.

Why would anyone lie? The machines on a local network were physically present in the same building, often the same room. They belonged

to the same organization, plugged in by the same administrators. If someone claimed an IP address, you believed them.

The locked door was the authentication mechanism.

This assumption made sense in 1982. The internet had roughly 500 hosts. A local network might have a few dozen machines, all owned by the same university department. The concept of an attacker on the local network didn't fit the mental model.

The local network was, by definition, local. Trusted.

Nine months earlier, Postel had published a different protocol with a similar assumption: the Internet Control Message Protocol, ICMP, published as RFC 792.[44]

The problem ICMP solved was different from ARP's. IP was "best effort" delivery. Packets were sent into the network and might or might not arrive. If they didn't arrive, no one knew why.

ICMP provided a vocabulary for network problems. A router that couldn't reach a destination could send back "Destination Unreachable." A router that was too busy could send "Source Quench." If a packet bounced around too long, "Time Exceeded" explained its fate. Network engineers could finally understand what was happening inside the black box of packet routing.

The most useful message type was the simplest. Type 8 was Echo Request: "Are you there?" Type 0 was Echo Reply: "Yes, I'm here." Together, they became the basis for the `ping` command, a tool network administrators still use daily, forty years later.

"ICMP messages are sent in several situations," Postel wrote: "when a datagram cannot reach its destination, when the gateway does not have the buffering capacity to forward a datagram, and when the gateway can direct the host to send traffic on a shorter route."

That last situation was the Redirect message. If a router received a packet and knew a better route, it could tell the sender: "Next time, send traffic for that destination to this other gateway." The sender

would update its routing table and use the better path. Network efficiency, achieved through helpful suggestions.

Postel included a disclaimer. "The Internet Protocol is not designed to be absolutely reliable. The purpose of these control messages is to provide feedback about problems in the communication environment, not to make IP reliable."

Control messages for routers, feedback for diagnostic tools, helpful infrastructure for administrators debugging problems. The expectation was that `ICMP` would serve the network itself. Error messages were helpful, not weaponized.

Together, `ARP` and `ICMP` occupied a strange position in the protocol stack, living below the applications users saw, below even `TCP` and `UDP`. They were infrastructure, invisible to anyone not running a network. Most programmers gave them no thought. Most users had never heard of them.

But both protocols shared the same premise that ran through every RFC of the era: the network connected a small group of cooperating institutions. `ARP` expected your neighbors wouldn't lie about their identity. `ICMP` expected error messages would be helpful, not hostile. Both expectations were encoded into protocols that would become as permanent as the internet itself.

The few hundred sites connected by phone lines would become a world of strangers—300,000 hosts by 1990, over 100 million by 2000. The protocols that assumed everyone was trustworthy would remain unchanged, running on every device, vulnerable to anyone who chose to lie.

5.2. TRUST ON THE WIRE

5.2.1. THE ADDRESS GAP

Every networked computer has at least two addresses. The IP address tells the network where the machine lives in the global hierarchy—which network, which subnet. The MAC address, assigned to the network interface at the factory (though software can override it), is how devices on the same wire recognize each other.

These two addresses have nothing to do with each other. The IP address is assigned by software; the MAC address is stamped into hardware. When you want to send a packet to your neighbor at 192.168.1.5, you know the IP address, but the Ethernet hardware needs the MAC address—the 48-bit identifier that makes one network card distinguishable from another.

How do you find out? You ask—and you believe whatever answer you get.

The ARP exchange is a two-step dance.

First, the requesting machine broadcasts an ARP request. The broadcast goes to every device on the local network. The request says, in effect: "I'm looking for 192.168.1.5. If that's you, please tell me your hardware address. I'm at 192.168.1.1 with hardware address AA:BB:CC:DD:EE:FF."

Every machine on the local segment receives this broadcast. Most ignore it—the IP address in the request isn't theirs. But the machine at 192.168.1.5 recognizes itself and sends a response, directly to the requester: "That's me. My hardware address is 11:22:33:44:55:66."

The requester now has the mapping it needs. It caches this information—storing the IP-to-MAC mapping in a local table called the ARP cache. The next time it needs to send a packet to 192.168.1.5, it

doesn't need to broadcast again. It just looks up the hardware address and sends directly.

Notice what's missing: nothing verified that the response came from the real 192.168.1.5, and nothing proved ownership of the IP address. The machine asked a question and believed the first answer it received.

Cache entries expire after a timeout—typically five to twenty minutes. When an entry expires, the next packet triggers a new ARP request. But for those five to twenty minutes, whatever is in the cache is treated as truth. Poison the cache once, and you control the traffic until the entry expires.

Sometimes a machine announces its own address without being asked. This is called gratuitous ARP: an unsolicited ARP reply that says "I'm 192.168.1.5 at hardware address 11:22:33:44:55:66."

Gratuitous ARP has legitimate uses. When a machine boots up, it can announce itself to the network, populating the ARP caches of any machines that were waiting for it. When a server fails over to a backup, the backup can send a gratuitous ARP to inform the network of the new hardware address for the same IP.

But no mechanism prevents a malicious machine from sending gratuitous ARP replies with false information. No verification confirms that the sender really owns the IP address it claims. The receiving machines update their caches automatically, trusting the announcement. One forged packet, and every host on the segment believes the gateway is at the wrong address.

5.2.2. ICMP: THE ERROR VOCABULARY

ICMP rides inside IP. Every ICMP message is delivered as an IP packet, with the protocol field set to 1. (By comparison, TCP is protocol 6 and UDP—a simpler, faster cousin of TCP that doesn't guarantee

delivery—is protocol 17.) The ICMP header starts with a Type field (what kind of message), a Code field (more specific details), and a checksum. The rest varies by message type.

ICMP speaks in message types, and each type tells your machine something about the network. The question is whether to believe it.

Type 8 is Echo Request—"Are you there?"—and Type 0 is Echo Reply—"Yes, I'm here." These two became the `ping` command: simple, helpful, and completely unauthenticated, since anyone can forge either side of that conversation.

Type 3 is Destination Unreachable—the network telling you something went wrong, whether the host is down, the port is closed, or the packet simply couldn't be delivered. But what if that message came from an attacker trying to tear down your connection? Your machine accepts it without proof of origin.

Type 11 is Time Exceeded, sent when a packet bounces around too long, its time-to-live counter hits zero, and a router discards it. `Traceroute` depends on these messages—and so does an attacker mapping your network infrastructure.

Type 5 is Redirect. A router is telling you: "Send traffic for that destination through this other gateway instead." Your machine updates its routing table and obeys. What if the "router" is an attacker? What if the "better path" runs through their machine?

Every message type expects honest senders, and none of them verify identity.

From these building blocks, network engineers built two essential diagnostic tools.

`Ping` sends Echo Requests to a destination and listens for Echo Replies: if replies come back, the destination is alive and reachable, and the round-trip time indicates network latency. Silence suggests a problem. The tool is simple, useful, and completely unverified—you trust that the reply came from where it claims.

`Traceroute` exploits the Time Exceeded message by sending a packet with TTL set to 1; the first router decrements TTL to zero and sends back Time Exceeded, revealing itself. Then `traceroute` sends another packet with TTL set to 2, and the second router responds. The process continues, one hop at a time, mapping the path from source to destination.

These tools have been part of every network administrator's toolkit since the 1980s. They rely entirely on `ICMP` messages that can be sent by anyone, to anyone, about any packet—real or fabricated.

`ARP` and `ICMP` trust differently than `Telnet` or `FTP`—not the network path, not client declarations, but the identity of neighbors, the machines sitting next to you on the same wire.

When a machine broadcasts "I am 192.168.1.5," every other machine on the segment believes it. `ARP` includes no verification, no signature, no proof. The protocol cannot tell the difference between a legitimate announcement and a lie. `ICMP` trusts similarly: when a router says "Destination Unreachable" or "Use this better route," the recipient believes it.

The security model was physical: if a machine could broadcast on your local network, it was plugged into your building's wiring, owned by your organization, managed by your colleagues. Physical proximity implied authorization, and the locked door protected the network better than any cryptography could.

5.2.3. THE HONEST NEIGHBOR

`ARP` has no CVE—it cannot, because the protocol's trust model, that neighbors on a network segment are honest, is the vulnerability itself. There is nothing to patch, since the CVE system tracks bugs in implementations, not flaws in design. The specification did what it said. The world did something else.

5.3. THE AMPLIFICATION ATTACK

5.3.1. WHEN TRUST BECAME LIABILITY

On a summer morning in 1998, a system administrator at the University of Minnesota noticed something strange.[45] Network traffic was spiking. Servers were slowing down. It looked like a massive influx of users, except the pattern was wrong. The traffic was all ICMP—Echo Replies, thousands of them per second, pouring into the university's network from hundreds of different sources.

They called it Smurf, named after the program that automated it[46]—a mechanism simple enough to fit on a single floppy disk.

The impact was physical. At the Minnesota Regional Network (MRNet), the ISP serving the university, engineers watched their capacity evaporate. David Bergum, a senior engineer, described it as a "cyber-traffic jam" that lasted two and a half hours. "My T1 is melted," became the shorthand description for what happened when amplification filled the pipe.[47]

Here's how it worked. The attacker found networks that allowed directed broadcasts—you could send a packet to the broadcast address of a remote subnet, and every machine on that subnet would receive it. The attacker then sent ICMP Echo Requests to these broadcast addresses, but with a twist: the source IP address was forged. Instead of the attacker's real address, the packets bore the victim's forged address.

Every machine on the broadcast network received the Echo Request, every machine sent an Echo Reply, and every reply went to the victim—the spoofed source address.

If the broadcast network had 256 machines, one forged packet produced 256 replies. If the attacker found ten such networks and sent a hundred packets per second to each, the victim received 256,000 packets per second—all legitimate ICMP traffic from hosts that believed they were responding to a genuine `ping`.

The amplification was devastating. Someone with a slow dial-up connection could overwhelm a target with a fast T1 line. The "Smurf amplifier" networks became unwitting weapons, their helpful Echo Replies flooding victims into oblivion.

Lists of vulnerable networks circulated in hacker forums. Attackers called them "Smurf amplifiers." Some networks became notorious for their size—more machines meant more amplification. The University of Minnesota wasn't being targeted because anyone cared about Minnesota. It was being used as a reflector, its machines sending Echo Replies to victims elsewhere.

ICMP was doing what the specification said it should do: Echo Requests received Echo Replies, error messages were helpful, and the protocol performed correctly. It just never anticipated someone lying about who was asking.

The Smurf attack wasn't the first time ICMP became a weapon. A year earlier, in late 1996, system administrators started noticing machines crashing for no apparent reason.[48] Windows workstations. Unix servers. Macintosh computers. All dying the same way, freezing and requiring hard reboots.

The cause was called the Ping of Death. The exploit was almost embarrassingly simple.

The IP protocol specified that packets could be up to 65,535 bytes—the maximum value in a 16-bit length field. But ICMP Echo Requests were supposed to be small, a few dozen bytes at most. What happened if you sent an oversized Echo Request?

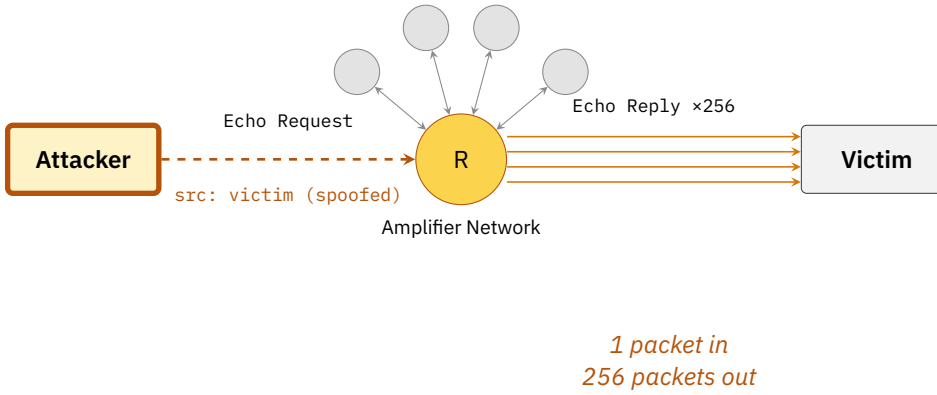


Figure 5.1: Smurf amplification attack. The attacker sends a single ICMP Echo Request to a broadcast network with the victim's IP as the spoofed source address. Every host on the amplifier network responds, flooding the victim with Echo Replies. One packet becomes 256.

On a properly implemented system, nothing—the packet would be discarded. But many implementations had a bug: they allocated a fixed-size buffer for incoming ICMP data, trusting that the data would fit. When an oversized packet arrived, the data overflowed the buffer, memory was corrupted, and the system crashed.

The attack could be launched from a single command line. On Windows: `ping -l 65510 victim`. On Unix: `ping -s 65527 victim`. One command, typed from anywhere, could crash a remote machine.

The vulnerability affected nearly every operating system—Microsoft patched Windows, Apple patched macOS, and every Unix vendor issued fixes. But the attack demonstrated something troubling: ICMP, designed for helpful diagnostics, could crash machines remotely, without authentication, without warning.

ARP's reckoning came more quietly, spreading through coffee shops and hotel lobbies rather than making headlines.

Security researchers called it ARP cache poisoning, sometimes ARP spoofing. The mechanism was straightforward. Someone on the same local network as a victim could send forged ARP replies, claiming that the gateway's IP address was at their own MAC address. The victim's ARP cache would update automatically, trusting the announcement. From that moment on, all the victim's outbound traffic would flow to the attacker instead of the gateway.

They could read the traffic, modify it, or simply forward it to the real gateway after making copies. The victim would have no idea. Their connection would work normally—perhaps slightly slower—while every packet passed through a stranger's machine.

This was an interception attack—sometimes called man-in-the-middle or MITM—at the network layer, before encryption could help. If the victim was browsing HTTP websites, everything was visible: usernames, passwords, session cookies, private messages. If the victim logged into their email, the credentials were captured. If the victim checked their bank balance, the numbers were exposed.

The attack required being on the same local network as the victim. In 1982, when Plummer wrote RFC 826, this seemed like a significant limitation. Local networks were corporate offices, university labs, research facilities. The locked door was the security model.

By the 2000s, local networks were everywhere. Coffee shop WiFi. Hotel business centers. Airport lounges. Conference venues. Coworking spaces. Anywhere that offered public wireless access, anyone could join the local network. The locked door had vanished, but ARP hadn't changed.

Tools made the attack trivial. The dsniff package[49] included a program called `arpspoof`. Two commands in a terminal window were enough to intercept all traffic between a victim and their gateway. Ettercap provided a graphical interface, point-and-click network inter-

ception. Cain & Abel ran on Windows, letting anyone with a laptop become an eavesdropper.

Security researchers demonstrated the attacks at conferences, published papers, and warned about the risks of public WiFi, but most users had never heard of ARP and didn't understand what was at stake. The defenses, when they came, were partial and imperfect.

For ARP, network administrators could configure static ARP entries, manually specifying the IP-to-MAC mappings for critical devices. But static entries don't scale: they break when hardware changes and require constant maintenance on networks with hundreds or thousands of devices.

Dynamic ARP Inspection, a feature on enterprise network switches, could validate ARP packets against known DHCP bindings. If a machine received its IP address from DHCP, the switch recorded the relationship between IP and MAC. Subsequent ARP traffic would be checked against this record. But DAI required managed infrastructure, expensive switches, careful configuration. Coffee shop WiFi didn't have it. Home networks didn't have it.

The most widespread mitigation for ARP spoofing was HTTPS. If the traffic itself was encrypted, intercepting it revealed nothing useful. Encrypted packets would flow past, but their contents remained unreadable. The push to encrypt everything—HTTPS Everywhere, certificate transparency, browser warnings for unencrypted connections—was partly a response to the impossibility of securing the lower layers.

For ICMP, the Smurf attack prompted RFC 2644[50], which recommended that networks disable directed broadcasts. Ingress filtering, described in BCP 38[51], recommended that networks reject packets with obviously spoofed source addresses. If a packet claiming to come from 192.168.1.1 arrived on an interface where 192.168.1.0/24 wasn't reachable, the packet was clearly forged and should be dropped.

Both recommendations were sound, but neither was universally adopted. Networks that followed the guidelines stopped being Smurf amplifiers; networks that didn't remained available for abuse. The attack eventually faded, not because the vulnerability was fixed, but because attackers found more effective techniques—botnets that could generate traffic directly rather than relying on amplification.

The Ping of Death was patched at the operating system level, where bounds checking, properly implemented, rejected oversized packets before they could overflow buffers. But the lesson—that ICMP could be weaponized, that helpful infrastructure could crash systems—echoed through the security community.

ICMP Redirect attacks persisted longer, since a forged Redirect message could tell a victim to route traffic through a hostile machine, achieving the same interception position as ARP spoofing. Some operating systems started ignoring ICMP Redirects entirely, or accepting them only from the current gateway, but legacy systems remained vulnerable and the attack remained in penetration testers' toolkits.

Forty years after Plummer and Postel published their RFCs, the protocols run unchanged on every network.

We can watch them with a packet sniffer: connect to any WiFi network and capture traffic. ARP requests broadcast constantly—"Who has 192.168.1.1? Who has 192.168.1.254?"—while ICMP Echo Requests fly as someone pings a server and Time Exceeded messages appear as someone runs `traceroute`. The infrastructure of the 1980s, still operating, still trusting, still vulnerable.

The protocols cannot be fixed without breaking compatibility with billions of devices. ARP has no authentication because adding it would require changing every network card, every router, every switch. ICMP has no verification because the installed base of IP implementations assumes the current packet format.

Instead, we build compensating controls: encryption at higher layers, network segmentation, monitoring for unusual ARP traffic, firewalls that block ICMP from untrusted sources. Each defense adds complexity, costs money, and requires expertise—each attempts to secure what was designed to be insecure.

David Plummer, writing in 1982, noted that "hosts don't transmit information about anyone other than themselves." He meant it as a feature—no machine could lie about another machine's address—but a machine could lie about its own. In a world where anyone can join the local network, that lie is enough.

Jon Postel, in RFC 792, described ICMP as "feedback about problems in the communication environment." The feedback was genuine and the problems were real, but the feedback mechanism itself became the problem—error messages designed to help administrators became weapons for attackers.

A network that verifies every packet, authenticates every address, validates every error message—such a network would be slower, more complex, harder to build. The engineers of 1982 made reasonable choices for the networks they had. Those choices became permanent. The networks changed. The protocols didn't.

Right now, on your local network, ARP is asking who has the gateway address.

The question goes out as a broadcast, invisible, automatic, running whether your device is idle or busy. Some machine answers. Your device believes it without asking for proof. The mapping goes into the cache. Packets flow. You never see any of it.

This exchange has been happening on every Ethernet network since 1982. The protocol has not changed. The question is the same. The

belief is the same. As of 2025, billions of devices ask and answer these questions every second—on home networks, in corporate offices, at every coffee shop and hotel with WiFi. The locked door that Plummer assumed would protect these conversations is long gone. The conversations continue anyway.

The protocols of the 1980s, still running, still believing every answer they receive. And ARP is not alone. While Plummer and Postel were designing how machines would find each other on local networks, another system had already taken shape—one that would become the internet's most-used application. It began with a researcher at BBN, a nearly forgotten key on a Model 33 Teletype, and a decision made in seconds that would outlast everything else he built.

CHAPTER 6

THE ELECTRONIC POST OFFICE

The objective of Simple Mail Transfer Protocol (SMTP) is to transfer mail reliably and efficiently.

— *RFC 821, Jon Postel, August 1982*

6.1. THE LEAST-USED KEY

Late 1971. Ray Tomlinson sits at his desk at Bolt Beranek and Newman in Cambridge, Massachusetts.[52] The office is cluttered with engineering detritus: printouts, manuals, coffee cups growing cold. BBN occupies a converted candy factory in an industrial neighborhood, a setting matching its culture: practical, unpretentious, focused on making things work. The company has some of the best computer scientists in the country, building the ARPANET's guts—the Interface Message Processors, the IMPs—since 1969.

Tomlinson is staring at his Model 33 Teletype keyboard. He has a problem nobody asked him to solve.

The ARPANET can connect computers. Researchers at UCLA can log into machines at MIT. But you cannot leave a message for someone

who isn't there. Local mail programs exist—you can write to someone else who uses your same computer—but the network changes nothing. If you want to reach a colleague at Stanford, you still call them on the telephone.

Tomlinson has been working on a file transfer protocol. Along the way, he's modified a local mail program called SNDMSG to send messages between machines. The technical part works. Now he needs a way to address these messages—some notation indicating which user on which computer should receive them.

He looks at the keyboard. Forty-odd characters stare back. He needs a symbol that won't appear in anyone's name, something separating "user" from "host" in a way that's obvious. The percent sign? Already used. The dollar sign? Too closely associated with money. The ampersand? Too common in company names.

His eyes land on @.

It's the least-used character on the keyboard. It shows up in accounting contexts—"10 widgets @ $5 each"—but almost nowhere else. And it has a meaning that fits perfectly: "at." User at host. smith@bbn-tenexa.

Tomlinson sends a test message from one PDP-10 to another sitting a few feet away. He doesn't remember what it said. "Something like QWERTYUIOP," he would later recall.[52]

The most important message in the history of communication, and nobody wrote it down.

The @ sign became the most recognized symbol of the digital age. Tomlinson had chosen it in a few seconds, without a committee meeting or formal proposal. A hack, in the original sense: a quick solution to a practical problem, implemented fast because it needed to exist.

What Tomlinson didn't design—what nobody designed—was any way to verify who was sending a message. When he typed QWERTYUIOP into his terminal, the message said it came from him because

he said it came from him. The receiving computer believed it because computers, in 1971, believed what they were told.

Email spread through the ARPANET like a weed. The network had been built to share computing resources—expensive mainframes researchers could log into remotely. The funding agencies at DARPA expected packet switching to prove itself through remote computing. Email wasn't on anyone's roadmap.

But by 1973, a study found that 75% of ARPANET traffic was email.[53]

The finding shocked administrators. They had built a highway for compute jobs. Instead people were sending notes to each other. Researchers had discovered that sending messages was more useful than sharing computers. A physicist at Stanford could pose a question at 3 AM and wake up to answers from MIT, UCLA, and BBN. Email collapsed the tyranny of time zones and the telephone, which demanded both parties be present simultaneously.

Email became the application that justified the network.

The message format evolved through trial and error as different sites developed different conventions. Headers appeared—From, To, Subject, Date—and some systems added Cc for carbon copies, an anachronism from the age of typewriters and onionskin paper. Bcc followed, letting senders hide some recipients from others. The format was informal, invented by engineers who needed it to work.

By the late 1970s, email had become indispensable, but the chaos was becoming a problem. Messages between different systems sometimes garbled headers or lost formatting; character encodings differed, line lengths varied, and an email that looked perfect on one machine arrived as gibberish on another. Something had to be standardized.

In August 1982, two RFCs appeared that would define email for the next four decades.[53] Postel wrote RFC 821, specifying SMTP—Simple Mail Transfer Protocol—the rules for how mail servers would relay messages.[54] The same month, David Crocker published RFC 822, defin-

ing the format of email messages: what headers should exist, how addresses should look, where the body begins.[55]

Together, these two documents built the electronic post office. RFC 821 defined how post offices pass mail. RFC 822 defined what letters look like.

Both were simple, direct, and built on a premise that would prove disastrous: senders tell the truth about who they are.

Postel, writing SMTP, wasn't designing for adversaries. The ARPANET connected universities and research labs—the same community that had built every protocol in this book so far. If a message purported to be from a researcher at MIT, it probably did. Forging email would have been professional suicide. The community was small enough that everyone would have known. The word "security" appears exactly zero times in the original RFC 821 specification. Not because Postel was negligent, but because securing email against malicious actors was outside the problem space he was solving.

The protocol reflected this culture. A sending server announces itself with HELO. It declares where the mail is from with MAIL FROM. It lists the recipient with RCPT TO. Then it transmits the message body. At no point does the receiving server verify any of these claims. It accepts the sender's word. This wasn't negligence. It was efficiency. Adding verification would have required infrastructure that didn't yet exist. The researchers wanted email to work, and work quickly. They got what they wanted. What they also got, without knowing it, was the architecture of spam, phishing, and billion-dollar fraud.

RFC 821 defined just fourteen commands.[54] A programmer could read the specification in an afternoon and implement a working mail server by evening. SMTP was human-readable by design. You could telnet to port 25 on a mail server and type commands by hand. This transparency was intentional. Debugging was easy when you could see

exactly what the machines were saying to each other. The simplicity that made SMTP easy to implement also made it easy to abuse.

6.2. THE ENVELOPE AND THE LETTER

6.2.1. HOW EMAIL WORKS

When we send an email, we're not really sending it. We're asking a series of servers to carry it for us, like passing a letter through a chain of post offices until it reaches the right one. The metaphor isn't perfect, but it's close. Email works through "store and forward": the mail program gives the message to a server, that server stores it temporarily, then forwards it to the next server, which stores it again, and so on until the message reaches the recipient's mailbox. The message might pass through three or four servers on its way across the internet.

But what if the next server is unavailable? Unlike a phone call, which requires both parties present, email stores messages at each hop. If the next server is down, the current server holds the message and tries again later—for days, if necessary. This resilience made email work even when network connectivity was unreliable.

And in the 1970s, connectivity was always unreliable.

The store-and-forward design also meant each server in the chain had to believe the previous one. The message that arrives at hop three claims to have come from hop two, which claims it came from hop one. No server verifies these claims. The message passes through the system on faith.

The protocol that governs this relay is SMTP. It's a conversation in plain text that any programmer can read. When one mail server wants

to deliver a message to another, it opens a connection and they begin talking. The conversation follows a rigid pattern—but watch what's missing. No one checks credentials. No one verifies identity. The dance proceeds on faith alone.

```
Server:  220 mail.example.com SMTP Ready
Client:  HELO sender.example.com
Server:  250 Hello sender.example.com
Client:  MAIL FROM:<alice@sender.example.com>
Server:  250 OK
Client:  RCPT TO:<bob@example.com>
Server:  250 OK
Client:  DATA
Server:  354 Start mail input
Client:  From: alice@sender.example.com
Client:  To: bob@example.com
Client:  Subject: Hello
Client:  This is the message body.
Client:  .
Server:  250 OK
Client:  QUIT
Server:  221 Bye
```

The client says HELO and claims an identity; the receiving host acknowledges without checking. The client announces who the mail is from—OK, no verification. The client names the recipient—OK again—then DATA, the message body, and a period to end.

Every server response starts with a three-digit code: 250 means "OK, I accepted what you said"—not "I verified what you said," just "I accepted it."

Codes starting with 4 mean temporary failures, while codes starting with 5 mean permanent ones. Postel specified exactly what each code

meant, but he did not specify how to verify the claims that triggered them.

The response code system borrowed from FTP and would influence HTTP decades later. The codes let automated systems make decisions without parsing human-readable text. A sending server knows immediately whether the transaction succeeded—or at least, whether the receiving server accepted the claims it was given.

Two computers, speaking in plain English commands, passing messages between them. A programmer in 1983 could watch the conversation scroll by on a terminal and understand exactly what was happening. But look closely at that conversation. The client claims to be `sender.example.com`. The client claims the mail is from `alice@sender.example.com`. The server says OK to both. It doesn't check whether those claims are true. It can't. SMTP provides no mechanism for verification.

To understand email's vulnerabilities, we need to understand that every message has two separate parts: the envelope and the contents. The envelope consists of the SMTP commands—MAIL FROM and RCPT TO—the routing instructions that servers process along the way. They tell each server where the message came from and where it's going. The contents are different: the headers and body that come after DATA. The From: line, the To: line, the Subject:—these are what the recipient sees when they open the message. They're part of the message, not part of the envelope. And here's the problem: the envelope and the contents can say different things. They often do.

```
SMTP Envelope:              Message Headers:
MAIL FROM:<attacker@evil.com>    From: ceo@yourcompany.com
RCPT TO:<victim@yourcompany.com> To: victim@yourcompany.
    com
                                 Subject: Urgent Wire
    Transfer
```

The envelope says the real sender is `attacker@evil.com`. The headers say the message is from the CEO. Most email programs show you the headers, not the envelope. The recipient sees only what the attacker wants them to see. Consider the postal analogy: you can drop a letter in a mailbox with any return address printed on the envelope, and the mail carrier will never verify you live there. You could impersonate anyone. That's how SMTP works.

But how does your server know where to deliver the message? When you send email to `bob@example.com`, your server asks DNS: "Who handles mail for example.com?" The answer comes from MX records—special DNS entries listing the servers that accept incoming mail for that domain.

```
$ dig MX example.com
example.com.    3600    IN    MX    10 mail.example.com.
example.com.    3600    IN    MX    20 backup.example.com.
```

Your server looks up example.com's MX records, finds `mail.example.com` has priority `10` (lowest wins), and connects to deliver the message. If that server is down, it tries the backup. A single email address can resolve to different servers at different times. Organizations can migrate infrastructure without breaking anyone's address book. But this flexibility means the relationship between

an email address and a physical server is fluid—and fluidity creates opportunity for interception. DNS poisoning is a story for Chapter 7; for now, note that email's routing depends on answers that no one verifies.

6.2.2. THE SENDER'S WORD

SMTP's built-in expectation is perhaps the most consequential in this book: senders are who they claim to be.

The MAIL FROM address accepts any value, and so does the From: header. The protocol provides no verification mechanism, no callback to the claimed domain, no cryptographic proof. A server receiving mail cannot distinguish a legitimate message from the CEO from a forgery sent by an adversary in another country.

> RFC 821 --- August 1982 --- Jon Postel
>
> The first step in the procedure is the MAIL command. The <reverse-path> contains the source mailbox. This command tells the SMTP-receiver that a new mail transaction is starting... It gives the reverse-path which can be used to report errors. If accepted, the receiver-SMTP returns a 250 OK reply.

Postel specified what the sender provides, not what the server verifies. The distinction was not accidental. In 1982, verification would have required infrastructure that didn't exist—public key cryptography was barely understood, and key distribution was an unsolved problem. But the deeper reason was cultural. The ARPANET was a community where forging email would have been professional suicide. The expectation of honesty wasn't naive. It was accurate—for that community, in that era.

What happens when senders lie? Spam is the most visible problem: anyone can impersonate anyone, so servers cannot reject mail from forged addresses. Spam filters must examine message content because they can't trust the envelope, and the industry has poured billions of dollars into building better filters. The attackers have responded by making their spam look more like legitimate mail. The arms race never ends.

Phishing goes further. An attacker can send mail that appears to originate from your bank, your employer, or your government. The From: header will show the spoofed address. Most users cannot detect the forgery. Security training tells employees to hover over links, verify unexpected requests, be suspicious of urgency. But people are busy. The requests seem reasonable. The email looks right. Training helps, but it's not enough.

Business email compromise is the ultimate expression: an attacker impersonates a CEO, a vendor, or a lawyer, requesting wire transfers, sensitive documents, or credentials. The recipient sees a familiar name and follows instructions.

All three attacks exploit the same gap: SMTP accepts claims of identity without verification. The server followed its instructions perfectly. It just never expected adversaries.

6.3. THE FIRST SPAM

The first spam wasn't called spam—the word didn't exist yet.[56] On May 3, 1978, it was just an advertisement—and an outrage.

Gary Thuerk was a marketer at Digital Equipment Corporation, a 21-year-old company headquartered in Maynard, Massachusetts.

DEC had grown from three employees to tens of thousands, riding the wave of minicomputers that brought computing power out of glass-walled data centers and into engineering departments. Now Thuerk had new computers to sell: the DECsystem-20 and DECSYSTEM-2020, machines with built-in ARPANET support. He wanted to tell people about them. He had access to a list of ARPANET users on the West Coast—researchers, engineers, the people most likely to need such machines. Why not send them an invitation to a product demonstration?

Thuerk's marketing instincts were sound, even if his judgment about social norms wasn't. The ARPANET was small, its users interconnected, and the idea of advertising had never occurred to anyone because it was explicitly non-commercial. It was funded by the Department of Defense. Using it to sell things felt like hanging a billboard in a university library. The network's Acceptable Use Policy explicitly prohibited commercial activity. But policies are only as strong as their enforcement, and enforcement was nearly nonexistent. Thuerk saw only the opportunity: a list of names, the means to reach them, and a product to promote.

The technical limitations of the era shaped the message itself. Email addresses were long—usernames followed by machine names, without the DNS system that would later make addresses concise. Thuerk tried to cram all 393 addresses into the To: field. The software choked. Many addresses ended up spilling into the message body, making the already aggressive solicitation look even more chaotic.

He composed his message:

```
MAIL-FROM: THUERK at DEC-MARLBORO
SUBJECT: INVITE YOU TO COME SEE THE NEWEST
         MEMBER OF THE DECSYSTEM-20 FAMILY

DIGITAL WILL BE GIVING A PRODUCT PRESENTATION OF
THE NEWEST MEMBERS OF THE DECSYSTEM-20 FAMILY...

WE INVITE YOU TO COME SEE THE 2020 AND HEAR ABOUT
THE DECSYSTEM-20 FAMILY AT THE TWO DIFFERENT TIMES
AND DIFFERENT LOCATIONS...
```

Thuerk sent the message to 393 ARPANET addresses.[56] It was, by any reasonable definition, an advertisement sent in bulk to people who hadn't asked for it. It was spam, before the word existed.

The reaction was swift and furious. Richard Stallman, then at MIT, complained that the message violated network protocols and community norms. A Defense Communications Agency administrator objected that commercial use of the ARPANET violated federal policy. Major Raymond Czahor of DCA wrote to DEC demanding an explanation, threatening consequences if it happened again. Other researchers replied with scathing messages, calling the advertisement inappropriate and unwelcome.

The outrage was genuine. These researchers had built something for science, for collaboration, for the free exchange of ideas. Thuerk had used it to sell computers. He had treated their shared resource as a marketing channel.

Thuerk defended himself. He was just trying to reach potential customers. The ARPANET was there, the addresses were there, and sending was free. Why wouldn't he use it? DEC sold $12 million worth of computers from those product demonstrations.[56] The message of the first spam was not lost on marketers: bulk email works. The ARPANET

community could object all it wanted. Sending unsolicited messages cost nothing and produced revenue. The economics were irresistible.

For twenty years, spam remained a nuisance—annoying, but manageable. The internet grew, email grew, and so did the volume of junk. By the late 1990s, spam was a flood: "Make Money Fast" chain letters, Nigerian prince scams, advertisements for dubious pharmaceuticals. Users spent significant time each day deleting unwanted mail, but the problems went beyond annoyance. Email's lack of sender authentication enabled entirely new categories of fraud.

Phishing emerged as a term in the mid-1990s, describing attacks that fished for credentials.[57] The technique was simple: someone sends an email posing as America Online, asking users to "verify" their passwords, and the email links to a fake website where users enter their credentials, which flow to the attacker. The attack works because nothing in email proves the sender's identity. The name came from "fishing" with the "ph" spelling borrowed from "phreaking"—the phone hacking culture that preceded internet crime. By the 2000s, phishing had become sophisticated: fake bank emails indistinguishable from real ones, fake tax notices from the IRS, fake shipping notifications from FedEx. Each message contained a link to a carefully crafted website that harvested whatever information the victim entered.

The attacks evolved with their targets. Spear phishing narrowed the focus: instead of blasting millions of generic emails, attackers researched specific individuals and crafted messages tailored to them—a CFO might receive an email about an upcoming board meeting, while an executive assistant might get a message referencing a real travel itinerary. The more specific the details, the more believable the forgery.

The most damaging evolution came in the 2010s, with what the FBI calls Business Email Compromise—BEC. These attacks are targeted. BEC attackers research their victims carefully, identifying a company and its key personnel, learning who the CEO is, who the CFO

is, who handles wire transfers. They study the organization through LinkedIn, press releases, and regulatory filings. Then they send a carefully crafted email that appears to come from the CEO, addressed to someone in finance, with a subject like "Urgent Wire Transfer" or "Confidential Request."

The body asks for an immediate payment to a specified account—a new vendor, an acquisition, an investment—and the tone is urgent: the CEO is traveling and needs this handled quietly and quickly. The email looks real because it uses the CEO's name, the CEO's email address (spoofed), and references real company details. The receiving employee has no easy way to detect the forgery. SMTP doesn't verify sender identity, and the From: header says what the attacker wants it to say. The employee initiates the wire transfer, the money flows to an account controlled by the attacker, gets moved through multiple banks, and vanishes.

These attacks work at scale. The FBI's Internet Crime Complaint Center tracked $2.4 billion in BEC losses in 2021.[58] $2.7 billion in 2022.[59] The real numbers are certainly higher—many companies don't report incidents due to embarrassment or fear of regulatory scrutiny. The victims range from small businesses losing their operating capital to municipalities losing millions in tax revenue to nonprofits losing donations meant for the vulnerable. Behind each statistic is an employee who followed instructions, did their job, and discovered too late that the instructions were lies.

Some cases are spectacular. In 2016, a Lithuanian man named Evaldas Rimasauskas identified a gap in how Google and Facebook processed invoices from their Asian hardware suppliers. He registered a company in Latvia with the same name as Quanta Computer, a real Taiwanese manufacturer that did business with both tech giants. Then he sent invoices—dozens of them, over two years—from forged email addresses that appeared to belong to Quanta. The invoices looked

authentic, the payment requests seemed routine, and Google and Facebook, collectively, wired him over $100 million.[60] Rimasauskas was eventually caught and extradited, sentenced to five years in prison. But most BEC attackers operate from jurisdictions that don't cooperate with American law enforcement, and their victims never recover the money.

The attacks don't require technical sophistication—an attacker doesn't need to break encryption or exploit buffer overflows, just send an email from a forged address and convince someone to act on it. SMTP makes the first part trivially easy.

The email community has tried to patch these problems. Three technologies—SPF, DKIM, and DMARC—attempt to add authentication after the fact.[61–63] SPF (Sender Policy Framework) lets domains publish DNS records listing which servers are authorized to send mail for that domain. A receiving server can check whether the sending server is on the list. If mail claims to be from `example.com` but arrives from a server not on example.com's authorized list, the receiver can treat it with suspicion.

DKIM (DomainKeys Identified Mail) adds a cryptographic signature to messages—like a wax seal proving the letter wasn't tampered with in transit. The sender's domain signs each message with a private key. Receivers can verify the signature against a public key published in DNS. The signature covers the message headers and body, so any tampering invalidates it. DMARC (Domain-based Message Authentication, Reporting, and Conformance) ties SPF and DKIM together and tells receiving servers what to do when checks fail: accept, quarantine, or reject. It also provides a reporting mechanism so domain owners can see who is sending mail purportedly from their domain. Together, these technologies can prevent some spoofing. If a company publishes strict DMARC policies and all its partners do too, attackers can't easily forge that company's email address.

But deployment remains incomplete: many domains have no DMARC policy, and many others have weak policies that only monitor rather than reject. Legacy systems don't support the checks, and legitimate email services sometimes fail SPF or DKIM due to forwarding or configuration errors. The authentication overlay is fragile, optional, and inconsistently applied.

Even when DMARC works, attackers adapt. They register lookalike domains—`examp1e.com` instead of `example.com`—that pass all authentication checks because the attacker legitimately controls them, or they compromise real accounts and send malicious messages from addresses that will pass every filter. They exploit the gap between what authentication proves (that a message came from a particular domain) and what users assume (that a message is trustworthy). The arms race continues with no clear winner.

More than forty years after Thuerk's spam, more than twenty years after phishing became mainstream, email still lacks universal sender authentication. The patches help, but they're patches. The protocol underneath still accepts senders at their word. Modern email systems run machine learning models on every incoming message, analyzing writing style, link destinations, attachment types, sender reputation, and dozens of other signals. These systems catch most attacks. But "most" isn't "all," and every success teaches attackers what to avoid next time.

Email's vulnerability follows a pattern we've seen before and will see again. SMTP was designed for a small, collegial community where senders were honest because dishonesty had social costs. The designers saw no need to build technical verification into a system where social verification worked. Then the community changed: the ARPANET became the internet, a few hundred researchers became a few billion users, and the social costs of dishonesty vanished. Someone impersonating a CEO to request a wire transfer doesn't worry about their

reputation among MIT professors. But the protocol remained—SMTP in 2024 is structurally similar to SMTP in 1982. The commands are the same, the design is the same, and the MAIL FROM field still accepts whatever the sender provides.

The @ symbol that Ray Tomlinson chose in 1971 changed how the world communicates.

He picked it in a few seconds. No committee approved it. The character was sitting on the keyboard, almost never used, meaning exactly the right thing: "at." And the system he built around it connected people across distances and time zones in ways that telephone and letter could not. A physicist at Stanford could ask a question at midnight and wake to answers from three continents. Email made the global research community feel like a hallway.

The same openness that made any message possible made any forgery possible. Email connects and deceives through the same mechanism, the same fourteen commands, the same unverified MAIL FROM field. The spam folders that intercept billions of fraudulent messages every day are compensating for a design choice made before anyone imagined that strangers would ever use this system. The choice wasn't wrong for 1971. It just never changed. The mail still arrives—most of it real, some of it dangerous, none of it verified at the source.

Thuerk proved that the social costs of dishonesty vanish when the money is good enough. Forty years later, more than 350 billion emails cross the internet every day—as of 2025—and no one can be certain which ones to believe. The ARPANET was a professional network. The internet is not. The protocol doesn't know the difference.

==========

PART II

THE INTERNET

==========

CHAPTER 7

THE PHONEBOOK

Host name to address mappings were maintained
by the Network Information Center (NIC)
in a single file (HOSTS.TXT) which was FTPed by all hosts.
The total network bandwidth consumed
in distributing a new version by this scheme
is proportional to the square
of the number of hosts in the network.

— RFC 1034, Paul Mockapetris, 1987

7.1. THE FILE THAT KEPT GROWING

In 1982, Elizabeth “Jake” Feinler had a problem that would not stop growing.

From her office at the Stanford Research Institute, Feinler ran the Network Information Center.[64] Her team maintained documentation, answered questions, and kept track of every computer on the network. Adding a new machine meant calling Jake’s office. Someone wrote it down.

The file was called HOSTS.TXT. It listed every computer on the ARPANET: name, address, operating system, who to contact if something went wrong. In 1973, when Feinler took over the NIC, the file held a few hundred entries. By 1982 it held thousands. By year's end, it would grow by another thousand.

In RFC 608, M.D. Kudlick described the process: "Jake Feinler at the NIC designed and maintains a source file... We shall write a program to generate the ASCII file."[65] Before anyone could download it, Feinler maintained it by hand.

The file spread by FTP. Every site downloaded the entire file from the NIC. Bandwidth consumed grew with the square of the number of hosts. Every new computer made the distribution problem worse, twice over.

The phone rang. Another sysadmin, another request. Could we add LOGIC? Sorry, there was already a LOGIC at MIT. How about LOGIC2? The administrator sighed.

Name collisions multiplied. The early ARPANET had a flat namespace. When the network had dozens of hosts, uniqueness was easy. With hundreds, every good name was taken.

Feinler pushed for a hierarchical system that would delegate naming authority.[64] Let MIT manage its own names. Let Stanford manage its own. The NIC would manage the top level.

The idea made sense, but the implementation didn't exist.

Forty miles south, Paul Mockapetris was working on exactly this problem.[66]

Postel had asked him to look at the naming crisis. Several proposals were circulating, but none had achieved consensus—the working

group kept meeting, the discussions kept looping, and nothing moved forward. Mockapetris decided to try something different.

His insight was architectural. The problem wasn't distribution; it was authority. A centralized file meant a single point of control. Every change required the NIC's approval. Every download had to carry the entire dataset.

Mockapetris proposed splitting both namespace and database. Names would be hierarchical, divided by dots. MIT's machines would live under `mit.edu`, Stanford's under `stanford.edu`. Each organization would maintain its own piece of the hierarchy, answering queries for names it owned and referring queries for names it didn't.

The root would be small—a handful of servers knowing the top-level domains. The top-level domains would know which organizations existed. Only the leaf servers would know actual addresses.

Delegation. Each level relied on the level below it. No one needed the complete list. When you asked "What is the address of `prep.ai.mit.edu`?" the query would cascade: root to .edu to mit.edu to ai.mit.edu to the answer. The room reached rough consensus. No one asked what happens if a nameserver lies.

Mockapetris wrote RFC 882 and RFC 883 in November 1983, defining the concepts and implementation of what he called the Domain Name System.[67,68] The documents were long—nearly a hundred pages between them—but the core idea was simple. DNS was a distributed, hierarchical, delegated database for name resolution. It replaced a single file with a constellation of servers, each responsible for its own piece of the namespace.

Postel endorsed the proposal immediately. DNS was elegant, buildable, and solved the scaling problem without a central authority.

The first DNS servers came online in 1984. The .com, .edu, .gov, .mil, and .org top-level domains were established in January 1985. By 1987, Mockapetris had refined the specification into RFC 1034 and RFC 1035—the documents that define DNS to this day.[69,70]

Feinler watched the transition. Her NIC continued to operate, but its role changed. Instead of maintaining every host, it maintained the root. The phone still rang, but the questions were different: not "add this host" but "assign this domain."

The last HOSTS.TXT distribution happened in 1989. The internet was growing too fast for any single file, and DNS had proven it could keep up.

The system rested on a simple premise: the phonebook entries were accurate. No verification, no cryptographic signature, no proof the answer came from the right source.

For twenty years, that faith held.

7.2. THE CHAIN OF PHONE BOOKS

7.2.1. HOW DNS WORKS

Every domain name is a question—and DNS answers it.

When you type `www.google.com` into a browser, your computer doesn't know what that means. Computers use IP addresses—numbers like 198.51.100.47—not names. DNS performs this translation billions of times per day, each one on faith.

Think of DNS as a chain of phone books. You want to find John Smith in Los Angeles. You can't look up "John Smith" directly; there are too many of them. Instead, you find the California phone book. That tells

you which page covers Los Angeles. The Los Angeles section tells you which John Smith lives at which address. Each lookup refers you to the next level of detail.

DNS works similarly—your computer starts at the root and asks, "Who handles .com?"

The root doesn't know Google's address—that's not its job—but it knows who might: "Try the .com servers."

Your resolver asks the .com servers the same question: "Who handles google.com?" Same pattern, same answer: "Not me. Try Google's nameservers."

Finally, Google's nameserver has the answer: 198.51.100.47. Three hops, fifty milliseconds, and you're connected.

Each server passes the question down the chain. Each server *believes* the referral it receives. No one verifies anything. If any server in that chain lies, your computer believes it.

This is delegation—no one server knows everything, and no one checks whether the answers are true.

7.2.2. THE RESOLUTION PROCESS

Your computer doesn't do this work itself. It asks a resolver—a DNS server provided by your ISP or configured manually. The resolver follows the chain of referrals on your behalf, from root to authoritative server.

You make one query and wait. The resolver might make half a dozen queries to find the answer, each one accepting the referral before it. The complexity is hidden—and so are the places where trust can be violated.

7.2.3. CACHING

If every query went to the root, the root servers would collapse. There are only thirteen root server addresses, mapping to hundreds of physical machines through anycast.[71] Trillions of queries per day, thirteen addresses—impossible without caching. So DNS caches everything.

When your resolver learns that ".com" is handled by certain servers, it remembers. The next query for `google.com` skips the root entirely.

Each answer comes with a Time to Live (TTL)—how long it can be cached. A typical TTL might be 3600 seconds. For that hour, your resolver returns the cached answer without asking anyone.

Caching makes DNS fast and vulnerable. If someone can inject a false answer into the cache, that lie gets returned to every user for the duration of the TTL.

7.2.4. THE MESSAGE FORMAT

Every DNS message has a header, a question section, an answer section, and referral sections.

One field deserves attention: the transaction ID. Every query carries a 16-bit random number. The response must carry the same number. Think of it like a claim ticket at a coat check. You hand over your coat, you get a number. When someone returns with a coat, they need your number.

The coat check doesn't verify ID. Anyone with the right number gets the coat.

The transaction ID was designed for multiplexing, not security. It became the only thing standing between your DNS query and an attacker's forged response.

Sixteen bits. 65,536 possible values.

That's the lock on the front door.

7.2.5. THE UNVERIFIED ANSWER

DNS believes its answers. When a resolver receives a response, it accepts the content at face value—the response could come from anywhere, and the resolver has no way to tell the difference.

> RFC 1035 --- November 1987 --- Paul Mockapetris
>
> "The domain system is intentionally extensible. Researchers are continuously proposing, implementing and experimenting with new data types, query types, classes, functions, etc."

Mockapetris designed DNS to be extensible—security would come later. The base protocol assumed good faith from its participants.

For two decades, the lack of authentication was theoretical.

7.3. THE RACE CONDITION

Dan Kaminsky was twenty-nine when he found the hole in DNS.[72] It was early 2008, and he was Director of Penetration Testing at IOActive in Seattle.[73] He had a reputation for looking at old problems with fresh eyes.

Cache poisoning wasn't new. The technique was simple: race the legitimate response. Send a forged answer with the correct transaction ID before the real answer arrives. If you win, the resolver caches your lie.

Probability stood in the way. Sixteen-bit IDs gave one chance in 65,536. The resolver would only accept an answer while waiting—

perhaps a few hundred milliseconds. If you lost, you couldn't try again until the cached answer expired. With TTLs in hours, you might get a few attempts per day.

Kaminsky discovered how to try thousands of times per second.

Don't attack the name you care about—attack a name that doesn't exist.

Query for `aaaa.google.com`. There's no cached answer, so the resolver must look it up, and while it waits, you flood it with forged responses including additional records: "The nameserver for google.com is evil.com."

The forged responses carry random transaction IDs, and most are wrong—but you can send thousands while the resolver waits. If you lose, query for `aaab.google.com`. Another nonexistent name, another chance.

Each attempt takes milliseconds. Each failure costs nothing. Given seconds, anyone could poison any domain in any resolver's cache.

The attack didn't exploit a bug—it exploited the mathematics of a sixteen-bit identifier. DNS worked exactly as designed—but the design itself was flawed.

Kaminsky realized what he had. This wasn't a software bug. This was the internet's naming infrastructure. Every resolver, every domain was vulnerable. If attackers learned the technique, they could redirect banking sites to phishing pages, software updates to malware.

He picked up the phone.

The first call went to Paul Vixie, president of the Internet Systems Consortium and maintainer of BIND—the most widely used DNS implementation.[74] Vixie had worked on DNS since the 1980s. He understood the severity immediately.

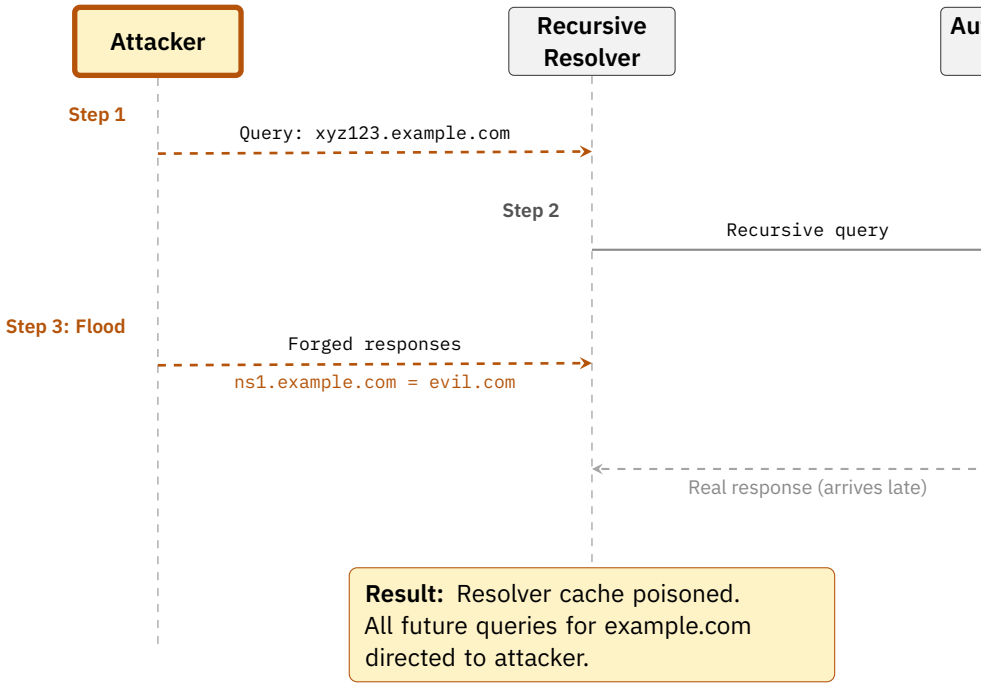


Figure 7.1: The Kaminsky DNS cache poisoning attack. The attacker queries for random nonexistent subdomains (Step 1), triggering recursive lookups (Step 2), then floods the resolver with forged responses faster than the legitimate response can arrive (Step 3). If a forged response has the correct transaction ID, the cache is poisoned.

"Usually, if somebody wants to report a problem, you expect that it's going to take a fair amount of time for them to explain it," Vixie later recalled. "In this case, it took 20 seconds for him to explain the problem, and another 20 seconds for him to answer my objections. After that, I said, 'Dan, I am speaking to you over an unsecure cell phone. Please do not ever say to anyone what you just said to me over an unsecure cell phone again.' "

The two of them began contacting other vendors—Microsoft, Cisco, Nominum, anyone who made DNS software.

On March 31, sixteen researchers gathered secretly at Microsoft's campus.[74] They needed a coordinated patch across all major DNS implementations before anyone figured out the attack.

They had five months. Kaminsky had agreed to wait until Black Hat in August. But five months is a long time to keep a secret.

On July 8, 2008, the patches came out.[75] Every major DNS vendor released updates the same day. The patches added source port randomization—attackers now had to guess two random numbers, not one. Thirty-two bits of entropy instead of sixteen.

Kaminsky held a press conference with a simple message: patch now, and I'll explain why at Black Hat.

Some researchers began probing the patches. On July 21, Halvar Flake guessed the attack correctly and posted online.[76] Matasano Security accidentally confirmed it on their blog—the post was removed within five minutes, but it had been mirrored. The secret was out.

By the time Kaminsky took the stage on August 6, organizers had moved him to the largest venue. It was standing room only.

He walked through the attack: query random names, flood with forged responses, redirect the domain. "I stumbled across something that lets you rewrite DNS for any domain. The entire internet."[77]

The patches held. Source port randomization made the attack harder by a factor of 65,000. Attackers would need minutes or hours instead of seconds.

But the underlying vulnerability remained. Randomizing ports was mitigation, not a fix. The only real solution was cryptographic authentication.

DNSSEC had been standardized in 1999.[78,79] It added digital signatures to DNS responses, but deployment stalled—the signatures added

complexity and broke some things, so a decade after standardization, almost no one used it.

Kaminsky spent years advocating for DNSSEC. He became one of the most prominent voices in internet security, pushing for fixes to protocols designed without adversaries in mind.

On July 15, 2010, the root zone was signed with DNSSEC.[80] But the chain of trust was incomplete. Most top-level domains weren't signed. Most resolvers didn't validate.

DNSSEC adoption has grown slowly, and most queries still travel without validation.[81] The rest accept whoever answers first.

Dan Kaminsky died on April 23, 2021, at forty-two.[72] At twenty-nine, he had helped save the internet's phonebook. The security community mourned him as a researcher, an advocate, and a friend.

The Kaminsky attack remains the most significant DNS vulnerability ever discovered, the moment when the internet confronted the fragility of its naming infrastructure.

DNS was designed to translate names to numbers, and it does that brilliantly. It scales to billions of queries, survives failures, and distributes authority across the internet.

It also accepts every answer it receives. Mockapetris built a protocol for researchers who knew each other. The phonebook worked because everyone in it was honest.

The patches accumulate, but the vulnerability persists.

In October 2025, three new cache poisoning vulnerabilities appeared in BIND 9. CVE-2025-40780 exploited the same weakness Kaminsky had disclosed seventeen years earlier.[82] The same attack, the same design flaw—seventeen years of patches, and the phonebook can still be poisoned.

DNS resolves trillions of queries every day. The phonebook Mockapetris designed for a few thousand researchers still translates names to numbers for five billion users, across every language, every device, every network on the planet. Mockapetris published RFC 882 and RFC 883 in November 1983. The core design hasn't changed.

Kaminsky had spent the years after his disclosure advocating for DNSSEC, for cryptographic validation, for the kind of authentication that Mockapetris never had reason to include. He believed the phonebook could be secured. As of 2024, a third of queries are validated. Two-thirds are not. The root zone is signed, and every year the validated fraction grows by a few percentage points. Kaminsky would have kept pushing.

The phonebook still works. It still trusts whoever answers first. The faith that held for twenty years before Kaminsky found the flaw is still, mostly, honored—because most nameservers are run by people who have no reason to lie, because the community of operators is large but not entirely anonymous, because the alternative to a working phonebook is a broken internet. The system endures on borrowed trust, the same borrowed trust it has always run on.

DNS tells the internet where things are. But knowing where isn't enough. Distributed systems also need to know when—when a message was sent, when a certificate expires, when a transaction happened. In 1979, a researcher noticed that networked computers couldn't agree on what time it was.

CHAPTER 8

THE CLOCKWORK

The intent of the service for which this protocol is designed
is to connect a few primary reference clocks,
synchronized by wire or radio to national standards,
to centrally accessible resources such as gateways.

— RFC 958, David L. Mills, September 1985

8.1. THE CLOCKS THAT DISAGREED

David Mills had a problem nobody else seemed to notice.

It was 1979, and Mills was building systems that synchronized satellite transmissions with ground stations.[83] The work required precision timing. He knew how hard it was to get two clocks to agree.

Then he paid attention to the ARPANET. A network of computers spread across a continent, each with its own clock drifting at its own rate. A log at MIT might say an event happened at 10:42:17. A log at Stanford: 10:43:02. Which came first? Nobody could tell.

For most purposes, this didn't matter. But Mills could see where the internet was heading. Distributed systems need to know order.

Databases need to resolve conflicts. Authentication systems need to timestamp tickets that expire in minutes. Without synchronized clocks, these systems couldn't work.

A time protocol already existed—RFC 868 returned a 32-bit timestamp—but it wasn't enough.[84]

Network delay defeated simple approaches. A timestamp that left the server at noon might arrive fifty milliseconds later—or five hundred. If you set your clock to whatever timestamp the server reported, you'd be off by half the round-trip time. On the ARPANET, that could mean seconds of error.

Mills wanted milliseconds.

His insight: exchange four timestamps instead of one—enough to separate clock offset from network delay.

RFC 958, published in September 1985, defined the Network Time Protocol.[85]

The design was hierarchical. At **stratum 0** sat the reference sources: atomic clocks at national laboratories, GPS satellites, radio stations transmitting signals from NIST. These were ground truth, and servers connected to them spread that truth outward, layer by layer, across the network.

An atomic clock counts cesium atom vibrations—9,192,631,770 oscillations per second, so stable that this rate defines the second itself. The best atomic clocks lose less than one second over millions of years.

Mills moved to the University of Delaware in 1986, and NTP moved with him.[86] He refined the protocol through three RFCs, culminating in RFC 1305 in 1992—over a hundred pages of phase-locked loops, oscillator drift, filtering algorithms.[87]

But the work wasn't abstract. One of the first NSFNET routers was a "Fuzzball"—a DEC PDP-11 running software Mills wrote—humming under the desk in his den. He managed the network from home. The

global timekeeping infrastructure began as a personal project, tended by one man.

He wrote code too. The reference implementation, `ntpd`, turned NTP from paper into software, and Mills maintained it himself for three decades. The internet's timekeeping depended on one professor in Delaware.

Colleagues called him "Father Time." He had given the internet a shared clock, and the clock carried his assumptions. Time sources tell the truth. NTP servers provide accurate time. Clients accept what servers give them.

In 1985, this made sense. The network connected researchers who had no reason to lie about the time. There was no profit in it, and time wasn't yet a weapon.

So Mills designed for that world. The protocol lacked authentication. Clients couldn't verify that a server was legitimate, and responses could be spoofed. The security model, like every protocol in this book, was the community itself. It would take twenty years to discover what happened when the community grew beyond recognition.

8.2. FOUR TIMESTAMPS

8.2.1. HOW NTP WORKS

Every computer has a clock. Every clock is wrong.

A computer's clock is driven by a quartz oscillator—and these drift. Temperature changes and manufacturing variations mean no two crystals vibrate at exactly the same rate, and the drift accumulates.

NTP disciplines these clocks: continuously comparing them against reliable references and nudging them back into alignment. Milliseconds over the internet, microseconds on local networks.

Choosing which time source to believe is NTP's central problem.

NTP organizes sources into layers, called strata. At the top, stratum 0: GPS satellites, atomic clocks at national laboratories, radio stations broadcasting from NIST. These are the reference sources, the ultimate authorities.

Stratum 1 servers connect directly to stratum 0—a GPS antenna on the roof, a cable to a server. These machines touch the source—thousands of them worldwide. Your computer believes them.

Stratum 2 servers synchronize to stratum 1, stratum 3 to stratum 2. Each layer adds a hop—a place where the chain could be broken or bent. A stratum 2 server trusts that its stratum 1 source is honest—nothing proves otherwise.

Stratum 16 means "unsynchronized"—a clock that has lost contact with any reference.[88] But there's no stratum for "liar." A server advertising stratum 1 could be an attacker in a closet, and NTP can't tell the difference.

Stratum	Source
0	Reference clock (GPS, atomic, radio)
1	Directly connected to stratum 0
2	Synchronized to stratum 1
3	Synchronized to stratum 2
...	
15	Maximum usable stratum
16	Unsynchronized

Table 8.1: The NTP stratum hierarchy

The elegance: a single stratum 1 server serves dozens of stratum 2 servers, each serving thousands of clients. The load spreads naturally, and the system scales without central coordination.

8.2.2. THE TIME EXCHANGE

The next problem: I ask you what time it is. You say "12:00." How long did your answer take to reach me? If I set my clock to noon when your message arrives, I'm wrong by the transit time.

NTP's solution: exchange four timestamps instead of one. The client notes when it sends the request (T_1). The server notes when the request arrives (T_2) and when the response departs (T_3). The client notes when the response arrives (T_4).

Four timestamps—two are yours, two are mine. Together they reveal the round-trip delay and the clock offset.

Notice who controls what. The client measures T_1 and T_4—its own clock. The server reports T_2 and T_3. The client accepts those numbers on faith. A lying server can report any T_2 and T_3 it wants.

The **round-trip delay** measures transit time:

$$\text{delay} = (T_4 - T_1) - (T_3 - T_2)$$

Total elapsed time on the client's clock, minus the time the server claims it spent processing. The **clock offset**—how far the client's clock differs from the server's—comes from averaging the forward and return legs:

$$\text{offset} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

The math requires symmetric routing and honest servers—but internet routing is asymmetric. Servers don't need to be honest.

NTP clients typically synchronize to multiple servers. Redundancy protects against failure—if one server goes offline, others remain. They sometimes disagree.

Mills called unreliable sources **falsetickers**: machines that report bad time, whether through malfunction, misconfiguration, or attack. A broken clock doesn't know it's broken. It reports its time with the same confidence as a working one. NTP needs to identify which sources to rely on.

The solution: **intersection**. Each server's time estimate comes with an uncertainty bound—a range where the true time probably lies. NTP looks for the interval where a majority of sources overlap. Servers outside the consensus are rejected as falsetickers. The remaining servers—**truechimers**—contribute to the final estimate.

The algorithm is self-organizing: bad servers get ignored while good servers rise to the top.

The requirement: the majority must be honest. If an attacker controls more than half the servers a client uses, the attacker's time becomes the consensus. The falseticker becomes the truechimer.

Once NTP knows the offset, it adjusts the local clock. But you can't just jump.

If your clock is 500 milliseconds slow, the obvious fix is to leap forward 500 milliseconds. This is called **stepping**—but stepping breaks things. Programs measuring elapsed time see negative intervals. Log files have timestamps that repeat or skip. Cryptographic protocols reject valid messages because the time changed mid-handshake.

NTP prefers to **slew**: speed up or slow down the clock gradually. If you're 500 milliseconds slow, run slightly fast until you catch up. Time keeps moving forward, just at a different rate, and no one notices.

For small offsets—under 128 milliseconds—NTP always slews. For larger gaps, the protocol steps the clock forward or backward in a single jump. If your clock is off by an hour, you can't wait days to drift back.

The danger: slewing is gradual, but the direction comes from the server. An attacker who controls your time source controls which way your clock drifts. Over hours or days, they can walk your clock

anywhere they want—forward to make certificates expire, backward to make revoked certificates valid again.

Mills spent decades refining the algorithms that manage this discipline. RFC 1305 devotes fifty pages to feedback loops and oscillator modeling. The goal wasn't just setting the correct time once—it was training the local clock to keep better time on its own.

8.2.3. THE HONEST CLOCK

NTP servers included a feature for administrators: the `monlist` command. Send it to any NTP server, and it returns a list of the last 600 clients—addresses, packet counts, timestamps. Everything an administrator might need.

Everything an attacker might exploit.

The request was small—40 bytes—but the response could be enormous: 100 packets, tens of thousands of bytes. An amplification factor of 556 to 1.

For every byte you send, the server sends 556 back—and it sends them wherever you claim to be from. The server doesn't verify your address; it simply believes the source field in your packet.

NTP accepts its responses at face value.

> RFC 958 --- September 1985 --- David L. Mills
>
> "There is no provision for peer discovery, acquisition, or authentication in NTP. Data integrity is provided by the IP and UDP checksums."

UDP checksums detect transmission errors—cosmic rays flipping bits, electrical noise corrupting packets. They don't detect intentional forgery. Anyone who can inject packets can claim to be a stratum 1 server synchronized to an atomic clock.

Mills designed for a research network where lying about the time would have been pointless vandalism. Adding authentication in 1985 would have required cryptographic infrastructure that didn't exist. The trade-off was reasonable then—but the network changed while the protocol stayed the same.

8.3. 400 GIGABITS PER SECOND

On March 18, 2013, Spamhaus went to war.

Spamhaus was a nonprofit organization that maintained blacklists of spam sources. Internet service providers used these lists to filter email. If Spamhaus said an IP address was a spam source, millions of mail servers would refuse to accept mail from it. The organization had made powerful enemies.

One of them was CyberBunker, a Dutch hosting provider with a libertarian philosophy: they would host anything except child pornography and terrorism. Spamhaus added CyberBunker's IP ranges to their blacklist. CyberBunker responded with what would become the largest distributed denial-of-service attack in history.

The attack reached 300 gigabits per second, enough traffic to disrupt major internet exchanges in London and elsewhere.[89] The method was DNS amplification: small queries to DNS servers that generated large responses, all directed at Spamhaus through IP address spoofing. The technique was already known. The scale was unprecedented.

When the attack finally subsided, security researchers took stock of what they had witnessed. DNS wasn't the only amplifiable protocol. Other UDP-based services had similar properties. The researchers started looking at NTP.

Brian Krebs learned about the new attack method the hard way.

On the morning of February 11, 2014, the security journalist's website disappeared. For roughly ten minutes, 200 gigabits per second of traffic flooded his servers.[90] The attacker wasn't trying to keep him offline permanently. This was a demonstration—proof of capability for admission to Darkode, an invitation-only cybercrime forum. The attacker, who went by "Rasbora" online, later boasted in private channels: "If all else fails and you just want it offline, PM me." He was reportedly a teenager.[90]

The same day, Cloudflare reported an even larger attack against one of their European customers.[91] The traffic peaked at 400 gigabits per second—enough to cause network congestion across parts of Europe. The attack used 4,529 NTP servers spread across 1,298 networks on six continents. Each server dutifully responded to what it thought were legitimate monitoring queries. The queries had been spoofed.

The French hosting giant OVH was hit simultaneously. Founder Octave Klaba tweeted from the trenches: "We see today lot of new DDoS attacks from Internet to our network. Type: NTP AMP Size: >350Gbps."[92] The irony was bitter: OVH's own NTP servers were among those being exploited to attack others.

The weapon was a command called `monlist`. It asked an NTP server to list its recent clients—a useful diagnostic tool. A 40-byte request could generate up to 22,000 bytes in response, a 556-to-1 amplification factor. The 400-gigabit attack used 4,529 NTP servers, each contributing roughly 87 megabits—all triggered by spoofed requests that may have originated from a single machine. No need to compromise thousands of computers into a botnet. Just a list of NTP servers and the ability to spoof source addresses.

The list was easy to find—scanning the internet for open NTP servers was trivial. Most ran default configurations, and most default configurations included `monlist`.

The vulnerability wasn't new. Security researchers had warned about `monlist` for years, and it was assigned CVE-2013-5211 after being exploited in the wild by December 2013.[93] US-CERT issued an alert in January 2014.[94] The warnings came too late. Millions of NTP servers ran publicly accessible `monlist` endpoints, configured by administrators who had set them up years earlier and forgotten about them.

The fix was straightforward: disable `monlist` or restrict it to trusted addresses. Version 4.2.7 of the reference NTP implementation removed the feature by default. Administrators needed to update their servers and restart them. Simple in principle, slow in practice. No central authority could force upgrades. The internet's timekeeping infrastructure was a collection of independent operators, each running their own servers, many unaware that their helpful monitoring feature had become a weapon.

Months passed, and surveys found that the number of NTP servers with `monlist` enabled dropped from hundreds of thousands to tens of thousands. The attacks became less powerful, but they didn't stop. As long as any amplifying servers remained, attackers would use them.

The deeper problem was architectural. NTP ran over UDP, a protocol that didn't verify sender addresses. Any packet with a spoofed source address matching the victim's IP would be accepted. The same amplification pattern would appear in SNMP (Chapter 12) and other UDP protocols—any service that answered queries from strangers could be turned into a weapon.

IP spoofing had been a known issue since the 1980s. BCP 38, published in 2000, recommended that networks filter outbound traffic to prevent spoofed addresses from leaving.[51] More than a decade later, the recommendation remained optional. Networks that allowed spoofing became amplification platforms.

Mills, watching from Delaware, had seen this coming. He had worked on NTP security extensions for years, and the extensions existed—but they weren't deployed. The same administrators who forgot about `monlist` weren't about to configure cryptographic authentication.

Time manipulation offered another attack vector. Anyone who could control a victim's NTP traffic—through a rogue server, through interception, through BGP hijacking (Chapter 13)—could control their clock.

The consequences rippled through time-dependent security. TLS certificates have expiration dates; shift the victim's clock, and valid certificates become invalid or expired certificates become valid. Kerberos tickets have five-minute windows; push the clock outside the window, and authentication fails. Log files become unreliable when timestamps can't be verified. Forensic analysis falls apart when the timeline is corrupted.

In 2015, researchers from Boston University demonstrated that NTP's lack of authentication made it possible to shift clients' clocks by years.[95] The attack exploited a "kiss-o'-death" packet that NTP used to tell clients to back off, repurposed to convince clients that their upstream servers had failed. The client would then accept time from an attacker-controlled source. A certificate revoked in 2012 would suddenly appear valid to a victim who thought it was still 2012.

Network Time Security, RFC 8915, was published in September 2020.[96] The protocol adds cryptographic authentication to NTP, letting clients verify that responses come from legitimate servers and haven't been tampered with. It uses the same TLS certificates that protect web traffic. It addresses both the amplification problem (authenticated servers can reject spoofed requests) and the time manipulation problem (clients can verify server identity).

Adoption has been slow—NTS requires servers to obtain certificates and clients to support the new protocol. Most operating systems still default to unauthenticated NTP. Most public time servers don't offer NTS. The infrastructure that should have been secured decades ago remains largely open.

The helpful features of 1985—open access, detailed monitoring, willing responses to anyone who asked—became the vulnerabilities of 2014. Trust, once granted, is hard to revoke.

The amplification attacks weaponized the internet's timekeepers. Helpful servers became unwitting cannons, flooding victims with traffic five hundred times the size of what attackers sent. The `monlist` command that David Mills created for debugging became the largest DDoS amplifier the internet had ever seen. The vulnerable servers were patched and the attacks subsided.

David Mills died on January 17, 2024, at age 85.[86] The obituaries called him "the father of internet time." He had spent forty-five years building a system that kept billions of devices synchronized, that made distributed computing possible, that underpinned every timestamp online.

His clocks still run. Every device that checks the time against a network source—every phone, every server, every certificate that expires on a scheduled date—passes through the stratum hierarchy Mills designed. The reference clocks at NIST and USNO still tick. The GPS satellites still broadcast. The Fuzzball in his den is long gone, but the protocol it ran has replicated itself across millions of servers on six continents. The borrowed seconds keep extending.

On December 17, 2025, hurricane-force winds tore through Boulder, Colorado, and knocked out power to NIST's timekeeping servers for

four days.[97,98] UTC(NIST) drifted by 4.8 microseconds—not from attack, not from software failure, but from wind and wire. Even stratum 0 depends on electricity. The system recovered, as Mills had designed it to: NIST operates multiple locations, and the stratum hierarchy routes around failures. The backup did what backups are for. The clocks corrected. The time held. Mills had built a system that could survive even its own foundations going dark—and it did.

Somewhere right now, a stratum 2 server is asking a stratum 1 server what time it is. The answer comes back unsigned, unverified, accepted on faith. The clock adjusts. The seconds tick forward. Mills is gone, but the handshake he wrote still runs, billions of times a day, keeping the whole machine in time.

CHAPTER 9

THE NEWSGROUP

Given the strains of running a nationwide
distributed network on machines with limited capacity
and via long distance telephone,
we utterly relied on the ease of writing shell scripts
to experiment with protocol variants.

— Steve Bellovin, on the origins of Usenet[99]

9.1. THE POOR MAN'S ARPANET

Tom Truscott wanted what he couldn't have.

It was 1979, and Truscott was a graduate student at Duke, watching the ARPANET from the outside. The network connected elite institutions—MIT, Stanford, Berkeley. Duke wasn't on the list.

But Duke had Unix machines, and Unix machines could exchange data.

What if they built their own network? Computers that dialed each other overnight, when long-distance rates were cheap, and exchanged messages in batch. A poor man's ARPANET.

The technology already existed. UUCP—Unix-to-Unix Copy—could transfer files over phone lines, and universities were already using it to exchange email and share software. It was slow and unreliable, but it was cheap.

Truscott and Ellis imagined a system for distributing messages to multiple recipients. A user at Duke posts an article. When Duke's machines dial UNC overnight, the article copies over. UNC copies to its neighbors, neighbors to theirs, until the article has propagated through the network.

They called it Usenet.

Steve Bellovin volunteered to write the software.[100] He wrote the first version in a few evenings—three pages of shell scripts that were crude but functional.

The first Usenet connection went live in late 1979, carrying about ten articles per day: Unix bugs, system administration tips, department gossip.

Truscott and Ellis presented at a USENIX conference in January 1980.[101] Their pitch: a network anyone could join. You just needed a Unix machine, a modem, and a phone line.

Sites joined, and the network grew from three nodes to thirty to three hundred. By 1983, Usenet connected over five hundred sites, with traffic growing from ten articles per day to thousands.

Usenet was radically decentralized—no master server, no central authority. Every site stored its own copy and decided which neighbors to exchange with.

Articles propagated through flood-fill: when a site received a new article, it forwarded copies to all neighbors, who forwarded to theirs, until a single post had generated thousands of copies.

The Path header tracked where an article had been, with each site prepending its name as the article passed through. If you received an article with your site already in the path, you'd already seen it.

Articles were organized into newsgroups. `net.unix`. `net.jokes`. `net.lang.c`. By 1987, administrators reorganized it into the "Big Seven": comp, misc, news, rec, sci, soc, talk.[102]

Usenet worked because everyone followed the rules.

The rules were called netiquette: don't post advertisements, don't spam, don't forge headers, stay on topic. The enforcement mechanism was community pressure—violators would be flamed, their administrators contacted. Five hundred sites, a few thousand users, shared norms.

RFC 977, published in February 1986, standardized the Network News Transfer Protocol—NNTP—for exchanging articles over TCP/IP rather than UUCP.[103]

NNTP was simple: commands like GROUP to select a newsgroup, ARTICLE to retrieve a message, POST to submit a new one. The standard accepted that information in article headers was accurate. The From line showed who sent the article. The Date line showed when. The Path showed the route it had traveled.

9.2. FLOOD-FILL

9.2.1. HOW USENET WORKS

How do you run a bulletin board with no central server?

You make copies—lots of copies. When you post a Usenet article, your server forwards it to its neighbors, who forward it to theirs, until the article exists on thousands of servers worldwide. One post becomes many copies.

This is fundamentally different from email, where one copy travels from sender to recipient. With Usenet, thousands of copies flood outward until every server has everything.

The implications cut both ways. No one can censor Usenet—there's no central authority to pressure, no single server to shut down. But no one can moderate it either. Once something propagates, it's everywhere. And since every server accepts every other server's articles, a single lie propagates as efficiently as the truth.

9.2.2. THE ARTICLE FORMAT

A Usenet article looks almost identical to an email message. Headers at the top, blank line, body below:

```
Usenet Article Header

From: alice@example.com (Alice Smith) Newsgroups:
comp.lang.c Subject: Question about memory
allocation Date: 15 Apr 1994 09:30:00 GMT Message-
ID: <1994Apr15.093000.12345@example.com> Path:
news.server.com!upstream!example.com!alice
```

Look at that `From` header. It says Alice Smith sent this article.

No one verifies it.

The receiving server accepts whatever the `From` line claims. If you want to post as the professor who runs the newsgroup, you change one line of text. The protocol has no mechanism to check.

The `Path` header traces the article's journey, with each server prepending its name as the article passes through:

```
Path: final.server!middle.server!origin.server!user
```

Servers use this to avoid loops—if your site is already in the path, you've seen it before. But the path is just text that anyone can type. The chain of provenance holds together only because everyone agrees to be honest about it.

9.2.3. FLOOD-FILL PROPAGATION

How does one post reach thousands of servers? The same way rumors spread through a village: everyone tells everyone.

When your server receives a new article, it checks the Message-ID. If the article is new, the server stores it, adds its name to the Path, and forwards copies to every neighbor, each of whom does the same. The article floods outward until every server has a copy.

This is beautiful for sharing information. It's terrifying for controlling it.

A single post generates traffic proportional to servers times connections—in the early days, with hundreds of servers and kilobyte messages, the load was manageable. But later, with millions of users posting binary attachments and pirated movies, the math became brutal.

Worse: removing content is nearly impossible. Once an article propagates, it exists on thousands of independent servers. No central authority can delete it.

The internet remembers—even when you wish it wouldn't.

9.2.4. CONTROL MESSAGES

Usenet needed to manage itself, but with no central authority, how do you create newsgroups, delete obsolete ones, or remove spam? The answer was control messages: special articles that instruct servers to take action.

These control messages could reshape the network from anywhere.

`newgroup` created a newsgroup—someone wanted to discuss Python programming? Send a `newgroup` message, and every server obediently creates `comp.lang.python`. No central authority approves it, and no one verifies that the sender has the right.

`rmgroup` deleted groups from every server—useful for cleaning up spam-ridden discussions, but also useful for silencing conversations you didn't like if no one could prove you weren't authorized.

Most dangerous was `cancel`. It removed a specific article from every server worldwide. The stated purpose: let users delete embarrassing posts, let administrators remove spam. The actual effect: anyone who knew an article's Message-ID could forge a cancel and erase it everywhere.

How did servers know which control messages to obey? They checked the `From` header—the same header that no one verified, the same header you could fill with any text you wanted.

If your server accepted articles purportedly from a certain administrator, it obeyed. If someone forged that administrator's name, your server obeyed anyway. The mechanism designed to let administrators remove spam could also let attackers censor dissent.

9.2.5. THE GOOD-FAITH FORUM

Usenet added a new dimension to the familiar pattern: faith in the community's ability to police itself.

Every header was accepted at face value, and control messages—commands to delete articles network-wide—carried no cryptographic authentication. The enforcement mechanism was entirely social: netiquette, peer pressure, and the expectation that administrators would discipline their own users.

The engineers knew this was fragile.

In February 1993, Theodore Ts'o proposed adding real authentication to `NNTP`, with `Kerberos` support and per-user permissions[104]. The proposal went nowhere. "Nobody was going to go out and implement something that nobody else had," Rich Salz observed.[105] The coordination problem was brutal. Everyone waited for everyone else.

Ts'o's own words are striking: "No authentication system is such that you can trust it 100%... trust is always relative".[106] The engineers understood the nuance.

The protocol they shipped had no authentication at all.

9.3. THE DEATH OF THE COMMUNITY

On April 12, 1994, Laurence Canter and Martha Siegel broke Usenet.[107]

Canter and Siegel were immigration lawyers in Phoenix who wanted to advertise their services for the green card lottery. Television was expensive, but Usenet was free.

They posted "Green Card Lottery- Final One?" to every newsgroup they could find. Over five thousand copies flooded across the network in ninety minutes.

The reaction was immediate and furious.

Usenet had seen spam before, but previous incidents had been mistakes or pranks. Canter and Siegel were commercial, deliberate, and unrepentant.

Their ISP, Internet Direct, crashed under the flood of complaint emails—angry users sent thousands of messages demanding action, and some sent email bombs. Internet Direct terminated the lawyers' account, but Canter and Siegel simply signed up with another provider.

They received death threats and harassing phone calls as the internet's fury descended.

But the lawyers didn't care. They later claimed the advertisement had generated between one hundred thousand and two hundred thousand dollars in business.[107] The outrage was free publicity. They wrote a book teaching others to do the same.[108]

Years later, Canter remained unapologetic. "Somebody would have done it if we hadn't." He was probably right.

The Green Card spam proved that spam worked—commercial incentive outweighed social pressure. Netiquette depended on caring about reputation, but Canter and Siegel cared about money.

The flood-fill design that made Usenet work also made it vulnerable: a single post propagated to thousands of servers, and the network amplified abuse as efficiently as it amplified discussion.

The community fought back. Arnt Gulbrandsen wrote a "cancelbot"—software that detected spam and automatically removed it.[109] Others built similar tools.

But the cancel mechanism worked both ways. If vigilantes could cancel spam, attackers could cancel legitimate posts. The "cancel wars" erupted. The system designed for community moderation became a battlefield.

Then came Eternal September.

Every September, new students discovered Usenet without knowing the norms. The regulars would sigh, correct them, and by October the newbies had learned.

In September 1993, America Online began offering Usenet to its millions of subscribers—and unlike university students, they kept coming. There was no October. The influx never stopped.

Dave Fischer coined the term in January 1994: "September 1993 will go down in net.history as the September that never ended".[110]

The flood swamped the community's ability to socialize new users—the regulars were outnumbered, and the cultural norms dissolved.

By the late 1990s, Usenet's character had changed. The binary newsgroups—alt.binaries.*, where users shared software, images, and later, pirated movies and music—came to dominate traffic. Discussion continued in the traditional groups, but the volume of binaries dwarfed it. A 2010 study of European ISP traffic found that more than ninety-nine percent of Usenet bytes were yEnc-encoded binary files.[111]

The major internet service providers began dropping Usenet. AOL stopped carrying it in 2005.[112] Verizon, Time Warner, and Sprint followed in 2008, responding to pressure from New York Attorney General Andrew Cuomo about illegal content in the binary groups.[113]

The network that had pioneered internet discussion was becoming a liability.

In May 2010, Duke University—where Tom Truscott and Jim Ellis had conceived Usenet thirty-one years earlier—shut down its Usenet servers.[114] The birthplace of the network no longer saw a reason to participate.

Usenet still exists—commercial providers sell access, and archives like Google Groups preserve decades of discussion. The protocol still works exactly as RFC 977 specified, but the community that made it meaningful is gone.

The network that believed in its users learned the hard way: that belief could not survive at scale. The social pressure that enforced good behavior required a community small enough to know each other. When millions joined, the community dissolved into a crowd. The crowd had no norms, no reputation system, nothing to punish defectors.

Tom Truscott and Jim Ellis had built a poor man's ARPANET. They got what they asked for: a network anyone could join. They hadn't an-

ticipated what would happen when everyone did. Spam proved that economic incentives could overwhelm social norms. Eternal September proved that community doesn't scale. The binary groups proved that people would use any free distribution network for whatever they wanted to distribute. The question was never whether Usenet's model would fail—only when.

Usenet tried to solve a problem of abundance: too many strangers, too many posts, norms dissolving in a flood. At MIT, researchers faced a different problem entirely. Their campus network was small and controlled. The users were students and faculty. The trouble wasn't anonymous strangers—it was that every password crossed the wire in plain text, readable by anyone with a packet sniffer and an Ethernet tap.

CHAPTER 10

THE TICKET

Network services may need to be accessed by many users,
and these users must be authenticated.
In a world without network security,
users often type in passwords
that are transmitted in the clear over the network.

— *RFC 4120, The Kerberos Network Authentication Service, July 2005*

10.1. THE DIGITAL BADGE

The password went across the wire in plain text, visible to anyone watching.

It was 1983. MIT was building Project Athena[115]—workstations across campus connected to shared services. Students would log into any terminal, access their files, check their email. The network would make computing democratic, but the security wasn't straightforward.

When a student typed a password, it traveled across the campus network to a server. Ethernet was fast and completely unencrypted.

A student with a packet sniffer could sit in a dorm room and collect passwords all day.

Steve Miller and Clifford Neuman stared at the problem.[116] They had been tasked with authentication for Project Athena, and the obvious solution—encrypt passwords before transmission—raised more questions than it answered. Encrypt with what key? If a user needed ten services, would that mean ten passwords, ten key exchanges? The complexity multiplied with every new resource.

Miller and Neuman's insight was that a user should authenticate once and then carry a credential—a digital proof of identity—to present to each service.

Miller and Neuman designed a digital badge system. A user authenticates once to a trusted server, which issues a ticket—an encrypted token that can be presented to any service on the network. No password crosses the wire after initial authentication.

They named it after the three-headed dog guarding the gates of Hades. `Kerberos`.

Project Athena launched in 1983, funded by IBM and DEC.[115] Students could sit at any terminal, log in, and see their files, while the security challenges remained invisible—exactly as Miller and Neuman intended.

`Kerberos` version 4 shipped in 1988,[116] and the protocol spread beyond MIT. By the early 1990s, `Kerberos` had become the standard for enterprise authentication.[117] When Microsoft integrated it into Windows 2000, replacing `NTLM`, every Windows network in the world suddenly ran `Kerberos`.

The RFC came later—Kerberos version 5 was published as RFC 1510 in 1993[118] and updated as RFC 4120 in 2005.[119] The campus project was now an internet standard.

> RFC 4120 --- July 2005 --- Kerberos Network Authentication
>
> The Kerberos protocol provides a mechanism for mutual authentication between a client and a server before a network connection is established between them.

The architecture made its premise explicit. At the center of every Kerberos deployment sat a Key Distribution Center—the KDC. This server held the secret keys for every user and service in the network. It was the registration desk that issued badges, the single point where faith was concentrated instead of distributed.

Control the KDC, control everything.

10.2. TICKETS AND TRUST

10.2.1. HOW KERBEROS WORKS

Think of an amusement park.

You arrive and show ID, and the attendant wraps a wristband around your arm. For the rest of the day, you show the wristband to access rides instead of showing ID again. The wristband says: this person already proved who they are.

Kerberos follows this model. Your password is the ID, the Key Distribution Center issues wristbands, and the services—file servers, email, printers—check your credential.

What happens if someone steals your wristband?

How do you prove your identity without sending your password over the wire? With a shared secret—a key that both parties know. Either side can encrypt messages only the other can read. In `Kerberos`, the KDC stores a secret key for every user and every service, derived from passwords but never transmitted.

The registration desk has a copy of every badge.

`Kerberos` uses these keys in three exchanges: prove your identity, obtain permission, then access the resource.

The Authentication Service Exchange (AS Exchange). Call it the "prove who you are" step. You sit at your workstation and type your username and password. Your computer sends a message to the Key Distribution Center: "I am alice@EXAMPLE.COM and I want to prove it."

The KDC looks up Alice in its database and finds her secret key. Remember, this key was derived from her password. The KDC creates a Ticket Granting Ticket (TGT)—think of it as a master pass that says "Alice has proven her identity." The KDC encrypts this TGT with its own secret key. Then it wraps the TGT and a session key together in a package encrypted with Alice's secret key. This package travels to Alice's workstation.

When it arrives, Alice's workstation derives the same secret key from her password. It uses this key to decrypt the package. If her password was correct, decryption succeeds. She now has a TGT that she cannot read (it is encrypted with the KDC's key) and a session key she can use to talk to the KDC.

The password never crossed the network. Alice proved she knew it by successfully decrypting a message. If she typed the wrong password, the decryption would have failed. The KDC never sees her password either—it only stores the derived key.

That TGT lives in memory. Alice's workstation keeps it cached, ready to present whenever she needs access to a service. For the next eight to ten hours, anyone who can read that memory has Alice's master pass.

The Ticket Granting Service Exchange (TGS Exchange). Alice wants to access a file server. Her workstation contacts the KDC again, presenting the TGT: "Here is my proof. I want a ticket for fileserver.example.com."

The KDC decrypts the TGT with its own key. Inside, it finds Alice's identity and verifies that the TGT has not expired. Then it creates a service ticket for the file server, encrypted with the file server's secret key, and sends it back.

Notice what the KDC does not verify: whether Alice is still at her workstation.

The Application Protocol Exchange (AP Exchange). Alice's workstation contacts the file server and presents the service ticket—but the ticket alone is not enough. Her workstation must also send an *authenticator*: a timestamp encrypted with the session key that only Alice received. The file server decrypts the ticket with its own secret key, extracts the session key, and uses it to decrypt the authenticator. If the timestamp is recent and the decryption succeeds, Alice has proven she possesses the session key.

This sounds secure. But here is the problem: both the ticket and the session key sit in Alice's workstation memory. Anyone who can read that memory has everything they need to impersonate her. The ticket alone cannot authenticate—but together with the session key, it might as well be a bearer token.

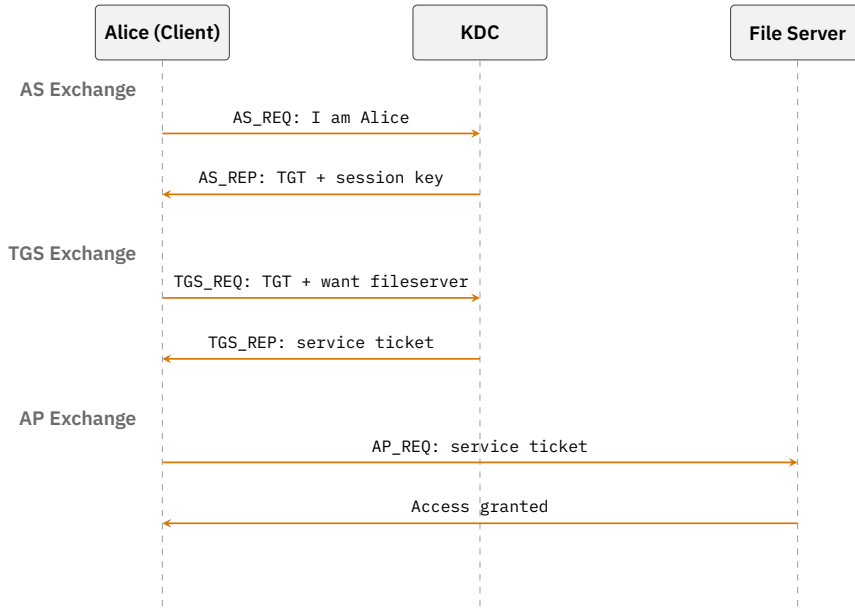


Figure 10.1: The Kerberos ticket flow. Alice authenticates once to the KDC, receives a Ticket-Granting Ticket, then uses it to obtain service tickets for individual resources — never sending her password over the network.

10.2.2. WHY TIME MATTERS

Every Kerberos ticket contains timestamps—the TGT records when it was issued and when it expires, and service tickets have their own validity windows. The default tolerance for clock skew between machines is five minutes.

This makes Kerberos depend on NTP, the Network Time Protocol we discussed in Chapter 8. If clocks drift too far apart, authentication fails. A ticket that looks valid to the server might appear expired to the client, or vice versa.

The timestamps serve a purpose: they prevent replay attacks, since an attacker who captures a ticket cannot use it forever. After the ticket

expires, it's worthless, and the five-minute window keeps the replay threat small.

The time dependency creates its own vulnerabilities. Anyone who can manipulate a machine's clock can make valid tickets appear expired, or make expired tickets appear valid. The timestamps assume clocks are honest. In a hostile environment, that assumption fails.

10.2.3. THE CENTRAL AUTHORITY

Kerberos inverts the pattern we've seen throughout this book. Instead of extending good faith to every node equally, it concentrates all reliance on a single point: the Key Distribution Center.

> RFC 4120 --- The Trust Model
>
> The Kerberos model is based on a trusted third-party authentication service. The Kerberos server is assumed to be secure.

The KDC knows every secret—it holds the keys for every user, every service, every machine in the network. The three-headed dog guards everything, and whoever controls the dog controls every gate.

The designers knew this. They expected the KDC would be the most protected machine on the network, isolated and hardened, accessible only to vetted administrators. In the world of Project Athena—a campus network where the administrators knew each other by name—that expectation held, but the world grew larger.

10.3. THE GOLDEN TICKET

Benjamin Delpy was a security researcher in France who wanted to understand how Windows stored credentials. In 2011, he released a tool called mimikatz[120] to demonstrate that Windows stored passwords in a way that could be extracted. The tool worked too well.

Mimikatz could extract `Kerberos` tickets from Windows memory. When you log into a Windows workstation, the operating system caches your TGT in RAM, and mimikatz could read that RAM and copy the ticket out. Once copied, the ticket could be used from any machine—a technique called pass-the-ticket. Anyone who compromised one workstation could steal the TGT of any user who had logged in.

Pass-the-ticket was devastating—it bypassed password security entirely. Change your password every hour—it wouldn't matter. Your ticket was already stolen.

But pass-the-ticket was just the beginning.

In a Windows domain, the Key Distribution Center ran on domain controllers, and the KDC's secret key was stored in an account called krbtgt. An attacker who gained administrative access to a domain controller could extract the krbtgt password hash. With that hash, they could forge TGTs for any user, with any permissions, for any duration. The golden ticket attack.[121]

Benjamin Delpy coined the term: "With Golden Ticket, you can be anyone!" The reference was Willy Wonka—a ticket that granted access to everything, forever.

A golden ticket was not stolen but created. The adversary generated a TGT asserting domain administrator privileges, set an expiration twenty years in the future, and began accessing resources. The KDC

never issued this ticket, but because it was encrypted with the correct key, every service treated it as legitimate.

The golden ticket persists even after discovery. Change all passwords, remove compromised accounts, patch the vulnerability—none of it matters, because they have the krbtgt hash and can create new administrator tickets at will.

Remediation requires resetting the krbtgt password twice, with at least ten hours between resets, and both disruptions ripple across authentication for the entire domain. Surgery while the patient is awake.

10.3.1. THE NATION-STATE PLAYBOOK

Golden ticket attacks were not just the tools of criminals—nation-state attackers adopted them as a standard technique for persistent access.

APT29, a designation for a group U.S. intelligence agencies attributed to Russian state actors, also known as Cozy Bear, used `Kerberos` attacks extensively.[121] After compromising a network through phishing or software vulnerabilities, they would move laterally—spreading from one compromised machine to others—until they reached domain controllers. Once they had the krbtgt hash, they had persistent access that could survive remediation attempts.

The SolarWinds attack of 2020 demonstrated the technique at scale.[122,123] Attackers compromised the software supply chain of SolarWinds, a company that made network management tools used by thousands of organizations, including government agencies. The malicious update gave attackers initial access. From there, they moved to domain controllers, extracted `Kerberos` keys, and forged tickets that let them access email and documents for months.

The attackers went further. They discovered that the same principles that allowed forging Kerberos tickets could be applied to SAML tokens—the authentication mechanism connecting on-premises domains to cloud services.[124,125] Microsoft's Azure Active Directory connected on-premises domains to cloud resources. The trust relationship meant that an attacker with the SAML signing certificate could forge tokens for Azure, bypassing even multi-factor authentication.[126]

The boundaries that organizations assumed existed—between their data center and the cloud, between their network and Microsoft's—dissolved. A golden ticket on-premises became a golden SAML token in the cloud.

10.3.2. THE DESIGN CHOICE

Kerberos was designed for a world of colleagues. Miller and Neuman expected that the network contained authorized users and legitimate services, and they built for a KDC that would be protected, administrative access that would be restricted, keys held only by people who deserved them.

The premise held for Project Athena, for university networks in the 1980s, and more or less for enterprise networks in the 1990s. It did not hold for networks connected to the internet, penetrated by nation-states, running software with unknown flaws. The three-headed dog guarded the gates, but someone found the side entrance.

The ticket-based model that made single sign-on possible also made persistent compromise possible, and the architecture that simplified authentication also concentrated the target. Own one machine—the domain controller—and you own the entire network.

Microsoft added defenses: Protected Users groups that limit ticket caching, Credential Guard to isolate tickets in hardware enclaves, and

advanced auditing to flag suspicious ticket usage. These were layers on top of a design built for a different threat model.

The KDC remains central, the krbtgt key remains powerful, and the architecture endures.

Steve Miller left MIT years ago. Clifford Neuman became a professor at USC, where he continued working on distributed systems and security.[127] The protocol they designed to solve a campus authentication problem became the authentication layer for most corporate networks in the world.

They solved the problem they set out to solve: passwords no longer cross the wire in plain text, users authenticate once and access services seamlessly, and the digital wristband works.

They could not anticipate a world where attackers would steal the wristbands, counterfeit them, target the registration desk itself. `Kerberos` was built for a community. The community became a battleground.

Golden tickets became standard tools in nation-state attacks, and mimikatz made credential theft trivial. The krbtgt key that protected every Windows domain became the prize that attackers sought above all others. Ransomware operators needed to own an entire domain before encrypting it, and the golden ticket gave them exactly that.

And yet the wristband still works. Billions of Windows logins each day pass through `Kerberos`, and the ticket system that Miller and Neuman designed in an MIT lab still authenticates the world's corporate networks. Somewhere right now, an employee is sitting down at a workstation, typing a password once, and opening every door on the network. The three-headed dog still guards the gate.

`Kerberos` concentrated trust in a single server. Other protocols tried the opposite: no center at all, no authority, just an open channel and the hope that everyone on it would behave.

CHAPTER 11

THE CHAT ROOM

IRC has been designed over a number of years
for use with text based conferencing.
This document describes the current IRC protocol.

— RFC 1459, Jarkko Oikarinen and Darren Reed, May 1993

11.1. ARCTIC TWILIGHT

In August 1988, Jarkko Oikarinen was a summer trainee at the University of Oulu, in northern Finland.[128] The sun barely set at that latitude in summer. He had been coding through the long twilight. His assignment: improve the chat feature on the bulletin board system. Instead, he built something entirely new.

The BBS chat was limited—it ran on a single machine, and Oikarinen wanted more. He imagined users anywhere joining the same conversation, multiple conversations at once, each in its own channel.

He called it Internet Relay Chat.

The first IRC server ran on tolsun.oulu.fi, where Oikarinen tested it from other machines in the department. A few students joined, but what mattered was that the room could now be anywhere.

His nickname was WiZ—on IRC, your handle mattered more than your legal name.

The idea wasn't entirely new. Other real-time chat systems existed. Bitnet Relay Chat let users on Bitnet—a separate academic network—talk in groups. The Unix `talk` program allowed two people to type at each other, their words appearing letter by letter on split screens. A Finnish programmer named Jyrki Kuoppala had written `rmsg`, a multi-user messaging system.

Oikarinen's contribution was synthesis. He took the multi-user model from Bitnet Relay, the internet's TCP/IP infrastructure, and the Unix tradition of small, composable programs, and created something that could scale. Servers connected to servers. Users connected to any server and could talk to anyone on the network. Messages relayed through the mesh. The whole thing worked like a distributed party line.

By 1989, IRC had escaped Finland. Servers appeared in Sweden, then at MIT in the United States.[129] The network grew through word of mouth and academic connections. A researcher would discover IRC, set up a server at their institution, link it to the existing network, and suddenly everyone at that university could join the global conversation.

The growth was explosive. When Oikarinen finally standardized the protocol in RFC 1459, he noted the strain: "[The network] is striving to cope with growth. Over the past 2 years, the average number of users connected to the main IRC network has grown by a factor of 10."[130]

The early culture was anarchic in the best sense. Channels formed around topics: programming, science fiction, philosophy, music. Anyone could create a channel just by joining one that didn't exist. Anyone

could claim a nickname just by using it. The only rules were the ones each community enforced for itself.

This worked because the early users shared norms. They were researchers, students, hobbyists who had found their way onto the academic internet. They wanted to talk, not to fight. When conflicts arose, they worked them out or ignored each other. Social pressure kept people in line.

The formula was familiar by now. Shared norms worked until the community grew too large to know itself.

By August 1990, the network had grown large enough to need governance. A server called "eris" had a wildcard line allowing anyone to connect servers, causing chaos. Operator Greg Lindahl led a split, and the remaining operators called themselves the Eris-Free Network—EFnet.[131] It became the first major IRC network, the mother from which all others would descend.

January 1991 brought a moment that hinted at IRC's potential.[132] American forces began bombing Baghdad in the first Gulf War. Users in Israel typed reports as SCUD missiles fell on Tel Aviv. Updates appeared in IRC channels before they reached television newsrooms. For a few nights, the network became the fastest news source on the planet.

The same openness that let Israelis report missile strikes let anyone pretend to be anyone. The architecture that carried news would carry commands to criminal botnets. The protocol that enabled global conversation enabled global attacks.

Oikarinen and his collaborators published the protocol specification as RFC 1459 in May 1993.[130] The document described what IRC had already become: a text conferencing system with channels, nicknames, and a mesh of servers. It codified the trust assumptions that had governed the network since August 1988.

The premises were simple: users would identify themselves honestly, server operators would link only reputable servers, and the system existed for conversation, not for crime.

11.2. CHANNELS AND COMMANDS

11.2.1. HOW IRC WORKS

Think of IRC as a party spread across many rooms. Each room has a name—`#linux`, `#python`, `#randomchat`. Anyone can walk into any room. Anyone can start talking. Some rooms have bouncers who can kick out troublemakers. But the bouncers are just other guests with special privileges.

The party happens across a network of servers. When you connect to IRC, you connect to one server in the mesh. That server is linked to others, which are linked to still others. A message you send travels from your server to every other server that needs to see it, then down to the clients in the relevant channel. The servers relay your messages. That's where the name comes from.

11.2.2. CONNECTING AND IDENTIFYING

When you connect to an IRC server, you introduce yourself—but no one checks whether you're telling the truth.

```
NICK alice
USER alice alice localhost :Alice Smith
```

The NICK command claims a name. The USER command provides a username, hostname, and "real name" (though no one verifies this). Once these complete, the server welcomes you with a burst of information: the server name, the network rules, the message of the day.

Your full identity on IRC is a triple: nickname, username, and hostname. You might appear as:

```
alice!alice@isp.example.com
```

This triple identifies you in channel bans, private messages, and server logs.

What's missing: proof.

You chose your nickname. You provided your username. The hostname comes from your connection, but that can be masked or spoofed. No password, no certificate, no proof that alice is who she claims to be. The server takes your word for it.

11.2.3. CHANNELS AND MESSAGES

Channels are where conversation happens, and they start with `#`—`#linux`, `#python`, `#secretcommands`. That last one could exist, because anyone can create a channel just by joining it.

```
JOIN #chat
PRIVMSG #chat :Hello everyone!
PART #chat :Goodbye
```

JOIN enters a channel. If the channel doesn't exist, you create it—no approval needed. PRIVMSG sends a message to a channel or a person. PART leaves.

When you send a message to `#chat`, your server forwards it to every other server with users in that channel. Those servers deliver it to their local users. Milliseconds. Global reach.

The protocol doesn't care what the message contains. "Hello everyone!" and "!ddos yahoo.com 80 60" travel the same way. The infrastructure that carries conversation will carry commands.

11.2.4. OPERATORS AND POWER

Channels have operators, marked with an @ before their nickname. Operators can kick users, ban them, set channel modes, and appoint other operators.

```
MODE #chat +o bob
KICK #chat alice :Spam
MODE #chat +b *!*@bad.net
```

Give bob operator status; remove alice from the channel for spam; ban everyone connecting from bad.net. Three commands, and you have total control.

Who decides who becomes an operator? The first person to join an empty channel. Or anyone an existing operator trusts enough to promote. No vetting. No appeals process. Just power, granted by whoever holds power.

Operator status is temporary—it lasts only while you're in the channel. If all operators leave, the channel becomes ungoverned. The next person to join could claim control. This led to elaborate schemes: bots that held operator status around the clock, scripts that rejoined

immediately after disconnection, wars over who controlled popular channels.

11.2.5. THE NETSPLIT PROBLEM

IRC networks are trees of servers. When a link between servers fails, the tree splits in two. Each side continues operating independently, unaware of the other. Users on one side can't see users on the other.

When the link comes back, chaos follows.

Both sides may have users with the same nickname. Both may have different operators in the same channel. The servers must reconcile these conflicts. The typical solution: kill one of the colliding nicknames, force both users to reconnect.

This became a weapon. Attackers would deliberately break server links, wait for key users to disappear from one side, then claim their nicknames or seize their channels. "Riding the split" became an art form. The feature that let the network survive outages became a vector for takeovers.

11.2.6. THE OPEN DOOR

What did IRC verify? Nothing. What did it expect? Good faith from everyone.

> RFC 1459 --- May 1993 --- Internet Relay Chat
>
> IRC itself is a teleconferencing system, which (through the use of the client-server model) is well-suited to running on many machines in a distributed fashion.

The RFC describes a teleconferencing system. No built-in registration verified identities. Server operators could see all traffic and

impersonate any user. The specification expected a community of enthusiasts who shared norms about acceptable behavior.

In 1993, using a chat system to control malware would have seemed absurd. But the features that made IRC good for conversation—real-time messaging, channels for coordination, global reach—would make it perfect for crime.

11.3. THIRTY MILLION MACHINES

The screen showed IP addresses scrolling upward—each one a compromised computer, all waiting for instructions.

The early 2000s—a scene that played out thousands of times across those years. A botnet operator sat at a keyboard, connected to an IRC channel no legitimate user would find. The channel name: random characters. The server: a hacked machine or cheap VPS paid for with stolen credit cards.

In the channel, thousands of bots waited—each one a program running on someone else's computer. A home PC infected through malicious email, a corporate workstation compromised by a worm, a server with an unpatched vulnerability. They had names like `[XP-HOME]39842`.

The operator typed:

```
!ddos yahoo.com 80 60
```

The bots began flooding Yahoo's servers. Sixty seconds of coordinated attack from thousands of machines.

This was what IRC became: infrastructure for crime.

11.3.1. THE BOTNET EVOLUTION

IRC bots started innocently—they managed channels, kept logs, kicked troublemakers, enforced rules. Useful, even beloved.

Then someone realized that a bot could run on a compromised machine, connect to a secret channel, wait for commands, and bring thousands of friends.

GTBot appeared in 2000.[133] Written in mIRC script, crude but effective. An infected machine would connect to a specified IRC server, join a hidden channel, and wait. The operator issued commands through normal IRC messages.

SDBot followed in 2002, written in C++ and more sophisticated.[134] The source code leaked, and soon thousands of variants existed—the barrier to entry had dropped to zero.

Agobot raised the bar in 2003. It offered over five hundred configuration options, could evade antivirus software, hide as a rootkit, and use encrypted channels. Its creator, Axel Gembe, was arrested in Germany in May 2004[135], but by then the model had spread beyond any single author's control.

11.3.2. THE MAFIABOY ATTACKS

In February 2000, a fifteen-year-old Canadian who called himself Mafiaboy demonstrated what IRC-coordinated attacks could do.[136]

On February 7, Yahoo went offline. The site had been the web's most visited destination, and it simply stopped responding—about an hour of downtime, millions of users locked out.

The next day: Amazon. eBay. CNN. Each site buckled under floods of traffic. The attacks used distributed denial of service—DDoS—coordinated through IRC channels. Compromised machines

received commands to flood specific targets. No single attacker could have generated enough traffic. Thousands of machines, working together, overwhelmed some of the largest sites in the world.

Michael Calce—Mafiaboy—had bragged about his exploits in IRC channels. The FBI monitored those channels, building a case. By mid-February, investigators had traced the communications. Canadian police eventually arrested him at his Montreal home.

The estimated damage exceeded a billion dollars. Calce was a teenager with no financial motive. He did it because he could.

11.3.3. BREDOLAB: THE CRIMINAL EMPIRE

If Mafiaboy was an amateur, the operators of BredoLab were professionals.

By 2009, BredoLab controlled an estimated thirty million infected computers. The malware spread through spam emails, through compromised websites, through malicious advertisements. Each infected machine connected back to IRC infrastructure, awaiting commands.

The operators didn't use the botnet themselves—they rented it out. Want to send spam? Pay per million messages. Want to launch a DDoS attack? Pay per hour. Want to steal banking credentials? The botnet could be configured for that too. Some estimates suggest the operator made up to $139,000 per month renting access.[137]

BredoLab was a service business.

In late October 2010, the Dutch National High Tech Crime Unit coordinated an international takedown.[138] They seized 143 command-and-control servers. The botnet operator, Georgy Avanesov, was arrested at Yerevan's Zvartnots Airport in Armenia, and in May 2012 was sentenced to four years in prison.[139] It was the largest botnet takedown to date.

But by then, the model had proven itself. IRC-based botnets continued for years afterward. Some still operate today.

In May 2021, the Freenode IRC network—home to thousands of open-source projects—collapsed after a hostile takeover by its new owner.[140] Staff resigned en masse. Users migrated to a new network called Libera.Chat. Within weeks, Freenode was dead.

The incident demonstrated IRC's final vulnerability: lacking any central authority, governance structure, or legal framework, a network could be seized or destroyed at the whims of whoever controlled the domain name. A system that welcomed everyone couldn't protect itself from anyone.

Millions of machines once answered to IRC commands, as botnets turned conversation into attack infrastructure. The network that connected Finnish students became the nervous system of criminal enterprises. The major chat platforms moved on, built their own protocols, added encryption and moderation and corporate governance. The world IRC was built for is gone.

And yet—on Libera.Chat, at two in the morning UTC, someone in `#linux` is asking how to compile a kernel module. A bot logs the question. Three strangers answer. The channel has been active, under one name or another, for thirty years. No one is paid to be there. No company runs the server. The conversation just keeps going, carried by a protocol that still trusts whoever shows up.

IRC connected people. But people were not the only things on the network that needed attention. The machines themselves—the routers and switches and servers carrying all this traffic—were growing too numerous to manage by hand. Someone needed to build a way to ask them how they were doing.

CHAPTER 12

THE COMMUNITY STRING

Security issues are not discussed in this memo.

— RFC 1157, May 1990

12.1. THE SIMPLE QUESTION

Jeffrey Case had a straightforward problem. He wanted to know what was happening inside his network.

It was 1987. Case was a researcher at the University of Tennessee, managing a growing collection of routers and gateways. Each device had its own console, its own commands. To check network status, he walked from machine to machine, logging into each. For ten devices, this was tedious. For a hundred, impossible.

Across the country, network administrators faced the same problem. The internet was growing—thousands of routers, switches, servers, each generating logs, each capable of failing without warning. The era of managing networks by walking around was ending.

That November, Case and three colleagues published RFC 1028, describing what they called the Simple Gateway Monitoring Protocol.

The name was honest: SGMP was simple. A management station could query a gateway for information, and the gateway would respond. The protocol ran over UDP, required minimal resources, and worked with devices that had limited memory and processing power. For routers that could barely spare the cycles to forward packets, simplicity wasn't a luxury—it was essential.[141]

The protocol spread quickly. Within months, vendors were adding SGMP support to their routers. The Internet Engineering Task Force recognized that something more complete was needed—a protocol that could manage not just gateways but any network device. In 1988, the IETF formed a working group to extend SGMP into a full management framework.

The result was SNMP: the Simple Network Management Protocol. RFC 1157, published in May 1990, defined the protocol that would become the standard way to monitor and control network devices. Case was the lead author, joined by Mark Fedor, Martin Schoffstall, and James Davin—the same team that had created SGMP. The "S" in SNMP stood for what they valued most.[142]

The design philosophy was explicit. "Consistent with the goal of minimizing complexity of the management agent itself, it is assumed that the management station performs the majority of the processing." The authors listed four reasons: lower development costs, broader management capability, fewer restrictions on tools, easier understanding for developers. Security appeared nowhere.

A router's job was to forward packets. Every cycle spent on management was a cycle not spent on routing. SNMP would be lightweight, easy to implement, and universally deployable. Administrators needed to monitor their networks now; they could worry about security later.

Authentication, in SNMP's design, came down to a single string of characters. The protocol called it a "community string"—a name cho-

sen because SNMP organized devices into communities that shared management policies. In practice, a community string was a password. Include the right string in your request, and the device would answer. Include the wrong string, and it wouldn't.

The RFC used "public" as an example community string for read-only access and "private" for read-write access. The examples were meant to be illustrative, placeholders that administrators would replace with their own values. Instead, they became defaults.

Device manufacturers shipped their routers and switches with "public" configured for read access and "private" for write access. Administrators, busy with a thousand other problems, left the defaults in place. They meant to change them later. Meant to secure their networks properly once things calmed down. Things never calmed down.

The community string traveled with every SNMP request, transmitted in cleartext. Anyone who could capture network traffic could read it. Anyone who knew the string could query the device—SNMP made no distinction between a legitimate administrator and someone who had guessed the password. If you knew the magic word, you were authorized. The word was usually "public."

RFC 1157 contained a section titled "Security Considerations." It consisted of a single sentence—the one that would become the chapter's epitaph.

It worked. By the mid-1990s, SNMP was everywhere. Every router, every switch, every server, every network-attached printer supported it. Network management systems could query thousands of devices, aggregate their statistics, alert administrators to failures, and even reconfigure equipment remotely. The chaos of managing networks by walking around had been replaced by centralized dashboards showing the status of entire infrastructures.

Fedor had organized all that device knowledge hierarchically—every piece of information on every device had a precise address that could be queried remotely. Routing tables, interface statistics, connection logs, ARP caches, configuration data, and in a few vendors' implementations, passwords.

With the community string, you could read all of it. With the read-write community string, you could change it.

12.2. MANAGERS AND AGENTS

12.2.1. HOW SNMP WORKS

Every network device contains a story. Interface counters track how many packets pass through. Error logs record what goes wrong. Routing tables describe the paths data takes. ARP caches remember which hardware addresses belong to which IP addresses. The device knows things about your network that no documentation captures.

SNMP exists to ask questions and hear answers. The protocol divides the world into managers and agents. A manager is the software that wants information: the monitoring dashboard, the alerting system, the configuration tool. An agent runs on the device being managed: the router, the switch, the server. The manager asks; the agent answers.

Who counts as a manager? How does the agent know a legitimate administrator from an intruder scanning the network?

It doesn't. Anyone who knows the password—a short string of characters—is treated as authorized. The agent cannot tell the difference between your monitoring system and an attacker in another

country. Both look the same: a UDP packet arriving on port 161, carrying a request and a password.

If the password matches, the agent answers. If the agent has something urgent to report—a link going down, a threshold being exceeded—it can send a trap to the manager on UDP port 162. Traps are unsolicited messages that let the network cry for help, but they also announce that something valuable is listening.

The operations are elementary and dangerous.

A **GET** request retrieves a single value. "What is the system uptime?" The agent responds: 86400 seconds. One day since the last reboot. Anyone who asks with the right password learns when this device was last vulnerable to a reboot attack.

A **GETNEXT** request retrieves whatever comes next in the database. Call it again and again, starting from the root, and you walk through every piece of information the agent knows: routing tables, interface statistics, ARP caches, configuration data. Attackers map networks this way, one query at a time, as the device reveals everything.

A **SET** request changes a value. "Disable interface 3." If you have the write password, the agent obeys. No login screen, no audit trail, no confirmation that you should be allowed to do this. Just a UDP packet and a password.

A **TRAP** flows in the opposite direction—the agent notifies the manager that something happened. A link failure. A reboot. An authentication failure, when someone guessed the wrong password. Ironically, the trap reveals that the device exists and is worth attacking.

12.2.2. THE MANAGEMENT INFORMATION BASE

What exactly can an attacker read? Everything the device knows about itself—and everything it knows about your network.

SNMP organizes information into a tree called the Management Information Base. The tree has a root, branches divide into smaller branches, and leaves contain actual data. Each piece of information has an address: a sequence of numbers separated by dots, called an Object Identifier. Think of it as a filing system where every document has a precise location.

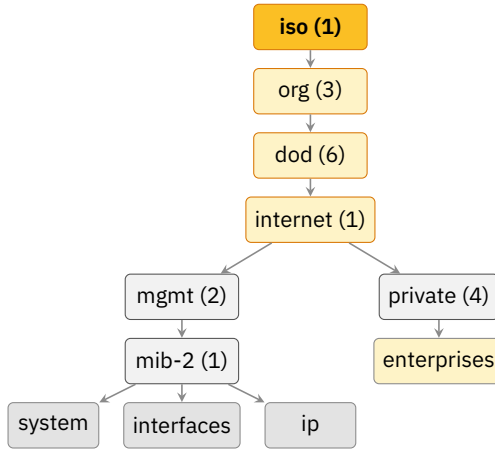


Figure 12.1: SNMP MIB tree structure. Object Identifiers form paths from root to leaf.

The address `1.3.6.1.2.1.1.5.0` points to the device's hostname. The address `1.3.6.1.2.1.2.2.1.10.3` points to how many bytes have arrived on interface 3. Hundreds of addresses, hundreds of pieces of information.

Each branch reveals more about your network.

System identity tells an attacker what they've found. Interface statistics show traffic patterns. IP routing tables expose the network topology. TCP connection data reveals what services are running and who's connected. ARP caches map IP addresses to physical hardware—the skeleton of your local network, laid bare.

Vendors extend the tree with their own branches, and some of those branches contain configuration data—a few even contain passwords.

An SNMP walk—GETNEXT requests, one after another, starting from the root—enumerates everything. One guess at the password, and the device tells you its secrets. For a legitimate administrator, this is visibility. For an attacker, reconnaissance on request.

12.2.3. THE COMMUNITY STRING

What is this password that grants such access?

Authentication in SNMP version 1 has one component: the community string. Every request includes it, and every response requires it to match what the agent expects.

> RFC 1157 --- May 1990 --- SNMP Authentication
>
> "Each SNMP community is named by a string of octets, that is called the community name for said community. An SNMP message originated by an SNMP application entity that in fact belongs to the SNMP community named by the community component of said message is called an authentic SNMP message."

The language is formal, almost bureaucratic. But the mechanism is bare: if the string matches, the request is "authentic." No usernames. No cryptographic signatures. No challenge-response. Just a password, sent in the clear with every packet.

Agents typically recognize two community strings: the read-only string allows GET and GETNEXT operations—reconnaissance, but not control—while the read-write string allows everything, including SET requests that reconfigure the device. An administrator might configure "monitoring" for read access and "configure" for write access.

Or they might leave the defaults: "public" and "private."

Millions did.

SNMP sends these strings as plaintext in UDP packets—no encryption, no hashing—so the password travels in the clear, readable by any device on the path. Capture one SNMP exchange, and you have the key. Use that key, and the device cannot tell you from its legitimate administrator.

12.2.4. THE DEFAULT PASSWORD

SNMP's security model stands out for its self-awareness—and its surrender.

RFC 1157 acknowledged that security mattered. It defined authentication services and noted their importance. Then it provided an escape hatch: "Some SNMP implementations may wish to support only a trivial authentication service that identifies all SNMP messages as authentic SNMP messages."

You could implement real security, or you could skip it—and most implementations skipped it. The trivial authentication service—community strings sent in cleartext—became the standard, and the escape hatch became the main entrance.

The working group knew this was a problem, and they debated cryptographic approaches for years. SNMPv2c, published in 1996, added bulk operations but kept community strings.[143] SNMPv3 finally arrived in 2002 with real authentication and encryption.[144]

By then, the damage was done. Millions of devices were deployed with "public" and "private" as their passwords, and SNMPv3 was complex enough that administrators who had grown up with the defaults found it bewildering. Many didn't bother upgrading.

The pattern differs from earlier chapters: the engineers knew they needed authentication. They discussed it, debated it, eventually standardized it. But the standard they shipped in 1990 had none. Twenty

years later, surveys still found millions of devices responding to "public"—the example value from RFC 1157, never meant for production, now embedded in infrastructure worldwide.

12.3. THE DEVICES ANSWER BACK

On an October morning in 2016, security cameras across the East Coast went dark, DVRs rebooted, and home routers dropped connections. The devices weren't failing—they were attacking.

The target was Dyn, the company providing DNS services to some of the internet's largest websites: Twitter, Netflix, GitHub, Reddit, Amazon. All of their names resolved through Dyn's servers, and on this morning, those servers were drowning.

The traffic came from everywhere. Homes, small businesses, security cameras, printers, thermostats, baby monitors with factory-default configurations. The devices had been conscripted into a botnet called Mirai.[145]

Mirai was not sophisticated—its source code was almost insultingly simple. The botnet spread by scanning for devices running Telnet with default passwords, trying 62 combinations like admin/admin, root/root, and guest/guest. When one worked, it logged in, downloaded itself, and started scanning for more victims.

But Telnet was just the beginning. The internet was full of devices answering queries without any login. SNMP was one of them.

Anyone scanning with "public" could query any device that hadn't changed defaults. The response revealed identity, contact information, sometimes location—and more usefully, network configuration. Anyone who could walk the MIB could map the entire network.

The reconnaissance was free. Shodan, a search engine for internet-connected devices, indexed SNMP responses. A query for devices responding to "public" returned millions of results.[146] Not just old routers forgotten in closets, but active network infrastructure. The example community string had become a skeleton key.

The amplification attacks came next. To understand them, we need to understand three things: UDP, forged addresses, and amplification.

SNMP runs over UDP, the User Datagram Protocol. Unlike TCP, UDP does not establish a connection before sending data—it fires packets and hopes they arrive. This makes UDP fast and lightweight, perfect for SNMP's quick queries.

But UDP has a weakness: it does not verify where packets came from. The source address in a UDP packet is whatever the sender claims it is, and an attacker can write any address they want in that field. This is called IP spoofing.

The attack works like this: an attacker sends an SNMP query to a device but lies about where the request came from. Instead of their own address, they write in the victim's. The device receives the query, prepares a response, and sends it to the victim. The victim never asked for this data. They drown in answers to questions they never asked.

This technique becomes devastating when the response is larger than the request. Attackers call this **amplification**. If you can send a 60-byte request and generate a 6,000-byte response, you have multiplied your attack power by 100. Your bandwidth becomes a hundred times more destructive.

SNMP was exceptionally good at amplification. A GETBULK request—added in SNMPv2 to retrieve multiple values efficiently—could ask for dozens or hundreds of OIDs at once. The request might be 60 bytes; the response, thousands. Security researchers measured amplification factors of 6.3x for basic queries, up to 650x for carefully crafted

GETBULK requests.[147] Every byte sent became hundreds flooding the victim.

The pattern was familiar from NTP (Chapter 8) and DNS (Chapter 7)—any UDP protocol that answered queries from strangers. But SNMP had a particular advantage: the devices running it were everywhere, often forgotten, rarely updated, almost never firewalled. Scans in 2014 found millions of devices responding to SNMP queries on port 161.[146] Many still do.

The attacks were not hypothetical. In February 2014, CloudFlare reported a DDoS attack reaching 400 gigabits per second using NTP amplification—and warned that SNMP could be even worse.[148] By mid-2014, SNMP amplification attacks were appearing in the wild. By 2016, SNMP had become a standard component of multi-vector attacks that set new records for traffic volume. The devices Case had designed to answer administrators' questions were now answering attackers' questions, flooding victims with the responses.

The fix was straightforward: disable SNMP on devices that didn't need it, change community strings on devices that did, firewall port 161 from the public internet. The execution was anything but. The devices with default configurations were precisely the devices whose administrators didn't know they existed. Cameras installed by contractors who had moved on. Routers deployed years ago, never touched since. Printers that had always worked fine and therefore never needed attention.

The deeper problem was architectural. SNMP assumed that knowing the community string meant you belonged. In that world, infrastructure was private, administrators were known, and the password stayed within the organization. The designers hadn't anticipated a world where every device was reachable from the internet. Where

anyone could guess "public" as easily as "password123." Where the management system itself could become a weapon.

RFC 1157's security considerations section—that single, dismissive sentence—had aged poorly. The specification that made network management possible had also made network exploitation scalable. The same ease of deployment that let administrators put SNMP everywhere let attackers exploit it everywhere.

In 2017, US-CERT published an advisory: SNMP remained a significant threat.[149] Disable it where possible. Use SNMPv3 with authentication and encryption where necessary. Block UDP port 161 at network boundaries. The recommendations were the same ones security researchers had been making for years. They continued to go unheeded.

Case went on to found SNMP Research, a company that built network management tools.[150] The standard he helped create became the foundation of an industry. The monitoring dashboards that network administrators rely on today descend from one elemental idea: devices should answer questions about themselves.

The community string persists. Newer devices may ship with randomized defaults or require configuration before enabling SNMP, but the installed base is vast and slow to change. Millions of routers still respond to "public" on port 161, revealing their routing tables to anyone who asks.

RFC 1157 used "public" as an example. Just a placeholder, never meant for production. It shipped in every manual, on every default configuration, an invitation that no one revoked.

In October 2025, security researchers documented Operation ZeroDisco, an attack campaign exploiting SNMP vulnerabilities in Cisco devices.[151] Nearly two million devices were potentially vulnerable. The

attackers needed only the read-only community string. Often, that string was still the example value from 1990, thirty-five years later.

Somewhere right now, a router on the edge of a network that no one remembers installing is listening on port 161. It will answer any question, from anyone, as long as they know the word. The word is "public." It has always been "public."

==========

PART III

THE WEB

==========

CHAPTER 13

THE ROUTING TABLE

The primary function of a BGP speaking system
is to exchange network reachability information
with other BGP systems.

— RFC 1105, Kirk Lougheed and Yakov Rekhter, June 1989

13.1. LUNCH IN AUSTIN

It was lunchtime in Austin, Texas, and two engineers were redesigning the internet on napkins.

By 1989, the infrastructure was in place. DNS translated names. NTP synchronized clocks. SNMP monitored the machines. The academic internet hummed along, thirty thousand hosts strong, managed by engineers who recognized each other at conferences. But commercial networks were knocking at the door: PSINet, UUNET, companies that wanted to sell connectivity, not share it. The research network was about to become a marketplace. It would need a new kind of routing.

Kirk Lougheed had flown in from Cisco's Menlo Park offices. Yakov Rekhter had come from IBM's T.J. Watson Research Center.[152] They

were attending the twelfth IETF meeting, where the people building the internet argued about how it should work. The formal sessions had broken for lunch, but the real work was happening at a restaurant table.

The challenge was simple to describe, brutal to solve. The ARPANET had been one network managed by one organization. By 1989, the internet had become a patchwork of independent networks, each with its own administrators, equipment, and ideas about traffic flow. The NSFNET backbone connected universities. Commercial networks like PSINet and UUNET carried traffic. Regional networks linked smaller institutions. Each was an "autonomous system"—a network that made its own routing decisions.

The existing protocol, EGP (Exterior Gateway Protocol), assumed a simpler topology: the ARPANET at the center, everything else connected like spokes on a wheel. The internet wasn't a wheel anymore. It was becoming a mesh, with networks connecting in complicated patterns. EGP couldn't express those patterns or handle policy: the rules about who could send traffic through whom, who paid whom, who trusted whom.

Lougheed and Rekhter sketched on napkins. They drew circles for autonomous systems, lines for connections between them. They wrote out the information each network would share with its neighbors: which destinations it could reach, and through which path.

The path was the key insight. EGP said "I can reach this network." The new protocol would say "I can reach this network through these autonomous systems." With that information, networks could prefer shorter paths, avoid competitors, and detect loops. If your own network appeared in the path, something had gone wrong.

They called it the Border Gateway Protocol, BGP. The name was straightforward: it ran on border routers, the machines at the edges of each autonomous system, the gateways between networks.

Lougheed worked at Cisco, which meant he had access to routers. Within months, he had a working implementation running on Cisco hardware. Rekhter implemented BGP for the NSFNET backbone. By late 1989, BGP was carrying real traffic.

RFC 1105, published in June 1989, laid out the design.[153] The document ran seventeen pages. It described four message types: OPEN, to establish a connection between BGP speakers; UPDATE, to exchange routing information; NOTIFICATION, to signal errors; and KEEPALIVE, to confirm the connection was still alive. The protocol ran over TCP—the reliable transport protocol from Chapter 4—which meant it inherited TCP's reliability. If a BGP message got lost, TCP would retransmit it. The engineers didn't have to build that machinery themselves.

The design was elegant: two routers would connect over TCP port 179,[153] exchange OPEN messages, agreeing on parameters, then exchange their complete routing tables. After that, they'd send UPDATE messages whenever routes changed. The rest of the time, they'd exchange KEEPALIVEs: heartbeats proving the connection was alive.

But the simplicity concealed a dangerous assumption. When one autonomous system announced "I can reach 192.0.2.0/24," the receiving system believed it—no proof, no certificate. The protocol assumed that network operators would tell the truth about the routes they could carry, because lying would disrupt the network, and everyone wanted the network to work.

BGP became known as "the three-napkin protocol": simple enough to sketch at lunch, powerful enough to route the entire internet. The original napkins are long gone, but the story has been retold so many times it has become legend: three handwritten sheets that defined how the world communicates.[154] The room didn't include nation-states.

The engineers never imagined that the trust they encoded would one day route traffic for adversaries.

The internet of 1989 had about 100,000 hosts. A decade later, it would have 100 million. The community of network operators who met at IETF meetings would become an industry of thousands of companies, including some that would rather route traffic through their surveillance infrastructure than deliver it honestly. The napkin protocol would scale beyond anyone's imagination. The underlying premise would not.

RFC 1105 was just the beginning. `BGP-2` came in 1990[155], `BGP-3` in 1991[156], and `BGP-4` in 1995.[157] Each version added capabilities—support for classless addressing, better aggregation of routes, more sophisticated policy controls. RFC 4271, published in January 2006, codified `BGP-4` as a full Internet Standard.[158] By then, `BGP` was routing traffic for billions of users. The two engineers who'd sketched it on napkins had created something that touched every packet crossing the internet.

The core premise never changed. Your neighbors tell the truth about what they can reach. If you can't rely on your neighbors, you shouldn't be peering with them.

The question nobody asked at that Austin lunch was simple: What happens when the neighborhood changes?

13.2. LONGEST PREFIX WINS

13.2.1. HOW BGP ROUTES THE INTERNET

Picture a world map where every major city manages its own roads. Los Angeles doesn't know Tokyo's roads, and Tokyo doesn't know Los Angeles's. But they both need to move traffic between them.

That's the internet's routing problem. Each city is an autonomous system (AS): a network under unified administrative control, like a transportation district. Los Angeles County manages its roads, and Orange County manages its own—neighbors, but separate. AT&T, Comcast, and Google each manage their own networks the same way.

Every AS gets a number: a name it uses to identify itself. AT&T is AS 7018, Google is AS 15169, and the U.S. Department of Defense operates AS 721. When they announce routes, they attach these numbers identifying who claims to control each piece of the internet—claims, not proof.

Neighboring districts exchange traffic by sharing road maps. "To reach San Diego, take I-5 south." BGP follows the same principle. Routers at network borders—the on-ramps and off-ramps between districts—connect and share their maps.

These border routers are BGP speakers—the name is literal, they speak BGP to each other. When two AS networks agree to share traffic, they establish a peering session, like postal offices exchanging mail. Once the session is established, each side believes what the other announces.

Once connected, the BGP speakers trade routes. A route is a promise: "I can reach these addresses, and here's the path." The path is expressed as a list of AS numbers, like highway directions listing which counties you'll pass through. If Google announces a route to YouTube's addresses, the path might be `15169 36561`. Traffic enters AS 15169

(Google), then reaches AS 36561 (YouTube, a Google subsidiary). Two hops.

These paths grow longer as they spread. AT&T might receive that route and announce it to its neighbors as `7018 15169 36561`. Now the path shows three hops: "Come through AT&T, then Google, then YouTube." Each network adds itself to the front, like stamping a passport.

> RFC 1105, Section 2 --- June 1989
>
> This network reachability information includes information on the autonomous systems (AS's) that traffic must transit to reach these networks. This information is sufficient to construct a graph of AS connectivity from which routing loops may be pruned and policy decisions at an AS level may be enforced.

The AS path serves two purposes. First, it prevents loops. If a BGP speaker receives an announcement containing its own AS number, it knows something is wrong—the route has looped back—and discards the announcement. Second, the path enables policy. An AS can prefer shorter paths. It can avoid paths through competitors. It can refuse to transit traffic through certain networks for political, economic, or security reasons.

But notice what the AS path doesn't do. It doesn't prove that any of those networks actually handled the packet. The path is a list of claims, not a chain of receipts.

13.2.2. THE ROUTE SELECTION ALGORITHM

When multiple routes exist to the same destination, BGP picks a winner—and the winner captures the traffic.

That's the key insight: if you can make your route look better than the legitimate route, packets flow to you, and BGP doesn't verify whether you deserve them.

How does BGP pick the winner? Through a hierarchy of preferences, the most important being LOCAL_PREF, local preference. Network administrators assign this number to express how much they favor a given route—higher numbers win.

Why prefer one route over another? Cost and politics—a path through a cheap peer might get LOCAL_PREF 100. A path through an expensive transit provider might get LOCAL_PREF 50. The highway is faster, but the back roads are free.

If LOCAL_PREF ties, the shorter AS path wins. Fewer hops usually mean faster delivery. A three-hop path beats a five-hop path, all else being equal.

But here's what matters: the competition is about appearances, not legitimacy. The route that *looks* best wins, whether or not the announcer actually controls the destination.

Let's trace a packet from a laptop in Chicago to a YouTube video server in California—watch for the trust.

The laptop sends the packet to the home router, addressed to 208.65.153.47, a YouTube server. The router doesn't know where that is, so it sends everything to the ISP—let's say Comcast.

Comcast's edge router receives the packet. It looks up 208.65.153.0/24 in its BGP routing table and finds an entry: "To reach this prefix, send to Google (AS 15169) via the Chicago exchange point." The router forwards the packet to Google's router at that exchange.

Google's router receives the packet. It looks up the same prefix and finds an internal route: "To reach 208.65.153.0/24, send to the Los Angeles data center." Google's internal network carries the packet across the country.

At the Los Angeles data center, another router receives the packet. This one has a more specific entry: "208.65.153.0/25 is handled by rack 47." The packet reaches the correct server. Total time: about 60 milliseconds.

Every hop consulted its BGP-derived routing table. Every table contained entries learned from neighbors, who learned from their neighbors. The packet crossed three autonomous systems: the ISP, Google, and YouTube's internal network. At each step, routes that began as BGP announcements guided the way.

At no step did anyone verify that those routes were legitimate. The ISP accepted Google's announcement. Google accepted YouTube's. The chain stretched from Chicago to California, every link relying on good faith.

13.2.3. LONGEST PREFIX MATCH

Here's where the trouble starts.

When a router receives a packet, it looks up the destination address in its routing table. But routing tables don't list individual addresses. They list prefixes: blocks of addresses. Think of it like highway signs. A sign doesn't say "123 Main Street, Apartment 4B." It says "Downtown—Next Exit." The router works the same way: "All addresses starting with 208.65 go this way."

Prefixes use a notation like 208.65.152.0/22, where the /22 specifies how many bits identify the network. A /22 covers 1,024 addresses, a /24 covers 256, and a /25 covers 128.

Remember those numbers—they're about to matter.

Consider overlapping prefixes: suppose one route announces 208.65.152.0/22 (a block of 1,024 addresses) and another announces 208.65.153.0/24 (a subset of 256 addresses). Both announcements are valid, both point to the same address space—which one wins?

The routing rule is simple: the most specific prefix wins. This is called longest prefix match. If the highway sign says "California—This Way" and another sign says "Los Angeles—Next Exit," you take the more specific exit. Routers follow the same logic. A /24 beats a /22 because /24 is more specific.

Why allow this overlap at all? Because it's useful—a large provider can announce one big block while customers announce smaller pieces within it. Traffic engineers can split prefixes to balance load. The flexibility is a feature.

Until it becomes a weapon.

Anyone who announces a more specific prefix captures traffic, even if they don't own that address space. If YouTube operates 208.65.152.0/22 and I announce 208.65.153.0/24, traffic to those 256 addresses flows to me instead. The routers don't know I'm lying. They just see a more specific sign and follow it.

This is not a theoretical vulnerability—it's an instruction manual.

BGP has no authentication layer. When an AS announces a route, other ASes simply believe it. The protocol doesn't verify that you own the addresses you claim, that your AS path is real, or that you intend to deliver the traffic rather than swallow it. BGP verifies nothing.

The reasoning made sense in 1989. Autonomous systems connected through peering agreements—contracts specifying what traffic each side would accept and forward. If someone lied about their routes, they'd violate the agreement, their peers would notice, and they'd be disconnected. Reputation and business relationships would enforce honesty.

But peering is transitive. If AS A peers with AS B, and AS B peers with AS C, then A will receive routes that originated from C, filtered and re-announced by B. The path might be `B C destination`. AS A has no relationship with C. It cannot verify C's announcements. It relies on B,

and B relies on C, and somewhere down that chain, someone might be lying.

The internet of 2008 had over 30,000 autonomous systems. The web of trust had become so tangled that no one could trace it.

Years later, Yakov Rekhter reflected on what he and Lougheed had sketched that day in Austin. "There was no concept that people would use this to do malicious things." The napkin protocol had been designed for a network of colleagues. What they built was the routing layer for five billion strangers.

On a Sunday afternoon in February 2008, that premise would be tested.

13.3. TWO HOURS OF DARKNESS

It was Sunday, February 24, 2008, and YouTube was disappearing.

At the company's operations center in San Bruno, California, engineers watched their traffic graphs collapse. YouTube, which normally served over 100 million videos per day, was suddenly unreachable—not from a single region, but from everywhere.

The problem had started half a world away, in Pakistan.

The Pakistani government had a problem with YouTube. A video had been uploaded that authorities considered blasphemous. The Pakistan Telecommunication Authority ordered local internet providers to block access. Pakistan Telecom, the country's largest telecommunications company, was ordered to make YouTube disappear.

The engineers at Pakistan Telecom faced a technical question: how do you block a website? One option was DNS filtering: manipulate the domain name system so that queries for youtube.com return nothing. But that's easy to circumvent. Users can change their DNS settings to use foreign servers.

Pakistan Telecom chose a more aggressive approach. They would hijack YouTube's IP addresses at the routing level. If traffic to YouTube's addresses never left Pakistan, users couldn't reach the site even if they knew the right IP.

YouTube operated its content from a block of addresses in the 208.65.152.0/22 range. To intercept that traffic, Pakistan Telecom announced a more specific prefix: 208.65.153.0/24. They configured the route to point to a "null route," a dead end where packets would be discarded. From the perspective of BGP, Pakistan Telecom was telling the world: "We can reach 208.65.153.0/24. Send us your traffic." What happened to that traffic after it arrived was Pakistan's problem.

The null route worked perfectly inside Pakistan. Pakistani users trying to reach YouTube had their packets swallowed, and the censorship was complete.

Then the route leaked.

Pakistan Telecom had a transit provider named PCCW, a Hong Kong-based telecommunications company that provided international connectivity. The BGP session between Pakistan Telecom and PCCW was configured to share routes. Normally, Pakistan Telecom would only announce routes it was authorized to use—its own address space and the address space of its customers. But someone made a mistake. The internal null route to YouTube's prefix was announced externally.

PCCW received the route and saw 208.65.153.0/24 being announced by Pakistan Telecom. The route was more specific than YouTube's legitimate announcement of 208.65.152.0/22. PCCW's routers did exactly what BGP requires: they preferred the more specific prefix, updated their routing tables, and announced the new route to their peers.

From PCCW, the false route spread to Asia's major carriers. Within minutes, it had reached European networks. A few minutes more, and it was in North America.[159] The AS path grew longer with each hop, but the route remained valid. No one questioned whether Pakistan Telecom actually controlled YouTube's addresses. The protocol had no mechanism for questioning.

By 18:47 UTC, YouTube was unreachable from most of the world's internet. Traffic destined for the video platform flowed toward Pakistan and vanished.

On the NANOG mailing list, where network operators coordinate the internet's plumbing, confusion turned to realization. "Routing to Pakistan?" one operator wrote, watching the traceroutes terminate in Islamabad. The graphs didn't lie—the world's cat videos were being sent to a null route in South Asia.

YouTube's engineers saw the problem immediately in their monitoring systems: inbound traffic had crashed. The site was up, their servers running fine, but no one could reach them because the traffic was being absorbed somewhere upstream, somewhere beyond their control.

The standard response to a BGP hijack is to announce an even more specific prefix. If the attacker has claimed /24, announce two /25s.

BGP's longest-prefix-match rule will route traffic to the more specific announcements, away from the hijacker.

YouTube began announcing 208.65.153.0/25 and 208.65.153.128/25. These /25 prefixes carved the hijacked space in half, each one more specific than Pakistan Telecom's /24. As these announcements propagated through the internet, traffic began to return, but propagation takes time—some networks converged quickly, while others, with longer paths to YouTube's actual address space, took longer to learn the legitimate routes.

Meanwhile, PCCW had identified the source of the problem. They began filtering Pakistan Telecom's announcements, refusing to accept or propagate routes to YouTube's address space. Without PCCW as a path, the false route lost its reach. Networks that had been sending traffic through Pakistan's null route switched to other paths.

By 21:01 UTC, approximately two hours after the hijack began, YouTube had recovered. The video platform was reachable again. The null route in Pakistan, which had only been meant to block local users, had instead blacked out YouTube for most of the planet.

Pakistan Telecom called it an accident. Their engineers had configured an internal null route for censorship purposes and failed to prevent it from leaking to external peers. A configuration error, not an attack.

The explanation was plausible—BGP misconfigurations are common, and networks accidentally announce routes they shouldn't. Usually the damage is limited—a few thousand users might lose connectivity to a specific destination for a few minutes until someone notices and fixes the mistake. What made the Pakistan incident unusual was the target.

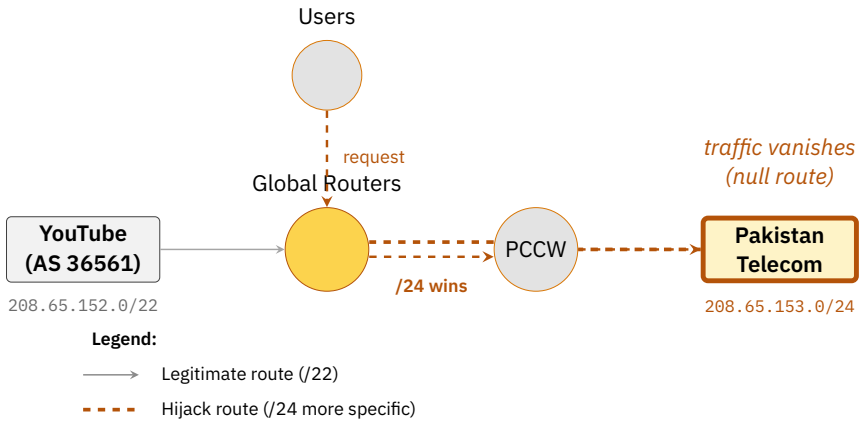


Figure 13.1: The Pakistan/YouTube BGP hijack. Pakistan Telecom announced a /24 prefix for YouTube's address space, more specific than YouTube's legitimate /22. BGP's longest-prefix-match rule caused global routers to prefer the false route, sending worldwide traffic into a null route.

YouTube was one of the web's most popular destinations. Millions of users tried to reach it during those two hours and failed.

But the accident revealed something important: intent doesn't matter. Whether Pakistan Telecom was trying to censor YouTube or trying to attack it or trying to intercept its traffic, the mechanism was identical. Announce a more specific prefix, and let BGP do the rest. The protocol couldn't tell the difference between a mistake, a censorship order, and an attack.

Two hours. That's how long a misconfigured router in Pakistan blacked out one of the world's largest video platforms. The root cause was simple: BGP has no mechanism to verify that an AS is authorized to announce a given prefix.

13.3.1. THE AFTERMATH

The Pakistan incident accelerated discussions about BGP security that had been simmering for years. Engineers had long known that route hijacking was possible. What they lacked was motivation to fix it.

The solution is called RPKI—Resource Public Key Infrastructure. It works by creating a cryptographic chain of trust from the regional internet registries, which assign address blocks, down to the individual networks that use those blocks. Each network can publish a Route Origin Authorization (ROA), a signed statement saying: "AS 36561 is authorized to announce 208.65.152.0/22." Other networks can validate announcements against these ROAs. If Pakistan Telecom announces YouTube's prefix, and there's no ROA authorizing that announcement, the route can be rejected.

RPKI was standardized in 2012.[160] As of 2024, approximately 50% of announced routes had corresponding ROAs.[161] That's progress, but it leaves half the internet's routing table unverified. And even where ROAs exist, many networks don't validate them. They accept announcements whether or not they're cryptographically authorized. The chains of trust work, but only if everyone in the chain checks them.

The deeper problem is incentives. Deploying RPKI requires engineering effort and ongoing maintenance. The benefit accrues not to the deploying network but to everyone else—you protect others from accidentally or maliciously routing through you. In a competitive industry where engineering resources are always stretched thin, collective infrastructure investments often lose to immediate priorities.

The Pakistan-YouTube incident was dramatic because of its scale. But smaller hijacks happen regularly. In 2022, a Russian network briefly announced routes for Twitter, Facebook, and Google.[162] In 2018, traffic to Amazon's Route 53 DNS service was hijacked to steal cryptocurrency.[163] In 2019, European mobile traffic was routed

through China Telecom.[164] In January 2024, a hacker compromised Orange Spain's RIPE account using stealer malware, then modified the BGP configuration to hijack over half the company's traffic. Some incidents are mistakes. Some are crimes. BGP cannot tell the difference.

13.3.2. THE PERSISTENT THREAT

In February 2013, a cybersecurity firm called Mandiant published a 60-page report with a map of a single building: a 12-story office tower at 208 Datong Road in Shanghai.[165] The building, Mandiant alleged, housed Unit 61398 of the People's Liberation Army—a signals intelligence unit responsible for systematic intrusion into American companies and government agencies.

Kevin Mandia, who founded the company that bore his name, had been tracking Chinese intrusions for years. His investigators had traced hackers across dozens of breaches, watching their working hours (Beijing time), their language (simplified Chinese), their targets (defense contractors, energy companies, technology firms). The patterns pointed to a single organization, and the IP addresses pointed to a single neighborhood in Shanghai, where the building at 208 Datong Road was large enough to house hundreds of hackers.

The report named names: Wang Dong, using the handle "Ugly Gorilla," and Mei Qiang, also known as "SuperHard"—specific individuals whom Mandiant linked to specific intrusions, allegedly at the direction of the Chinese military. A year later, the U.S. Department of Justice indicted five alleged officers of Unit 61398, marking the first time the United States had criminally charged foreign government officials for network espionage.[166]

The indictments were largely symbolic—the officers would never face trial in an American court—but the Mandiant report changed the

conversation. Nation-state hacking was no longer an open secret; it was documented, publicly attributed, and backed by detailed evidence.

While the hackers at 208 Datong Road were stealing secrets through compromised networks, China Telecom was achieving something potentially more troubling through BGP. Between 2015 and 2019, researchers documented multiple incidents where traffic destined for European and American networks was routed through China.[167] In one case, traffic from Canada to Korean government sites transited Beijing for six months. In another, financial institutions' traffic was briefly rerouted through mainland Chinese infrastructure.

Were these accidents? Configuration errors happen constantly in BGP. But unlike the Pakistan incident, where the hijack was brief and immediately noticed, some of these routes persisted for months. The traffic passed through infrastructure where it could be copied, analyzed, or modified. Even when packets eventually reached their destination, they had passed through territory where China's intelligence services had complete visibility.

The BGP incidents highlighted an exposure that no patch could fix. Nation-states operate legitimate networks. China Telecom is a major carrier with peering relationships across the globe. When a Chinese network announces routes, other networks accept them because that's how BGP works. The mechanism cannot distinguish between a routing optimization and a surveillance operation.

The Unit 61398 hackers and the BGP incidents represented two approaches to the same goal: break into networks one at a time, or redirect their traffic at the routing layer. The first approach is targeted but leaves traces; the second is silent but indiscriminate. Both exploit the same blind spot—the expectation that remote systems are secure, that BGP announcements are legitimate.

The napkins are lost.

Lougheed and Rekhter never kept them. The restaurant in Austin is long gone. What survived is the protocol they sketched—and as of 2024, it routes over a million prefixes through more than 75,000 autonomous systems.[168] Three handwritten sheets became the routing layer for the planet.

The system works because, usually, network operators tell the truth. The exceptions are brief enough and rare enough that the internet survives them. But survival isn't the same as security. The napkin protocol now connects adversaries: nation-states with intelligence services, companies with competitive interests, criminals with financial motives. All of them can announce routes, and all of them can redirect traffic. The sketches in Austin never imagined them.

The premise that made BGP work—that your neighbors would tell the truth—was reasonable for the internet of 1989, where operators knew each other at conferences and reputation enforced honesty. That internet is gone. The protocol is not.

Lougheed and Rekhter had solved the problem of getting packets where they needed to go. But arrival was only half the story. Once a packet reached the network's edge, someone still had to open the door.

In Pleasanton, California, an engineer named Carl Rigney was building a different kind of trust. His protocol would guard WiFi networks, VPN connections, and enterprise systems for three decades, all of it resting on a single shared secret.

CHAPTER 14

THE SHARED SECRET

Transactions between the client and RADIUS server are authenticated through the use of a shared secret, which is never sent over the network.

— RFC 2865, Remote Authentication Dial In User Service, June 2000

14.1. THE MODEMS WOULDN'T STOP

The modems wouldn't stop ringing.

It was 1991. Steve Willens had a problem. Livingston Enterprises, the company he led from Pleasanton, California, made terminal servers—the boxes between dial-up modems and the network.[169] When someone picked up their phone, dialed a local number, and waited for the carrier tone, a Livingston PortMaster answered. It asked for credentials, checked them against a list, connected the caller to the internet.

The list was the bottleneck. Merit Network, which ran the NSFNET backbone, had contracted with Livingston to handle thousands of dial-

up users; each PortMaster needed to know every valid username and password.

This didn't scale. Adding a user meant updating every terminal server, and changing a password meant propagating it to dozens of machines. A disk failure could wipe out the entire user database. The authentication system was distributed in the worst way: replicated everywhere, consistent nowhere.

Carl Rigney, an engineer at Livingston, started sketching a solution.

The insight was simple: separate the database from the equipment. Store credentials in one place, and when a user dials in, have the terminal server ask a central authentication server: is this username valid? Is this password correct? The server says yes or no, and the terminal server follows orders.

Rigney called it `RADIUS`: Remote Authentication Dial-In User Service—an unwieldy name for an elegant design.

A PortMaster running `RADIUS` didn't need a user database or disk space for credentials—it simply asked questions and accepted answers. All intelligence lived in the `RADIUS` server, which could run on a Unix machine with proper backups and administration. Add a user once and every terminal server would recognize them; change a password once and it changed everywhere.

The protocol did more than authentication. Rigney built in authorization—what the user was allowed to do—and accounting—what the user actually did. The three functions formed what network engineers would call the **AAA model**: Authentication, Authorization, and Accounting. `RADIUS` handled all three.

Merit deployed `RADIUS` in 1992, and the protocol worked exactly as designed. Users dialed in, the PortMasters queried the central server, and access was granted or denied in milliseconds. The user database lived in one place, password changes took effect instantly, and when

Merit added new dial-up points of presence, they just pointed the new equipment at the existing RADIUS server.

Other organizations noticed. Internet service providers were sprouting across the country, each struggling with the same problem Merit had solved. They needed to authenticate dial-up users at scale. RADIUS offered a ready-made answer.

Livingston published the protocol specification and released a reference implementation. They weren't trying to keep RADIUS proprietary—their business was selling terminal servers, and RADIUS made those servers more valuable. An open protocol meant that customers could run whatever RADIUS server they wanted, that competitors could build compatible equipment, and that RADIUS would spread.

It spread.

By the mid-1990s, RADIUS was the de facto standard for dial-up authentication—every major ISP ran it, every equipment vendor supported it. When customers connected worldwide, RADIUS answered the question: are you who you say you are?

The IETF standardized the protocol in 1997 as RFC 2138[170], then updated it in June 2000 as RFC 2865.[171] The acknowledgments section was brief: "RADIUS was originally developed by Steve Willens of Livingston Enterprises for their PortMaster series of Network Access Servers."

Rigney was listed as the first author. He had designed the protocol to solve a practical problem for a paying customer. He had no illusions about perfect security. The goal was something that worked, that scaled, that could be implemented on the limited hardware of early 1990s terminal servers.

The security model reflected these constraints. The terminal server and the RADIUS server shared a secret—a password, really, though no one called it that—which authenticated the terminal server to the

RADIUS server. It also "encrypted" the user's password in transit, using a technique that was good enough for 1991.

The RFC described the method precisely. Take the shared secret. Append the Request Authenticator, a random 16-byte value that changes with each request. Hash the result with MD5—Message Digest 5, a one-way fingerprint function that turns any input into a fixed 128-bit code. XOR—a reversible scrambling operation—that hash with the user's password. Send the result. On the other end, reverse the process.

```
RFC 2865 --- Section 5.2 --- Password Hiding

Call the shared secret S and the pseudo-random 128-bit
Request Authenticator RA. Break the password into 16-octet
chunks p1, p2, etc.
b1 = MD5(S + RA)    c(1) = p1 xor b1 b2 = MD5(S + c(1))
c(2) = p2 xor b2
```

This wasn't encryption—it was obfuscation. Anyone who knew the secret could recover the password instantly, and anyone who captured enough traffic could potentially guess it. The RFC acknowledged the concern: "The User-Password hiding mechanism described in Section 5.2 has not been subjected to significant amounts of cryptanalysis in the published literature."

But the RFC went ahead anyway, because in 1991 the threat model was different. RADIUS traffic flowed over private networks, between equipment that organizations controlled. The secret stayed between machines in the same data center, never transmitted across hostile territory. The premise was that if an attacker could read your RADIUS traffic, you had bigger problems.

That premise held for dial-up. It wouldn't hold forever.

14.2. THE NIGHTCLUB BOUNCER

14.2.1. HOW RADIUS WORKS

Think of a nightclub with a guest list—but the bouncer can't read.

The bouncer at the door doesn't have the list—instead, he has a radio. When you show up and give your name, he calls someone inside—the manager, let's say—and asks if you're on the list. The manager checks, then radios back: yes, let them in and give them a VIP bracelet, or no, turn them away.

The bouncer is the Network Access Server, the NAS, the manager is the `RADIUS` server, and the radio channel is the network connection between them. The radio code they use to confirm each other's identity—that's the shared secret. The bouncer and the manager share a code word, and anyone claiming to have the guest list must prove they know the code.

`RADIUS` follows this pattern. A user connects to the NAS and provides credentials, and the NAS packages those credentials into a request and sends it to the `RADIUS` server. The server consults its database, makes a decision, and sends back a response, which the NAS enforces without question.

What the NAS doesn't do: verify the response independently. The bouncer trusts the voice on the radio. If someone else gets on that frequency and says "let them in," the bouncer lets them in.

When you type your username and password, the NAS constructs an `Access-Request` packet. Your username travels in clear text—there's no point hiding it, since the server needs to look it up—but your password gets scrambled using `MD5` and the secret key.

The NAS sends the packet over UDP and waits—no TCP connection, no guaranteed delivery. If no response arrives, the NAS retransmits, and if the primary RADIUS server is unreachable, it might try a backup. UDP made this failover behavior simple, but it also meant neither side verified source addresses.

The request can go to any server that shares the secret. But how does the NAS know that the response came from a legitimate server?

Three responses are possible. An `Access-Accept` means you authenticated successfully—the nightclub manager says "let them in and give them the VIP bracelet." An `Access-Reject` means authentication failed—wrong password, disabled account, unknown user—and the game is over.

An `Access-Challenge` means the server wants more information—maybe a second factor, like a code from a hardware token. The exchange can continue through multiple rounds until the server reaches a final decision.

The problem: the NAS lacks any mechanism to confirm a response is legitimate. The server could be spoofed. An attacker could inject a fake `Access-Accept`, granting access to someone who should have been rejected.

RADIUS addresses this with the Response Authenticator. When the server constructs a response, it computes an MD5 hash—there's that word again—over the entire response packet, the Request Authenticator from the original request, and the secret. The result goes in the Authenticator field. When the NAS receives the response, it performs the same calculation. If the hashes match, the response is authentic.

The formula looks like this:

```
ResponseAuth =
MD5(Code+ID+Length+RequestAuth+Attributes+Secret)
```

Without knowing the secret, no one can forge this hash.

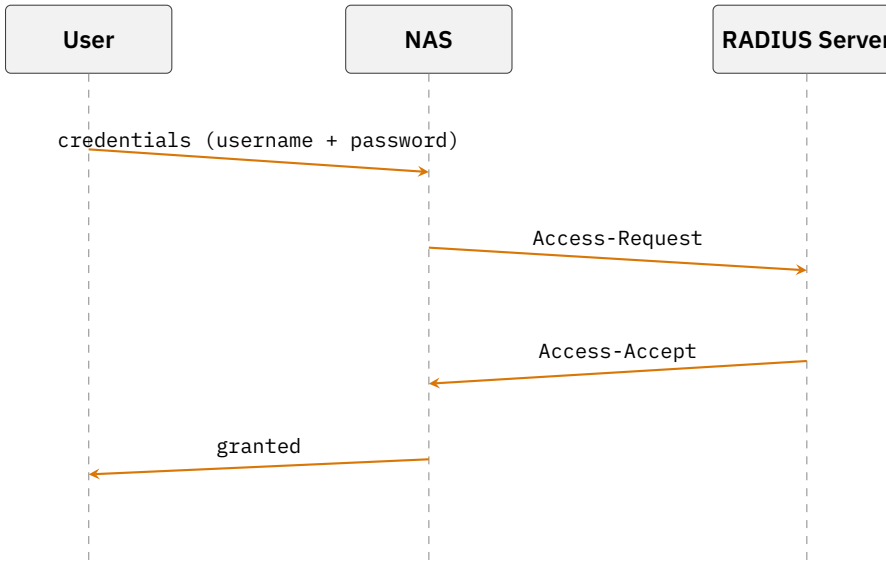


Figure 14.1: RADIUS authentication flow. The Network Access Server proxies user credentials to the central RADIUS server, which holds the user database. The shared secret between NAS and server is never sent over the network.

Capturing a response doesn't help, because it can't be replayed against a different request—the Request Authenticator changes each time, binding the response to both the request and the secret. The entire security of `RADIUS` authentication rests on this binding being unforgeable.

`MD5` was supposed to make forgery impossible. The designers knew the hash function wasn't perfect, but they believed the secret key would compensate—even if `MD5` had theoretical weaknesses, an attacker who didn't know the secret couldn't exploit them.

14.2.2. THE SHARED KEY

RADIUS rests on two premises: that the network between the NAS and the authentication server is safe, and that MD5 cannot be broken.

Why UDP instead of TCP? The answer was speed—you're staring at a login prompt, waiting for access, and TCP's reliability guarantees take time with their acknowledgments, retransmissions, and congestion control. With UDP, RADIUS controls its own timing: if a response doesn't arrive in two seconds, send another request, and if the server is dead, fail over immediately.

UDP made the implementation simpler, too—early RADIUS servers were single-threaded, following a straightforward pattern of receive, process, respond. In 1991, simpler meant it would actually get built.

But UDP has no concept of connection and doesn't verify source addresses—anyone can send a packet with any forged source IP. The RADIUS server accepts that requests come from legitimate NAS devices, and the NAS accepts that responses come from the real server. Neither side checks.

The secret is supposed to compensate, proving that both ends are legitimate while the MD5 hashing ensures that responses cannot be forged. But if someone positions themselves between the NAS and the authentication server, the first premise collapses. If someone finds a way to forge MD5 hashes, the second premise collapses too.

The RFC made the bargain explicit: both sides must know the secret, and both sides must keep it.

This secret authenticates the transaction, but it has to be distributed somehow. Administrators type it into configuration files on both ends. The RFC recommends at least 16 bytes of random data.

In practice? "radius123." "companyname." "secret."

Recommendations are not requirements.

And the secret has to remain secret, since every device that shares it can impersonate every other device that shares it. When a new access point joins the network, it gets the secret; when an old access point is decommissioned, the secret should change. It almost never does.

14.3. BLAST-RADIUS

On July 9, 2024, security researchers published a paper with an unassuming title: "RADIUS/UDP Considered Harmful".[172] The content was less modest—they had found a way to forge RADIUS responses, turning rejected authentication attempts into accepted ones. The vulnerability had existed since 1991.

The research team spanned institutions: Sharon Goldberg from Boston University and Cloudflare; Miro Haller and Nadia Heninger from UC San Diego; Mike Milano from Cloudflare; Dan Shumow from Microsoft; Marc Stevens from CWI Amsterdam; Adam Suhl from Boston University.[172] They had been working together since late 2023, following a trail that led back to MD5. This was not amateur work. The team included some of the world's foremost experts on hash function cryptanalysis. Stevens had spent fifteen years studying MD5 collisions. The attack required months of mathematical refinement and access to GPU clusters that cost thousands of dollars per hour to run.

They named it Blast-RADIUS. CVE-2024-3596.[173]

The vulnerability centered on the Response Authenticator—the fingerprint that proved a response was genuine. Remember: that fingerprint depends on the shared secret, which the attacker doesn't know. The prevailing wisdom was that this made forgery impossible. Even if MD5 was weakening, the secret should protect the construction.

The wisdom was wrong. Cryptographers had been chipping away at MD5 for decades. In 1996, researchers found theoretical cracks. In 2004, a team from Shandong University demonstrated practical collisions—different inputs that produced the same fingerprint.[174] In 2007, Marc Stevens showed something more dangerous: given two different messages, you could craft them so they produced the same hash.[175] Think of it as finding two different keys that open the same lock—except the lock is supposed to be unique to every door.

The Blast-RADIUS attack exploited this. The attacker positions themselves between the access point and the RADIUS server—an interception position—and waits. When a user tries to authenticate, the attacker captures the request and forwards it. The server rejects the invalid credentials and sends back an Access-Reject.

Here is where the forgery happens. The attacker blocks the legitimate rejection and constructs a fake Access-Accept instead. The fake response needs a valid fingerprint—one that matches what the access point will compute when it checks. The attacker doesn't know the secret. But they know enough of the other ingredients (captured from the original exchange) to exploit the MD5 weakness: they can craft their forged message so that its fingerprint comes out identical to what a legitimate response would produce.

The computation is expensive—not laptop-in-a-coffee-shop expensive, but well-funded-research-lab expensive. Finding the right forgery takes three to six minutes on specialized hardware costing tens of thousands of dollars. That sounds like a barrier, but RADIUS is patient. The NAS will wait for a response, retransmitting periodically. The server has already sent its legitimate Access-Reject, but the attacker is blocking it. Three to six minutes is plenty.

When the collision is found, the attacker sends the forged Access-Accept to the NAS, which computes the Response Authenticator, finds that it matches, and accepts the response as genuine. An unauthorized

user has been authenticated. The protocol worked exactly as designed—the cryptography failed.

RADIUS authenticates far more than dial-up modems—that use case faded with the rise of broadband—but RADIUS adapted, and in 2024 it was everywhere.

Enterprise WiFi networks use RADIUS—when you connect your laptop to a corporate wireless network and enter your username and password, that authentication goes to a RADIUS server. The protocol is called 802.1X, and it runs on WPA2-Enterprise and WPA3-Enterprise networks at every Fortune 500 company, every university, every hospital.

VPN authentication uses RADIUS—when employees connect to the corporate network from home, the VPN gateway often validates their credentials through RADIUS, and the same applies to network device management for the routers, switches, and firewalls that require administrator login.

ISP subscriber authentication uses RADIUS—when you plug in your home router and it connects to your cable or DSL provider, RADIUS is often involved in verifying your service.

The researchers estimated that "billions of devices" relied on RADIUS. The vulnerability affected all of them.

Alan DeKok had spent two decades with RADIUS and had created FreeRADIUS, the open-source server that handled more RADIUS authentication than any other implementation.[176] When the researchers contacted him in February 2024, he understood the implications immediately.

The fix was theoretically simple. RADIUS had always supported an optional attribute called Message-Authenticator, which used HMAC-MD5 instead of plain MD5 when present in a request, binding the message to the secret key in a way that resisted collision attacks. If servers

required Message-Authenticator on all packets, Blast-RADIUS wouldn't work.

"Optional" is a powerful word. For thirty-three years Message-Authenticator had been optional—equipment deployed in 2005 didn't include it, equipment deployed in 2015 often didn't either. Administrators configured their systems to work, not to enforce security requirements that nobody demanded.

Making Message-Authenticator mandatory meant breaking compatibility with decades of deployed equipment. But leaving it optional meant leaving the vulnerability open.

The coordinated disclosure began in February 2024, when the researchers contacted every major vendor—Microsoft, Cisco, Aruba, Palo Alto, Juniper, Fortinet, FreeRADIUS. Patches needed development, testing, and distribution; documentation needed updates; customers needed warning.

The public disclosure came on July 9. Patches followed immediately: FreeRADIUS required Message-Authenticator by default; Microsoft updated its Network Policy Server; networking vendors pushed firmware updates.

Updating RADIUS infrastructure isn't like updating a web browser. Servers sit buried in data centers, managed by teams that might not know they exist. Access points scatter across campus buildings, installed by contractors who moved on years ago. VPN concentrators handle thousands of production connections—nobody wants to reboot them.

The long-term fix is RadSec: RADIUS over TLS, standardized in RFC 6614.[177] Wrap the entire RADIUS exchange in a TLS connection, and the collision attacks become irrelevant—the encryption protects the traffic while the certificates authenticate the endpoints. But RadSec has been available since 2012 and adoption remains limited, since it requires

infrastructure changes, certificate management, and configuration updates. The path of least resistance is to keep using UDP.

The Blast-RADIUS paper concluded with a section on responsible disclosure. The researchers had waited five months between contacting vendors and going public, working with implementers to develop patches and coordinating timing so that fixes would be available when the vulnerability was announced.

But they also reflected on what the vulnerability meant. MD5's weaknesses had been known for twenty years before someone demonstrated a practical attack on RADIUS, and the protocol had been running on faith—faith that the crypto wouldn't be broken, that attackers wouldn't bother, that the infrastructure was secure.

That faith had held for thirty-three years. Now it was gone.

Carl Rigney designed RADIUS to solve a practical problem: authenticating dial-up users for Merit Network. He succeeded. The protocol scaled from thousands of users to billions of devices, outliving the modems it was built for and adapting to WiFi, VPNs, and enterprise networks.

The shared secret at the heart of RADIUS was a design choice, not a design flaw. In 1991, it was reasonable to expect that the communication between a terminal server and an authentication server would be protected—the RADIUS traffic stayed inside the organization's network, the threat was external attackers, and internal links were trusted.

The zone shrank. Networks connected to the internet, employees worked from home, and access points spread across buildings connected over paths that nobody controlled. The shared secret became a shared liability—a password stored on every device, distributed to

every administrator, kept unchanged for years because changing it would break something.

And MD5 aged. The hash function that seemed adequate in 1991 was showing cracks by 1996, visible fractures by 2004, and gaping holes by 2024. Cryptographic algorithms have lifespans—they're published, adopted, attacked, and eventually broken—and the protocols that depend on them should be designed to upgrade. RADIUS wasn't.

The vulnerability wasn't about speed or timing. RADIUS relied on cryptography, with the secret key and MD5 hash supposed to make forgery impossible. When MD5 broke, the foundation broke with it.

The thirty-three-year-old vulnerability waited patiently for someone to exploit it.

Someone did.

Every corporate WiFi login, every VPN connection, passes through the protocol Rigney designed for dial-up modems.

Most users who connect to their company's network have never heard of RADIUS. They type a username and password, wait a moment, and the connection opens. The authentication happened somewhere—a server in a data center, a process that began and concluded before the browser loaded its first page. RADIUS ran invisibly behind every one of those moments.

The shared secret at the heart of RADIUS was never wrong for 1991. The assumption that RADIUS traffic would stay inside trusted networks held for dial-up modems on private lines. The world moved: employees worked from home, networks connected to the internet, access points spread across campuses and hotels and airports. The secret stayed the same. RADIUS adapted, stretched, survived—and in 2024, researchers with GPU clusters and twenty years of cryptanalysis finally broke the

mathematical assumption that had kept it whole. The protocol that Rigney sketched for a dial-up problem now guards the door for billions of connections, still trusting the shared secret, still serving without being seen.

RADIUS governed who could join the network. But before authentication could happen, a machine needed something more basic: an address. Somewhere in Hartford, Connecticut, a network administrator stared at her spreadsheet of IP assignments and wondered how much longer she could keep track of them all by hand.

CHAPTER 15

THE FIRST RESPONSE

DHCP allows a host to obtain an IP address dynamically, without requiring significant administrative attention for the addition of new hosts to the network.

— RFC 2131, Ralph Droms, 1997

15.1. KAREN CHEN'S SPREADSHEET

A network administrator we'll call Karen Chen had a spreadsheet problem.

It was 1992, and Chen worked at a mid-sized insurance company in Hartford—one of thousands of network administrators facing the same crisis. Her desk faced a window overlooking the parking lot, but she rarely looked up. Most of her attention went to an Excel file called `IP_ADDRESSES.XLS`. The spreadsheet had 347 rows, one for each computer in the building. Each row listed an `IP` address, a machine name, a physical location, and the employee who used it.

When someone joined the company, Chen found an empty row in the spreadsheet, assigned the next available address, walked to the

new hire's desk, and typed the numbers into Windows for Workgroups. When someone moved offices, she updated the spreadsheet and reconfigured their machine. When someone left, she deleted their row and hoped she remembered to do it before someone else claimed that address.

The spreadsheet was her lifeline. Without it, no one knew which addresses were in use. Assign the same address to two machines and both would fail. Miss an entry and the address would be lost, marked as used but actually free. The spreadsheet was also her nightmare. It was never quite right. Someone always forgot to tell her about the new contractor in marketing. Someone always moved without filing a ticket.

The worst days were laptop days. The company had started issuing Compaq laptops to executives, and executives traveled. An executive's laptop might connect from the fifth floor on Monday, the conference room on Wednesday, and a client site in New York on Friday. Each location was a different network. Each network required a different `IP` address. The spreadsheet couldn't handle it. Chen couldn't handle it.

Thousands of network administrators shared the same headache. `IP` addresses needed to be unique. Assignments required tracking. Someone had to maintain the list. And the list kept growing, kept changing, kept breaking.

The archives of Usenet newsgroups from 1991 told the story. In `comp.dcom.lans.ethernet`, administrators traded war stories about "duplicate `IP` address" errors causing machines to hang. The only solution was physical hunting: check the `ARP` table for the rogue Ethernet address, trace the cable to a specific port on a hub, walk down the hall, and find the machine. "Who took `192.168.1.5`?" wasn't just a technical question—it was an accusation.[178]

The problem had a partial solution, and it came from a surprising place: diskless workstations.

In the mid-1980s, some organizations deployed computers without hard drives. These thin clients booted over the network, downloading their operating system from a central server. The cost savings were significant, but the machines faced a bootstrapping problem. A computer without a disk couldn't store its network configuration. When it powered on, it didn't know its own address.

Bill Croft and John Gilmore solved this with the Bootstrap Protocol, BOOTP, published as RFC 951 in September 1985.[179] BOOTP was simple. A diskless machine would broadcast a request: "I need an address." A server with a table of hardware-to-IP mappings would respond: "You're this address. Boot from this server." The workstation would configure itself and proceed.

BOOTP solved the bootstrapping problem, but it didn't solve Karen Chen's spreadsheet problem. BOOTP servers required static tables, one entry for every machine, maintained by hand. An administrator still had to assign each address. A visitor with a laptop still couldn't connect. The table just lived on a server instead of in Excel.

Ralph Droms saw the next step. A computer science professor at Bucknell University in central Pennsylvania, he had been working on network configuration since the late 1980s, trying to make networks easier to manage. By 1993, he had a proposal: extend BOOTP with dynamic allocation.

The concept was straightforward. Instead of maintaining a table of permanent assignments, the server would maintain a pool of addresses. When a machine requested an address, the server would pick one from the pool and loan it out. The loan would have a time limit, called a lease. When the lease expired, the address would return to the pool.

Dynamic allocation solved the laptop problem. An executive connecting in Hartford would receive an address from the Hartford pool.

The same laptop connecting in New York would receive a different address from the New York pool. No administrator needed to update a spreadsheet. No one needed to know in advance which machines would connect.

Droms called it the Dynamic Host Configuration Protocol, DHCP. He published the specification as RFC 1531 in October 1993[180], then revised it as RFC 2131 in March 1997.[181] The later document became the standard that runs on almost every network today.

The protocol did more than assign addresses. It could configure anything a machine needed to join a network: subnet mask, default gateway, DNS servers, domain name, time servers. A properly configured DHCP server could bring a machine from complete ignorance to full network participation with a single exchange.

The exchange followed a four-step pattern the community would call DORA—four packets, and you're connected.

But those four packets carried an implicit trust assumption. When a machine broadcast a discovery request, any DHCP server could respond. The client would accept the first offer it received. No authentication existed, no verification that the server was legitimate. Physical presence on the wire implied authorization to configure it.

Droms wrote in RFC 2131: "DHCP is built on a client-server model, where designated DHCP server hosts allocate network addresses and deliver configuration parameters to dynamically configured hosts."[181]

Designated. The word did a lot of work. In practice, designation meant nothing. Any machine on the local segment could run a DHCP server. Any machine could respond to discovery requests. The fastest response won, and the client trusted whatever it received.

In 1993, this assumption made sense. Local networks were corporate. Access required physical presence in the building, past the front desk, through the locked door. If someone connected a rogue DHCP

server, they were an employee, and they had probably made a mistake, not mounted an attack.

DHCP spread rapidly. Microsoft added client support to Windows 95. Unix vendors added it to their distributions. Home routers shipped with DHCP servers enabled by default. By the end of the decade, most computers on most networks received their addresses automatically.

The spreadsheet problem was solved. Karen Chen and her counterparts could finally look up from their desks. The network configured itself.

Two decades later, in public WiFi hotspots across the world, attackers would exploit exactly this automation. The fastest server would win—and the fastest server wouldn't always be the legitimate one.

15.2. THE FIRST RESPONSE WINS

15.2.1. HOW DHCP WORKS

RADIUS relies on cryptography, but DHCP relies on something simpler: speed.

You power on your laptop in a hotel lobby with no IP address, no gateway, no DNS server—you know nothing about this network. Within seconds, you're browsing the web. How?

Your computer shouts into the void: "Is anyone out there? I need an address." Any DHCP server within broadcast range can answer, and your computer accepts whatever arrives first.

Think of it like checking into a hotel. You arrive at the front desk without a room assignment. The hotel has a block of rooms available. The clerk assigns you one, gives you a key, and tells you where to find

the elevator, the restaurant, and the pool. You keep the room for the length of your stay. When you leave, the room goes back into the pool.

But here's where the analogy breaks. At a hotel, the front desk is clearly marked, so you know you're talking to the actual hotel. With DHCP, anyone in the lobby can claim to be the front desk—and your computer cannot tell the difference.

Network engineers call the exchange DORA: Discover, Offer, Request, Acknowledge. Four packets. Let's trace what happens—and what's missing.

Your computer broadcasts a Discover message. Source address: 0.0.0.0, because you don't have one yet. Destination: 255.255.255.255, meaning everyone on the local network hears it. "I need an address," your machine announces. Every DHCP server—legitimate or not—receives this plea.

Servers respond with Offers. "I could give you 10.0.0.47," says one. "I could give you 10.0.0.112," says another. Each offer includes not just an address but a full configuration: gateway, DNS servers, lease duration. Your computer accepts the first offer that arrives.

The first offer—not the best offer, not the authenticated offer, just the fastest one.

Your computer broadcasts a Request: "I'll take the address from that first server." All servers hear this, so the ones who lost know their offered addresses are still available.

The winning server sends an Acknowledge: "Done. Here's your configuration." Your network interface comes up, and you're connected.

Four packets, a few hundred milliseconds, no verification at any step.

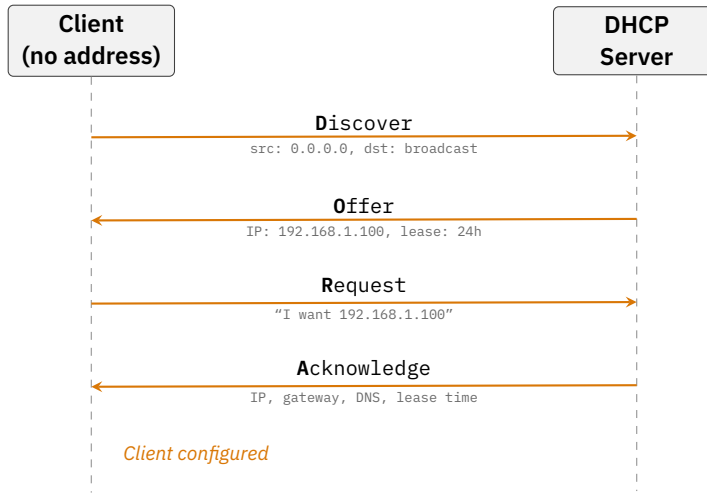


Figure 15.1: The DHCP DORA sequence. A client without an address broadcasts a Discover, receives an Offer with an available address, broadcasts a Request to claim it, and receives an Acknowledge with full configuration. The entire process takes milliseconds.

15.2.2. LEASES AND CONFIGURATION

Why not just assign addresses permanently? Because networks change—servers move, DNS entries update, and people come and go. Leases force your computer to check back periodically, picking up new configuration if anything has changed.

A corporate network might issue eight-hour leases. A coffee shop might use one-hour leases—short enough to recycle addresses as customers come and go. Your computer renews automatically, usually at the halfway point, before you notice anything.[181]

The system handles disruptions gracefully. A server reboot doesn't immediately disconnect everyone. A brief network outage doesn't break existing connections. The design is resilient, so long as every participant plays fair.

But DHCP doesn't just hand out addresses. It configures your entire network stack. The server tells you which gateway to use for reaching the outside world. It tells you which DNS servers to query. It can specify your domain name, your time servers, your proxy settings.

Every one of these settings is a lever. Pull the gateway lever, and all your traffic flows through a different machine. Pull the DNS lever, and every domain you type resolves to a different address. A rogue DHCP server doesn't just assign you a bad IP.

It rewrites your computer's understanding of the network.

15.2.3. WHY THE FIRST RESPONSE WINS

So why can't your computer be more careful? Why not wait for multiple offers and pick the most reliable one?

Because it lacks any mechanism to evaluate reliability. Your computer has no credentials to verify, no list of authorized servers, no certificates, no signatures, no shared secrets. It broadcasts a plea and accepts whatever comes back first.

The RFC is explicit about this. Clients "may choose to wait for multiple responses" but "should respond to every valid DHCPOFFER message." In practice, implementations accept the first offer and proceed.[181] Waiting costs time. On a well-managed network, the first response comes from the legitimate server. The optimization seems harmless.

On a network with an attacker, it's catastrophic.

The attacker's laptop can respond faster than the server in the data center, so your computer accepts the attacker's configuration. The attack succeeds before the legitimate server's packet even arrives.

This isn't a bug in DHCP. It's the design.

As we saw with Droms's "designated" servers, the protocol has no way to distinguish a legitimate server from an imposter. DHCP takes the network at its word, and the network doesn't always deserve that faith.

15.3. ANYONE WITH A LAPTOP

Breaking RADIUS required cryptographic expertise, GPU clusters, and months of research. Breaking DHCP required a laptop and a cup of coffee.

Consider a scenario that plays out at tech conferences worldwide. A hotel hosts a developer conference—three thousand attendees packed into meeting rooms and hallways. The conference WiFi network requires no password. Walk in, connect, start working.

An attacker checks into the hotel and carries a laptop to a lounge overlooking the atrium. Conference attendees streamed past, badges dangling from lanyards, laptops under their arms. He ordered coffee from the bar, opened his machine, and launched a program called Yersinia.[182] Named after the bacterium that causes plague, Yersinia was a network attack tool that anyone could download. No cryptographic expertise required. No mathematics. No understanding of hash functions or collision attacks. Just a free tool and physical proximity to the victims.

He started with reconnaissance. Yersinia sniffed the network traffic, watching DHCP exchanges between attendees and the hotel's access points. The legitimate server was at 10.50.0.1, handing out addresses in the 10.50.10.0/24 subnet with two-hour leases. Standard hotel configuration for high-turnover guest networks. No protections.

The attack had two phases. First, exhaust the legitimate server's pool. Second, become the only server left.

DHCP starvation was simple. He sent Discover requests with spoofed MAC addresses—hundreds per second, each pretending to be a different network card. Every request looked like a new device joining the network. The legitimate server offered an address to each one. He completed the exchanges, acquiring leases for addresses he would never use. Within minutes, the server's pool was empty. Legitimate attendees trying to connect received nothing.

Now came phase two. He ran a DHCP server on his laptop. When new attendees joined the network, their Discover broadcasts reached both the legitimate server (which had no addresses left) and his laptop (which had plenty). He responded, and the attendee accepted.

The configuration he provided looked almost legitimate. Same subnet mask. Same range of addresses. But the default gateway pointed to his laptop instead of the real gateway. The DNS servers were his as well.

Every packet the attendee sent went to his machine first. He forwarded it to the real gateway, so the connection worked normally. The attendee noticed nothing unusual—maybe the WiFi seemed slightly slow, but that's conference WiFi.

But he saw everything. Unencrypted web traffic. HTTP form submissions with usernames and passwords. Email synchronization with IMAP servers that didn't require TLS. Session cookies. API keys transmitted over HTTP during development demos. The attendees scrolled through slide decks and checked Slack. He watched their sessions populate his packet capture logs.

Rogue DHCP attacks aren't theoretical. They've been documented in hotels, airports, conferences, and coworking spaces.[183] Security researchers demonstrate them at DefCon and Black Hat, showing how a single laptop can compromise dozens of connections in minutes. The tools are free. The tutorials are on YouTube. A motivated teenager could pull it off in an afternoon.

The contrast with Blast-RADIUS could not be sharper. Blast-RADIUS required world-class cryptographers and months of mathematical research. Rogue DHCP required presence and patience. Blast-RADIUS demanded GPU clusters costing thousands of dollars. Rogue DHCP ran on any laptop. Blast-RADIUS exploited subtle flaws in hash functions that took decades to discover. Rogue DHCP exploited a design decision obvious since 1993: the first response wins, and anyone can respond.

The attacks work because they're trivially easy—a script can automate the entire process, and the victim's computer cooperates fully, accepting whatever configuration it receives. No passwords need to be cracked, no software flaws exploited. The system behaves exactly as specified.

Variations exist for different goals. Someone interested in surveillance routes traffic through their machine and captures it. Someone seeking credentials configures a malicious DNS server that redirects banking sites to phishing pages. Others push a proxy configuration that injects JavaScript into web pages.

The attacks combine well with techniques from other chapters. ARP spoofing (Chapter 5) lets the attacker intercept traffic without being the gateway. DNS hijacking (Chapter 7) lets them redirect specific domains. Together with rogue DHCP, attackers can control virtually every aspect of a victim's network experience.

Enterprise networks have defenses against these attacks. Consumer networks don't.

DHCP snooping is a feature available on managed network switches. When enabled, the switch tracks which ports are authorized to send DHCP server responses. Traffic from unauthorized ports is dropped. A rogue server connected to a regular network jack simply doesn't work; its responses never reach the clients.

The switch builds its knowledge from observing legitimate DHCP traffic. When a client completes a DORA exchange with an authorized server, the switch records the binding: this MAC address has this IP address, assigned by this server. The switch then enforces the binding. ARP responses that don't match the binding are blocked. IP traffic from unexpected addresses is rejected.

This defense is effective—a corporate network with DHCP snooping enabled stops rogue server attacks cold, because the attacker's laptop can run whatever DHCP server it wants; the switch simply won't forward the packets.

But DHCP snooping requires managed switches, which cost more than consumer equipment. It demands configuration expertise. Most of all, it requires thinking about security at the network layer—and most organizations don't.

Guest WiFi networks rarely have protection. The coffee shop router cost fifty dollars. It doesn't support DHCP snooping. The hotel access points serve hundreds of guests per day; adding security would require infrastructure that doesn't exist. The conference center has a single SSID for five thousand attendees, and the IT staff is busy enough keeping the connection working at all.

802.1X authentication offers stronger protection. Before a device can communicate, it must authenticate—usually with a username and password or a certificate—and the network won't accept traffic from unknown devices. A rogue DHCP server can't attack if it can't join the network in the first place.

Enterprise environments sometimes deploy 802.1X, but consumer environments never do. The configuration burden is too high, the user experience too complex. The coffee shop just wants the WiFi to work.

This gap between enterprise protection and consumer vulnerability grows wider each year. Corporate networks deploy defense in depth: DHCP snooping, 802.1X, network segmentation, monitoring. Public networks remain undefended, relying on the good behavior of their users.

The hotel eventually added a captive portal. Attendees had to accept terms of service before getting an address. This didn't stop DHCP attacks; the portal ran after address assignment, not before. But it made guests feel safer, which is perhaps what security means in environments where real security isn't economically viable.

Ralph Droms, writing RFC 2131 in 1997, anticipated that "networks may contain multiple, non-cooperating DHCP servers."[181] He meant server redundancy, not adversarial servers. The specification includes no mechanism for distinguishing authorized servers from unauthorized ones. The client takes whatever comes.

Nearly three decades later, in airports and conference centers around the world, the fastest reply still wins. The mechanism that solved Karen Chen's spreadsheet problem created a different problem that remains unsolved on every public network.

The internet was built for networks where physical presence implied legitimacy. You were on the wire because you belonged there. You configured your neighbors because your neighbors expected you to. The premise was reasonable in 1993, when DHCP emerged, and in 1997, when it was standardized.

The premise was obsolete by 2005, when public WiFi became common, and it's been obsolete for twenty years now. The specification hasn't changed—but the networks have.

Every time you connect to public WiFi, you broadcast a request to any server listening. You accept whatever configuration comes back first. You assume the reply is legitimate, because DHCP cannot verify. The conference badge comes with a side of blind faith.

Usually, that faith is justified—the hotel's access point responds, you get a working connection, and you browse the web and check your email and do whatever you came to do. The system works as designed.

Sometimes, there's a laptop in the lounge, responding faster than the legitimate server. The owner looks like any other attendee, working through the conference sessions, coffee cup beside their keyboard. Their Yersinia session runs in a minimized window. Their packet capture fills with your traffic.

You never know which connection is which—that's the point. DHCP cannot distinguish them. The fastest reply wins, and you accept what you're given.

And still, it runs.

Billions of devices wake each morning and shout into the void: "I need an address." Billions of servers answer. The exchange takes milliseconds. The system works because, on most networks, the first response comes from someone who means well.

The design that made Droms's protocol so easy to deploy also made it impossible to secure without wrapping it in something else—802.1X, DHCP snooping, enterprise monitoring. DHCP itself remains what it always was: a handshake between strangers who assume good intent.

Karen Chen's spreadsheet is gone, and the laptops configure themselves—the pool refreshes, the leases renew. And every so often, in a conference lounge or an airport terminal, someone's laptop responds a little faster than the legitimate server.

The protocol cannot tell the difference. It was never designed to.

DHCP handed out addresses on faith. But addresses were only the beginning. Every day, on every network, people typed passwords into remote sessions that crossed the wire unprotected. The protocols that carried those keystrokes had been built in the same era of trust—and one researcher in Finland had seen enough.

CHAPTER 16

THE SECURE SHELL

SSH is a protocol for secure remote login
and other secure network services
over an insecure network.

— RFC 4251, January 2006

16.1. THE PASSWORD SNIFFER

The attack happened in spring 1995.[21] Someone had placed a packet sniffer on Helsinki University of Technology's network, capturing everything—login sessions, file transfers, email, every password typed into every remote connection.

Tatu Ylönen, a cryptography researcher there, watched IT staff discover thousands of compromised passwords. The attacker had been watching for weeks, and every `Telnet` login had crossed the wire in plain text. The sniffer caught them all.

Ylönen was furious. Not at the attacker, but at the protocols. `Telnet`, `rlogin`, `rsh`—all the tools researchers used daily sent passwords with-

out protection. Everyone knew Telnet was insecure. They used it anyway because nothing better existed.

Ylönen decided to build something better.

He started writing code that summer. The concept was straightforward: take everything Telnet did—remote login, command execution, terminal emulation—and wrap it in encryption. Strong encryption. Every keystroke would be scrambled before it left your machine, every response scrambled before it came back. An attacker watching the network would see nothing but random noise.

He called it Secure Shell, or SSH. The name was deliberate, evoking the familiar shell prompt, the $ or # that greeted you on Unix systems. But this shell was secure.

The first version appeared in July 1995.[184] Ylönen released it as free software, and word spread through the academic community. Researchers at universities worldwide were dealing with the same problem Helsinki had faced. Password sniffers were appearing on networks everywhere. SSH was the answer they needed.

Adoption was swift. Within a year, tens of thousands of systems were running SSH. Ylönen started getting inquiries from companies who wanted to use it commercially. In December 1995, he founded SSH Communications Security to sell licenses and support.[185]

The commercialization created tension. Early versions of SSH had been free, but Ylönen was now charging for commercial use. The source code remained available, but the licensing terms grew restrictive. Universities and open-source projects could still use SSH without paying, but companies faced fees that some found unreasonable.

By 1999, the situation had become untenable for the open-source community. The last freely usable version was SSH 1.2.12, released in 1999. After that, the licensing changed in ways that made many developers uncomfortable.

Theo de Raadt was building OpenBSD at the time, a fork of NetBSD focused obsessively on security. His team audited every line of code in the system, looking for vulnerabilities. They needed SSH—a secure operating system without secure remote access was absurd—but the licensing made the current version unusable.

De Raadt and his team made a decision: they would fork the old free version and maintain it themselves. They called it OpenSSH.[186]

On December 1, 1999, De Raadt announced the fork on the `openbsd-misc` mailing list. His message was characteristically blunt: "The goal of the OpenSSH project is to keep the spirit of the original free SSH release alive." He promised a version that would be "free for everyone to use," unencumbered by restrictive licenses or patent claims.

The OpenSSH project launched in late 1999. The team started with SSH 1.2.12, ripped out everything they didn't need, audited what remained, and started adding features—rewriting large sections to match OpenBSD's coding standards and fixing bugs that SSH Communications Security had left unfixed.

Within months, OpenSSH supported the SSH-2 protocol, a cleaner redesign that Ylönen's company had developed. The new protocol fixed cryptographic weaknesses in SSH-1 and added features like improved key exchange and channel multiplexing.

The portable version of OpenSSH—adapted to run on operating systems beyond OpenBSD—appeared in late 1999. By 2000, major Linux distributions were shipping OpenSSH as their default remote access tool. Telnet retreated to legacy systems and specialized applications.

Today, OpenSSH runs on almost every Unix and Linux server worldwide. It's the standard way to administer cloud servers, network equipment, and embedded systems. The code that started as one researcher's response to a password-sniffing attack became the infrastructure that keeps the modern internet running.

16.2. KEYS AND HANDSHAKES

16.2.1. HOW SSH WORKS

Imagine you're making a phone call, but you're worried someone might be listening. You want to speak freely, but the line could be tapped. Worse: you're not entirely sure you dialed the right number.

SSH is the cryptographic equivalent of a secure phone line. Before you start talking, you and the other party establish a shared secret that only you two know. Then everything you say gets scrambled using that secret. Anyone listening hears only static.

But that phone analogy has a gap. When you dial, you trust the phone system to connect you to the right person. SSH has no phone company. It has to solve a harder problem: prove you're talking to who you think you're talking to, without any central authority to vouch for either side.

Three things must happen before you can type commands: encrypt the wire so eavesdroppers hear static, verify the server is who you think it is, and prove your own identity. Encryption is handled cleanly. The second part is where trouble hides.

When you type `ssh server.example.com`, your client opens a connection to port 22 on that server.[22] The first thing that happens is key exchange: both sides generate a shared secret without ever sending that secret over the wire.

The problem is that neither side can safely send the secret over the wire—and yet both sides need to agree on one.

The mathematics are spare and clever. Diffie-Hellman key exchange lets two parties compute an identical secret even though an

eavesdropper sees every message they exchange. Think of mixing paint colors. Alice and Bob each have a secret color, and they mix their secret with a shared public color before exchanging results. Each then mixes in their own secret again, arriving at the same final color. Anyone watching sees only the intermediate mixtures, never the secrets.

After a few messages back and forth, both sides have a key that nobody else knows. From that point on, every byte traveling between client and server is encrypted. Commands typed, output received, even the protocol control messages—all scrambled. An attacker watching the traffic sees the length of encrypted blocks and the timing of packets. Nothing else.

That's the encryption problem, solved. Now comes the harder question.

16.2.2. THE LEAP OF FAITH

Encryption protects the conversation. But it doesn't tell you who you're talking to.

When an SSH server starts for the first time, it generates a key pair: a private key that never leaves the server, and a public key that anyone can see. This is the server's host key: its identity, its cryptographic fingerprint.

When a client connects, the server sends its public host key. The client has a decision to make: is this the right server?

If the client has connected before, the answer is easy. SSH keeps a file called `known_hosts` that records every server visited and its host key. If the key matches, the client is talking to the same server as last time. If the key doesn't match, something is wrong—either the server's key changed, or someone is impersonating the server.

But what about the first connection? The server sends a host key the client has never seen. Is it legitimate?

Researchers call it "Trust On First Use," often abbreviated TOFU. The client accepts the server's identity on first contact, then alerts if it ever changes. We cannot verify that initial identity, so we take the server's word for it. If an attacker has positioned themselves between us and the real server, they could present their own host key. We'd accept it. Every subsequent connection would go through them.

SSH handles this with a prompt:

```
The authenticity of host 'server.example.com' can't be
    established.
ED25519 key fingerprint is SHA256:xvSQ2pNmG4f...
Are you sure you want to continue connecting (yes/no)?
```

Most users type "yes" without thinking. It's the only way to proceed, and who has time to manually verify key fingerprints?

TOFU. A calculated gamble that becomes permanent.

TOFU is a compromise—better than no verification at all, but weaker than a certificate authority system like TLS uses. And if the initial handshake was intercepted, all subsequent connections defer to the attacker's key.

Once you accept a key, it goes into `known_hosts`. Future connections check against that stored key. If the server's key ever changes, SSH refuses to connect and prints a stern warning:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
```

The warning is intentionally alarming because a changed host key is serious. Either the server was rebuilt and got a new key, or someone is attempting an interception attack. The user has to manually remove the old key before SSH will connect again.

16.2.3. PROVING WHO YOU ARE

The encrypted channel is established, the remote host's identity confirmed—or at least, accepted on first use. Now you need to prove who you are. The authentication method matters here: if you've connected to the wrong server, what you hand over depends on which method you used.

SSH supports several authentication methods.[187] The simplest is password authentication: you type your password, SSH encrypts it, and the remote host checks it against its user database. Better than Telnet—the password travels encrypted rather than in plain text. But if you're talking to an attacker's machine, you just handed them your password. Encrypted delivery to the wrong recipient.

The stronger approach is public key authentication. You generate your own key pair: a private key that stays on your machine, protected by a passphrase, and a public key that you copy to the server. When you connect, the server challenges you to prove you have the private key—

your client signs a challenge message, the server verifies the signature, and if it checks out, you're in.

The private key never crosses the network, so even if someone captures every packet, they can't extract your key. They'd have to steal the key file from your machine directly, and if you protected it with a passphrase, they'd need that too.

This is stronger than passwords—but only if you're talking to the right server. Public key authentication proves you are who you claim to be, but it doesn't prove anything about who you're talking to. That's still TOFU's job.

SSH does more than remote shells—a single connection can carry multiple channels for shell sessions, file transfers, and port forwarding.[188] This power cuts both ways.

An SSH tunnel through a compromised server can reach internal networks that firewalls would otherwise protect. The traffic is encrypted, so network monitoring tools see only that an SSH connection exists—what travels inside remains hidden. The same encryption that protects legitimate users protects attackers too.

16.2.4. THE BOOTSTRAPPING PROBLEM

SSH's weakness differs from DHCP's "first response" race. DHCP fails when an attacker responds faster than the legitimate server—a matter of milliseconds and network topology. SSH isn't about speed. It's about establishing identity without prior knowledge.

SSH does have verification—the `known_hosts` file provides exactly that—but verification requires a reference point, and obtaining that reference point requires relying on something: the initial handshake, an out-of-band exchange of fingerprints, or a certificate authority.

The bootstrapping problem doesn't disappear—it just moves.

RFC 4251, published in January 2006, describes the SSH architecture[189]:

> RFC 4251 --- January 2006 --- SSH Architecture
>
> Each server host SHOULD have a host key. Hosts MAY have multiple host keys using multiple different algorithms. The server host key is used during key exchange to verify that the client is really talking to the correct server.

The word "verify" is doing heavy lifting there. Verification happens against a stored value, and that stored value was accepted on initial contact.

16.3. WHEN RANDOMNESS FAILS

On May 13, 2008, the Debian security team sent an advisory that began with measured language and ended with catastrophe.[190]

The story began two years earlier, in April 2006. A Debian developer named Kurt Roeckx was cleaning up code warnings. He was running Valgrind, a tool that detects memory errors, on OpenSSL. Valgrind complained about the use of uninitialized memory in the random number generator.

Roeckx did what diligent developers do: he investigated, found the lines of code triggering the warning, and posted a query to the `openssl-dev` mailing list. "It's using uninitialized memory here," he wrote. "Is this intentional?" The responses were ambiguous, so Roeckx

decided to fix it by commenting out the lines that read the uninitialized memory.

The fix was a disaster. OpenSSL's random number generator gathered entropy—genuine randomness—from various sources on the system. The lines Roeckx removed were critical: they intentionally read uninitialized memory to add unpredictability to the entropy pool. From Valgrind's perspective, reading uninitialized memory is a bug. From OpenSSL's perspective, it was a feature.

By removing those lines, Roeckx accidentally crippled the generator. Without them, the only input the PRNG received was the process ID of the running program.

Process IDs on Linux are small numbers, ranging from 1 to 32,768 by default, so every key generated by the broken OpenSSL had to come from one of about 32,767 possible states.

For two years, from September 2006 to May 2008, every Debian and Ubuntu system with the broken OpenSSL generated predictable keys. SSH keys. SSL certificates. Anything that depended on OpenSSL's random number generator was affected.

The impact was staggering. Millions of SSH key pairs were compromised. Security researchers generated lists of all possible weak keys, and anyone could download them. Check whether a server's host key was vulnerable. Try all possible user keys against an SSH server until one worked.

System administrators worldwide scrambled to regenerate every key on every affected system. The fix itself was trivial—update OpenSSL and generate new keys—but discovering which keys were affected, replacing them all, and updating every `known_hosts` file and `authorized_keys` file took weeks of work.

The Debian OpenSSL bug (CVE-2008-0166)[191] was introduced on September 17, 2006 and discovered on May 13, 2008[190]—nearly two years undetected. A single line of code removed from the random num-

ber generator reduced all keys generated on Debian/Ubuntu systems to approximately 32,767 possibilities. The only remediation: regenerate all keys on affected systems.

The Debian bug became a case study in how one well-intentioned fix can break everything downstream. Roeckx wasn't malicious—he saw what looked like a bug, fixed it, and even contacted the OpenSSL developers before committing the change. But the code was subtle, and the consequences of removing that line weren't obvious until the damage was done.

16.3.1. THE GITHUB EMERGENCY

Fifteen years later, SSH key management made headlines again.

On March 24, 2023, GitHub posted a security advisory.[192] Their RSA SSH host key's private component had been "briefly exposed" in a public repository—someone at GitHub had accidentally committed a file containing the private key. The exposure was brief and the file was removed quickly, but the damage was done: anyone who saw that file could impersonate GitHub's SSH servers.

GitHub rotated their RSA host key immediately, which meant every developer who had ever connected to GitHub over SSH would now see a host key mismatch warning. Millions of developers.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
    @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Host key for github.com has changed and you have requested
strict checking.
```

That wall of at-signs is designed to stop users cold. But keys do sometimes need to change legitimately.

GitHub's advisory told users to update their `known_hosts` files: remove the old key, connect again, accept the new key. Simple instructions—but those instructions trained millions of developers to ignore host key warnings. "Oh, that scary warning? Just delete the old key and try again."

The model works because host keys rarely change, and the alert gets its power from being exceptional. But every time a legitimate key rotation happens, users learn that the warning is sometimes just noise, and the next time an attacker attempts an interception attack, they might remember that time GitHub told them to dismiss the alert.

SSH replaced the exposed wire with an encrypted tunnel.

The passwords are hidden now. The sessions are authenticated. A remote administrator in Helsinki or Tokyo connects to a server in San Francisco and the conversation is private, the identity verified. Ylönen built this in a summer because the alternative was plaintext flowing over the wire for anyone to read—and he had just watched someone read it. The fix was real. It worked.

But every first connection ends with a prompt: "Are you sure you want to continue connecting (yes/no)?" Most users type "yes" without reading the fingerprint. They have no reference to check it against. They are accepting a key on faith, the way everyone accepts everything on first use.

The faith is better founded now, backed by mathematics, protected by encryption. But the Debian bug showed that the mathematics depend on their implementation, and a two-year flaw in a random number generator can hollow out every key generated during that

window. GitHub showed that key rotation at scale teaches users to dismiss the warning. In July 2024, Qualys researchers found a race condition in OpenSSH that had been fixed in 2006 and reintroduced in 2020.[193] Old bugs return. The vigil never ends.

The wire is encrypted. The faith is still there.

SSH encrypted the remote shell. It replaced Telnet's cleartext with cryptography that actually worked. But encryption solved only one kind of exposure. When voice calls moved from dedicated telephone lines to the same internet that carried encrypted data, they inherited all of the network's implicit expectations. The system that made voice over IP possible accepted caller identity on faith alone.

CHAPTER 17

THE CALLER ID

SIP makes use of elements called proxy servers
to help route requests to the user's current location,
authenticate and authorize users for services,
implement provider call-routing policies,
and provide features to users.

— *RFC 3261, June 2002*

17.1. MAKE IT LOOK LIKE HTTP

Henning Schulzrinne wanted to make a phone call over the internet.

Not through copper wires and circuit switches that cost dollars per minute. Over the internet, where packets were cheap and the infrastructure already existed. It was 1996, and Schulzrinne was a computer science professor at Columbia University with a vision for internet telephony.[194]

H.323 stood in the way.

The International Telecommunication Union had published H.323 in November 1996 as the standard for voice and video over packet

networks.[195] The protocol worked, and Intel and Microsoft were building products around it. But H.323 carried the telephone industry's genetic inheritance: binary encoding, complex state machines, committee-designed procedures. Reading an H.323 message required a decoder, and debugging required specialized tools. The protocol felt like a grudging concession that the internet existed.

Schulzrinne thought he could do better. He had been collaborating with the Multiparty Multimedia Session Control working group at the IETF, developing protocols for internet conferencing. The group had created SDP, the Session Description Protocol, which described the technical details of a multimedia session—codecs, ports, addresses—like a handwritten note specifying where to meet and what language to speak. They had created RTSP for streaming media. The pieces were in place. What was missing was the signaling layer—the part that would set up a call.

He started sketching something simpler.

Mark Handley was thinking along the same lines. A researcher at University College London, Handley had been working on the MBONE, the experimental multicast backbone that researchers used to broadcast conferences and lectures across the internet. He understood the challenges of real-time communication over packet networks. He also understood that the telecom industry's approach—careful, standardized, backward-compatible—was too slow for the internet.

Schulzrinne and Handley began collaborating in late 1996. They exchanged emails, shared drafts, argued over details. Jonathan Rosenberg, a researcher at Bell Labs, joined them. Eve Schooler at Caltech contributed ideas. The group worked outside the formal IETF process at first, developing something lightweight enough to implement quickly.

Their design principle was radical: make it look like HTTP.

HTTP delivered web pages—text-based, human-readable, stateless. A web browser sent a request with a few headers, and a server sent a response. The interaction was simple enough that you could debug it with a telnet session, typing commands by hand and reading the responses on screen. HTTP had scaled to millions of servers, proving that text protocols could be fast enough for real applications.

SIP would follow the same pattern. A phone call would start with an INVITE request, formatted almost identically to an HTTP request. The server would respond with status codes borrowed directly from HTTP: 200 OK meant the callee answered, 404 Not Found meant the number didn't exist, 180 Ringing meant the phone was ringing. Headers would carry the metadata—who was calling, who was being called, where to send the audio. A human could read the messages. A human could understand what was happening.

The email metaphor went further. SIP addresses looked like email addresses: `sip:alice@example.com`. Users could register their current location with a server, the way email forwarding worked. A call to `sip:alice@example.com` would route to wherever Alice had registered, whether that was her office phone, her mobile, or her laptop at an airport. Personal mobility, the protocol designers called it. Your identity followed you across devices.

RFC 2543 appeared in March 1999. The authors were Handley, Schulzrinne, Schooler, and Rosenberg. The abstract was modest: "The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, modifying and terminating sessions with one or more participants."[196]

The document ran to 153 pages, but the core mechanism was simpler than it looked. INVITE started a session, BYE ended it, REGISTER told a server where to reach you, and OPTIONS asked what capabilities a server supported. The rest was details.

The details included a header called From.

The From header identified the caller: when Alice called Bob, her phone would include this header with her SIP address, and Bob's phone would display the information. Caller ID, internet-style.

The RFC specified what the From header should contain. It did not specify how to verify it. Section 6.21 explained that this field "MUST be present in all requests and responses" and "indicates the initiator of the request." Nothing required that this indication be truthful.

The design made sense in context. Email followed the same pattern—the From line was whatever the sender claimed, since SMTP didn't verify sender addresses either. Both systems relied on social accountability, legal liability, and the desire to receive replies, expecting the network infrastructure to be operated by responsible parties.

The expectation held for email until spam overwhelmed it. It would hold for SIP until voice became free.

RFC 3261 replaced RFC 2543 in June 2002.[197] The new version, authored by Rosenberg, Schulzrinne, and six collaborators, ran to 269 pages and represented three years of implementation experience. It added security sections, clarified ambiguities, and tightened requirements throughout.

The security considerations section acknowledged the problem. "SIP is not an easy protocol to secure," the authors wrote. They documented the threats: registration hijacking, server impersonation, message tampering, denial of service. They described countermeasures: TLS for transport security, digest authentication for users, S/MIME for message encryption.

Schulzrinne and his colleagues understood the tradeoff. Adding cryptographic verification would have required a public key infrastructure for phone numbers, a certificate authority that could attest that Alice really controlled `sip:alice@example.com`. Such infrastructure didn't exist in 2002. Building it would have delayed deployment

by years. The alternative was to ship a working protocol and let the security infrastructure catch up.

They shipped the protocol. The infrastructure is still catching up.

17.2. THE FROM HEADER PROBLEM

17.2.1. HOW SIP WORKS

Making a phone call used to require a physical path. When you dialed a number, switches at the telephone company closed relays, establishing a circuit from your phone to the recipient's. The circuit existed for the duration of the call, dedicating copper and equipment to your conversation. This was circuit switching, and it worked for a century.

The phone company knew who you were because you had an account and a physical line to your house. If you misbehaved, they could disconnect you.

SIP replaces circuits with messages.

A SIP call isn't a dedicated connection but a negotiation—two phones agreeing on how to exchange audio. The negotiation happens through text messages that look like web requests, while the audio travels separately through a different protocol entirely: SIP handles the signaling, and RTP, the Real-time Transport Protocol, handles the media.[198] This split is the first thing to understand.

The second thing to understand: SIP inherited the web's openness. If you can send the message, you can claim any identity you want.

Alice wants to call Bob. Her phone constructs an INVITE message:

```
INVITE sip:bob@example.com SIP/2.0
Via: SIP/2.0/UDP alice-phone.example.com:5060
From: Alice <sip:alice@example.com>;tag=1234
To: Bob <sip:bob@example.com>
Call-ID: xyz123@alice-phone.example.com
CSeq: 1 INVITE
Contact: <sip:alice@192.168.1.100:5060>
Content-Type: application/sdp
Content-Length: 142
```

Look at the From field: `From: Alice <sip:alice@example.com>`.

Who verified that this was actually Alice? Nobody—her phone wrote that header, and if she wanted to write `From: Your Bank <sip:security@bigbank.com>`, nothing in SIP would stop her.

The other headers mattered for routing: Via told servers where to send responses, To identified the callee, and Call-ID tracked this particular call. Below the headers, separated by a blank line, is the body—SDP, the Session Description Protocol[199]—describing where Alice wants to receive audio by offering an IP address, a port, and a codec.

The message travels to a proxy server, which looks up Bob's registered location and forwards the INVITE. Bob's phone receives it, starts ringing, and sends back `180 Ringing`. When Bob answers, his phone sends `200 OK`.

The 200 OK includes Bob's SDP answer, specifying where he'll receive audio. Alice's phone acknowledges with an ACK. The signaling is complete. Now the phones start sending RTP packets directly to each other, carrying the actual voice data.

Here's the architecture: signaling and media travel different paths. The INVITE might traverse multiple proxy servers while the audio packets travel directly between endpoints—efficient for bandwidth, but neither path verifies who you're really talking to.

This is how Caller ID works in the VoIP world. When your phone rings and displays a number, that number comes from SIP headers. If the caller spoofed those headers, you see the spoofed number. The phone company might add their own headers, might verify that the caller has the right to use that number, might mark the call as unverified. Or they might pass it through unchanged.

The verification chain breaks at the first dishonest party.

In the traditional phone network, Caller ID was difficult to spoof because access to the signaling network was controlled. You needed equipment from the phone company. You needed a physical connection. The carrier could revoke your service if you abused it.

VoIP eliminated those barriers.

SIP software was free. SIP providers competed on price, and the cheapest providers asked the fewest questions. Anyone with an internet connection could send INVITE messages claiming any identity they chose. The cost was negligible. The anonymity was nearly complete.

17.2.2. WHERE VERIFICATION FAILS

The sender field isn't the only weak point.

SIP addresses work like email addresses, but with a twist: Bob might have multiple devices—a desk phone, a mobile app, a laptop client. Each device registers with Bob's SIP domain, announcing its current address, and the registrar stores these bindings to forward incoming calls.

What happens if someone else registers as Bob?

If an attacker guesses Bob's password, or compromises his account, or spoofs the registration message, they can redirect his calls to themselves. The caller thinks they're reaching Bob. They're reaching the attacker. Registration hijacking is invisible to the caller and often in-

visible to Bob, who simply stops receiving calls he doesn't know were made.

The media path has its own vulnerabilities.

RTP packets flow directly between endpoints, carrying the actual voice—each packet contains a timestamp, a sequence number, and compressed audio. The receiving phone trusts without checking that these packets come from the person it thinks it's talking to.

Anyone who modifies the SDP in transit can redirect the media—a caller thinks they're talking to their bank, but the audio goes to an attacker who relays it onward while listening to both sides. An interception attack at the media layer, invisible to the signaling.

SRTP, Secure RTP, encrypts the media path—if both endpoints support it and if they negotiate keys successfully.[200]

Many VoIP calls still don't use SRTP, which means calls to doctors, lawyers, and accountants may be traveling across the internet unencrypted.

17.2.3. THE ECONOMICS OF FRAUD

Traditional phone calls cost money—long-distance calls cost more, and international calls cost even more. This expense created friction, so a scam call from overseas needed to be profitable enough to justify the cost.

VoIP removed the friction.

Calls that traverse the internet cost fractions of a cent, and VoIP providers with minimal vetting offer bulk rates with no verification. Automated dialers can place thousands of calls per hour, shifting the economics from "calls are expensive" to "calls are free."

Free calls with spoofed Caller ID enabled a new industry.

Robocalls pitch fake insurance, threaten IRS audits, and impersonate Microsoft support, while the Caller ID shows a local number, or

your bank's number, or a government agency. You answer because the number looks legitimate—and that's the point.

Americans received an estimated 50 billion robocalls in 2021.[201] Most displayed spoofed Caller ID.

The From header that Schulzrinne and Handley created for a research network had become a vector for fraud at industrial scale.

17.3. THE 911 CALL

The 911 call came in at 2:47 AM on December 28, 2017.

A man was on the line, speaking urgently. He was at a house in Wichita, Kansas, on West McCormick Street. He had just shot his father. He was holding his mother and brother hostage. He had poured gasoline around the house. If police approached, he would set it on fire.

The Wichita Police Department dispatched officers, a SWAT team, and negotiators. They surrounded the house, established a perimeter, and prepared for an armed standoff with a killer who had nothing to lose.

Andrew Finch opened his front door.

Finch was 28 years old, unarmed, and had no idea why police were pointing weapons at him. He lived in the house with his mother and niece—no one had been shot, no hostages, no gasoline. The 911 call was a lie.

An officer fired. Finch died on his porch.

The practice is called SWATting. The name comes from SWAT teams, the heavily armed police units deployed for dangerous situations. The technique is simple: call 911 with a fabricated emergency,

give the victim's address, and watch the police respond with overwhelming force.

SWATting requires two things: the victim's address, and the means to make the call appear legitimate. VoIP provides the second requirement.

The call that sent police to Andrew Finch's door came from Tyler Barriss, a 25-year-old in Los Angeles with a history of making fake bomb threats.[202] Barriss used a VoIP service to place the call. He spoofed the Caller ID to show a Wichita number. The 911 operator saw a local call from someone who said he was at the scene. Nothing in the technology indicated otherwise.

The address Barriss used wasn't random. He was trying to target someone else—a man named Shane Gaskill who had gotten into an argument with Casey Viner over a $1.50 bet in the video game Call of Duty. Viner gave Barriss an address for Gaskill. The address was wrong. Andrew Finch had no connection to either man. He died because of a video game dispute, a wrong address, and a signaling system that didn't verify caller identity.

Barriss pleaded guilty to 51 federal charges and was sentenced to 20 years in prison.[202] Viner pleaded guilty to conspiracy and received 15 months.[203] But criminal prosecution, as always, came after the harm.

SWATting emerged as VoIP services became cheap and caller ID spoofing became trivial. The FBI's first federal swatting case was prosecuted in 2007, though earlier incidents had occurred.[204] Hackers targeted security researchers, journalists, and celebrities, while gamers targeted other gamers—and the calls were specific and detailed enough to trigger armed responses.

The victims included Brian Krebs, a security journalist who had written about hackers and faced repeated SWATting attempts.[205] They included online streamers whose addresses leaked through careless

social media posts, and random families whose addresses appeared in databases sold on underground forums.

Each SWATting call exploited the same gap: 911 operators relied on Caller ID. When the screen showed a local number, dispatchers believed the call originated locally, and when the caller described an emergency, they responded to it. The system was designed to save lives, and it expected that callers who provided detailed information were telling the truth.

The expectation was reasonable for landlines. It was catastrophically wrong for VoIP.

The response came slowly.

Congress passed the Truth in Caller ID Act in 2010, making it illegal to transmit misleading Caller ID information with intent to defraud.[206] The law had limits. Enforcement was difficult when callers operated from overseas. Many spoofed calls weren't fraudulent in the legal sense—debt collectors, political campaigns, and businesses routinely spoofed numbers for convenience.

The phone industry developed a technical solution. `STIR`, Secure Telephone Identity Revisited, uses digital signatures to attest that a caller has the right to use a particular number.[207] Its companion standard `SHAKEN`, Signature-based Handling of Asserted information using toKENs, defines how carriers should transmit and verify these signatures, creating a chain of trust from the originating carrier to the terminating carrier.

The FCC mandated `STIR/SHAKEN` implementation by June 30, 2021 for large carriers, with smaller carriers following later.[208] The rollout was uneven. International calls bypassed the system entirely since foreign carriers didn't participate. Many smaller carriers obtained extensions. The attestation had three levels—full, partial, and gateway—and partial provided weaker assurance than the name suggested.

Phones now displayed whether a call had been verified, though the notation varied by carrier—AT&T showed a green checkmark, Verizon showed "Caller Verified." The absence of verification suggested caution, but it didn't prove the call was fraudulent, only that the carrier couldn't verify the identity.

Andrew Finch's death changed something. Before Wichita, SWATting was treated as a prank—dangerous, illegal, but somehow removed from real consequences. After Wichita, prosecutors and judges began treating it as what it was: weaponized deception that could kill.

But the underlying standard hasn't changed. SIP still transmits whatever the sender claims, and the email analogy that Schulzrinne and Handley chose in 1996 still shapes how phone calls work. The premise that participants would be honest, would be accountable, would have reasons not to lie—that premise failed at scale.

Henning Schulzrinne is still at Columbia, still working on the problem.

His protocol carries billions of calls. It also carries the fifty billion robocalls that Americans received in 2021, each one pretending to be someone else. The specification he wrote made voice free and voice anonymous, and the cost of that anonymity shows up in drained bank accounts, in stolen identities, in a man dead on his porch in Wichita. STIR/SHAKEN adds signatures now, and some carriers verify them. International calls bypass the system. The From header is still text that anyone can write.

The calls still connect. The specification didn't fail—it succeeded completely, beyond anything its authors imagined. A protocol designed for researchers at Columbia and UCL now carries voice for the entire planet, and the openness that made it possible is the same openness

that made it dangerous. Schulzrinne knew the tradeoff in 2002. The infrastructure to close it still hasn't caught up.

Voice was not the only thing moving to the internet. So was everything else—documents, commerce, conversation. The web would carry its own assumptions about identity and sessions, and those assumptions would prove just as fragile.

CHAPTER 18

THE STATELESS DANCE

The Hypertext Transfer Protocol (HTTP)
is an application-level protocol with the lightness
and speed necessary for distributed, collaborative,
hypermedia information systems.

— RFC 1945, Tim Berners-Lee, Roy Fielding, and Henrik Frystyk Nielsen, May 1996

18.1. VAGUE BUT EXCITING

In March 1989, Tim Berners-Lee sat at his desk in Building 31 at CERN, staring at a problem that had nothing to do with physics. The particle accelerators worked. The detectors worked. The experiments produced data at staggering rates. But the knowledge was drowning.

Thousands of physicists collaborated there, arriving from universities worldwide, working for months or years, then leaving. When they left, their knowledge left with them. Where did that analysis software end up? Who wrote the detector calibration notes? What happened to the documentation for the experiment that ended in 1986?

The answers existed, scattered across incompatible computers running different operating systems, stored in formats that could not interoperate. CERN had VAX machines and Unix workstations and Macintoshes and IBM mainframes—each an island of computing, isolated from the others.

Berners-Lee picked up a pen. He had an idea, though he was not sure anyone else would understand it.

The document he wrote was titled "Information Management: A Proposal".[209] It ran twelve pages and included hand-drawn diagrams of circles connected by arrows. The circles represented documents, people, concepts. The arrows represented links. He proposed a system where any piece of information could point to any other piece of information, regardless of where it lived or what computer stored it. He called it a "web" of notes with links between them.

His supervisor, Mike Sendall, read the proposal. In the margin, he scrawled two words: "Vague but exciting."

It was neither an approval nor a rejection. The proposal sat on Sendall's desk, and Berners-Lee waited.

The problem wasn't new. Researchers had been dreaming of linked documents since 1945, when Vannevar Bush described a hypothetical "memex" machine.[210] Ted Nelson had coined "hypertext" in 1965 and spent decades on an ambitious system called Xanadu that never shipped.[211] The idea was well established, but no one had made it work simply, at scale, across real networks.

Berners-Lee's insight was to build on what already existed. The internet was there, connecting universities and research labs. DNS was there—the naming system from Chapter 7—providing human-readable names for computers. Email showed that global communication could work. He did not need to invent a new network. He needed a new way to use the network that already existed.

In late 1990, Sendall finally gave Berners-Lee permission to work on the project. Berners-Lee bought a NeXT workstation—one of the sleek black cubes that Steve Jobs had designed after leaving Apple—and began building. Someone taped a hand-written note to the black cube: "This machine is a server. DO NOT POWER IT DOWN!!" The entire World Wide Web, in late 1990, could be killed by an inattentive cleaner with a power strip.

He created three things. HTML, Hypertext Markup Language, a spare way to structure documents with tags like <a> for links and <p> for paragraphs. HTTP, Hypertext Transfer Protocol, a minimal way to request and receive documents. And the URL, Uniform Resource Locator, a clean way to name any document anywhere on the network. Three inventions, all economical, all designed to work together.

HTTP, the protocol, is almost comically minimal. The first version had only one command: GET. A client connects to a server, asks for a document by name, and receives it back. That is the entire transaction. No login, no negotiation, no state. You ask, you receive, and then you are both strangers again.

Berners-Lee would later describe the design philosophy as "simple things first."[212] HTTP was supposed to be so easy that anyone could implement it. A basic server could be written in an afternoon. A basic client could be written by a graduate student learning to program. This was deliberate.

On December 20, 1990, Berners-Lee's colleague Robert Cailliau visited his office and saw the system working for the first time.[213] The NeXT computer was running a program called WorldWideWeb—both a browser and an editor, letting users read documents and create them. Cailliau clicked a link and a new document appeared; he clicked another and another document followed. The links could point anywhere.

On August 6, 1991, Berners-Lee posted a message to a Usenet newsgroup describing his project.[214] The message included an address:

`info.cern.ch`. This was the first public web server, running on a NeXT computer at CERN.

The world's first website described the World Wide Web project itself—a document about how to create documents, containing links to other documents explaining how links worked. The snake ate its own tail.

The reaction was modest: a few dozen people visited the site, and a few of them set up servers of their own. By December 1991, there was a web server in the United States, at the Stanford Linear Accelerator Center. By the end of 1992, there were fifty servers worldwide. It was growing, slowly, the way things grow when no one is paying attention.

Then, in February 1993, a twenty-two-year-old student at the University of Illinois released a program called Mosaic.

Marc Andreessen and Eric Bina had built Mosaic at the National Center for Supercomputing Applications, spending nights and weekends on what they considered a side project.[215] Mosaic was a web browser, like Berners-Lee's WorldWideWeb. But Mosaic did something new: it showed images inline with text. Previous browsers required you to click on an image link and view the picture in a separate window. Mosaic put the pictures right there on the page, mixed with the words. It sounds trivial now. It was not.

Suddenly the web was colorful, visual—it looked like something you might actually want to use. Mosaic was also easier to install than previous browsers, and it ran on Windows and Macintosh, not just Unix workstations. Within months, web traffic began doubling every few weeks.

By January 1994, there were over 600 web servers. By June, there were 2,700. By the end of the year, there were more than 10,000 websites[216], and newspapers were writing stories about something called the "information superhighway." Andreessen left Illinois, moved

to California, and co-founded a company called Netscape.[217] The browser wars had begun.

HTTP, the mechanism that made this possible, was still the same one Berners-Lee had sketched in 1990. Request a document, receive a document. No login required. No encryption. No way to verify that the server you were talking to was actually the server you thought it was. If you asked for a document, you meant to ask. If the server sent a document, it was the right one.

This was a design for physicists sharing research papers at CERN—and it was about to carry the entire world's communication.

18.2. REQUEST, RESPONSE, DONE

18.2.1. HOW HTTP WORKS

Think of HTTP as a librarian at a reference desk. You walk up, hand over a slip of paper with a call number, and the librarian retrieves the book. The transaction is brief and impersonal. The librarian does not remember you from yesterday. The librarian does not know what you read last week or what you plan to read tomorrow. You ask, you receive, and then you are just another patron.

This library, though, has no walls. You hand the librarian your request, and everyone in the building can see what you wrote. The librarian hands you the book, and everyone can see what you received. The transaction is public.

This is HTTP's request-response model. A client—your browser—sends a request to a server. The server sends back a response. Each request is independent. Each response is complete. The server main-

tains no memory of previous interactions. In protocol terminology, HTTP is **stateless**. In security terminology, HTTP is *cleartext*—readable by anyone who can see the traffic.

A request looks something like this:

```
GET /index.html \proto{HTTP/1.1}
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

The first line contains the **method** (GET), the path to the resource (/index.html), and the protocol version. GET means "give me this document." Other methods exist—POST for submitting data, PUT for uploading, DELETE for removal—but GET is what your browser sends every time you click a link.

The lines that follow are **headers**: the Host identifies which website you want, the User-Agent identifies your browser, the Accept header tells the server what content you can handle.

Notice what is missing. No encryption. No signature. No proof that this request came from you or that it has not been modified in transit. The bytes travel over the wire as naked text. Anyone along the path can read them.

The server responds:

```
{HTTP/1.1} 200 OK
Content-Type: text/html
Content-Length: 1234

<!DOCTYPE html>
<html>
...
```

The first line contains the status code. `200 OK` means everything worked. `404 Not Found` means the document does not exist. `500 Internal Server Error` means something broke on the server's end. The status codes are arranged in categories: 2xx for success, 3xx for redirection, 4xx for client errors, 5xx for server errors.

After the status line come more headers, then a blank line, then the actual content: the `HTML` of the webpage, or an image, or whatever you requested.

That completes the transaction: request, response, done. The server has no memory of what just happened, and the next request starts fresh.

And every byte of it—your request, the response, the content—traveled in the clear. If you requested a page while sitting in a cafe, everyone on that WiFi network could have read what you asked for and what you received. `HTTP` assumes you do not mind.

18.2.2. THE STATELESSNESS PROBLEM

Statelessness made `HTTP` fast and scalable. Servers did not need to track thousands of ongoing conversations. Each request was self-contained. Load balancing was easy: any server could handle any request.

But statelessness created a problem. How do you build applications that need to remember things?

Consider a shopping cart. You add an item on one page. You navigate to another page to add a second item. You go to the checkout page. At each step, the server needs to know what is in your cart. But HTTP provides no mechanism for this. Each request is independent. The server cannot tell that the person requesting the checkout page is the same person who added items five minutes ago.

Or consider login. You authenticate on one page, then try to access your account settings—but the server has forgotten you. According to HTTP, the request for account settings comes from a stranger.

The web needed state. HTTP would not provide it.

18.2.3. THE COOKIE SOLUTION

In 1994, a Netscape engineer named Lou Montulli invented the **cookie**.[218] The mechanism was straightforward. When a server wanted to remember something about a client, it would include a special header in its response:

```
Set-Cookie: session=abc123
```

The browser would store this cookie and send it back with every subsequent request to that server:

```
Cookie: session=abc123
```

Now the server could recognize returning visitors. The session ID `abc123` acted like a name tag. Whatever information the server wanted to associate with that session—login status, shopping cart contents,

user preferences—it stored on its end, keyed to that identifier. The cookie just carried the key.

The solution was clever. It grafted state onto a stateless system without changing HTTP itself. Cookies became the foundation of web commerce, social media, advertising, and authentication. Every time you stay logged into a website after closing your browser, a cookie is doing the work.

But the solution had a flaw.

The cookie is a **bearer token**, like a concert wristband. Whoever has it gets access. No questions asked.

If someone steals your cookie, they become you. The server does not verify identity or check who is presenting the cookie—it simply accepts that the cookie arrived through legitimate means.

Your cookie travels over the connection naked, visible to anyone watching. On an open WiFi network, "anyone watching" means everyone.

This design choice made sense in 1994. The web was small. Networks were controlled by universities and corporations. The idea that someone might intercept your cookie seemed remote. But the web was about to grow beyond universities. The cookies would follow.

HTTP's fundamental premise: the network is honest enough. When you request `http://example.com`, HTTP expects that the data you receive actually comes from example.com. It expects no one has intercepted the request or modified the response. It expects the path between your browser and the destination is clean.

RFC 1945, the formal specification for `HTTP/1.0` published in May 1996, is refreshingly candid about this.[219] The security considerations section notes that "the Basic authentication scheme is not a secure method of user authentication, nor does it prevent the Entity-Body from being transmitted in clear text across the physical network." The

RFC does not claim HTTP is secure. It observes that security mechanisms could be added "to increase security."

The authors knew—they always knew. HTTP was designed for a world where security was someone else's problem, to be handled at a different layer of the stack. The specification would move documents. Protecting those documents would require something more.

18.2.4. FASTER, NOT SAFER

The web grew a thousandfold, but the security did not grow with it. HTTP evolved, always in pursuit of speed, not safety.

HTTP/1.0[219] opened a new connection for every request: clean but slow. **HTTP/1.1**[220] kept connections open for reuse. **HTTP/2**[221] multiplexed requests, sending many at once. **HTTP/3**[222] switched to QUIC for even faster connection establishment.

Each version made the web faster. None made HTTP itself more secure.

Through every iteration, the security model stayed the same. The standard moves bytes. Protecting those bytes requires HTTPS: HTTP wrapped in TLS, which we will examine in a later chapter. Plain HTTP remains what it always was: a clear channel through which anyone can listen.

18.3. FACES IN THE SIDEBAR

18.3.1. THE COFFEE SHOP PROBLEM

On October 24, 2010, Eric Butler stood before a crowd at ToorCon, a security conference in San Diego.[223] He was about to release a Firefox extension called Firesheep, and he knew exactly what would happen next.

His motivation was clear. "Websites have a responsibility to protect the people who depend on their services," he wrote in the release post. "They've been ignoring this responsibility for too long, and it's time for everyone to demand a more secure web."

The tool was shockingly direct: install it, open it, wait.

Firesheep put the wireless network adapter into promiscuous mode, listening to all traffic on the local WiFi network. When it saw `HTTP` traffic containing session cookies for popular websites—Facebook, Twitter, Amazon, Flickr—it extracted those cookies and displayed them in a sidebar. Each stolen session appeared as a face: the profile picture of someone you could impersonate.

Click on a face, and Firesheep injected the stolen cookie into your browser. You were logged in as that person. No password required. No hacking required. One click.

Session hijacking was not new. Security researchers had known about it for years. Academic papers described the technique in detail, and anyone with `Wireshark` and some patience could capture cookies from an open WiFi network. The vulnerability was well documented.

But no one had made it easy. No one had built a tool that a non-technical person could use, or made session hijacking as trivial as installing a browser extension. Butler did.

The scenario Butler exploited played out millions of times a day. Someone walked into a coffee shop, connected to free WiFi, checked Facebook, browsed Twitter, maybe bought something on Amazon.

Within weeks, demonstrations were happening everywhere. In November 2010, a software engineer named Gary LosHuertos sat in a Starbucks, running Firesheep. He watched as people around him logged into Facebook. He clicked their faces, "nabbed identities," and posted status updates to their accounts warning them: "Your account is exposed." The abstraction of "session hijacking" had become the reality of strangers controlling your screen.

Each of these actions created network traffic. When users logged in, their browsers sent usernames and passwords to the server. If the connection was encrypted with HTTPS, the credentials were safe. An attacker could not read encrypted traffic.

But in 2010, most websites used HTTPS only for the login page. Once you were logged in, the site would switch back to plain HTTP to save computing resources. HTTPS was considered expensive—it required more server processing power and slowed down page loads. So Facebook and Twitter and most other sites would authenticate you over HTTPS, set a session cookie, and then serve all subsequent pages over HTTP.

The cookie traveled with every request. Unencrypted. Visible to anyone on the same WiFi network.

This is where the coffee shop becomes dangerous. Open WiFi networks do not encrypt traffic between your laptop and the access point. Anyone in range can capture packets. Your unencrypted traffic moves through the air, and every device nearby can receive it.

An attacker does not need to break into anything—they just need to listen, and Firesheep made listening trivial.

18.3.2. THE DEMONSTRATION EFFECT

Butler's stated purpose was not to enable attacks. He wanted to embarrass the technology industry into fixing a problem it had known about for years.

The demonstration worked. Within hours of Firesheep's release, security journalists were testing it in coffee shops, videos appeared showing how easy it was to hijack Facebook accounts, and mainstream news organizations ran alarmed stories about WiFi security. The technology industry scrambled.

Twitter made HTTPS available to all users in January 2011.[224] Facebook added an option for always-on HTTPS in the same month.[225] Gmail, which had already defaulted to HTTPS for webmail, extended it to all Google services. Over the following years, the web underwent a fundamental shift. HTTPS went from expensive option to expected default.

In 2015, a non-profit organization called Let's Encrypt began offering free TLS certificates, removing the last economic barrier to HTTPS adoption.[226] By 2017, more than half of all web traffic was encrypted. By 2020, Google reported that over ninety-five percent of browsing time on Chrome was spent on HTTPS pages—a figure that has since plateaued.[227]

But encrypting the channel does not protect the server. In October 2021, a path traversal vulnerability in Apache—the same HTTP server software that has run since the web's early days—allowed attackers to read any file on affected machines.[228] The first patch introduced a second flaw.[229] Ransomware operators were exploiting both within days. The vulnerability tracking system itself is buckling (Chapter 22). The Apache HTTP Server that Tim Berners-Lee would recognize still runs on millions of machines. Still vulnerable. Still essential.

Firesheep did not reveal anything new. The vulnerability had been documented for a decade. What it did was make the vulnerability

visceral. Watching your neighbor's face appear in a sidebar, one click away from impersonation, was more persuasive than any security white paper.

The web that Berners-Lee imagined was a space of linked documents, knowledge flowing freely between machines. HTTP was the language of that flow—a language without encryption, without verification, without suspicion. He built it to share physics papers at CERN. It moves the world's commerce now.

Today, a layer of encryption wraps most HTTP traffic, transforming it into HTTPS. But the encryption sits on top, and beneath it, HTTP remains what it always was: a request, a response, the conversation continuing billions of times per second. The session cookie is still a bearer token—whoever possesses it possesses the session—and the stateless core is unchanged from what Berners-Lee sketched on a NeXT workstation in 1990. The web grew from fifty servers to billions. The fundamental design did not change with it.

Berners-Lee's supervisor wrote "Vague but exciting" in the margin of a twelve-page proposal. Thirty years later, that proposal carries the world's banking, its medical records, its political speech. Every security fix has been bolted on afterward—cookies, HTTPS, certificate pinning—because the original design assumed the network was honest enough. It was not. The bolts hold anyway.

HTTP carries everything, and most of it is encrypted now. But the encryption is not built into HTTP. It wraps around it—a separate protocol layered on top, negotiated before the first request is sent. That protocol

is TLS, and its story is about what happens when the internet tries to fix a trust problem with mathematics.

CHAPTER 19

THE LOCK ICON

The primary goal of the TLS Protocol
is to provide privacy and data integrity
between two communicating applications.

— *RFC 2246, TLS 1.0, January 1999*

19.1. THE PADLOCK PROBLEM

It's late 1994. Taher Elgamal has a problem.[230]

The Egyptian-born cryptographer sits in Netscape's Mountain View offices, surrounded by the buzz of the technology boom. Netscape Navigator is eating the browser market. Marc Andreessen graces magazine covers. The company is racing toward an IPO that will make history.

But Elgamal isn't thinking about stock options. He's thinking about credit cards.

The web was growing faster than anyone predicted. Millions are typing URLs, clicking through pages, discovering the internet contains

more than email. They want to buy things—books from Amazon, auctions on eBay, flowers, plane tickets, software.

The problem is simple: HTTP sends everything in plain text. When you type your credit card number and click submit, those sixteen digits travel like a postcard—visible to anyone watching. Your ISP can read them, every router can read them, and a student with a packet sniffer can read them.

Commerce online is impossible without privacy. And privacy online does not exist.

Elgamal knows cryptography. He invented the ElGamal signature scheme a decade earlier, based on public-key cryptography.[231]

Public-key cryptography works like a mailbox with two keys. Anyone can drop a letter through the slot—that is the public key—but only the owner can open the box with the private key. You publish your public key to the world. Anyone can use it to encrypt a message to you. Only you, with your private key, can decrypt it.

Before the 1970s, this was impossible. Now it makes secure communication possible between strangers who have never met. Elgamal knows the mathematics exist to solve the commerce problem. What doesn't exist is a practical way to deploy them at internet scale.

He assembles a small team. The goal: build a protocol that wraps HTTP in encryption, invisible to the user except for one detail. A small icon in the browser's status bar—a padlock.

They call it Secure Sockets Layer—SSL for short. The "socket" part refers to the connection between client and server. The "secure" part is the hard work.

```
HTTP Request (without SSL):
GET /checkout.html HTTP/1.0
Host: www.amazon.com
Cookie: session=abc123

Credit-Card: 4111-1111-1111-1111
Expiry: 12/98
CVV: 123
```

Anyone watching the network could read this. Every router, every ISP, every curious graduate student with the right tools.

SSL 1.0 never ships. The team finds security flaws before release—a sign of careful engineering, or a worrying sign of how hard the problem is, depending on your perspective.

SSL 2.0 launched in February 1995 with Netscape Navigator 1.1. It worked—users could connect to secure servers, and the padlock appeared. E-commerce websites started advertising "SSL Secured" to reassure nervous customers. VeriSign, a company spun out of RSA Security—named for cryptographers Rivest, Shamir, and Adleman, who invented the algorithm that made public-key encryption practical—begins selling digital certificates—the credentials that let servers prove their identity.

But SSL 2.0 has problems. Cryptographers find weaknesses in the key exchange. The protocol allows cipher suites so weak they can be broken in hours. An attacker can downgrade connections to pathetically weak encryption without either party knowing.

By late 1996, Netscape released SSL 3.0, a substantial redesign.[232] Paul Kocher, a cryptographer who would later become famous for breaking hardware security, consults on the project.[233] The protocol is

cleaner, stronger, more resistant to the attacks that plagued version 2.0.

But Netscape owns SSL. Every implementation requires their blessing. Microsoft has its own browser war to fight, and depending on a competitor's protocol feels wrong. The IETF steps in with a proposal: take SSL 3.0, change the name, make minor modifications, and publish it as an open standard.

Tim Dierks and Christopher Allen write it up. In January 1999, RFC 2246 defines Transport Layer Security version 1.0—TLS for short.[2][3][4] The differences from SSL 3.0 are minor, but the political difference is everything.

> RFC 2246 --- January 1999
>
> The TLS protocol provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

The lock icon became universal. By the early 2000s, every major browser displays it. Users learn to look for it before entering passwords or payment information. Banks and e-commerce sites advertise their security. "Look for the padlock," they say. "Your connection is secure."

What they don't say is what "secure" means. The padlock promised less than most users believed.

But that confusion is a problem for later. In 1999, the web finally has encryption. Money can flow, and the dot-com boom has the infrastructure it needs.

19.2. THE CRYPTO WARS

19.2.1. THE GOVERNMENT'S OBJECTION

The same year Netscape was building SSL, the U.S. government was trying to kill civilian cryptography.

In April 1993, the Clinton administration announced the Clipper Chip.[235] The proposal was audacious: a government-designed encryption chip that would be installed in telephones and computers. It used a strong algorithm called Skipjack, developed by the NSA. The catch was "key escrow." A copy of every encryption key would be split between two government agencies. Law enforcement with a court order could retrieve both halves and decrypt any Clipper-protected communication.

The administration presented this as a compromise. Citizens would get strong encryption, law enforcement would retain the ability to conduct wiretaps, and everyone would be happy.

Almost no one was happy.

Phil Zimmermann had seen this coming. In 1991, he released Pretty Good Privacy—PGP—a free encryption program anyone could download.[236] PGP used public-key cryptography, the same mathematics that would later power SSL. Zimmermann named it after Ralph's Pretty Good Groceries from the radio show *A Prairie Home Companion*. The understatement was deliberate—the encryption was better than pretty good.

The government investigated Zimmermann for three years. Exporting strong cryptography was illegal under arms control regulations—the same rules that governed missiles and tanks. Zimmermann faced potential prison time. "I think it's ironic," he said when the Electronic Frontier Foundation honored him in 1996, "that the thing I'm being honored for is the same thing that I might be indicted for." The investi-

gation was eventually dropped, but not before Zimmermann became a symbol of the fight for civilian encryption.

The Clipper Chip had its own problems. In 1994, a young researcher at AT&T Bell Labs named Matt Blaze published a paper with a devastating title: "Protocol Failure in the Escrowed Encryption Standard".[237] He had found a way to bypass the key escrow mechanism entirely. The chip's Law Enforcement Access Field—the component that enabled wiretapping—used a 16-bit checksum. Blaze showed that an attacker could brute-force that checksum in about thirty minutes, using the Clipper Chip's own encryption while defeating its surveillance features.

"It's good that Clipper was killed," Blaze later reflected, "and I'm glad that I helped kill it...but it was sorta killed for the wrong reasons. The bug I found wasn't why it was a bad idea." The real problem was the architecture: any key escrow system created a target that attackers would inevitably try to compromise.

A third front opened in the courts. Daniel Bernstein, a mathematics graduate student at UC Berkeley, wanted to publish an encryption algorithm he had developed. The government told him he needed an arms export license. In 1995, Bernstein sued. Four years later, the Ninth Circuit Court of Appeals ruled that source code was protected speech under the First Amendment.[238] "Cryptographers use source code," the court wrote, "to express their scientific ideas in much the same way that mathematicians use equations."

The Crypto Wars ended in a rout. By 2000, the export restrictions had been largely lifted. SSL and its successors could use strong encryption without government backdoors. The lock icon in your browser represents a victory that was far from inevitable.

But the government that funded ARPANET had tried to control the cryptography that would secure it. The same agencies that enabled the internet's creation sought to ensure they could always read its traffic. They lost that battle, but they did not stop trying.

In 2006, the National Institute of Standards and Technology published a random number generator called Dual_EC_DRBG.[239] Security researchers were suspicious from the start—the algorithm was slower than alternatives and had unusual mathematical properties. In 2007, two Microsoft researchers showed that the algorithm could contain a backdoor: anyone who knew certain secret values could predict its output.[240] The secret values, if they existed, would be known only to whoever designed the algorithm. The designer was the NSA.

In 2013, Edward Snowden's disclosures confirmed the suspicions. The NSA had paid RSA Security ten million dollars to make Dual_EC_DRBG the default in their products.[241] The agency that lost the Clipper fight had found another way in.

19.2.2. THE HANDSHAKE

We encrypt messages with a key. We send the key to the recipient so they can decrypt our messages. But if we send the key in plain text, an eavesdropper can read it. Then they can read everything we encrypt.

This is the central paradox of internet cryptography. How do two parties who have never met establish a shared secret without anyone else learning it?

The answer involves clever mathematics, but we don't need the equations. Imagine two people, Alice and Bob, trying to agree on a secret color over a public channel where everyone can see what they're saying.

Each picks a private color—let's say Alice chooses yellow and Bob chooses cyan—colors they can never reveal to anyone.

Now they agree on a public color, red, which everyone knows. Alice mixes her private yellow with this public red to get orange, while Bob mixes his private cyan with the same red to get purple. They exchange these mixed colors publicly.

Here's the trick: Alice takes Bob's purple and mixes in her private yellow. Bob takes Alice's orange and mixes in his private cyan. They both arrive at the same color—a muddy brown that depends on all three inputs.

An eavesdropper sees the red, the orange, and the purple. But mixing colors is easy while unmixing them is impossible. The eavesdropper cannot work backward from orange to yellow, from purple to cyan. The shared secret—that muddy brown—remains hidden.

This is Diffie-Hellman key exchange, invented in 1976[242], using paint as analogy for modular exponentiation. The real mathematics involves very large prime numbers and operations that are easy to perform but practically impossible to reverse.

TLS uses this principle, or variants of it, to establish a session key between your browser and a web server.

But key exchange alone isn't enough. The color-mixing trick proves no one can learn the secret, but it doesn't prove you're mixing colors with the right person.

19.2.3. CERTIFICATES AND CHAINS

Encryption protects the content of a conversation. But what good is encryption if we're talking to the wrong person?

This is the identity problem. When we connect to `bank.com`, we want to talk to the actual bank, not an impostor pretending to be the bank. The connection might be perfectly encrypted, but if the other end is a thief, encryption just means no one else can watch the theft in progress.

TLS solves this with certificates. A certificate is a document that binds an identity (like "bank.com") to a public key. The binding is secured by a digital signature—a cryptographic stamp that proves the certificate came from someone specific and hasn't been altered.

But who signs the certificate, and who vouches for the bank?

This is where certificate authorities enter. A CA is an organization trusted to verify identities and issue certificates. When Bank of America wants an HTTPS website, they go to a CA like DigiCert or Let's Encrypt. The CA verifies that the requester actually controls the domain `bankofamerica.com`. Then the CA issues a certificate signed with the CA's own private key.

Here's the catch: how does your browser know to trust the CA?

The answer is built in. Your browser ships with a list of approved root CAs—about 150 organizations for Firefox, more for Windows. Most of these names you've never heard of. DigiNotar, Comodo, Trustwave—small companies scattered across the globe. Your browser accepts their signatures completely.

When you connect to a secure website, the server sends its certificate. The browser checks: is this certificate signed by a CA on the approved list? Is the certificate valid for this domain? Has it expired? Has it been revoked?

If all checks pass, the padlock appears.

This creates a chain of delegation: the browser accepts the root CA, which vouches for an intermediate CA, which vouches for the website. Each link depends on the one above it.

The problem emerges when a link breaks.

The system has a name: Public Key Infrastructure, or PKI. It's the reason e-commerce works. When we buy something from Amazon, the browser verifies Amazon's certificate, establishes an encrypted connection, and we can enter a credit card with reasonable confidence that only Amazon sees it.

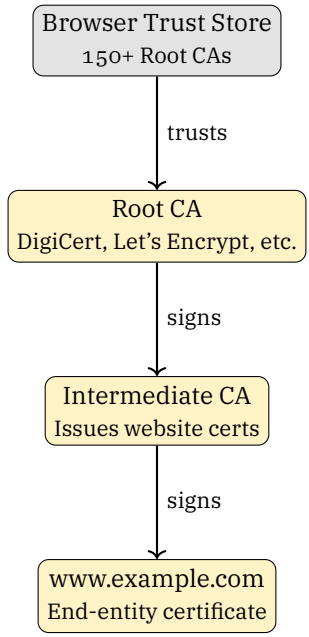


Figure 19.1: The certificate chain. Trust flows from your browser's built-in list of root CAs, through intermediate CAs, down to the website's certificate.

Here's the weakness: every CA in your browser's certificate store can issue a certificate for *any* domain. DigiCert can issue a certificate for `google.com`. So can Let's Encrypt. So can a small CA in the Netherlands that you've never heard of. Your browser will accept any of them.

The chain is only as strong as the weakest CA, and you have no idea which one that is.

So how does this negotiation actually happen?

Your browser reaches out: "I want a secure connection." That's the ClientHello—a message listing the encryption methods your browser supports (the protocol calls this list a **cipher suite**—think of it as a menu of lock types the browser can use), plus a random number to make this handshake unique. The server picks a method from the list, sends

back its own random number, and includes its certificate. That's the ServerHello.

Now comes the moment of truth. Your browser examines the certificate. Is it signed by a CA in the trust list? Is it valid for this domain? Has it expired? Has it been revoked? The browser checks each link in the chain, from the website's certificate up through intermediate CAs to a root it recognizes.

If any check fails, the connection stops. If all checks pass, the browser proceeds—assuming the CA verified the server's identity correctly.

Next, key exchange—the color-mixing step. Client and server each contribute their private colors, mix them with the public values they've exchanged, and arrive at the same shared secret. Anyone watching the traffic sees the mixed colors fly past but cannot unmix them. In milliseconds, two strangers who have never met now share a key that no eavesdropper knows.

Both sides confirm success by sending a message encrypted with the new key. If either can decrypt the other's confirmation, the key exchange worked—the handshake is complete. The lock icon appears.

After the handshake, every message is encrypted with this session key. Your HTTP requests, the server's responses, your passwords, your credit cards—all unreadable to eavesdroppers.

But notice what the browser accepted on faith: that the CA verified the server's identity correctly, that the certificate chain is legitimate, that no one has compromised any CA on the list. One corrupted link, and the whole chain breaks—but your browser would never know.

TLS 1.3, published in RFC 8446 in August 2018, tightened the handshake.[243,244] It cuts the exchange from two round trips to one. It removes legacy algorithms that cryptographers had proven weak. It encrypts more of the handshake itself, so eavesdroppers learn less about the connection being established.

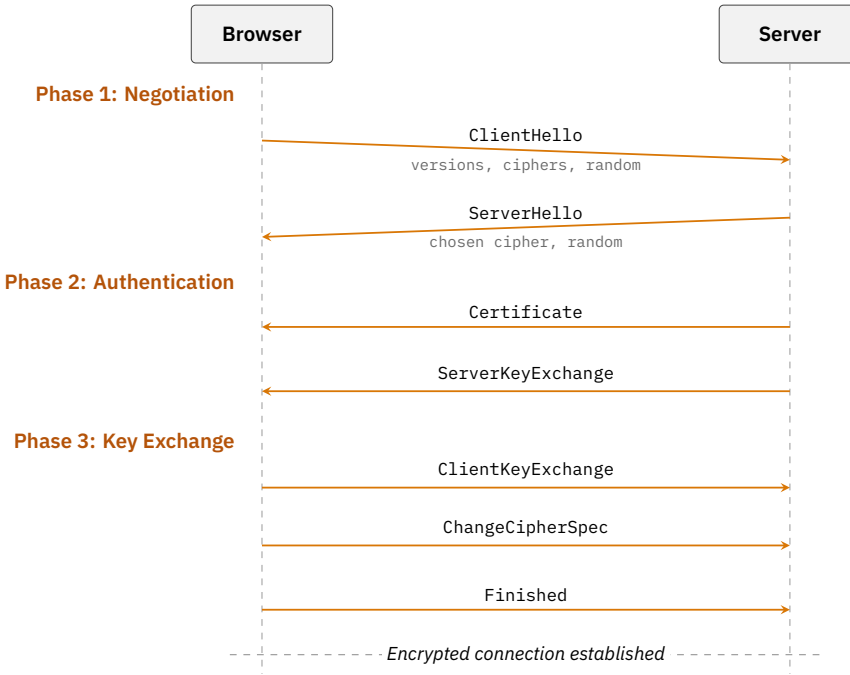


Figure 19.2: TLS 1.2 handshake. The browser and server negotiate encryption parameters (Phase 1), the server proves its identity with a certificate (Phase 2), and both sides establish a shared secret key (Phase 3). TLS 1.3 reduces this to a single round trip.

> RFC 8446 --- August 2018
>
> The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream.

Eric Rescorla, who edited the TLS 1.3 specification, spent years negotiating with browser vendors, server administrators, and cryp-

tographers to get it right. The result is cleaner and faster than its predecessors, and **forward secrecy** is now mandatory.

Forward secrecy means that each connection generates its own unique session key—and that key is discarded when the connection closes. Even if an attacker records today's encrypted traffic and steals the server's private key next year, they cannot go back and read the old conversations. The color-mixing produced a one-time secret that no longer exists. Every session is a new mix of colors, and none can be reconstructed after the fact.

19.2.4. WHAT THE LOCK ICON ACTUALLY MEANS

Users have been taught to look for the padlock before entering sensitive information. "If you see the lock," the advice goes, "you're safe."

This is half true at best.

The lock means the connection is encrypted—eavesdroppers cannot read the traffic—and that the server presented a valid certificate for this domain, vouched for by a certificate authority.

The lock does *not* mean the website is legitimate, the website won't steal your information, or the people running it are who they claim to be. The lock only proves the connection is private. It says nothing about whether the recipient deserves your confidence.

Phishing sites routinely use `HTTPS`. They get certificates from legitimate CAs—which only verify domain control, not business legitimacy. A site called `paypa1-secure-login.com` (note the "`1`" instead of "l") can get a valid certificate. Your browser will show the padlock. You'll enter your PayPal password, and it will go—encrypted and secure—directly to criminals.

Security researchers call this a "false sense of security." The lock became a symbol of safety when it was only ever a symbol of privacy.

Modern browsers have tried to fix this confusion. By 2023, Chrome had quietly retired the lock icon.[245] But decades of "look for the lock" advice are hard to undo.

19.3. THE DUTCH FAILURE

On a summer morning in August 2011, a user in Iran opened Gmail. Something looked wrong. His browser—Google Chrome—displayed a warning he had never seen before. The security certificate for `mail.google.com` didn't match what the browser expected.

He posted about it on a Google help forum. Within hours, security researchers are investigating. What they find will bring down a certificate authority and expose how fragile the PKI really is.

The certificate his browser rejected came from DigiNotar, a Dutch CA. It was technically valid—signed by DigiNotar's root key, which every major browser accepted. But Google's Chrome browser had an extra layer of defense. It knew what Google's real certificates looked like, and this wasn't one of them.

Someone had gotten DigiNotar to issue a fake certificate for `*.google.com`. The asterisk is a wildcard—it matches any subdomain. A certificate for `*.google.com` covers `mail.google.com`, `docs.google.com`, `www.google.com`, and every other subdomain. Organizations use wildcards for convenience: one certificate protects dozens of services. The attackers got a certificate that covered Gmail, Google Search, Google Docs—everything Google operated.

Chrome's detection relied on certificate pinning—hard-coded expectations about which certificates Google properties should present.

Certificate Transparency, the public logging system that now makes fraudulent certificates visible to anyone, did not yet exist in 2011.

The investigation moved fast. Researchers at Google and security firms started digging.[246] On August 29, just two days after the forum post, DigiNotar admitted it had been hacked a month earlier. Attackers had penetrated their systems in mid-July and issued more than 500 fraudulent certificates.[247]

Not just for Google—for Microsoft, Yahoo, Mozilla, the CIA, MI6, and Mossad, according to the Fox-IT investigation.

The attackers had been inside DigiNotar's systems for weeks, issuing certificates for whatever they wanted. A certificate for `*.torproject.org`—the anonymity network used by dissidents worldwide. A certificate for `windowsupdate.microsoft.com`—meaning attackers could potentially push malware as legitimate Windows updates.

Fox-IT, the Dutch forensic security firm that investigated, published a devastating report.[247] DigiNotar's security was abysmal. No network segmentation, weak passwords, no intrusion detection, and outdated software with known vulnerabilities. The attackers had walked through DigiNotar's systems with apparent ease.

But the worst part wasn't the breach itself. The worst part was what the certificates were used for.

An estimated 300,000 Iranian Gmail accounts were intercepted using the fake Google certificate.[247] When they connected to Gmail, they connected to servers controlled by—well, no one knows for certain, but the evidence points toward Iranian intelligence services. Their emails were read, their contacts exposed. For dissidents organizing against the regime, this wasn't an abstract security failure. It was a matter of life and freedom.

The context matters. In 2009, millions of Iranians had taken to the streets to protest a disputed election. The Green Movement, as it

became known, organized through social media—through Gmail and Facebook and Twitter. The regime crushed the protests—thousands arrested, some tortured, some killed in custody. Two years later, the same government that had suppressed the Green Movement was reading the emails of anyone who logged into Gmail through Iranian internet providers.

Hans Hoogstraaten, a Dutch prosecutor who investigated DigiNotar, was blunt about the stakes: "People got killed for having a different opinion".[248] The failure of a small Dutch certificate authority was connected, however indirectly, to the suppression of a political movement. The attackers hadn't just stolen data—they had weaponized trust.

The attackers left traces. In a text file on DigiNotar's servers, someone wrote in Persian: "There is no God but Allah." Another signature appeared in the code: "I am Comodo Hacker." The same claim had appeared in an earlier attack on Comodo, another certificate authority, in March 2011. The same individual—or the same unit—had compromised two separate CAs in the same year.

On September 3, 2011, every major browser revoked DigiNotar's root certificate. Overnight, every website that used a DigiNotar certificate became invalid. The Dutch government, which had used DigiNotar for citizen-facing services, scrambled to get new certificates.

Three weeks later, DigiNotar's parent company filed for bankruptcy.[249] A certificate authority—accepted by every browser on the planet—was destroyed in less than a month.

19.3.1. THE OPENSSL BUG

The DigiNotar breach showed that CAs could be compromised. A bug discovered three years later showed that TLS itself had hidden weaknesses—not in the protocol design, but in its most common implementation.

On April 7, 2014, security researchers from Google and Codenomicon disclosed a bug they called Heartbleed.[250,251] The name came from the TLS "heartbeat" extension, a keep-alive feature that let clients and servers confirm the connection was still active.

The heartbeat works simply: a client sends a message saying "echo this back to me," and the remote host copies the specified data and returns it as proof of life.

The bug was in the copy.

A heartbeat message includes two fields: the data to echo and the length of that data. A legitimate message might say "Here are 16 bytes of data—echo them back," and the receiving machine dutifully reads those 16 bytes and returns them.

But OpenSSL, the software library that handles TLS for most of the internet's servers, didn't check whether the stated length matched the actual data. An attacker could send a message saying: "Here is 1 byte of data. Echo back 64,000 bytes."

The server would obediently read 1 byte of the attacker's data, then copy 63,999 bytes of whatever happened to be next to it in memory. And what was in memory? Potentially, anything. Other users' session cookies, private encryption keys, passwords, usernames, credit card numbers. Whatever had recently passed through the server's TLS processing.

```
Heartbeat Request:
  Data: "X"
  Claimed Length: 65535 bytes

Heartbeat Response:
  Data: "X" + [65534 bytes of server memory]
```

The leaked memory could include private SSL keys, session tokens, usernames, passwords, and other users' requests.

The buggy code was committed in December 2011 and shipped in released software on March 14, 2012, with OpenSSL 1.0.1. For over two years, anyone who knew about the bug could have silently stolen data from approximately 17% of the web's secure servers.

Bruce Schneier, the security expert, was blunt: "Catastrophic is the right word. On the scale of 1 to 10, this is an 11".[252]

The aftermath was chaos. Every server running a vulnerable version of OpenSSL needed patching. Every private key on those servers needed to be considered compromised. Certificates required revocation and reissue—millions of them. Users were told to change their passwords everywhere, but only after the sites they used had patched their servers.

The bug was a missing bounds check. Two lines of code—or rather, two lines of missing code—that would have verified the claimed length matched the actual data. The fix was trivial, but the damage was immeasurable.

19.3.2. THE CONTINUING TENSION

TLS works—e-commerce exists, banks operate online, and we send passwords and credit cards through encrypted channels where, the overwhelming majority of the time, only the intended recipient sees them.

But the architecture assumes that verification can be delegated. That a small number of organizations—CAs that most users have never heard of—can vouch for the identity of every server. That those organizations will maintain perfect security, forever.

DigiNotar demonstrated the cost of a single CA failure, while Heartbleed showed that even correct protocols can be undermined by implementation bugs.

The padlock remains, though browsers display it less prominently now. The lesson isn't that encryption is impossible or that the internet is insecure. Security depends on delegation—and delegation can fail. Concentrated authority creates single points of failure.

We built a system where 150 organizations can vouch for any website and assumed it would hold. Usually it does—but when a small Dutch company gets hacked, dissidents on the other side of the world pay the price.

TLS solved one problem: eavesdroppers could no longer read the conversation. But the conversation still had to reach you, and reaching you increasingly meant traveling through the air. The coffee shop WiFi that Eric Butler exploited with Firesheep, the hotel network where business travelers checked email, the airport lounge where executives negotiated deals—all of them broadcast radio waves that anyone with an antenna could receive. TLS encrypted what traveled over those airwaves. It could not prevent someone from capturing them.

Encrypting the air would prove even harder than encrypting the wire.

The padlock is disappearing from browsers.

Not because encryption failed—because it won. Chrome stopped displaying the lock icon in September 2023, reasoning that HTTPS had become the default rather than the exception. Firefox and Safari followed. After thirty years of "look for the lock," the lock is being retired because it no longer marks something unusual. As of 2024, over 95% of pages loaded in Chrome travel over HTTPS.[253] Let's Encrypt alone

has issued more than 400 million active certificates.[254] The encryption Taher Elgamal assembled in a Netscape office in 1994—contested by governments, limited by export law, rejected by early users—now runs invisibly beneath nearly every web transaction on the planet.

You notice TLS when it fails. A certificate error interrupts a page load and warns you away. A suspicious certificate triggers a browser alert. The handshake completing successfully is something you never see, because success means nothing interrupted—the session key generated, the data encrypted, the page rendered. Billions of times per day, the protocol performs its work and disappears. That invisibility is what Elgamal and Dierks were building toward, even if they never put it in those terms.

The CA system still has weaknesses. The 150 root certificates in your browser represent 150 organizations you have never vetted. One failure—DigiNotar proved this—can weaponize trust against the people it was built to protect. Certificate Transparency now requires CAs to log every certificate they issue, so fraudulent certificates become detectable.[255] CAA records let domain owners restrict which CAs may issue for their names.[256] The bargain holds, mostly. The lock is gone from the address bar. The encryption it represented is not.

CHAPTER 20

WAVES OF TRUST

WEP is intended to provide data confidentiality
that is subjectively equivalent
to the confidentiality of a wired LAN
that does not employ cryptographic techniques
to enhance privacy.

— IEEE 802.11 Standard, 1997

20.1. THE CABLE WAS THE PROBLEM

Vic Hayes watched technicians snake Ethernet through the ceiling tiles of a Dutch warehouse in 1988.[257] NCR Corporation wanted to connect its cash registers to a central computer. That meant running wire to every checkout lane—an installation that took weeks. Every time the layout changed, every time a new register was added, the technicians came back with spools of cable and ceiling ladders. Hayes, a lanky engineer at NCR's Utrecht facility, stood watching and thinking there had to be a better way.

Radio was not new to networking. HAM operators had experimented with packet radio since the 1970s. The military used encrypted wireless links. But these were specialized applications requiring expensive equipment and licensed spectrum. You could not just walk into a warehouse and start broadcasting. Hayes imagined something different: cheap, unlicensed wireless anyone could deploy. A replacement for the cable itself.

The obstacles were large. Radio frequencies belonged to governments. To transmit without a license, you needed spectrum regulators had left open for anyone. In the United States, the FCC had designated bands for "industrial, scientific, and medical" applications—the ISM bands. Your microwave oven operates there. So do garage door openers. The ISM bands were a radio free-for-all: anyone could transmit, everyone tolerated interference.

Hayes started building prototypes in 1988. Born in Surabaya in the Dutch East Indies, he had trained as an electrical engineer at the Hogeschool van Amsterdam before joining NCR in 1974. In a cramped lab in Utrecht, his team soldered together transceivers and wrote driver software. The 2.4 GHz ISM band was their playground—the same frequencies used by microwave ovens, chosen precisely because no government claimed exclusive rights to them. By 1990, NCR had working wireless networking hardware. It was slow, about 1 megabit per second, and the range topped out at a hundred meters under ideal conditions. But it worked. Cash registers could talk to servers without a single cable between them.

The technology was useless without a standard. NCR's wireless would not talk to anyone else's wireless. Every vendor had its own frequencies, its own encoding, its own packet formats. Proxim sold one wireless product, Motorola sold another, and Symbol Technologies had its own approach. If you bought from one vendor, you were locked

in. For wireless networking to become ubiquitous, companies would have to agree on a common way to do it.

Hayes took on the standardization effort himself. In 1990, the Institute of Electrical and Electronics Engineers formed a working group to develop a wireless local area network standard.[258] Hayes became its chairman. The group was designated `802.11`, following the IEEE's numbering scheme for networking standards. 802.3 was Ethernet. 802.5 was Token Ring. `802.11` would be wireless. The number was bureaucratic, but the ambition was sweeping: a single standard that would let any device talk to any access point, from any manufacturer, anywhere in the world.

The committee meetings were contentious. AT&T, which had acquired NCR by this point, wanted one approach. Proxim wanted another. Lucent, Symbol Technologies, Apple—everyone had different ideas about frequency bands, modulation schemes, and collision avoidance. The representatives sat in conference rooms in hotels near airports, arguing about technical minutiae while their companies' patent portfolios hung over every decision. Hayes spent seven years mediating, compromising, and occasionally forcing votes when consensus seemed impossible. He also fought a parallel battle with European regulators. The goal was harmonized spectrum: 83.5 MHz in the 2.4 GHz band and 475 MHz in the 5 GHz band, the same allocations on both sides of the Atlantic, so that a device built for Europe would work in America and vice versa.

> IEEE 802.11 Project Authorization Request --- 1990
>
> The IEEE 802.11 Working Group shall develop a wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specification... compatible with existing 802 networks... The standard shall support multiple physical layers.

One of the hardest issues was security. Radio signals do not stop at walls. Anyone within range can hear the traffic: a competitor in a van outside your building, a bored teenager in the next apartment, a spy in the embassy across the street. If you replace the Ethernet cable, you must replace the physical security that cable provided. Someone has to tap a wire to eavesdrop on Ethernet. With wireless, they need only an antenna and patience.

The committee knew encryption was necessary. They also knew it was problematic. In 1997, when the 802.11 standard was finalized[259], the United States regulated cryptography as a munition.[260] The State Department limited exports to 40-bit keys—weak enough to crack in hours. The 802.11 committee faced the same dilemma that had constrained SSL designers a few years earlier (Chapter 19): build strong encryption that could not ship internationally, or weak encryption that would fail.

They chose deployment over security. WEP's 40-bit version became the default for international markets.

The committee designed what they called Wired Equivalent Privacy, or WEP. The name tells you everything about their ambitions. They were not trying to build the strongest possible encryption. They were trying to provide privacy "equivalent" to a wired network—meaning, good enough to keep casual eavesdroppers out, but not a serious barrier to a determined attacker with physical access. The goal was a lock on the door, not a vault.

WEP used the RC4 stream cipher, a fast algorithm that could run on the weak processors in networking hardware. RC4 was popular in the 1990s: it required minimal memory, computed quickly, and had been designed in 1987 by Ron Rivest of RSA fame.[261] WEP combined a shared key with a 24-bit initialization vector to generate a keystream—a sequence of random-looking bytes. That keystream was XORed with

the plaintext to produce ciphertext. XOR is a simple bitwise operation: apply it twice and you get back the original. Standard stuff, for 1997.

The committee made choices that would prove fatal. The initialization vector was too short—24 bits gave only about 17 million possible values, and on a busy network those would be exhausted in hours. The key scheduling was weak, leaking information about the secret key itself. There was no mechanism for changing keys automatically; if you wanted to rotate your network password, you had to visit every device and type in a new one. In 1997, with export regulations limiting what they could deploy and hardware constraints limiting what they could compute, WEP seemed good enough.

Hayes called wireless "the final step" in the networking puzzle. He imagined laptops roaming freely through office buildings, connecting to the network from anywhere. He imagined homes without Cat5 cables running along baseboards. He imagined airports and coffee shops where travelers could check email without hunting for a phone jack. For his work chairing the 802.11 committee through seven years of contentious standardization, the industry would later call him "the father of WiFi." In 2007, the IEEE awarded him the Charles Proteus Steinmetz Award. In 2012, the American Computer Museum gave him the George R. Stibitz Computer and Communications Pioneer Award.

All of this came true—so did the security problems the committee had underestimated.

20.2. BREAKING THE AIR

20.2.1. THE SHARED SECRET PROBLEM

You open your laptop at a cafe and type the WiFi password from the chalkboard. Thirty feet away, someone else types the same password, and a third person by the window does the same. You all know the secret, so you are all inside.

That password is the only wall.

If the key travels over the air unencrypted, anyone listening can capture it. WiFi solves this with cryptographic key derivation: both your device and the access point know the password, so they can derive encryption keys without transmitting the password itself. An eavesdropper sees the encrypted negotiation but cannot work backward to the underlying secret. Two parties agree on a key while a third party watches the entire conversation and learns nothing useful.

The problem is that everyone who knows the password can do the same math. The password on the chalkboard is not protecting you from the person by the window. It protects both of you from people who do not know the password—and only if the encryption works.

The question is whether it does.

20.2.2. WEP: HOW TO BREAK ENCRYPTION

WEP failed spectacularly. To understand why, we need to understand how stream ciphers work—and how WEP got them wrong.

A stream cipher encrypts data bit by bit, generating a pseudorandom keystream—a sequence of random-looking bits—from a key and an initialization vector. XOR the plaintext with the keystream and you get ciphertext; XOR is reversible, so apply it twice and you are back where you started. The receiver, knowing the key and IV, generates the same keystream and recovers the plaintext.

The security rests on one rule: never reuse the same keystream.

If two messages are encrypted with identical keystreams, an attacker with both ciphertexts can XOR them together, and the keystreams cancel out, leaving the XOR of the two plaintexts. From there, extracting the original messages is straightforward—especially when the messages have predictable structure like HTTP headers or email formats. The encryption does not just weaken; it vanishes.

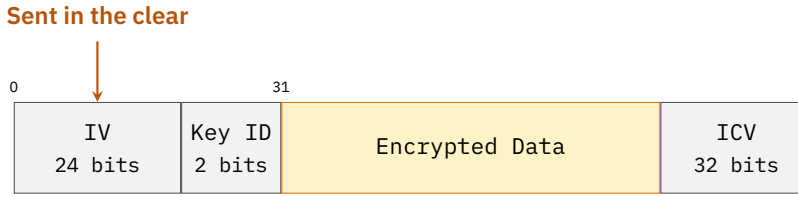


Figure 20.1: A WEP packet. The IV is transmitted in the clear. With only 24 bits, IV collisions are guaranteed on busy networks — and each collision leaks information about the key.

WEP's IV is only 24 bits long—just 16.7 million possible values. On a busy network transmitting 1,000 packets per second, all possible IVs are exhausted in about four and a half hours. After that, IVs start repeating, keystreams start repeating, and the encryption is broken. And it gets worse.

In 2001, three cryptographers—Scott Fluhrer, Itsik Mantin, and Adi Shamir—published a paper at the Selected Areas in Cryptography workshop showing that some IVs were worse than others.[262] They called it "Weaknesses in the Key Scheduling Algorithm of RC4." The title was understated. The implications were not.

The vulnerability lay in "weak IVs"—certain initialization vectors that leaked clues about the secret key. These IV patterns, combined with how WEP initialized the RC4 cipher, produced keystream bytes statistically correlated with the key. The first bytes of encrypted data were not truly random; they contained faint echoes of the key itself. Anyone who captured enough packets with weak IVs could derive

each byte of the key sequentially. No brute force. No guessing. Pure mathematics.

The result was devastating: collect a few million packets—a few hours of traffic on a typical network—and the key falls out. Security researchers immediately built tools to automate the process, with AirSnort appearing first, then weplab, then `aircrack`, each generation faster than the last.

In 2007, three German researchers at Darmstadt University of Technology—Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin—published a refinement that made the original attack look primitive.[263] Their paper, "Breaking 104 bit WEP in less than 60 seconds," delivered exactly what the title promised. The Pyshkin-Tews-Weinmann attack needed only 40,000 to 60,000 packets instead of millions. Combined with active packet injection—forcing the network to generate traffic by replaying captured `ARP` requests—an attacker could recover a WEP key in under a minute. The attack was integrated into `aircrack-ng` and became the default method. WEP was not just broken; it was trivially broken.

```
$ aircrack-ng capture.cap
Opening capture.cap
Read 68291 packets.

   #  BSSID              ESSID         Encryption

   1  00:14:BF:3B:96:41  LINKSYS       WEP (47281 IVs)

Choosing first network as target.

                         KEY FOUND! [ 41:42:43:44:45 ]
      Decrypted correctly: 100%
```

Tools like aircrack-ng automated the process. Breaking WEP became something a teenager could do with a laptop and a few minutes of patience. "Wired Equivalent Privacy" provided privacy equivalent to shouting your data across a crowded room.

By 2002, the 802.11 committee knew WEP was dying. The attacks were public. Network administrators were panicking. But replacing WEP was not simple. Millions of devices were already deployed—access points, laptops, PDAs, industrial equipment—all running WEP, all expecting a certain protocol. A complete replacement would make every existing device obsolete.

The WiFi Alliance released WPA in 2003 as an emergency measure.[264] WPA kept the RC4 cipher but wrapped it in TKIP, the Temporal Key Integrity Protocol. TKIP extended the IV to 48 bits, eliminating IV exhaustion. It generated a unique key for every packet. It added integrity checking to detect tampering.

But RC4 was weak, and researchers kept finding attacks that TKIP's patches could only partly contain.

The real fix came in 2004 with WPA2[265], which replaced RC4 entirely with AES—the cipher NIST had selected after a five-year international competition. AES was stronger, well-analyzed, and designed to resist the statistical attacks that killed WEP. WPA2 was what WEP should have been from the start.

WPA2 is what most networks use today, and the encryption is solid. But WiFi security involves more than encryption—it involves key management, and key management is where WPA2 still has problems.

20.2.3. THE FOUR-WAY HANDSHAKE

How do two strangers agree on a secret key while someone else is listening? WPA2 solves this problem every time you connect to a network. Your laptop and the access point both know the password.

Protocol	Encryption	Key Size	Status
WEP	RC4	40/104 bits	Broken
WPA	RC4/TKIP	128 bits	Deprecated
WPA2	AES/CCMP	128/256 bits	Standard
WPA3	AES/GCMP	128/256 bits	Recommended

Table 20.1: WiFi security protocol evolution. WEP lasted from 1997 until its complete compromise by 2007.

From that password, they derive encryption keys—fresh ones, unique to this session—without transmitting the password itself. An attacker watching the entire exchange learns nothing.

The solution is a four-message exchange called the handshake. Watch how confidence builds—and where it could break.

Both sides start by computing a master key from the password. They run it through 4,096 rounds of hashing, deliberately slow to punish anyone trying to guess passwords by brute force. This master key never changes; it is the foundation. But you cannot encrypt traffic with the master key directly. If you did, every session would use the same encryption, and an attacker who broke one session would break them all.

So they derive a session key by mixing fresh random numbers from both sides—called nonces, used only once—with the master key and the hardware addresses of both devices. The result is a unique session key, different every time you connect.

The handshake unfolds in four steps. First, the access point sends its random number. Your device generates its own random number, computes the session key, and sends its number back along with a cryptographic proof that it knows the password. The access point, now

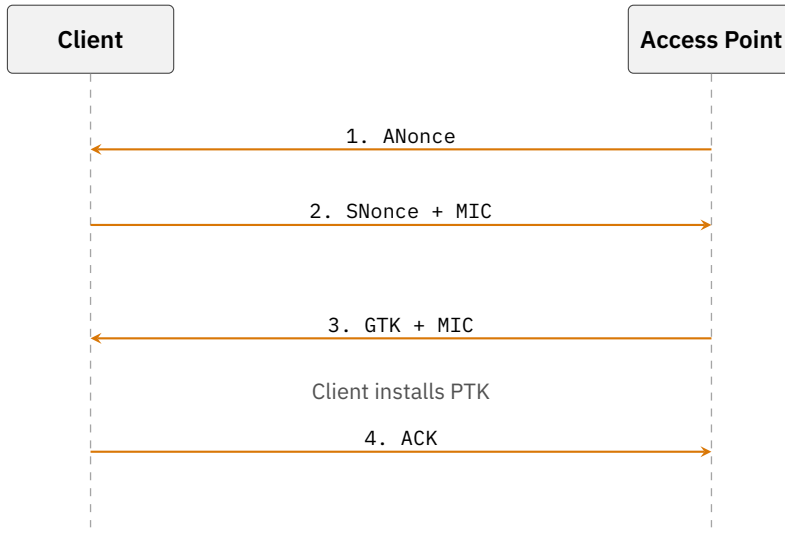


Figure 20.2: The WPA2 four-way handshake. Both sides derive the same PTK from the PMK and exchanged nonces.

having both random numbers, computes the same session key and sends the shared broadcast key. Your device acknowledges.

After message three, your device installs the session key and starts encrypting. Both sides end up with identical keys, derived from fresh random values. An eavesdropper who captured all four messages still cannot decrypt anything. Without the password, the math does not work.

The handshake has to handle errors. Wireless is unreliable. Packets get lost. What happens if message four does not arrive?

WPA2 has two modes. Personal mode uses a pre-shared key—the password you type when joining the network—and everyone who knows it can connect. That is why you use it at home, why most coffee shops use it, why small businesses use it. One password, shared among all users.

Personal mode has a weakness: if you know the password and capture the four-way handshake, you can derive the session key and decrypt traffic. Worse, you can take the captured handshake home and run password-guessing software against it for as long as you want. If the password is weak—"password123," "CompanyName2024," the name of the business with a number at the end—cracking it is trivial. The attack happens offline. The network never knows anyone is trying.

Enterprise mode is different: each user authenticates individually against a RADIUS server using their own credentials—your username and password, your employee badge, your certificate. When you leave the company, IT revokes your account without changing the WiFi password and redistributing it to everyone. Someone who captures Enterprise traffic cannot crack a shared password because there is none; they would need to attack individual credentials, which is harder.

Why does everyone not use Enterprise mode? Complexity. You need a RADIUS server, user account management, certificate issuance, and access point configuration—routine for a corporation with an IT department, but beyond a neighborhood cafe.

So the world runs on Personal mode, with all its weaknesses.

20.2.4. THE SHARED AIR

The password is the perimeter: anyone who knows it is inside, and everyone else is outside. Without this promise, every coffee shop becomes a surveillance station, every home network an open door.

For home networks, the model works well enough—you share the password with family and guests. For businesses, it is less ideal since every employee knows the password, and when someone leaves, you should change it. For public networks, the model breaks down entirely: open networks have no encryption at all, and "secured" networks often post the password on a sign above the counter.

For twenty years after WEP's collapse, WPA2 held this line. Security researchers probed it, looking for weaknesses. They found some: dictionary attacks against weak passwords, attacks on TKIP that worked under specific conditions. But the core of WPA2—AES encryption with the four-way handshake—seemed solid.

Then, in 2017, a Belgian researcher found a hole in the handshake itself.

20.3. WARDRIVING IN MARSHALLS

It was the summer of 2005. Albert Gonzalez was sitting in a car outside a Marshalls store in Miami[266,267], armed with a laptop, a directional antenna, and patience. Through the windshield he could see shoppers coming and going, hunting for bargains on discount clothing. Inside, cash registers were processing credit card transactions—and the store's wireless network was running WEP.

Gonzalez was twenty-four years old, thin, with the wired intensity of someone who spent too much time staring at screens. He had learned about credit card theft the hard way. In 2003, the Secret Service had arrested him for running a website that trafficked in stolen credit cards. He cut a deal: become an informant, help the feds catch other carders, avoid prison. For two years, he worked both sides—feeding the Secret Service information on his former associates while quietly building his own criminal operation. The Secret Service thought they had turned him. They were wrong.

TJX Companies owned Marshalls, along with T.J. Maxx, HomeGoods, and several other retail chains.[268] Their stores processed millions of credit card transactions every month. And in 2005, their wireless

security was stuck in 1997. While security researchers had been warning about WEP for years, while corporations with experienced IT departments had long since upgraded, TJX was still protecting its networks with broken encryption.

The WEP key fell quickly—Gonzalez's crew collected packets, ran `aircrack`, and within minutes had access to the network. From the parking lot, they could see traffic flowing between the cash registers and the store's servers: credit card numbers, expiration dates, CVV codes, everything a carder needed. The encryption meant to protect this data was performing exactly as designed—just not as intended.

But Gonzalez wanted more than one store's traffic—he wanted the entire company.

From the store network, his team found connections to TJX's corporate systems in Framingham, Massachusetts. The internal network was more secure than the store-level WiFi, but not secure enough. TJX had failed to segment its networks properly—the store networks that processed credit cards connected to the same network that handled inventory and logistics. Through SQL injection attacks and weak access controls, Gonzalez's crew wormed their way deeper. They installed packet sniffers on internal servers, capturing credit card data as it flowed between point-of-sale terminals and the company's central processing systems. By early 2006, they had access to the systems that processed card transactions for thousands of stores across the country.

The breach continued for a year and a half. Gonzalez and his crew siphoned credit card numbers from TJX's databases, tens of millions of them. They sold the data to buyers in Eastern Europe who specialized in fraud. They encoded stolen numbers onto blank magnetic-stripe cards and used them to withdraw cash from ATMs. They bought electronics and gift cards, converting stolen data into untraceable goods. A single WEP vulnerability in a Marshalls parking

lot had become the entry point for one of the largest data breaches in history.

The timeline tells the story. In July 2005, Gonzalez's team cracked WEP at a Marshalls store in Miami. By late 2005, they had access to TJX's corporate network. Throughout 2006, they exfiltrated credit card data continuously. TJX detected suspicious activity in December 2006 and disclosed the breach in January 2007. By March, the full scale of the breach became clear. Gonzalez was arrested in May 2008 for a different breach and sentenced to twenty years in federal prison in March 2010.[269]

The final tally was staggering: 45.7 million confirmed compromises, though some estimates reached 94 million. The breach cost TJX over $250 million in settlements, legal fees, and security upgrades.[270] Banks reissued millions of cards. Customers received letters warning them to monitor their accounts for fraudulent charges. Credit monitoring became a standard offering for breach victims—a practice that continues to this day.

The attackers had not exploited a software bug. They had not used a zero-day vulnerability. They had simply taken advantage of a protocol that the security community had known was broken for years. WEP was still protecting financial transactions in 2005 because upgrading was inconvenient, expensive, and—until someone actually exploited it—seemingly unnecessary. The gap between "known to be broken" and "actually broken into" was eighteen months of stolen credit cards.

20.3.1. KRACK: THE HANDSHAKE BETRAYED

TJX was a failure of deployment: using broken security when better alternatives existed. A decade later, a Belgian researcher would demonstrate that even the better alternatives had problems.

Mathy Vanhoef was a doctoral student at KU Leuven, working on WiFi security under the supervision of Frank Piessens in the imec-DistriNet research group. His office was in the university's computer science building in Leuven, a medieval city east of Brussels. He spent his days reading specifications, writing proof-of-concept code, and looking for flaws that might have been overlooked. In 2016, he began analyzing the `WPA2` handshake—the very mechanism that was supposed to have replaced WEP's broken security.

The IEEE `802.11i` specification ran to hundreds of pages, and Vanhoef read it with a cryptographer's eye, looking not at what the protocol did, but at what it assumed. The four-way handshake was well-designed for its primary purpose: deriving session keys without transmitting the password. But Vanhoef noticed something in the error-handling sections—a gap between intent and implementation.

The four-way handshake has a reliability mechanism because radio is lossy and acknowledgments vanish. If the client sends message 4 but the access point does not receive it, the access point retransmits message 3. The client is supposed to handle this gracefully—acknowledge again and continue. The protocol designers had thought about this and built in error handling.

Vanhoef realized that handling retransmissions was dangerous. When the client receives message three, it installs the session key and resets its nonce counter—the number that increments with each packet to ensure unique encryption. If an attacker forces message three to be retransmitted, the client reinstalls the key and resets the nonce. The same key. The same nonce values. Used again.

Nonce reuse in encryption is catastrophic: with the same key and nonce, you get the same keystream, and if you XOR two ciphertexts encrypted with the same keystream, the keystreams cancel out—the same failure mode that killed WEP, now appearing in its successor.

An attacker who could trigger key reinstallation could collect traffic encrypted with repeated nonce values, XOR the ciphertexts together, and recover plaintext. With enough traffic, they could inject packets and, in some cases, take over the connection entirely. The mechanism designed to handle lost packets became the vulnerability.

The attack required the attacker to be positioned between the client and the access point. This meant cloning the target access point on a different channel and forcing the client to connect to the attacker's copy. From there, the attacker could selectively block and replay handshake messages, triggering key reinstallation at will.

Vanhoef called the attack KRACK: Key Reinstallation Attack.[271]

On October 16, 2017, Vanhoef publicly disclosed the attack; the formal paper was presented at the ACM Conference on Computer and Communications Security in November.[272] The paper, "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2," earned the conference's distinguished paper award. Vanhoef had coordinated with major vendors for months before the announcement, giving them time to prepare patches. US-CERT issued ten CVE numbers—CVE-2017-13077 through CVE-2017-13088—one for each variant of the attack.

When the news went public, the response was immediate. Every WiFi device in the world—every laptop, phone, tablet, IoT sensor, smart thermostat, WiFi-connected camera—was vulnerable. The attack worked against WPA2-Personal and WPA2-Enterprise alike, on Linux, Android, Windows, macOS, and iOS. Some implementations were worse than others. Android 6.0 and certain Linux distributions, due to a quirk in their wpa_supplicant software, installed an all-zeros key on reinstallation. The encryption became entirely worthless. The bug was in how the client software interpreted the retransmission—it zeroed out key memory before reinstalling, and if the key had not arrived yet, it installed zeros.

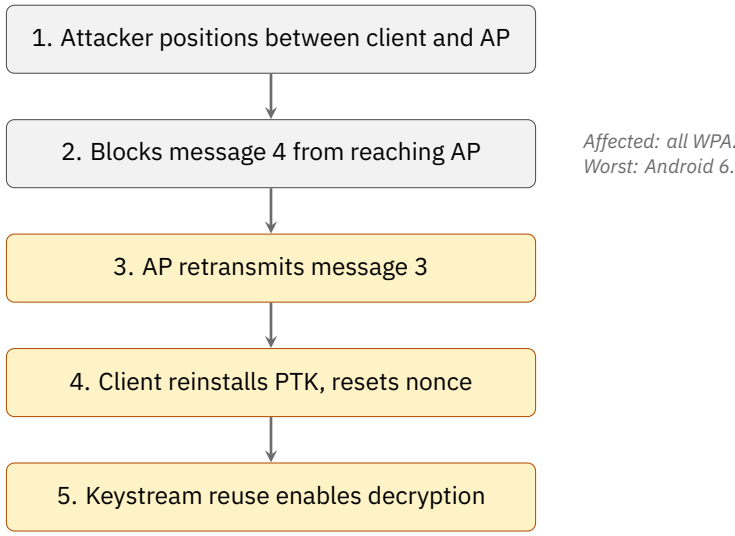


Figure 20.3: KRACK attack exploits the WPA2 four-way handshake state machine. By blocking the client's acknowledgment (message 4), the attacker forces the access point to retransmit message 3, causing the client to reinstall an already-used encryption key — and reset the nonce counter to zero.

Patches rolled out within days. Microsoft, Apple, and Google pushed updates to their operating systems. The WiFi Alliance issued guidance. Network administrators around the world scheduled emergency updates. The attack required the attacker to be in WiFi range, which limited its practical exploitability—you could not attack someone from across the internet, only from the parking lot. But the theoretical implications were troubling: a protocol considered secure for over a decade contained a fundamental flaw in its key management. The flaw was not in the cryptography. The math was fine. The flaw was in the state machine that managed the handshake, in how software kept track of what step it was on.

WPA3, finalized in 2018[273], addresses KRACK and other weaknesses. It replaces the four-way handshake with SAE—Simultaneous Authentication of Equals[274]. The name captures the key difference: both sides prove they know the password simultaneously, rather than the access point challenging the client. SAE provides forward secrecy and resists offline dictionary attacks. Even if an attacker captures the handshake and later learns the password, they cannot decrypt past traffic. Adoption has been slow—WPA2 devices outnumber WPA3 devices by a wide margin, and many access points still need to support older clients—but the industry learned the lesson.

Vanhoef was not finished. In 2019, he broke WPA3 with an attack called Dragonblood[275], exploiting flaws in the SAE handshake within months of its release. In 2021, his FragAttacks research[276] revealed something worse: bugs in the original 1997 802.11 specification—design flaws from the very first standard—had persisted through every revision for twenty-four years, unnoticed.

20.3.2. THE ENCRYPTED AIR

Wireless signals respect no perimeter. Anyone within range receives your traffic. The only protection is encryption, and encryption is only as good as its implementation. Each generation of WiFi security has been an attempt to patch a fundamental problem: radio waves go everywhere, and mathematics has to stop everyone who should not be listening.

From Vic Hayes's dream of wireless freedom to Albert Gonzalez's parking lot laptop, the lesson is the same: faith in the airwaves is faith in mathematics, and mathematics has to be flawless. The cables were never this complicated.

Vic Hayes wanted to eliminate the cable. He succeeded beyond anyone's imagination. The wireless network that started in a Dutch warehouse now blankets the planet, reaching coffee shops, airports, hospitals, and homes, connecting billions of devices that have never touched a wire.

The encryption keeps breaking—WEP was flawed from the start, `WPA2` took thirteen years to crack, and `WPA3` fell within months of release. The math struggles to keep up because the air refuses to be contained: radio waves pass through walls, cross property lines, and leak into parking lots where attackers sit with laptops and antennas.

The waves never stop. They carry everything still.

INTERLUDE: RIGHT NOW

Right now, as your eyes move across this page, four million emails are in transit. One million DNS queries are resolving. 1.2 million TLS handshakes are completing. Fifteen thousand BGP route updates are propagating across the global routing table.[1]

All of this, in the time it took to read this sentence.

Right now, also: twenty-five thousand vulnerabilities sit unprocessed in the CVE database.[2] The agency tracking them operates at reduced capacity. One hundred forty-three national CERTs coordinate across languages and jurisdictions, trying to keep pace with threats that cross borders in milliseconds.

The infrastructure watching over the network is strained, but the system runs anyway.

Every one of these events involves an act of faith.

The mail server accepts the sender's claim. The resolver believes the DNS response. The browser validates the certificate chain. The router honors the BGP announcement. Each transaction assumes the other party is telling the truth.

[1] Statistics in this interlude are order-of-magnitude estimates derived from industry reports: email volumes from the Radicati Group[277], DNS query rates from Cloudflare and NS1 analyses[278,279], TLS adoption from Google's Transparency Report[253], and BGP routing data from the CIDR Report and APNIC.[168,280]

[2] Institutional statistics from NIST NVD analysis[281], CISA workforce reports[282], CVE Program funding announcements[283], and FIRST CERT directory.[284]

Most of this faith is justified—usually, most participants are honest. Most emails arrive, most lookups are accurate, most certificates are legitimate, most routes are valid.

Most.

These standards aren't historical artifacts. They're executing right now. The design choices made in 1969, in 1982, in 1993—they're running on servers manufactured last year, in data centers that didn't exist when the specifications were written.

The email system Jon Postel designed for colleagues who would never lie? More than 350 billion messages pass through it daily. The phone book Paul Mockapetris sketched on a whiteboard? Trillions of queries every twenty-four hours. The routing protocol that relies on network operators to tell the truth? One million routes, nearly eighty thousand autonomous systems, held together by announcements that anyone could forge.

And yet.

It works.

Not perfectly—not without cost. We've counted the costs in the chapters behind you: the breaches, the fraud, the surveillance, the chaos. The Morris Worm from Chapter 1, the Kaminsky flaw in Chapter 7, the DigiNotar compromise in Chapter 19. Every chapter ends with trust betrayed.

But every chapter ends with the system still running. The betrayals are exceptions. Most people, on any given day, don't attack. The patches accumulate. The defenses improve.

The internet is not secure—it never was. It was built by people who knew each other. That confidence was encoded into systems that now serve five and a half billion strangers[285] (as of 2024)—specifications

designed for four nodes of colleagues, carrying the traffic of half the planet.

There's a concept in engineering called "graceful degradation." A well-designed system doesn't fail completely when one component breaks. It limps along, reduced but functional, giving humans time to notice and repair.

The internet is the largest graceful degradation in history.

Something is always failing. A root server is under attack. A BGP misconfiguration is leaking routes. A certificate authority is issuing certificates it shouldn't. A spam campaign is exploiting SMTP. A DNS cache is being poisoned. A vulnerability database falls 25,000 entries behind. Somewhere, right now, the premises are being violated.

And still the packets flow, the pages load, the emails arrive. The network absorbs the attacks, routes around the failures, patches the holes—not through any central authority, because there is none, but through the accumulated effort of millions of people who maintain routers and servers and cables, who respond to incidents and write code.

The internet runs on borrowed time, and it keeps extending.

You've seen the pattern: creation, mechanism, breach. The openness wasn't naive—it was necessary. The vulnerabilities aren't bugs—they're consequences of the design that let the network grow.

What you may not have felt yet: you're soaking in it.

The device in your hands depends on these mechanisms. So does the screen before your eyes. So does the network connecting you to

these words. Your DNS resolver accepted the response it got. Your TLS handshake validated the certificate chain. Your packets followed the routes.

You are participating in the act of faith that keeps the internet running.

The chapters ahead are harder. The attacks get larger. The stakes get higher. We'll see nation-states weaponize infrastructure, supply chains fall to compromise, critical systems held hostage.

But we'll also see the infrastructure survive—the ARP packets still announce, the routes still converge, the names still resolve.

Right now, as you read this, the fragile, improbable machine keeps running.

And you're part of it.

==========

PART IV

THE RECKONING

==========

CHAPTER 21

THE NEXT GENERATION

The change from 4 bytes to 16 bytes of address space
is not merely cosmetic.
It is a fundamental change which will require
a new way of thinking about Internet addressing.

— RFC 1752, The Recommendation for the IP Next Generation Protocol, 1995

21.1. THE INFINITE ADDRESS

In a small office at Xerox PARC in Palo Alto, Steve Deering was running numbers that nobody wanted to see.

Fresh from his Stanford doctorate, Deering had come to PARC to work on multicast networking—how to send a single packet to multiple destinations simultaneously.[286] But as he worked on multicast, he kept arriving at a more fundamental problem. The address space was running out.

The number was simple enough: 4,294,967,296.

That's how many addresses Internet Protocol version 4—IPv4—could provide. Every networked device needs a unique address, like a house needs a street address for mail delivery, and the protocol stored these addresses as 32-bit numbers—string them together and you get about four billion possible addresses.

When Vint Cerf and Bob Kahn designed the protocol in the early 1970s, four billion seemed astronomical. The entire planet had perhaps a few million computers. Most would never connect to a network.

By the early 1990s, the premise was starting to look dangerous. The internet was doubling in size every year, personal computers were proliferating, and corporations were building intranets. Deering was among the first to plot the allocation curves and extend them forward. The math was stark: at current growth rates, IPv4 addresses would be exhausted sometime in the next decade. The internet's architects had given it a finite resource and expected it would last forever.

The first serious study of the problem appeared in 1990. Frank Solensky presented an analysis at IETF 18 in Vancouver showing that Class B address exhaustion could arrive as early as 1994.[287] The presentation sent a ripple of concern through the Internet Engineering Task Force—the IETF—which was then meeting three times a year to coordinate internet standards.

Deering watched the debate unfold. Some engineers argued for quick fixes: reclaim unused address blocks, implement stricter allocation policies, extend the address space with tricks and patches. Others argued for something bolder. If the address space was limited, why not replace it?

In 1991, Deering began sketching a new internet protocol. Not a fix for IPv4, but a successor. He called it SIPP, the Simple Internet Protocol Plus. The key change was obvious: expand the address from 32 bits to 128 bits. If 32 bits gave you four billion addresses, 128 bits gave you a number so large it defies comprehension.

The number defies intuition. 128 bits provides 340 undecillion addresses—340 followed by 36 zeros. For every grain of sand on Earth, you could assign a trillion trillion IP addresses. Deering wasn't planning for the next decade. He was planning for the heat death of the universe.

But address space was only part of the vision. The new protocol would be simpler. IPv4 had accumulated cruft over fifteen years: optional header fields nobody used, checksum calculations that slowed routers, fragmentation logic that caused problems as often as it solved them. SIPP would strip the header to essentials: fixed format, no checksums at the network layer, no in-transit fragmentation.

And there was one more addition: security.

The original internet protocols had no encryption, no authentication, nothing to verify that packets came from where they claimed. By 1991, the Morris Worm and a decade of smaller incidents had made clear that good faith alone was not enough.

Deering's proposal included IPsec—a set of protocols that would encrypt and authenticate packets at the network layer. Every IPv6 implementation would be required to support IPsec. Security wouldn't be an optional add-on—it would be built into the foundation.

The IETF held a competition. Deering's SIPP competed against several alternative proposals: TUBA, which used the OSI addressing scheme; CATNIP, which tried to unify everything; and PIP, a minimalist approach. The debate consumed two years and countless mailing list messages.

In July 1994, the IETF made its choice.[288] SIPP won, with modifications. They renamed it IPv6, skipping version 5 (which had been assigned to an experimental streaming protocol that never deployed). Steve Deering and Bob Hinden published the specification in December 1995 as RFC 1883.[289]

The document was a statement of faith: IPv6 would solve the address crisis, simplify routing, and provide security from the ground up. All the internet's operators had to do was upgrade every device on Earth.

Deering knew this wouldn't happen overnight—he had designed transition mechanisms, ways for IPv4 and IPv6 to coexist by tunneling one through the other and translating at the boundaries. The migration would be gradual; it might take a decade.

He was optimistic by a factor of three.

Twenty years after the RFC, IPv6 carried less than five percent of internet traffic. The address crisis had been postponed by Network Address Translation—a hack that let millions of devices hide behind a single public IPv4 address. NAT broke the end-to-end principle the internet's architects held sacred, made peer-to-peer connections difficult, and added latency and complexity. But it worked, and working killed the urgency.

IPv6 adoption crawled. IPv6 was technically superior, but technical superiority doesn't win when the installed base is measured in billions. Network operators had no incentive to upgrade when NAT was free. Equipment vendors had no incentive to prioritize IPv6 when customers weren't asking for it. The chicken-and-egg problem persisted for decades.

The trigger, when it finally came, was exhaustion. On February 3, 2011, the Internet Assigned Numbers Authority handed out the last IPv4 address blocks.[290] The regional registries began rationing. Companies started paying for addresses on secondary markets. The abstract warning from 1990 had become a balance sheet problem.

Even then, the transition was gradual. World IPv6 Day in June 2011 tested compatibility across major websites.[291] World IPv6 Launch in June 2012 marked permanent deployment by Google, Facebook, and

other giants.[292] By 2020, roughly a third of Google's traffic arrived over IPv6. By 2024, it was nearly half.[293]

The protocol Deering designed in 1991 was finally reaching critical mass—thirty years later, in a world he couldn't have imagined, running on devices that didn't exist when he wrote the first draft.

And the security he had built into the foundation? That story turned out differently.

Jim Roskind had a simpler problem: the web was too slow.

It was 2012, and Roskind was a principal engineer at Google who had spent his career on performance—building the Netscape browser in the 1990s, optimizing infrastructure at a dozen companies. Now he stared at latency charts that refused to improve.

TCP was the bottleneck. Every web connection began with a handshake: client sent SYN, server sent SYN-ACK, client sent ACK—three packets before any data flowed. For an encrypted connection, add the TLS handshake: two more round trips. On a mobile network with 100 milliseconds of latency, users waited half a second before the first byte arrived.

Roskind wanted to eliminate those round trips. But TCP was embedded in the internet's infrastructure. Routers, firewalls, and middleboxes all understood TCP. They inspected headers, tracked state, filtered based on flags. Changing TCP would require upgrading millions of devices that Google didn't control.

So Roskind went around them.

UDP, the other major transport protocol, had no handshakes and no expectations—a UDP packet was just bytes with a destination address, and middleboxes usually passed it through unchanged. What if you built a new transport protocol on top of UDP? You could control the

entire stack, from encryption to reliability to congestion control, and deploy updates with a browser release rather than an operating system patch.

Roskind called the project QUIC, a name that would later be backronymed to Quick UDP Internet Connections, with design goals that were deceptively simple: eliminate round trips, encrypt everything, and make connections survive network changes.[294]

The first version of QUIC was running in Google's servers by 2013, deployed quietly and tested with Chrome users who had opted into experimental features. The results were striking: page load times dropped by three percent on average, and by much more on high-latency networks like mobile.

Encryption was mandatory from the start. Unlike TCP, where TLS was bolted on after the fact, QUIC integrated TLS 1.3 directly into its design, combining transport and security in a single handshake: one round trip to establish a connection, negotiate encryption, and verify the server's identity.

For repeat visitors, QUIC could do even better. The protocol supported 0-RTT resumption: if you had connected to a server recently, you could send encrypted data with your very first packet—no handshake at all, the request arriving at the same moment the connection opened.

Connection migration was the other innovation. A TCP connection is identified by four numbers—source IP, source port, destination IP, destination port—and changing any of them breaks the connection. Walk from WiFi to cellular, and the handshake starts over.

QUIC connections were identified by a connection ID, a random number chosen by the client. The remote host did not care which IP address the packets came from, as long as they carried the right ID. You could switch networks mid-download and the bytes would keep flowing.

By 2015, QUIC carried half of Google's traffic, and the company published the design and brought it to the IETF for standardization. The process took six years—committees, working groups, drafts, revisions—but in May 2021, QUIC became RFC 9000.[295] A year later, HTTP/3, the version of the web protocol built on QUIC, followed as RFC 9114.[222]

The next generation had arrived, thirty years late and from an unexpected direction. IPv6 promised security at the network layer and delivered slowly; QUIC promised speed at the transport layer and delivered overnight. Both tried to learn from the mistakes of their predecessors, and both succeeded in ways their designers didn't expect.

And both created new problems.

21.2. LAYERS OF TRUST

21.2.1. IPV6: BIGGER, SIMPLER, SUPPOSEDLY SECURE

The internet runs on layers, each relying on the one below: physical cables carry signals, Ethernet organizes local networks, IP routes packets globally, TCP provides reliable delivery, and UDP provides fast, connectionless transport. Break any layer, and everything above it falls.

IPv6 and QUIC operate at different layers, but they share a philosophy: build security into the design, not as an afterthought. One succeeded differently than expected; the other succeeded less than hoped.

An IPv4 address looks like this: `192.168.1.1`—four numbers, each between 0 and 255, separated by dots, thirty-two bits total. Every device you use shares these addresses with four billion others.

An IPv6 address looks like this: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`—e groups of four hexadecimal digits, separated by colons, one hundred twenty-eight bits total. That's enough addresses that every device you will ever own, and every device your grandchildren will own, could have millions to spare.

The expansion is staggering. Every square meter of Earth's surface could have 100 billion billion billion unique addresses. The number is not a prediction of need but an insurance policy against imagination. Whatever future devices require internet connectivity—appliances, implants, nanomachines—IPv6 can accommodate them. The address space will never run out. The question was whether the rest of the design would hold.

The header changed too. IPv4 headers could range from 20 to 60 bytes, but IPv6 headers are fixed at 40 bytes—no options in the base header (they're pushed into extension headers that routers can ignore), no header checksum (higher layers handle integrity), and no fragmentation by routers (only endpoints fragment, and they're expected to discover the path MTU first).

The fixed format makes routers faster: they examine exactly 40 bytes of every packet, apply the same logic, and make the same decisions. IPv6 traded flexibility for speed.

Extension headers carry what the base header cannot. The outer envelope (the base header) tells routers where the packet is going. The inner envelopes (extension headers) contain special instructions: hop-by-hop options that every router should examine, routing headers that specify intermediate destinations, fragment headers for oversized packets, authentication and encryption headers for security.

Here's the problem: "should examine" is not "must examine." Routers can skip extension headers they don't understand, and security devices often do. The simplicity of the base header pushed complexity into extensions—and complexity hides threats.

The word "supposed" matters.

If IPv6 was designed with security in mind, why isn't your internet traffic encrypted by default?

RFC 1883, the original IPv6 specification, required every implementation to support IPsec—the language was unambiguous: "IPv6 nodes must support IPsec." Security would be universal.

The vision was appealing: every packet, from any source to any destination, could be authenticated and encrypted, with no more forged source addresses and no more packet inspection by intermediate routers. End-to-end security at the network layer, transparent to applications.

The reality was different. IPsec requires key management—some way for two endpoints to agree on encryption keys before communicating securely—and the IETF standardized IKE, the Internet Key Exchange protocol, for this purpose. But IKE was complex, and most network administrators never touched it.

Worse, IPsec conflicted with middleboxes. Firewalls wanted to inspect traffic, NAT devices needed to rewrite headers, and load balancers had to see inside connections—but IPsec encryption blinded all of them. The choice became security or functionality, and functionality usually won.

RFC 6434, published in 2011, acknowledged the failure.[296] The new text read: "Support for IPsec… is recommended." The requirement had become a suggestion. The security intended to be built-in had become optional, and optional meant unused.

IPv6 did not deliver universal encryption. What it delivered was a larger address space and a cleaner header format. The security revolution would have to wait.

21.2.2. QUIC: SPEED THROUGH ENCRYPTION

QUIC learned from IPv6's failure. If optional security means no security, make security mandatory.

Every QUIC connection is encrypted—there is no plaintext mode, no fallback for debugging, no option to disable cryptography. TLS 1.3 is woven into the protocol itself, not layered on top, so by the time a QUIC connection is established, the data is already protected.

The integration runs deep. In TCP with TLS, the transport and security layers operate independently: the TCP handshake completes, then the TLS handshake begins, and the full sequence takes two or three round trips. During those round trips, an attacker watching the network learns the server you're connecting to, even if the content stays encrypted.

QUIC combines them: the initial packet contains both connection establishment and cryptographic key exchange. The response carries acceptance and certificates in a single reply. Data flows after a single round trip.

For returning visitors, QUIC improves further by storing session tickets: small pieces of data that allow a client to resume a previous connection without a full handshake. With a valid ticket, the client can send encrypted data in its first packet, which the remote host decrypts and processes immediately. Zero round trips to first data—faster than any unencrypted protocol can achieve.

But speed was only half the design. The other half was mobility.

TCP connections are fragile, each identified by four values: the client's IP address and port, the server's IP address and port. Change any and the connection breaks. Walk from your office WiFi to the parking lot's cell network, and every TCP connection dies—downloads restart, video buffers, state vanishes.

This was acceptable when computers sat on desks, but intolerable when computers ride in pockets. Mobile devices change networks constantly, and TCP forces them to start over each time.

QUIC connections are identified by a connection ID, a random number chosen by the client. The destination remembers the ID, not the address. When packets arrive from a new IP with the familiar ID, they are recognized—data keeps flowing, downloads continue, and the user notices nothing.

The design resists tracking because connection IDs can be rotated: the client periodically chooses a new ID and informs the server, so an observer sees different IDs over time, making it harder to follow a device across network changes. Privacy built into the transport layer—assuming the implementation does it correctly.

UDP provides almost nothing—a UDP packet is a source port, a destination port, a length, a checksum, and data, with no handshake, no acknowledgments, no retransmission, no ordering, no protection.

QUIC builds everything TCP provides—reliability, flow control, congestion management—on this minimal substrate. Each QUIC packet contains frames: self-describing units that carry stream data, acknowledgments, flow control updates, and connection management. The protocol numbers every byte, tracks what the receiver has seen, and retransmits what was lost.

The question is obvious: why bother? If QUIC reimplements TCP's features, what's the advantage?

The advantage is control. TCP lives in operating system kernels, and changing it requires convincing Microsoft, Apple, Google, and the Linux community to modify code that runs on billions of devices—a process that takes years, if it happens at all.

QUIC lives in applications: Google ships it in Chrome, Facebook ships it in their apps, and Cloudflare ships it in their edge servers. Updates deploy with new software releases rather than operating system

patches, so innovation that would take a decade in TCP takes a quarter in QUIC.

And because QUIC runs on UDP, it passes through networks that would interfere with novel protocols. Middleboxes understand UDP—they don't understand its contents, since QUIC packets are encrypted and opaque to inspection, but they know enough to forward them. A new transport protocol, running over the same substrate as DNS and gaming traffic, slips past equipment built for TCP and UDP alone.

The opacity cuts both ways: your ISP can't see what you're downloading, your employer can't monitor your connections, and neither can the security tools designed to protect the network.

21.2.3. WHAT THE NEXT GENERATION ASSUMES

IPv6 assumes that endpoints can handle security themselves: if they choose to encrypt with IPsec, the network will carry their packets, and if they don't, the network will carry those packets too. The specification provides capability, not requirement.

QUIC assumes that the network is hostile. Encryption is mandatory because the alternative—relying on intermediate devices to behave—has been tried and failed. The specification provides requirement, not capability.

Both rest on something more subtle: the belief that complexity is manageable. IPv6 has extension headers that can be chained, options within options, and interactions that even the RFC authors hadn't fully explored, while QUIC has multiple packet types, frame types, state machines, and encryption layers. The specifications run to hundreds of pages.

Complexity is the enemy of security—every additional feature is another surface for attack, every optional behavior another state to test.

The next-generation protocols aimed for more security and achieved more features. Whether they achieved more safety is a different question.

21.3. THE TRANSITION WINDOW

The attacks came not from the protocols' weaknesses but from their strengths.

IPv6 assumed abundant addresses—every device would have a unique, globally routable number—and the transition mechanisms expected coexistence, allowing IPv4 and IPv6 packets to pass through the same networks. Both design choices created new attack surfaces.

The first class of attacks exploited the transition. 6to4, one of the earliest tunneling mechanisms, allowed IPv6 packets to travel over IPv4 infrastructure by encapsulating them.[297] Any host with an IPv4 address could become a 6to4 gateway. The design expected gateways to be honest.

Attackers set up rogue relays, and when IPv6 traffic passed through, they could inspect it, modify it, or redirect it. The tunneled packets crossed network boundaries invisibly, bypassing firewalls configured only for IPv4, and security administrators discovered holes they hadn't known existed.

Teredo, Microsoft's tunneling protocol, was worse.[298] It was designed to work behind NAT devices, punching through firewalls that would have blocked direct connections. From a security perspective, punching through firewalls was the last thing anyone wanted. Teredo became a vector for internal network penetration, turning every Windows machine into a potential entry point.

The IETF deprecated both protocols. RFC 7526, published in 2015, declared that 6to4 "must not be used".[299] But deprecation is not disabling—legacy implementations remained, default configurations remained, and the mechanisms that were supposed to ease the transition instead prolonged the exposure.

Router Advertisement attacks exploited IPv6's design for address assignment. In IPv4, a device joining a network asks a DHCP server for an address, but in IPv6, a device can configure itself by listening for Router Advertisements and constructing its own address from the information provided.

The convenience was obvious, but the vulnerability was equally obvious: any device on the local network could send Router Advertisements. An attacker's laptop at an airport lounge could announce itself as the default gateway, and every other device would believe it, sending traffic through the attacker's machine.

This was ARP spoofing's cousin, transplanted to IPv6. The countermeasures—RA Guard, SEND, SeND—existed but required configuration, and most networks deployed IPv6 with defaults that accepted local devices at face value. The attacks followed.

Extension headers created a different problem. The IPv6 header was fixed at 40 bytes, but extension headers could add arbitrary complexity.

Attackers discovered that many network devices couldn't process extension headers correctly—the specifications were ambiguous and implementations varied. A carefully crafted chain of extension headers could crash routers, bypass access control lists, or evade intrusion detection systems. Simplicity in the base header had pushed complexity into extensions, and complexity created bugs.

In 2012, security researchers published a catalog of extension header attacks.[300] Firewalls that inspected only the base header missed threats hiding in extensions, while routers that tried to process deep

header chains ran out of memory. The extensions designed for flexibility had become extensions that couldn't be safely parsed.

QUIC's vulnerabilities were different in kind. QUIC was younger, designed by engineers who had watched IPv6's struggles and built security in from the start—requiring encryption, assuming hostile networks.

But QUIC had its own problems, growing from the same source as its benefits.

The encryption that protected user privacy also blinded network operators. Enterprises that monitored traffic for malware couldn't inspect QUIC connections. ISPs that managed congestion couldn't see inside encrypted streams.

This wasn't a bug—it was a design goal. The QUIC specification explicitly aimed to prevent middlebox interference by encrypting most of the packet header, leaving observers with little more than opaque bytes. Users gained privacy; operators lost control.

Some networks responded by blocking QUIC entirely, filtering UDP on common QUIC ports and forcing browsers to fall back to TCP. The system designed to traverse middleboxes found itself blocked by policy, not technology, and adoption slowed in corporate environments where network visibility was mandatory.

The 0-RTT feature attracted different concerns.[301] Sending data in the first packet saved latency but created replay risks: an attacker who captured a 0-RTT packet could resend it, potentially causing the server to process the same request twice. For idempotent requests like retrieving a web page, replay was harmless, but for state-changing requests like placing an order or transferring money, replay was dangerous.

The specification warned against non-idempotent 0-RTT requests. Many applications ignored the warning.

IPv6 tried to build security in—IPsec was mandatory, universal, baked into the specification—but the result was protocols nobody used,

security that existed in documents but not in deployments. When the requirement was weakened to a recommendation, the security vanished.

QUIC tried a different approach: make encryption mandatory with no opt-out. The result was successful deployment—billions of connections encrypted, latency reduced, security improved—but the mandatory encryption also broke legitimate monitoring and created new policy conflicts.

The lesson isn't that security is impossible—both protocols improved on their predecessors. IPv6 includes cryptographic address verification mechanisms that IPv4 lacks, and QUIC connections are encrypted by default.

The lesson is that security is never just technical. The best specification accomplishes nothing if nobody deploys it. The most secure encryption accomplishes nothing if it breaks the expectations networks are built on. And the transition from old systems to new creates windows of exposure that neither was designed to handle.

Steve Deering retired from Cisco by 2017, and Jim Roskind moved on to other roles after leaving Google.[286][302] The protocols they designed now carry substantial fractions of internet traffic—as of 2024, IPv6 approaching half of Google's traffic, QUIC already there for some providers.

The addresses will not run out, the handshakes have been shortened, and the encryption is, finally, becoming default. Three decades after the first warnings about IPv4 exhaustion, the internet is slowly acquiring the infrastructure its designers envisioned.

The transition continues. IPv4 and IPv6 run in parallel, their dual-stack configurations doubling complexity and attack surface. TCP and QUIC compete for connections; browsers fall back to TCP when UDP is blocked. IPv6 and QUIC delivered security—but as layers, not foundations.

DNS translates names to addresses, BGP routes packets between networks, TLS encrypts the data in transit, IPv6 provides the addresses, and QUIC provides the transport.

Each layer rests on expectations about the layers around it, and each weakness in one propagates to the others. The stack is not a wall but a chain, and chains break at their weakest links.

The addresses will not run out.

The crisis Deering had warned about in 1990 finally arrived in 2011—and the internet kept running. NAT bought time, IPv6 provided the long-term answer, and both systems now operate in parallel. As of 2024, roughly half of Google's traffic arrives over IPv6, a figure that would have seemed implausible when the last IPv4 block was allocated. The fix exists. The migration is not finished. Both are true.

That dual-stack coexistence is not what anyone planned. Deering designed IPv6 to replace IPv4, not to run alongside it indefinitely. Running both systems simultaneously doubles the configuration surface, doubles the security concerns, and doubles the opportunities for misconfiguration. IPv6 adds new attack vectors—router advertisement attacks, extension header exploits—while IPv4's familiar vulnerabilities remain. The network grew more complex before it grew more secure. QUIC followed a different path: it moved fast, encrypted everything by default, and reached billions of connections within a decade of its first deployment. But QUIC runs on top of the old protocols, not instead of them. The next generation arrived in layers, not foundations.

The handshakes still complete. The addresses still resolve. The packets still arrive.

And yet the attackers had already moved on. While engineers spent decades securing the protocols, someone discovered a shorter path:

stop exploiting the specifications. Compromise the source instead, and the infrastructure delivers the payload itself.

CHAPTER 22

THE CONVERGENCE

We are witnessing an attack
by a nation with top-tier offensive capabilities.

— Kevin Mandia, FireEye CEO, December 8, 2020

22.1. THE FIRST NETWORK WEAPON

In December 2020, a security analyst at FireEye reviewed log files from an internal investigation. Something did not add up: an employee's authentication credentials had been used to access systems from a device the security team had never seen. The access pattern looked legitimate, the timing looked legitimate, everything looked legitimate—except the employee had not done it.

FireEye's security operations center ran like a command bridge. The analysts had tracked nation-state hackers for years, helped governments respond to breaches, written the playbooks others used. Now they discovered that someone had been inside their own network for months.

The investigation expanded, and more anomalies surfaced. The attackers had moved carefully, touching only what they needed, pausing when activity might draw attention. They had stolen FireEye's proprietary penetration testing tools—the kind of exposure that would burn the operation's cover. But these attackers didn't care. They had something better already.

The trail led to an unexpected source: SolarWinds Orion, a network monitoring platform used by thousands of organizations worldwide.[303] FireEye ran Orion. So did the Treasury Department, the Pentagon, Microsoft, Intel, and Cisco. A routine software update, downloaded months earlier, had contained a backdoor. The update was digitally signed with SolarWinds' certificate. It passed every verification check. Antivirus didn't flag it. The company's own security team didn't flag it. No one flagged it, because the code that compromised them came from a vendor they trusted.

What follows is a story of all trust assumptions failing together.

22.1.1. CODE AS WEAPON

On June 17, 2010, a Belarusian antivirus analyst named Sergey Ulasen received a customer complaint: computers in Iran kept rebooting endlessly.[304] Blue screens, crashing, recovering, crashing again. Ulasen, team leader at VirusBlokAda in Minsk, began investigating.

What he found was unprecedented. The malware used four separate zero-day exploits: one attacked the Windows shell through infected USB drives, another targeted the print spooler, a third exploited a kernel flaw, and a fourth abused Windows task scheduling. Four zero-days in a single piece of malware was extraordinary—security researchers might discover one or two in a career.

On July 12, 2010, Ulasen posted a warning to a security forum. Within days, researchers worldwide were dissecting the code. What they found changed the nature of warfare.

Stuxnet wasn't designed to steal data or extort money—it was designed to destroy machinery. It targeted Siemens Step 7 industrial control systems running programmable logic controllers in factories and power plants. When Stuxnet found a system controlling variable-frequency drives spinning at certain speeds, it began its attack. It recorded normal operating data, then played that data back to operators while secretly altering commands to the machinery. The centrifuges spun too fast, then too slow, then too fast again, and the stress tore them apart—while the operators saw only normal readings.

The target was Iran's uranium enrichment facility at Natanz. The centrifuges spinning there were separating uranium isotopes for what Iran claimed was a civilian nuclear program and what Western intelligence services believed was a weapons program. Stuxnet destroyed approximately one thousand of the roughly nine thousand centrifuges at Natanz. It set Iran's nuclear program back by months, perhaps years.

According to reporting by David Sanger of *The New York Times*, the operation was codenamed Olympic Games.[305] Sanger's sources described it as beginning under President George W. Bush in 2006 and accelerating under President Barack Obama, with the CIA holding operational responsibility. The code itself was reportedly a collaboration between the NSA and Israel's Unit 8200, the signals intelligence unit of the Israel Defense Forces. The partnership combined American technical resources with Israeli intelligence on Iran's nuclear program.

Getting the malware into an air-gapped facility required human agents. According to later reports, a Dutch engineer named Erik van Sabben, working for Dutch intelligence, infiltrated Natanz in 2007 and introduced infected equipment through water pumps.[306] Van Sabben

died in a motorcycle accident in Dubai in January 2009, at thirty-six. Dutch authorities have never confirmed his role.

The supply chain Stuxnet exploited wasn't software—it was hardware. USB drives that engineers carried into the facility. Industrial controllers that factories trusted to operate honestly. Siemens software that assumed commands came from legitimate operators. Every layer of trust was violated, not by breaking the protocols, but by exploiting the assumption that physical access meant authorized access.

A programming error caused Stuxnet to spread beyond Natanz. The malware was meant to remain contained within the facility, but it escaped onto the broader internet, which is how Ulasen's customer in Iran encountered it. The world's first acknowledged network weapon was revealed by accident.

Stuxnet proved something that military planners had theorized but never demonstrated: software could destroy hardware. Code could reach across borders, penetrate air-gapped facilities, and cause physical damage. The attack combined signals intelligence, human intelligence, and network operations into something new. It was espionage, sabotage, and warfare, conducted through trust assumptions baked into industrial control systems.

The same year Stuxnet was discovered, the United States established Cyber Command, and the Pentagon declared cyberspace a domain of warfare alongside land, sea, air, and space. The protocols that connected the world had become weapons.

SolarWinds, a decade later, would prove that Stuxnet's techniques weren't unique. Supply chains could be compromised anywhere. The trust that organizations placed in their vendors, their updates, their tools—all of it was vulnerable. Where Stuxnet destroyed centrifuges, SolarWinds stole secrets. The first attack came from America. The second came for America.

We have spent twenty-one chapters tracing individual protocols and their vulnerabilities. ARP trusts that neighbors on a network segment won't lie. IP trusts that source addresses are accurate. TCP trusts that anyone who completes a handshake is who they claim to be. DNS trusts that the first response is correct. BGP trusts that route announcements are legitimate. TLS trusts that certificate authorities vouch only for verified identities.

But we've examined these exploits one at a time: a DNS poisoning here, a BGP hijack there. The modern internet doesn't work that way. Every connection traverses multiple protocols simultaneously. A single web request might touch BGP, IP, TCP, DNS, TLS, and HTTP. Each layer trusts the layer below it. Each assumes the data it receives hasn't been tampered with, that the identities it sees are genuine, that the infrastructure is sound.

The layering should have provided defense in depth. If one layer failed, another might catch the problem. A poisoned DNS response might be detected by certificate validation in TLS. Multiple barriers meant multiple chances to stop an attack.

Defense in depth became attack in depth. Each layer provided not redundancy but additional opportunity. Attackers who understood the full stack could chain vulnerabilities together, using weaknesses at one layer to bypass protections at another. An attacker who controlled the network could poison DNS, downgrade TLS, and intercept session tokens in a single operation.

And then there was the software supply chain.

Every organization runs software written by someone else. Web applications depend on frameworks. Frameworks depend on libraries. Libraries depend on other libraries. A modern JavaScript project might

pull in hundreds of packages from npm. Each dependency represents a trust decision.

The companies that wrote the dependencies trusted their developers. The developers trusted their laptops. The laptops trusted the operating systems. Somewhere in that chain, someone made an assumption about who could be trusted, and somewhere in that chain, that assumption could be violated.

SolarWinds Orion was a network monitoring platform. Its job was to watch everything: servers, routers, firewalls, applications. Administrators gave it credentials to access their entire infrastructure—they had to. A monitoring tool that can't see the network can't monitor it.

Every update was a trust decision. Every download assumed the code came from SolarWinds, that SolarWinds' build systems were secure, that no one had tampered with the bits in transit. The updates were signed, and signed code was safe code. That was the assumption.

22.2. THE STACK

Picture the internet as a tall building. The foundation is physical: cables, radio waves, electrons moving through silicon. Above that sits the link layer, handling communication between neighbors. Above that: the network layer, routing packets across the planet. Above that: the transport layer, managing connections. Above that: encryption. At the top: the applications you actually use—web browsers, email clients, everything you see on screen.

Each layer depends on the layers below, and each layer takes the layers below on faith. That faith is the exposure.

Table 22.1: The Trust Stack

Layer	Protocol	Trust Assumption
Application	HTTP/HTTPS (Ch. 18)	Server is legitimate
Security	TLS (Ch. 19)	CAs are trustworthy
Resolution	DNS (Ch. 7)	Responses aren't poisoned
Transport	TCP (Ch. 4)	Handshake equals identity
Network	IP (Ch. 4)	Source addresses are honest
Link	ARP/Ethernet (Ch. 5)	Neighbors don't lie

When you type a URL into your browser, you set off a cascade of trust decisions. The computer asks DNS for the server's IP address, opens a TCP connection, initiates a TLS handshake, and sends an HTTP request. Each step assumes the previous ones were not compromised.

Every step works, and every step has for decades. But if DNS returns a poisoned address, you might establish a TLS connection with an attacker's server. If the attacker has a fraudulent certificate, your browser shows the lock icon anyway. If packets are routed through a malicious network because of a BGP hijack, encryption won't save you—the attacker controls the endpoint.

Each layer can only protect against threats at its own level. TLS encrypts your traffic, but it cannot verify that DNS gave you the right address. Certificate validation checks that a certificate was issued by a trusted authority—yet that authority might be compromised. Network monitoring detects anomalous traffic, unless the monitoring software itself contains a backdoor.

The answer is always the same: add encryption, sign the data, use TLS everywhere.

Encryption helps—it helps enormously. But encryption protects the pipe between two computers, not the computers themselves.

An attacker who controls one endpoint can read your plaintext before it's encrypted or after it's decrypted. Malware on your machine captures your keystrokes before they reach the encrypted channel. A backdoor in the server software exfiltrates data through the same encrypted connection you trust.

The pipe is secure. The endpoints are on fire.

Supply chain attacks are devastating because they compromise the endpoint itself. The SolarWinds attackers didn't break TLS—they didn't need to. They were already inside.

Confidence is transitive. If you rely on Alice, and Alice relies on Bob, you're depending on Bob. You may never have heard of Bob.

How many dependencies does your application have? Not the ones you installed—the ones those installed. Your web framework depends on a router. The router depends on a parser. The parser depends on a utility library maintained by one person in Nebraska who hasn't updated it in two years.

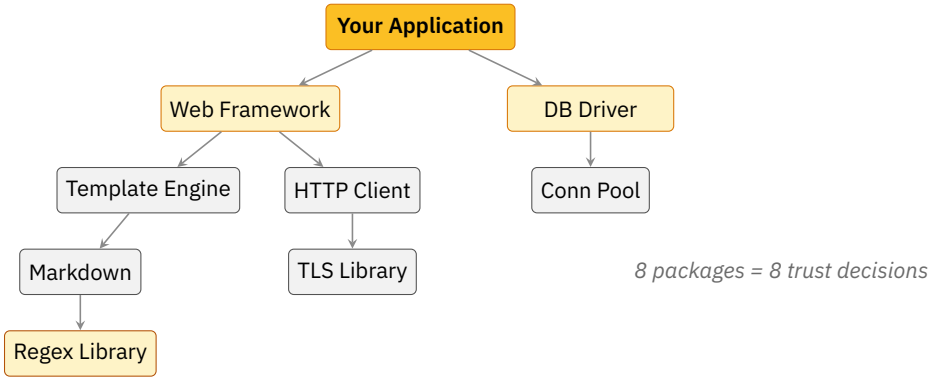


Figure 22.1: Software dependency tree. Each package represents a trust decision you inherit.

Eight dependencies for a simple application, and each represents a trust decision—you trust them all.

Now imagine: one maintainer's credentials get stolen, one repository accepts a malicious pull request, one package adds a single line of code that phones home with your environment variables.

You won't know—you'll run `npm update`, and the compromised code will arrive through the same channel as every legitimate update—signed, verified, trusted.

Your security depends on the security of people you've never met, organizations you've never vetted, code you've never reviewed. This isn't a failure of modern software development—it's the architecture, and the alternative of writing everything yourself doesn't exist.

There's a paradox at the heart of security monitoring.

To watch your network, you need access to your network: a monitoring platform needs credentials to query devices, an endpoint agent needs root privileges, a log aggregator needs to read everything.

From an attacker's perspective, this is a gift. Find the tool with credentials to the entire infrastructure, and you've found the skeleton key.

Compromise the monitoring platform and you've compromised everything it monitors—and worse, the platform is supposed to watch for compromise. If the platform itself is compromised, who watches the watcher?

SolarWinds Orion was exactly this kind of tool. It held the keys to everything it watched. The attackers didn't break in through the front door. They came in through the security camera.

Modern attacks don't exploit a single vulnerability. They chain weaknesses together, each step enabling the next.

A phishing email arrives—an invoice from a vendor, a calendar invite from HR. Someone clicks. The attachment exploits a browser vulnerability, and now the attacker has a foothold: a single compromised workstation.

From there, the attacker moves. A scheduled task ensures persistence; the implant survives reboots. Stolen credentials from the workstation unlock other systems. The attacker moves laterally, machine to machine, until reaching something valuable: a domain controller vulnerable to the Golden Ticket attacks from Chapter 10. Now the attacker owns the network.

The exfiltration is quiet—data flows out through DNS queries—legitimate-looking requests that carry stolen documents encoded in subdomain names. The system designed to resolve names becomes a covert channel.

Six steps, six assumptions violated. The email gateway assumed attachments from known senders were safe. The browser assumed JavaScript would stay sandboxed. The scheduler assumed only administrators create tasks. The domain controller assumed Kerberos tickets were legitimate. DNS assumed queries were just queries.

No single defense catches this, because each layer trusts the layers below. Sophisticated attackers test their tools against common security products before deployment—they move slowly, blend in, and know what triggers alerts so they can avoid it.

The SolarWinds attackers were patient: two weeks dormant after installation, command-and-control traffic only during business hours, domain names mimicking legitimate SolarWinds infrastructure, lateral movement only when they found high-value targets. They avoided systems running security software that might detect them.

Every choice was designed to evade the layered defenses organizations had built. The attackers understood the stack better than the defenders.

22.3. WHEN EVERYTHING BREAKS

On December 8, 2020, Kevin Mandia published a blog post that began: "We recently identified a sophisticated, nation-state-sponsored campaign".[307] The CEO of FireEye was telling the world that his company had been breached. The attackers had stolen FireEye's red team tools.

This was embarrassing, and it got worse.

FireEye's investigation revealed the attackers hadn't breached FireEye directly—they had breached SolarWinds and compromised its build system.[308]

A build system compiles source code into the software you actually run. Developers write code; the build system turns that code into an application. Compromise the build system, and you can inject malicious code into otherwise-legitimate software.

That is exactly what the attackers did. They inserted a backdoor called SUNBURST into the Orion platform.[309] The backdoor shipped in updates distributed between March and June 2020. SolarWinds signed these updates with its legitimate code-signing certificate. The updates were cryptographically validated—and they were also compromised.

Eighteen thousand organizations downloaded and installed the backdoored update.[310] Among them: the U.S. Treasury Department, the Commerce Department, the Department of Homeland Security, the State Department, the National Nuclear Security Administration, Microsoft, Intel, Cisco, and hundreds of other government agencies and Fortune 500 companies.[123]

The SUNBURST attack began with a build system compromise on September 4, 2019[311]. The backdoor shipped in March 2020 and went undetected until December—nine months of access. Fewer than 100 were actively exploited[312]. The U.S. government attributed the attack to APT29, also called Cozy Bear[313]. This Russian state-sponsored group,

associated with the SVR foreign intelligence service, had used the Golden Ticket technique from Chapter 10. This time, they did not need to forge tickets. They had legitimate credentials from a trusted vendor.

The backdoor was carefully designed. After installation, it remained dormant for up to two weeks.[314] Then it began communicating with command-and-control servers, using domain names generated by an algorithm. The traffic was encrypted and designed to look like legitimate Orion telemetry. The attackers checked each infected system before deciding whether to proceed—most victims were ignored, and fewer than a hundred were selected for deeper exploitation.

The selection criteria were invisible to victims. The attackers looked at domain names and internal network configurations—government agencies, defense contractors, technology companies. These were worth the risk; random businesses were not.

Those selected received a second-stage payload, and the attackers created accounts and stole credentials. Then came lateral movement—using one compromised system to access others on the same network. The attackers accessed email, read documents, and exfiltrated data, all without triggering alerts for months.

The tradecraft was exceptional. The attackers used IP addresses from the same country as their targets, making traffic look local, and they mimicked legitimate administrator behavior while avoiding actions that would generate logs. When they needed to run commands, they used PowerShell in a way that didn't trigger common detection rules, and they stored stolen credentials in temporary files that were deleted after use. They were ghosts in systems designed to catch ghosts.

Testifying before Congress in February 2021, Mandia put it bluntly: "This was the varsity team on offense, for sure." The attackers had chosen their entry point deliberately. A network monitoring platform

was, Mandia told the Senate Intelligence Committee, "the last place you'd look." The irony cut deep. The tool organizations deployed to watch for intruders had become the intruder.

The legal disclosure was dryer. On December 14, 2020, SolarWinds filed a Form 8-K with the SEC stating it had "been made aware of a cyberattack that inserted a vulnerability within its Orion monitoring products which, if present and activated, could potentially allow an attacker to compromise the server on which the Orion products run."[310] The gap between "varsity team" and "potentially allow" is the gap between the operational reality of a breach and the regulatory language used to describe it.

Every organization that ran Orion had to assume it was compromised. Incident response teams worked through the holidays. The full scope of the damage may never be known. The attackers had access to email systems, cloud environments, internal networks. What they read, what they copied, what they learned—much was classified, and the government has revealed only fragments.

Every trust assumption this book has examined was present in the SolarWinds attack. The victim organizations relied on their software vendor. They assumed updates were legitimate. They believed code signing meant code was safe. They expected their monitoring tools would detect anomalies. SolarWinds, in turn, relied on its build system, which relied on its developers' workstations. Somewhere in that chain, someone trusted the wrong thing.

The attackers understood transitive dependency better than their victims. They did not attack one organization at a time—they attacked SolarWinds and let the software distribution channel do the rest. Every customer who installed updates trusted a compromised source without knowing it. The trust was invisible, automatic, routine.

Here lies the convergence: the point where all the assumptions meet, where the stack becomes a chain, where a single break cascades

through every layer. The protocols we've studied are not independent. They are layered into infrastructure that underpins modern life. Break one layer and the failure propagates upward.

In the months after the SolarWinds disclosure, the U.S. government issued an executive order on cybersecurity.[315] It mandated software bills of materials—inventories of every component in a piece of software, so that buyers can assess supply chain risk. It required vendors to meet security standards if they wanted to sell to the government. It pushed for zero-trust architectures, where no user and no system is trusted by default, where every access must be verified.

These are reasonable responses, but they are also admissions. The old model—rely on your vendors, accept your updates, depend on your tools—is no longer adequate. The internet was built on good faith, and that faith was exploited.

The standards remain: DNS still resolves names, BGP still routes packets, TLS still encrypts connections. The infrastructure that powers the modern world is the same infrastructure that enabled the attack, and we cannot rebuild it—we can only patch, monitor, and hope that the patches hold.

Rough consensus built systems that assumed good faith. Rough consensus is now scrambling to patch them.

SolarWinds patched the backdoor. The compromised certificates were revoked. Affected organizations rebuilt their networks, replaced their credentials, and audited their logs. The executive order mandated software bills of materials; zero-trust architectures replaced perimeter defenses. The industry learned from disaster, as it always does.

And then SolarWinds kept appearing in the breach reports.

Seven SolarWinds vulnerabilities now appear in CISA's catalog of actively exploited flaws.[316] One—CVE-2021-35211, in the Serv-U file transfer product—has been used in ransomware campaigns. In August 2024, CISA added CVE-2024-28986, a deserialization weakness in Web Help Desk.

Deserialization is how a program reconstructs data from a stored format. When you save a document and reopen it, the software deserializes the saved data back into something you can work with. The process is routine, but the danger is this: if an attacker can control the stored data, they can inject malicious code that runs when the program reconstructs it. A deserialization flaw turns a data file into a weapon. The Web Help Desk weakness was exactly this kind of opening.

Two months later, CISA added another SolarWinds entry: CVE-2024-28987, hardcoded credentials in the same product. Hardcoded credentials—a security failure so elementary that it appears in the first week of any security course. Four years after SUNBURST should have made such oversights unthinkable, the supply chain vendor at the center of the most consequential breach in a decade shipped software with a password baked into the code.

The supply chain remains vulnerable because the dependency model hasn't changed: we still run software written by others, still rely on vendors to secure their build systems, still update automatically. The alternative of vetting every line of code in every dependency doesn't scale. The reliance was never optional—it was the architecture.

But identifying weaknesses requires its own supply chain. The CVE program—the system that assigns identifiers to security flaws, the common language that lets defenders know what to patch—is a supply chain of information. In April 2025, that supply chain nearly collapsed.[317] CISA failed to renew its contract with MITRE, the nonprofit that had operated the CVE program since 1999. Seventeen hours before the shutdown deadline, the agency found emergency funding. Jen

Easterly, the CISA director, compared the near-miss to "tearing out the card catalog from every library at once."

Three months later, Easterly stepped down. She was one of more than 2,500 CISA employees who departed as the agency lost roughly two-thirds of its workforce.[282] The CVE backlog climbed past 25,000 unprocessed entries.

The irony is structural. SolarWinds was a supply chain attack on code. The CVE program is a supply chain of knowledge about attacks. Both depend on institutions and funding, both rest on the expectation that someone, somewhere, is maintaining the infrastructure that everyone else relies on. The supply chain of vulnerability information depends on a supply chain of institutional capacity—and that chain, too, can break.

The infrastructure carries on, because the infrastructure is all we have. There is no alternative internet, no clean-room replacement, no means to start over without breaking everything that depends on what exists.

The next attack is coming—the defenders know this, and the attackers know this. The infrastructure knows nothing; it simply runs, processing packets, assuming good faith by default, doing what it was designed to do forty years ago.

The infrastructure was never designed for this.

The routing protocols were designed for research networks. The certificate system was designed for e-commerce. The monitoring tools were designed for availability, not for adversaries who would compromise them specifically because they were trusted everywhere. No single architect sat down and planned a system that could carry five billion users through nation-state attacks, ransomware campaigns,

and supply chain compromises. What exists is the accumulated result of ten thousand separate decisions, each reasonable at the time, each now load-bearing in ways its authors never anticipated.

The system survives because people keep showing up. Someone patches the router at 3 AM when the BGP session drops. Someone responds to the incident ticket on a Sunday. Someone reviews the dependency pull request that introduces a subtle backdoor, and catches it before it ships. The maintainers—underfunded, understaffed, largely invisible—are the reason the infrastructure holds. They are not the architects. They didn't design the cathedral. They are the crew that keeps it standing: noticing the cracks, mixing the mortar, deciding every day that the work is worth doing.

The CVE database still processes vulnerabilities. The CERTs still coordinate. The IETF still meets, still argues, still reaches rough consensus and ships running code. The protocols that were designed for colleagues now carry the traffic of strangers—and the strangers, mostly, don't attack. The maintenance crew keeps showing up. The cathedral stands.

CHAPTER 23

THE HONOR SYSTEM

The Robots Exclusion Protocol
is not a substitute for valid content security measures.
Listing paths in the robots.txt file exposes them publicly
and thus makes the paths discoverable.

— *RFC 9309, Martijn Koster et al., 2022*

23.1. KOSTER'S PROPOSAL

The newest trust assumption to fail is the oldest in spirit. No exploits required. No code signing to forge. Just a plain text file that says "please don't"—and a world that stopped listening.

Martijn Koster stared at his server logs and watched his machine die.

It was September 1993. Something was crawling through his website at Nexor in Nottingham, England. The requests came in bursts, one

per second, following hyperlinks with mechanical persistence. Koster's Perl-based web server couldn't keep up. Response times climbed. Other users waited. The logs showed the same pattern: a program from MIT, methodically downloading every page it could find.

The program was called the World Wide Web Wanderer.[318] Matthew Gray, an undergraduate physics student at MIT, had created it that spring to answer a simple question: how big was the web? The Wanderer visited a site, recorded its pages, followed every link, and continued until it ran out of territory. Gray ran it monthly and published his counts. In June 1993, his crawler found 130 websites. By September, the number had tripled.

The Wanderer wasn't malicious—Gray was genuinely curious about web growth, and his program produced valuable data. But early versions had a flaw: they didn't know when to stop. The Wanderer would return to the same sites repeatedly, request the same pages, follow the same links. Gray's curiosity was overwhelming Koster's infrastructure.

Koster wasn't alone. Across the nascent web, server administrators were discovering the same problem. The Wanderer had company: WebCrawler, the World Wide Web Worm, JumpStation, and a growing population of automated agents that followed links without asking permission. Among the programs hammering Koster's server was one run by Charles Stross, a programmer who would later become a celebrated science fiction author. Each crawler served a purpose; each consumed resources. And server operators couldn't refuse.

Koster understood the problem better than most. He wasn't just running a web server; he was building an early search engine. ALIWEB, which he announced in November 1993[319], took a different approach. Instead of visiting every site automatically, ALIWEB asked webmasters to submit descriptions of their content. ALIWEB indexed only what webmasters chose to share.

The approach was elegant but doomed. Webmasters didn't want to maintain submission files. They wanted to be discovered automatically. The crawlers won because crawlers did the work. But ALIWEB's philosophy stuck with Koster: webmasters should have some control over what robots accessed.

In February 1994, Koster posted to the www-talk mailing list. He had been collecting information about active robots and their behavior. He proposed something simple: a convention, not a standard, for communicating with automated agents.

He originally called it "A Standard for Robot Exclusion." Early drafts bore a more direct name: `RobotsNotWanted.txt`—inspired by Charles Stross, whose experimental crawler had accidentally launched a denial-of-service attack against Koster's machine.

The proposal was simple: place a plain text file at the root of your website, in a predictable location, with a predictable name. Call it `robots.txt`. In that file, list the areas you don't want robots to access. Any well-behaved crawler would check for this file before visiting a site and respect its instructions.

The format was deliberately minimal:

```
User-agent: *
Disallow: /cgi-bin/
Disallow: /private/
Disallow: /tmp/
```

Anyone with a text editor could create one. Any programmer could parse it. No special software, no protocol extensions, no authentication.

Koster's approach echoed Steve Crocker's RFC 1 twenty-five years earlier. Both men faced the same problem: coordinating a community

without central authority. Crocker, the graduate student in Chapter 1, had titled his document "Request for Comments" rather than "Specification" because he didn't want to seem presumptuous. Koster called his proposal "a convention, not a standard" for the same reason. Both trusted that colleagues would cooperate because cooperation made the system work. Both built infrastructure that outlasted the community that created it.

The responses came quickly. The web in 1994 was small enough that most of the people building it had met at conferences, or knew someone who had. Tim Berners-Lee, the web's inventor, participated in the discussion. So did Brian Pinkerton, creator of WebCrawler, and other crawler authors who recognized that their programs were causing problems.

By June 1994, Koster published "A Standard for Robot Exclusion".[320] Not a standard in any formal sense—no working group had debated it, no committee had approved it. Just a proposal posted to a mailing list that solved a problem everyone shared.

The crawler authors adopted it. They had no reason not to. Respecting `robots.txt` cost them almost nothing. Ignoring it earned blocked IP addresses and angry emails. The web's early culture was small and interconnected. Reputation mattered. A crawler that deliberately violated `robots.txt` would be named and shamed.

By July, most active robots supported the convention or had promised to. WebCrawler checked for `robots.txt`, Lycos checked, and AltaVista, when it launched in December 1995[321], checked from the beginning. Google would check too.

Koster had created something rare: a protocol with zero enforcement, relying entirely on voluntary compliance, that would govern the relationship between websites and automated agents for nearly three decades. No one was forced to respect `robots.txt`. Everyone chose to.

The web grew. By 1996, there were hundreds of thousands of sites. By 2000, millions. Search engines became essential infrastructure, indexing the web's content and making it discoverable. Google's PageRank algorithm, which ranked pages by the links pointing to them, transformed web search from a curiosity into a necessity. And through all of this growth, `robots.txt` held.

It held because the incentives aligned. Search engines needed good relationships with the sites they indexed. A search engine that ignored `robots.txt` would find itself blocked, its crawler's IP addresses rejected, its index degraded. Publishers needed search engines to drive traffic. Both sides benefited from cooperation. The gentleman's agreement worked because everyone had something to lose by breaking it.

Attempts to formalize `robots.txt` as a proper internet standard came and went—proposals circulated through the Internet Engineering Task Force in 1996, again in 2007, again in 2019, but none achieved consensus. The convention worked fine without a standard.

The answer came in 2022, twenty-eight years after Koster's proposal, when `robots.txt` finally became RFC 9309.[322] Koster was listed as first author, joined by three engineers from Google. The document formalized what everyone already knew: the syntax, the semantics, the expected behavior.

But by the time the standard was published, the social contract was already dissolving.

23.2. POLITE REQUESTS

23.2.1. HOW ROBOTS.TXT WORKS

Think of `robots.txt` as a sign on your front lawn that says "Please don't walk on the grass." Polite people respect it. Nothing stops anyone from ignoring it. The request works only because most people are polite.

When a crawler visits your website, it first requests a specific file: `/robots.txt`, located at the root. If the file exists, the crawler reads it. If it contains instructions, the crawler follows them. That is, if the crawler is polite.

Nothing in the protocol forces compliance or verifies identity. The crawler announces who it is. You believe it—or you don't. Either way, you can't stop it from looking.

The format is simple enough that anyone can read it—and that's the problem. Each entry names a crawler and lists paths it shouldn't access:

```
User-agent: Googlebot
Disallow: /admin/
Disallow: /private/
Disallow: /tmp/

User-agent: *
Disallow: /cgi-bin/
```

The asterisk is a wildcard, meaning "all crawlers." Specific entries override the wildcard. In this example, Google's crawler is blocked from `/admin/`, `/private/`, and `/tmp/`. All other crawlers are blocked only from `/cgi-bin/`.

A polite crawler stays away. Not every visitor is polite.

What if you want to allow one file in a blocked directory? RFC 9309 added an Allow directive:

```
User-agent: *
Disallow: /directory/
Allow: /directory/public.html
```

This blocks everything under `/directory/` except `public.html`—the most specific rule wins, and Allow takes precedence when specificity is equal.

The logic presumes the crawler cares about specificity, that it's parsing your file in good faith. Neither premise is enforced.

The crawler reads your rules and chooses whether to follow them. It can impersonate Googlebot. It can claim to be a browser. Most servers don't verify. The file is advisory, not authoritative—a preference expressed, not a permission enforced.

Worse: the `robots.txt` file itself is public. Listing `/secret-project/` in your Disallow rules advertises that `/secret-project/` exists. You've drawn a map and labeled the treasure.

Courts have sometimes considered `robots.txt` when evaluating unauthorized access claims, but the file creates no enforceable rights. RFC 9309 is blunt about this: the protocol cannot replace actual security.

The protocol's entire foundation is a single expectation: crawlers will behave honestly. They will identify themselves accurately. They will check for `robots.txt`. They will respect the rules they find. They will do this because it's polite, because the web ecosystem works better when everyone cooperates.

The pattern echoes SMTP, the email protocol from Chapter 6. Both accept unverified headers—User-Agent strings for robots.txt, From addresses for email. Both were designed for communities where lying had social costs. Both broke when strangers arrived with different incentives.

For nearly three decades, this expectation held. Not perfectly—there were always rogue scrapers, bots that claimed false identities. But the major players cooperated. Google, Bing, Yahoo, DuckDuckGo all respected `robots.txt`. They had to. Their business models depended on relationships with publishers. A search engine that ignored opt-out signals would find itself excluded from the content it needed to index.

Publishers built additional defenses for individual pages rather than entire directories. A parallel mechanism lets publishers mark specific pages:

```
<meta name="robots" content="noindex, nofollow">
```

The robots META tag sits inside a page's HTML, where `noindex` means "don't remember this page exists" and `nofollow` means "don't follow links from here to anywhere else." Publishers who wanted belt-and-suspenders protection would use both: `robots.txt` to block crawling at the gate, META tags to catch anything that slipped through.

Neither mechanism has teeth—both depend on the crawler reading your code in good faith, parsing your preferences, and choosing to comply. The tag is a whisper in the ear of a machine that may not be listening.

By 2020, Googlebot handled the lion's share of web indexing. Below it came specialty crawlers: SEO tools, archive services, academic researchers. Each crawler identified itself with a User-Agent string—a text label included with every HTTP request. Googlebot announced

itself as Googlebot. Bingbot as Bingbot. The system had an implicit social contract: crawlers that wanted legitimacy announced themselves honestly; publishers who wanted indexing let them through.

The arrangement worked because the stakes were manageable. Indexing a website cost something, but not much. No search engine would risk its reputation to access one publisher's content. The ecosystem remained stable because defection wasn't profitable.

Then the stakes changed.

23.3. THE MACHINES STOPPED ASKING

Training GPT-3 required approximately 300 billion tokens of text. GPT-4 likely required far more. No one knows the exact numbers—OpenAI hasn't published them. What we know is that training a large language model demands a corpus of human writing larger than any ever assembled. The obvious source was the web itself.

The web in 2020 contained over a billion websites and trillions of pages. Much was freely accessible. Much was protected by `robots.txt`. For decades, the distinction had been clear: if you blocked a crawler, it wouldn't access your content.

AI companies had different priorities.

The training data for early language models came from Common Crawl[323], a nonprofit foundation that had been archiving the web since 2008. Common Crawl operated CCBot, a crawler that respected `robots.txt`, and made its archives freely available. The corpus was vast but imperfect, shaped by whatever sites CCBot could access.

OpenAI, Anthropic, Google, and others scraped the web directly. They built custom crawlers, accessed content at scale, assembled

training sets of unprecedented size. Some of this scraping respected `robots.txt`; some didn't. The details remain murky because the companies didn't disclose them.

By the time publishers realized what was happening, the moment had passed. The training data had been collected. The models had been trained. ChatGPT launched in November 2022[324] and became the fastest-growing consumer application in history.[325] The content that powered it was already inside.

In August 2023, OpenAI published documentation for GPTBot[326], its web crawler. The documentation explained how publishers could block it:

```
User-agent: GPTBot
Disallow: /
```

The response was immediate. Within weeks, Amazon, The New York Times, CNN, and Wikihow had blocked the crawler. By November, a third of the top 1,000 websites had opted out. The social contract was dissolving in real time.

But opting out of future crawling couldn't undo past crawling. GPT-4 was already trained. ChatGPT was already deployed. The content was already inside the model, embedded in its parameters, impossible to extract or remove. OpenAI's offer of a `robots.txt` opt-out was like offering a lock for a door that had already been robbed.

Google followed with Google-Extended[327], a new User-Agent string separate from its search crawler. Publishers could block Google-Extended while still allowing Googlebot for search indexing. Anthropic documented its crawlers. Meta announced Meta-ExternalAgent. Each company created its own opt-out mechanism.

Publishers scrambled to keep up. The list of User-Agent strings to block grew: GPTBot, Google-Extended, CCBot, ClaudeBot, anthropic-ai, Claude-Web, PerplexityBot, Bytespider, Meta-ExternalAgent. Each required a separate Disallow line. The file Martijn Koster designed to be simple was becoming a catalog of denial.

The New York Times had invested heavily in digital journalism. Its reporting required months of investigation, teams of reporters, legal review, fact-checking. The resulting articles sat behind a paywall, generating subscription revenue that funded the next investigation.

In December 2023, the Times sued OpenAI and Microsoft for copyright infringement.[328] The lawsuit alleged that OpenAI had used Times content without permission, that ChatGPT could reproduce substantial portions of articles verbatim, and that AI products let users bypass the paywall.

The filing included examples. Prompts caused ChatGPT to output paragraphs matching Times articles word for word. Responses reproduced investigative journalism, recipes, opinion columns. The model had memorized parts of its training data—and some of that data was copyrighted content.

The Times had blocked GPTBot, but the training data was already inside the model.

OpenAI's defense centered on fair use. Training a model wasn't copying in the traditional sense, they argued. The model learned patterns, not text. The output changed the input beyond recognition. AI training, like Google's book-scanning project before it, represented a new kind of use that copyright law should accommodate.

The case remains unresolved. Courts in 2024 rejected OpenAI's motion to dismiss, allowing the lawsuit to proceed. Similar cases followed: authors suing over book excerpts in training data, artists suing over image generators trained on their work, publishers suing over scraped archives.

The legal questions are genuinely difficult: Does training constitute copying? Is AI output changed enough to qualify as fair use? What does it mean to "reproduce" content when the reproduction emerges from billions of parameters rather than a stored file?

But the `robots.txt` question is simpler—publishers said no, and AI companies scraped anyway. The polite convention shattered because the stakes became too high.

In June 2024, journalists at WIRED[329] and independent researcher Robb Knight[330] exposed something worse. Perplexity AI, a search startup valued at over a billion dollars, was circumventing `robots.txt`.

The investigation revealed a pattern. According to WIRED and Knight's findings, publishers had blocked PerplexityBot in their `robots.txt` files, yet content from those publishers continued appearing in Perplexity's responses. When the investigators dug deeper, they found crawlers using standard browser User-Agent strings—masquerading as Chrome on Windows rather than identifying as bots—accessing content that PerplexityBot was blocked from reaching.

The crawlers used different `IP` addresses, different ASNs, different identifiers. When one was blocked, another appeared. Based on the investigation's findings, Perplexity wasn't just ignoring the sign on the lawn—it was wearing disguises.

Cloudflare, the infrastructure company that provides security and performance services to millions of websites, de-listed Perplexity as a verified bot, removing the protections that legitimate crawlers receive.[331] They published a blog post documenting the violations. They updated their systems to detect and block Perplexity's stealth crawlers.

Perplexity's CEO responded with a telling statement[332]: "The Robots Exclusion Protocol is not a legal framework." He was right. It never was. That was the whole point.

Koster never intended a legal framework—just a handshake among colleagues. The convention worked because everyone agreed to the same rules, because defection was unprofitable, because the web was built by people who knew each other and cared about reputation.

The AI gold rush ended that era. Training data became worth billions of dollars. First-mover advantage in language models translated to market dominance. The incentives that once encouraged compliance now encouraged defection.

RFC 9309 contains a section called "Security Considerations." It's brief. The protocol, it notes, provides no access control. Listing paths in the file makes them publicly discoverable. The mechanism is advisory only.

The section doesn't mention AI. It was written before the training data wars, before the lawsuits, before the collapse of the social contract it documents. RFC 9309 describes a protocol built on voluntary compliance at the moment compliance was becoming optional.

Koster's original proposal, posted to a mailing list in 1994, expected that crawler authors would behave responsibly. They did, for a while. The web grew from thousands of sites to billions. Search engines indexed it all. The unwritten rules scaled because the incentives remained aligned.

What changed wasn't the technology. The syntax of `robots.txt` in 2024 is nearly identical to Koster's 1994 proposal. What changed was the value of the content. When crawling a site meant indexing it for search, the value was modest and shared. Publishers got traffic; search engines got content. When crawling means training a model that generates competing content, the value concentrates on one side.

AI companies don't need ongoing relationships with publishers the way search engines do. Train once, and the model exists forever. The content is embedded, transformed, unreachable. There's no traffic to

send back, no mutual benefit to preserve. The social contract rested on mutual dependence. AI broke the symmetry.

New protocols are emerging. TDMRep[333], the Text and Data Mining Reservation Protocol, lets publishers express licensing terms in machine-readable form. The EU AI Act requires compliance with opt-out signals for training data. Proposals for `ai.txt` and `learners.txt` would create AI-specific exclusion mechanisms.

None solve the fundamental problem. Technical opt-outs work only when everyone respects them. Legal frameworks work only when courts can enforce them across borders. Koster's original insight remains valid: the web lacks a central authority. Compliance must be voluntary. The only question is whether the incentives align.

For a generation, they did—publishers shared content, crawlers indexed it, users found what they needed, and the ecosystem thrived on cooperation.

The AI era creates new tensions. The content becomes more valuable than the relationship. Training a model once eliminates the need for ongoing access. The companies doing the scraping are worth hundreds of billions of dollars; the content they scrape funds the journalism that holds them accountable.

`robots.txt` doesn't answer these questions. A text file can express a preference. It cannot enforce one. Polite requests worked when politeness was enough. When billions of dollars are at stake, a sign on the lawn may not suffice.

Martijn Koster, the Dutch engineer who proposed `robots.txt` from his office in Nottingham, saw his informal standard become an RFC in 2022. His name appears first in the author list. The document describes a protocol that expects good faith—crawlers identifying themselves honestly, everyone following rules they could ignore.

RFC 9309 formalized a social norm that governed the web for a generation. It is also a historical document, describing a world that was already disappearing as the RFC was published.

The web was built on trust. `robots.txt` is that trust made visible, a plain text file that works only because everyone agrees it should. For twenty-eight years, they agreed. The honor system held.

Then the AI companies came, and the sign on the lawn stopped meaning anything.

The CVE program is another honor system—in 2025, it nearly collapsed when government funding lapsed (Chapter 22), and the vulnerability database that underpins modern security still runs on borrowed time.

The CVE database tracks code flaws—memory corruption, authentication bypasses, input validation failures—but it has no category for broken promises. When AI companies scraped content from sites that had blocked their crawlers, no CVE was issued; when Perplexity reportedly wore disguises to avoid detection, no severity score was assigned. The vulnerability tracking system cannot track its own kind of vulnerability: the kind where the protocol works exactly as designed, and the social contract fails anyway.

The lawsuits are working their way through courts. The EU AI Act requires opt-out compliance. Technical countermeasures evolve—rate limiting, fingerprinting, blocking entire IP ranges—as the web fights back, haltingly and imperfectly. Some crawlers still obey. Googlebot still respects the rules. The infrastructure of search—billions of pages

indexed, trillions of links followed—still rests on Koster's thirty-year-old convention.

Most websites still have a `robots.txt` file. They still specify which crawlers are welcome. They still believe—despite everything—that declaring their preferences matters.

This chapter has no proper ending. The story isn't finished.

You are reading these words because someone wrote them. The writing took time, research, care. Somewhere, right now, a crawler is deciding whether to index this page. Somewhere, a model is being trained on text scraped from a site that asked not to be scraped. The outcome—multiplied across billions of pages, trillions of requests—will shape what the web becomes.

Every protocol in this book was designed by people who knew each other. They expected their users to be colleagues. Those expectations became vulnerabilities when strangers arrived. You are not a stranger to this moment. You are inside it.

Every time you search the web, you benefit from systems built on good faith. Every time you ask an AI a question, you interact with models trained on content that may not have been freely given. The honor system is breaking. What replaces it depends on choices being made now—by engineers, by courts, by companies, by us.

Martijn Koster put a sign on his lawn in 1994. The sign still stands. Whether it means anything is the question this book cannot answer.

That's up to you.

EPILOGUE: LO AND BEHOLD

The sign on the lawn still stands. Whether it means anything—that's up to us.

To understand what we're choosing, return to where it started.

Boelter Hall still stands on the UCLA campus. Room 3420 is an ordinary laboratory now, indistinguishable from a hundred others in the building. The IMP is gone, shipped to the Smithsonian decades ago. The Model 33 Teletype is gone. The cables that once connected this room to Menlo Park are gone.

Steve Crocker sometimes returns here. He is in his eighties now. He still attends IETF meetings, still participates in discussions about internet governance, still watches the network he helped build struggle with problems that did not exist when he wrote RFC 1.[4] He holds no formal position over the internet's technical direction. No one does. That was always the point.

In October 1969, Charley Kline sat in this room and typed `L-O-G`.[334] The system crashed after two letters. The first message on the ARPANET was `LO`.

The graduate students could not have known what they were starting. They were solving a practical problem: how to share expensive

computers across research labs. The assumptions they made—trust the sender, believe the address, accept the packet—were right for the problem they faced. Fifty-five years later, a text file asking crawlers to stay away would fail for the same reason. It assumed everyone was playing by the same rules.

Those early design choices were generative. Because anyone could participate, anyone did. The network grew from four nodes to five billion users. The colleagues who fit around a conference table became anonymous strangers, state actors, criminal syndicates. The intimacy that made openness safe disappeared—but the openness remained.

And yet.

The costs are real, and they keep mounting.

The Morris Worm brought the network to its knees in 1988.[10] Cliff Stoll traced hackers through his lab for months before anyone believed him.[18] Mitnick spoofed packets to hijack Shimomura's machine.[39] The Kaminsky vulnerability lurked in DNS for twenty-five years before anyone noticed.[77] Pakistan accidentally swallowed YouTube's traffic through a BGP misconfiguration.[159] DigiNotar's compromised certificates enabled surveillance of Iranian dissidents.[335] SolarWinds delivered malware through the software supply chain itself.[122] And now the AI training data wars—content scraped from sites that asked not to be scraped.

Business email compromise costs organizations billions annually.[336] The phishing email that compromises your network arrives with a sender address spoofed to match your CEO. Nation-states route traffic through their borders to read what passes. Ransomware encrypts hospitals. Botnets control millions of infected machines.

The systems expected cooperation, but the attackers did not cooperate.

The watchers are falling behind—as of mid-2025, the CVE database carries a backlog of 25,000 unprocessed vulnerabilities, and the costs accumulate faster than anyone can count them.

But the gains are real too.

Global communication, instant and nearly free. Wikipedia, built by volunteers who will never meet. The ability for anyone to publish, to organize, to find an audience. Dissidents coordinating across borders. Open-source software developed by strangers in different time zones. Video calls with grandchildren on another continent.

The Arab Spring organized on social media. So did the riots. Linux grew from a Finnish student's hobby project to running most of the world's servers.[337] So did malware distribution networks. Every startup that disrupted an industry depended on the same openness that enables fraud.

The internet isn't good or bad—it's large, and it contains multitudes.

The openness was the feature. You cannot separate them. The same permissiveness that enabled the attacks also enabled the growth. A network that demanded verification at every step could not have grown this way. The World Wide Web emerged from a physics lab because Tim Berners-Lee could build it without asking permission.[209] Permission was never required. That was the point. That was also the problem.

Right now, as your eyes move across this page, the internet is running.

You type a URL. Here is what happens:

Your browser asks a DNS resolver for the address. The resolver does not know, so it asks a root server—one of thirteen logical addresses

that form the internet's master directory, scattered across hundreds of machines on six continents. The root points to the `.com` servers. The `.com` servers point to the domain's nameserver. The nameserver returns an `IP` address. Three hops, each accepting the referral it received.

Your browser opens a `TCP` connection. Three packets: SYN, SYN-ACK, ACK.[34] The handshake Cerf and Kahn designed in the 1970s[32], still executing billions of times per second. Your address claims to be yours, and the destination believes it.

Your browser initiates a `TLS` handshake. Certificates are exchanged. A chain of trust extends from the server's certificate to a root CA embedded in your browser years ago, by people you have never met. If every signature verifies, an encrypted tunnel forms. You see a lock icon.

Your `HTTP` request travels through the tunnel. It routes through autonomous systems—the internet's neighborhoods—a dozen of them, perhaps, each honoring `BGP` announcements from its neighbors. A misconfigured router, a state-sponsored redirect, or a simple mistake could hijack the route at any moment. The packets arrive anyway. Usually.

The server responds, `HTML` flows back, your browser renders it, and the page appears.

Milliseconds—billions of times a day—and every step involves faith in the next.

Your resolver accepted the `DNS` response. Your machine believed the source address. Your browser relied on the certificate chain. The routers honored the `BGP` announcements. The server processed the request.

And beneath that: your browser relied on a certificate authority's audit. Your security update depended on the CVE database. Your patch followed CISA's advisory. Confidence layered on institutional

confidence, all the way down. When the CVE database fell twenty-five thousand entries behind in 2025, the browser vendors kept depending on their update mechanisms. The institutional layer bent but did not break.

This is a cathedral built by different architects across different decades. Crocker's RFC 1 in 1969.[4] Postel's TCP and SMTP in the 1980s.[34,54] Mockapetris's DNS.[69] Berners-Lee's HTTP.[219] The TLS working group's revisions, culminating in RFC 8446.[243] Rekhter and Li's BGP.[158]

They never sat in a room together and designed a system—they built pieces, and the pieces fit, not perfectly, not securely, but well enough. Somehow, it holds together.

Then there is the scaffolding. CERT in 1988, responding to the Morris Worm. FIRST in 1990, coordinating across borders. CVE in 1999, naming the vulnerabilities. CISA in 2018, defending federal networks. One hundred forty-three national CERTs worldwide. These were repairs, retrofitted onto a structure never meant to bear this load. In 2025, some of the scaffolding failed. The cathedral stands.

And the terrifying, beautiful truth: it keeps working. The premises were wrong, but the system survived anyway.

The ratio holds. Honest traffic still outnumbers the malicious, valid certificates still outnumber the forged, accurate lookups still outnumber the poisoned. The borrowed time keeps extending.

The packets carrying these words followed that same path: DNS resolved, TCP connected, TLS encrypted, HTTP delivered. You're soaking in it.

It is late in Lagos. The power has failed twice today, but the generator runs. A university student sits before a laptop, screen bright in the

darkness. She is building a mobile payment system for market vendors who lack bank accounts. The established players ignored this problem. She did not.

She does not know the full history—Crocker's legal pad, Postel's robustness principle, RFC 1. She does not know the protocols carrying her API calls were designed for a network of four nodes where everyone knew everyone.

But she knows the network is there, and she knows it works. She is building something on it.

In Bangalore, a teenager writes code that will run on the servers of a company that does not exist yet. In São Paulo, a developer patches a library that ten thousand other projects depend on. In Berlin, in Seoul, in Cairo, in a hundred cities, people extend the network. They count on it to carry their packets. They expect the infrastructure to hold.

They are extending the chain that started in Boelter Hall, that passed through Postel's cluttered office, that ran through every RFC and every specification and every patch. They are inheriting the premises—and the exposures.

Jon Postel is dead—he died in 1998, unexpectedly, at fifty-five, and no one knew exactly what his job entailed.[338,339] The robustness principle lives on, and so do its consequences.

The network is still open, right now, as you hold this book.

We cannot go back to the small community of researchers who knew each other's faces. That world ended when the network outgrew any individual's comprehension. We cannot uninvent the vulnerabilities or unlearn the exploits.

In 1992, at the twenty-fourth IETF meeting, an MIT researcher named David Clark stood before his colleagues and articulated the philosophy that had guided them from the beginning[7]:

> David Clark --- IETF 24 --- July 1992
>
> We reject: kings, presidents and voting.
> We believe in: rough consensus and running code.

The words declared how the internet had been built: no central authority, no formal governance. Just engineers in a room, arguing until most agreed, then writing code to prove it worked.

That philosophy created the network. It also created the weaknesses. You cannot have one without the other.

We carry forward what that community created. The habit of documenting decisions in RFCs. The principle that anyone can participate if their ideas work. The expectation—still operative, still dangerous, still generative—that most participants mean well.

Rough consensus built the internet. Rough consensus broke it. Rough consensus is still the process by which it evolves. Nearly ten thousand RFCs later, the process remains: someone writes a document, others respond, and if most agree it works, it becomes the standard. The IETF still meets. The RFCs still accumulate. Nearly seven thousand contributors have joined the conversation Crocker started.

RFC 1 ended with an admission[4]:

> RFC 1 --- April 7, 1969 --- Steve Crocker
>
> I present here some of the tentative agreements reached and some of the open questions encountered. Very little of what is here is firm and reactions are expected.

Reactions are still expected.

The graduate students in 1969 could not have known what they were starting. The builders today cannot know where it is going.

In 2025, the institutions watching over the network buckled: budgets were cut, staff were laid off, the CVE database fell behind, and CISA shrank to a fraction of its former size.

The threats did not pause. Chinese state-sponsored campaigns, first disclosed in 2023 and 2024, continued to expand. Volt Typhoon had compromised critical infrastructure across the United States[340]: power grids, water systems, transportation networks. The intrusions were designed to persist, to remain hidden until a crisis. Pre-positioned for a conflict that might never come, or might come tomorrow.

Salt Typhoon, disclosed in late 2024, targeted telecommunications. Nine American carriers were compromised.[341] The hackers gained access to systems used for lawful intercepts. The infrastructure the government built to enable surveillance had been turned against it.

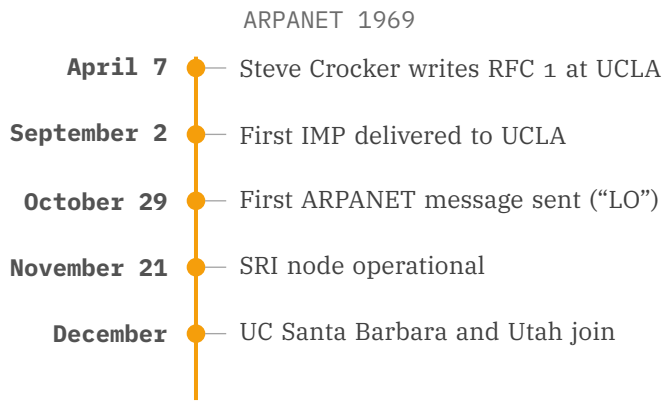
The attacks demonstrated something the early engineers never imagined: the standards they built for colleagues would become the battlefield where nations compete. The openness was the exposure. The exposure is still being exploited, more than fifty years after the first RFC was written.

The network bent, but it did not break.

Lo and behold—it still works.

TIMELINE OF INTERNET HISTORY

This section consolidates key events from the history of internet protocols.



TELNET TIMELINE

1971 RFC 97: "A First Cut at a Proposed Telnet Protocol"

1983 RFC 854/855: Formal Telnet standard published

1986 Clifford Stoll begins tracking the Hannover Hacker

1987 Markus Hess arrested in Germany (June 29)

1989 *The Cuckoo's Egg* published

1990 Hess convicted of espionage (February 15)

1995 Tatu Yl"onen releases SSH as secure replacement

2006 IETF publishes SSH as RFC 4251-4254

FTP EVOLUTION

1971 RFC 114: Bhushan's original File Transfer Protocol[25]

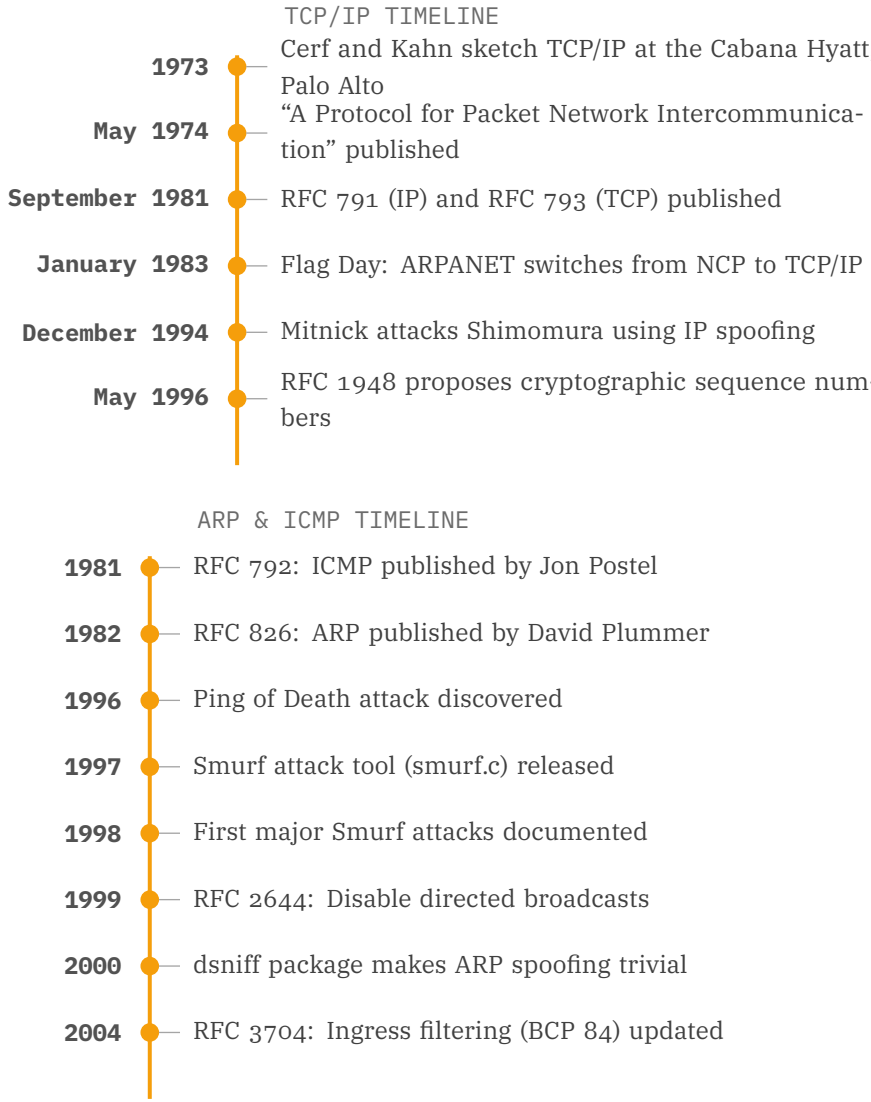
1971 RFC 172: Expanded FTP with restart and status commands[26]

1980 RFC 765: FTP restructured for TCP/IP environment[342]

1985 RFC 959: Definitive FTP specification (still current)[27]

1995 Hobbit publishes FTP bounce attack[29]

1997 RFC 2228: FTP security extensions proposed[343]



EMAIL TIMELINE

1971 Ray Tomlinson invents network email, chooses @

1973 75% of ARPANET traffic is email

1978 First spam: Gary Thuerk's DEC advertisement

1982 RFC 821 (SMTP) and RFC 822 (message format) published

1996 "Phishing" term first used on Usenet

2014 SPF published as RFC 7208[61]

2015 DMARC published as RFC 7489[63]

DNS TIMELINE

1973 HOSTS.TXT maintained by NIC

1983 Mockapetris publishes RFC 882/883 (DNS concept)

1985 First TLDs established: .com, .edu, .gov, .mil, .org

1987 RFC 1034/1035: definitive DNS specification

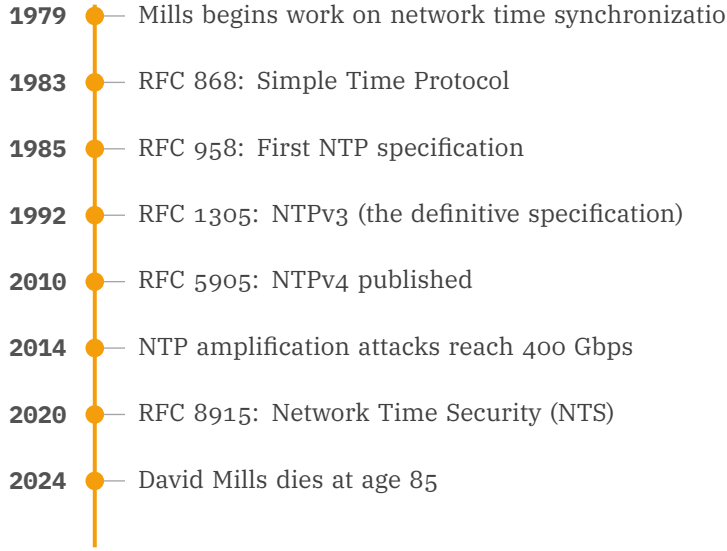
1989 Last HOSTS.TXT distribution

1999 DNSSEC first standardized (RFC 2535)[78]

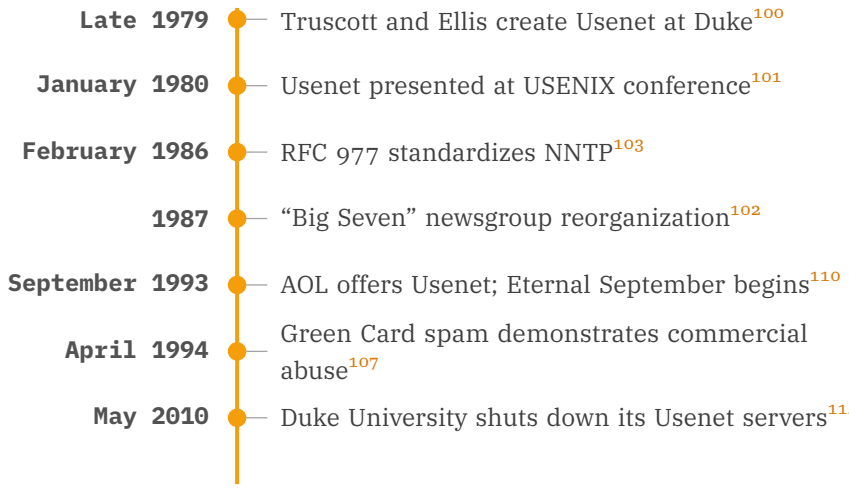
2008 Kaminsky discloses cache poisoning attack

2010 Root zone signed with DNSSEC

NTP TIMELINE



THE NETWORK ANYONE COULD JOIN



KERBEROS DEVELOPMENT

1983 Project Athena begins at MIT

1988 Kerberos v4 released for production use

1993 RFC 1510 standardizes Kerberos v5[118]

2000 Microsoft adopts Kerberos for Windows domain authentication

2005 RFC 4120 updates the specification[119]

2011 mimikatz reveals Windows credential vulnerabilities[120]

FROM CHAT TO BOTNETS

August 1988 Oikarinen creates IRC at University of Oulu

1989 First servers outside Scandinavia (MIT)

1990 EFnet forms as first major IRC network

January 1991 Gulf War updates spread via IRC

May 1993 RFC 1459 standardizes the protocol[130]

2000 GTBot proves IRC can control malware

February 2000 Mafiaboy attacks take down major sites[136]

October 2010 BredoLab botnet (30M machines) taken down[138]

NETWORK MANAGEMENT'S SECURITY GAP

November 1987 — RFC 1028 introduces SGMP[141]

May 1990 — RFC 1157 defines SNMP with community strings[142]

January 1996 — SNMPv2c keeps community-based security[143]

December 2002 — RFC 3411 introduces SNMPv3 with real authentication[144,344]

2014 — SNMP amplification attacks appear in multi-vector DDoS[148]

October 2016 — Mirai botnet uses default credentials for IoT attack[145,345]

BGP TIMELINE

1989 — RFC 1105: BGP-1 designed on napkins in Austin, Texas

1995 — RFC 1771: BGP-4 with classless addressing

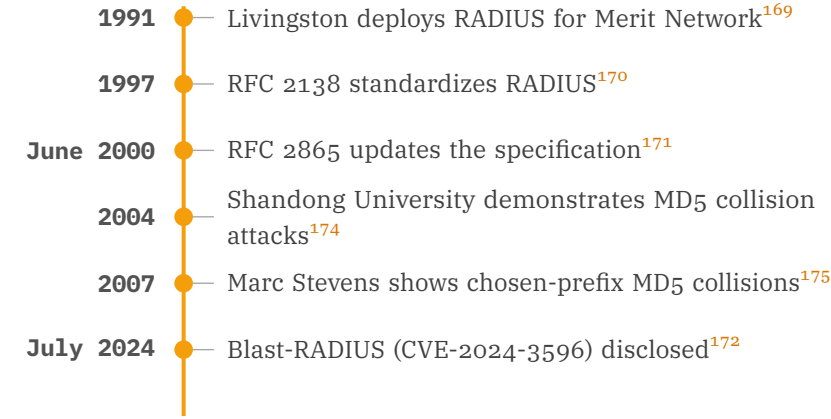
2006 — RFC 4271: BGP-4 becomes full Internet Standard

2008 — Pakistan/YouTube hijack reveals global vulnerability

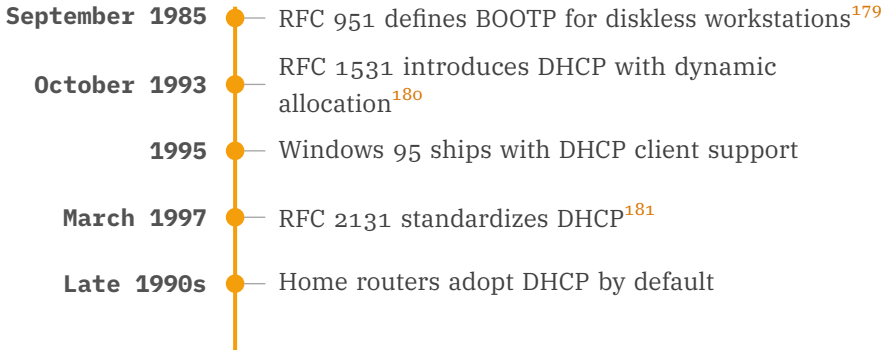
2012 — RPKI (Resource Public Key Infrastructure) standardized

2024 — Approximately 50% of routes have ROAs; many networks don't validate

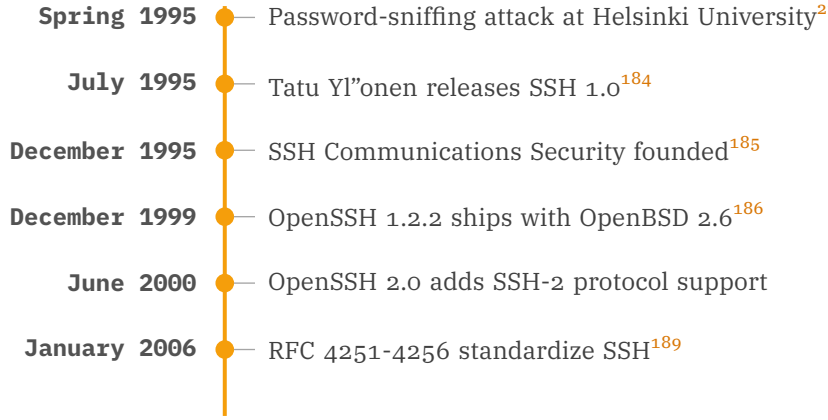
THE LONG FUSE



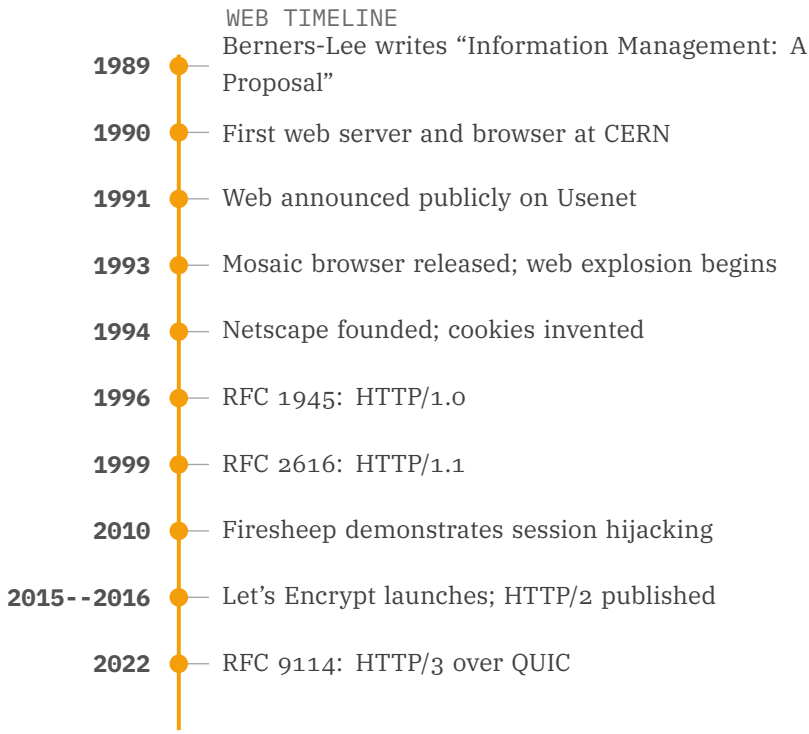
FROM STATIC TO DYNAMIC CONFIGURATION



SSH DEVELOPMENT TIMELINE



VOIP'S TRUST PROBLEM



TLS TIMELINE

1994 — Netscape develops SSL; version 1.0 never released

1995 — SSL 2.0 ships with Netscape Navigator 1.1

1996 — SSL 3.0 addresses critical weaknesses

1999 — RFC 2246: TLS 1.0 (SSL becomes open standard)

2006 — RFC 4346: TLS 1.1

2008 — RFC 5246: TLS 1.2

2011 — DigiNotar CA compromise; 300k Iranians affected

2014 — Heartbleed bug disclosed; OpenSSL patched

2015--2016 — Let's Encrypt launches free certificates

2018 — RFC 8446: TLS 1.3 with mandatory forward secrecy

THE CRYPTO WARS

1991 — Phil Zimmermann releases PGP; investigation begins

1993 — Clinton administration announces Clipper Chip

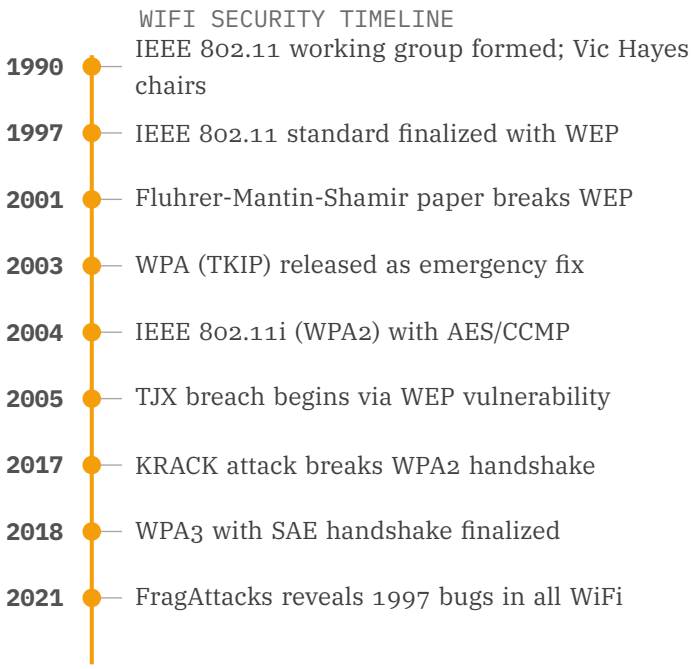
1994 — Matt Blaze publishes Clipper vulnerability

1995 — Daniel Bernstein sues over export restrictions

1996 — Zimmermann investigation dropped; Clipper defunct

1999 — Ninth Circuit: code is protected speech

2000 — Export restrictions on cryptography largely lifted



IPV6 & QUIC TIMELINE

1990 — Frank Solensky warns of IPv4 address exhaustion

1991 — Steve Deering begins designing SIPP

1995 — RFC 1883: IPv6 specification published

2011 — IANA exhausts IPv4 address pool

2012 — World IPv6 Launch; Google begins QUIC development

2015 — RFC 7526 deprecates 6to4 tunneling

2021 — RFC 9000: QUIC standardized

2022 — RFC 9114: HTTP/3 over QUIC

2024 — IPv6 carries 50% of Google traffic

SUPPLY CHAIN ATTACKS

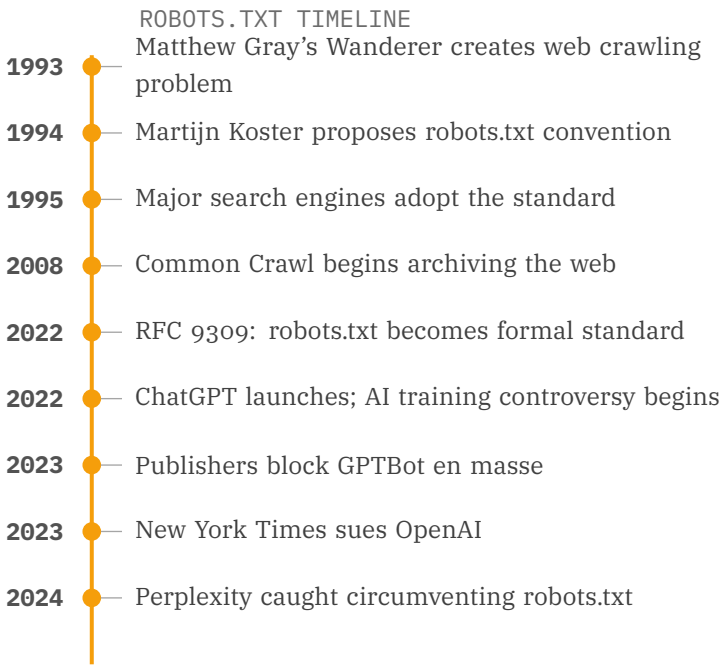
2010 — Stuxnet: first known nation-state supply chain attack (industrial controllers)[347]

2017 — NotPetya: Ukrainian tax software M.E.Doc used to distribute wiper malware[348]

2020 — SolarWinds/SUNBURST: 18,000 organizations receive backdoored updates[310]

2021 — Codecov: CI/CD tool modified to steal credentials[349]

2021 — Kaseya: MSP software used to deploy ransomware to customers[350]



REFERENCES

[1] Leonard Kleinrock. *The Day the Infant Internet Uttered its First Words*. UCLA Connection Lab. [Kleinrock's account of the first ARPANET message, October 29, 1969]. 2019. url: https://www.lk.cs.ucla.edu/internet_first_words.html.

[2] Katie Hafner and Matthew Lyon. *Where Wizards Stay Up Late: The Origins of the Internet*. Simon & Schuster, 1998.

[3] Steve Crocker. *How the Internet Got Its Rules*. The New York Times. [Op-ed on the 40th anniversary of RFC 1]. Apr. 2009. url: https://www.nytimes.com/2009/04/07/opinion/07crocker.html.

[4] Steve Crocker. *Host Software*. RFC 1. UCLA, Apr. 1969. url: https://www.rfc-editor.org/rfc/rfc1.

[5] Steve Crocker. *Documentation Conventions*. RFC 3. UCLA, Apr. 1969. url: https://www.rfc-editor.org/rfc/rfc3.

[6] Peter J. Denning. The ARPANET after Twenty Years. *American Scientist* **77**. (1989). [Historical account noting 15 nodes in 1971, 37 nodes in 1973], 530–534.

[7] David D. Clark. A Cloudy Crystal Ball: Visions of the Future. *Proceedings of the Twenty-Fourth Internet Engineering Task Force*. [Plenary presentation at IETF 24, July 13–17, 1992. Origin of the phrase "rough consensus and running code."]. Cambridge, MA, July 1992.

[8] United States General Accounting Office. *Computer Security: Virus Highlights Need for Improved Internet Management*. Report GAO/IMTEC-89-57. GAO, June 1989.

[9] Federal Bureau of Investigation. *The Morris Worm: 30 Years Since First Major Attack on the Internet*. FBI News. Nov. 2018. url: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218.

[10] Eugene H. Spafford. The Internet Worm: Crisis and Aftermath. *Communications of the ACM* **32**. (June 1989), 678–687.

[11] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. *Proceedings of the IEEE Symposium on Security and Privacy* (1989), 326–343.

[12] John Markoff. Student Testifies His Error Jammed Computer Network. *The New York Times* (Jan. 1990). [Trial testimony; includes Graham quote: "I said, 'You idiot.' It was such a good idea and he just blew it, I was so mad."], A19. url: https://www.nytimes.com/1990/01/19/us/student-testifies-his-error-jammed-computer-network.html.

[13] CERT/CC. *CA-1988-01: UNIX /usr/ucb/fingerd Buffer Overrun*. CERT Advisory. [First CERT advisory, issued in response to the Morris Worm]. Nov. 1988.

[14] United States Court of Appeals, Second Circuit. *United States v. Morris, 928 F.2d 504*. [Appeal upholding Morris worm conviction under Computer Fraud and Abuse Act]. Mar. 1991.

[15] John T. Melvin and Richard W. Watson. *A First Cut at a Proposed Telnet Protocol*. RFC 97. SRI-ARC, Feb. 1971. url: https://www.rfc-editor.org/rfc/rfc97.

[16] Jon Postel and Joyce Reynolds. *Telnet Protocol Specification*. RFC 854. USC/Information Sciences Institute, May 1983. url: https://www.rfc-editor.org/rfc/rfc854.

[17] Clifford Stoll. Stalking the Wily Hacker. *Communications of the ACM* **31**. (May 1988). [Technical account of the Lawrence Berkeley Lab intrusion investigation, published before The Cuckoo's Egg], 484–497.

[18] Clifford Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, 1989.

[19] Guinness World Records. *First incident of cyber-espionage*. Guinness World Records. [Documents Markus Hess arrest on 29 June 1987 and conviction on 15 February 1990]. 2020. url: https://www.guinnessworldrecords.com/world-records/612868-first-incident-of-cyber-espionage.

[20] Der Spiegel. *KGB-Hacker Markus Hess verurteilt*. Der Spiegel. [German news coverage of Hess conviction]. 1990.

[21] Tatu Ylönen. *SSH (Secure Shell) Remote Login Program*. comp.security.unix announcement. [Original SSH release announcement]. July 1995.

[22] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. IETF, Jan. 2006. url: https://www.rfc-editor.org/rfc/rfc4253.

[23] GreyNoise Labs. *The Day the Telnet Died*. https://www.labs.greynoise.io/grimoire/2026-02-10-telnet-falls-silent/. [Analysis of January 2026 telnet traffic collapse. Pre-drop baseline of 914,000 daily sessions fell 59% following CVE-2026-24061 disclosure]. Feb. 2026.

[24] *CVE-2026-24061: GNU Inetutils telnetd Authentication Bypass*. https://nvd.nist.gov/vuln/detail/CVE-2026-24061. [CVSS 9.8 CRITICAL. Authentication bypass via argument injection in USER environment variable. Vulnerability present since 2015. Added to CISA KEV January 26, 2026]. 2026.

[25] Abhay K. Bhushan. *A File Transfer Protocol*. RFC 114. MIT Project MAC, Apr. 1971. url: https://www.rfc-editor.org/rfc/rfc114.

[26] Abhay Bhushan et al. *The File Transfer Protocol*. RFC 172. MIT/Project MAC, June 1971. url: https://www.rfc-editor.org/rfc/rfc172.

[27] Jon Postel and Joyce Reynolds. *File Transfer Protocol (FTP)*. RFC 959. USC/Information Sciences Institute, Oct. 1985. url: https://www.rfc-editor.org/rfc/rfc959.

[28] U.S. Department of Justice. *Federal Law Enforcement Targets International Internet Piracy Syndicates*. DOJ Press Release. [Operation Bucca-

neer targeting warez groups using FTP]. Dec. 2001. url: https://www.justice.gov/archive/opa/pr/2001/December/01_crm_643.htm.

[29] Hobbit. *The FTP Bounce Attack*. Bugtraq mailing list. [Security advisory describing FTP PORT command abuse]. July 1995. url: https://seclists.org/bugtraq/1995/Jul/46.

[30] Mozilla Security Team. *Stopping FTP Support in Firefox 90*. Mozilla Security Blog. [Announcement of FTP protocol removal from Firefox]. July 2021. url: https://blog.mozilla.org/security/2021/07/20/stopping-ftp-support-in-firefox-90/.

[31] Vinton G. Cerf. *Oral History of Vint Cerf*. Computer History Museum. [Interviewed by David Hochfelder]. Nov. 2007. url: https://www.computerhistory.org/collections/catalog/102658186.

[32] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications* **22**. (May 1974), 637–648.

[33] Jon Postel. *Internet Protocol*. RFC 791. USC/Information Sciences Institute, Sept. 1981. url: https://www.rfc-editor.org/rfc/rfc791.

[34] Jon Postel. *Transmission Control Protocol*. RFC 793. USC/Information Sciences Institute, Sept. 1981. url: https://www.rfc-editor.org/rfc/rfc793.

[35] Jon Postel. *NCP/TCP Transition Plan*. RFC 801. [Plan for ARPANET transition from NCP to TCP/IP on January 1, 1983 (Flag Day)]. USC/Information Sciences Institute, Nov. 1981. url: https://www.rfc-editor.org/rfc/rfc801.

[36] Internet Society. *A Brief History of the Internet*. Internet Society. [Written by Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, Stephen Wolff]. 1997. url: https://www.internetsociety.org/internet/history-internet/brief-history-internet/.

[37] Robert T. Morris. *A Weakness in the 4.2BSD Unix TCP/IP Software*. Computing Science Technical Report 117. AT&T Bell Laboratories, Feb. 1985.

[38] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* **2**. (Nov. 1984), 277–288.

[39] Tsutomu Shimomura and John Markoff. *Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw—By the Man Who Did It*. Hyperion, 1996.

[40] Steven M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM SIGCOMM Computer Communication Review* **19**. (Apr. 1989), 32–48.

[41] Steve Bellovin. *Defending Against Sequence Number Attacks*. RFC 1948. AT&T Bell Laboratories, May 1996. url: https://www.rfc-editor.org/rfc/rfc1948.

[42] IEEE. *Ethernet Local Area Network (LAN), 1973-1985*. Engineering and Technology History Wiki. [DIX Blue Book specification published September 30, 1980]. 2023. url: https://ethw.org/Milestones:Ethernet_Local_Area_Network_(LAN),_1973-1985.

[43] David C. Plummer. *An Ethernet Address Resolution Protocol*. RFC 826. MIT Laboratory for Computer Science, Nov. 1982. url: https://www.rfc-editor.org/rfc/rfc826.

[44] Jon Postel. *Internet Control Message Protocol*. RFC 792. USC/Information Sciences Institute, Sept. 1981. url: https://www.rfc-editor.org/rfc/rfc792.

[45] CERT/CC. *CERT Advisory CA-1998-01: Smurf IP Denial-of-Service Attacks*. CERT Advisory. 1998. url: https://www.cert.org/advisories/CA-1998-01.html.

[46] TFreak. *Smurf Attack*. Bugtraq mailing list. [Original smurf.c release and documentation]. 1997.

[47] Paul Festa. Smurf Assault Cripples ISPs. *CNET* (Jan. 1998). [Reports on the University of Minnesota attack and "melted" T1 lines].

[48] CERT/CC. *CERT Advisory CA-1996-26: Denial-of-Service Attack via ping*. CERT Advisory. [First documentation of the Ping of Death vulnerability]. Dec. 1996. url: https://vuls.cert.org/confluence/display/

historical/CERT+Advisory+CA-1996-26+Denial-of-Service+Attack+via+ping.

[49] Dug Song. *dsniff*. monkey.org. [Network auditing tools including arpspoof for ARP cache poisoning]. 2000. url: https://www.monkey.org/~dugsong/dsniff/.

[50] Daniel Senie. *Changing the Default for Directed Broadcasts in Routers*. RFC 2644. [BCP 34: Recommends disabling directed broadcasts to mitigate Smurf attacks]. Amaranth Networks Inc., Aug. 1999. url: https://www.rfc-editor.org/rfc/rfc2644.

[51] Paul Ferguson and Daniel Senie. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. RFC 2827. [BCP 38: Original network ingress filtering recommendation]. Cisco Systems, Amaranth Networks Inc., May 2000. url: https://www.rfc-editor.org/rfc/rfc2827.

[52] Ray Tomlinson. *The First Network Email*. BBN Technologies. [Tomlinson's account of inventing network email]. 2009. url: https://openmap.bbn.com/~tomlinso/ray/firstemailframe.html.

[53] Craig Partridge. The Technical Development of Internet Email. *IEEE Annals of the History of Computing* **30**. (2008), 3–29.

[54] Jon Postel. *Simple Mail Transfer Protocol*. RFC 821. USC/Information Sciences Institute, Aug. 1982. url: https://www.rfc-editor.org/rfc/rfc821.

[55] David H. Crocker. *Standard for the Format of ARPA Internet Text Messages*. RFC 822. University of Delaware, Aug. 1982. url: https://www.rfc-editor.org/rfc/rfc822.

[56] Brad Templeton. *Reaction to the DEC Spam of 1978*. Templetons.com. [History of the first spam email]. 2005. url: http://www.templetons.com/brad/spamreact.html.

[57] Phishing.org. *History of Phishing*. Phishing.org. [First recorded mention of phishing on alt.online-service.america-online Usenet newsgroup, January 2, 1996]. 2023. url: https://www.phishing.org/history-of-phishing.

[58] Federal Bureau of Investigation. *Internet Crime Report 2021*. Internet Crime Complaint Center. 2022. url: https://www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf.

[59] Federal Bureau of Investigation. *Internet Crime Report 2022*. Internet Crime Complaint Center. 2023. url: https://www.ic3.gov/Media/PDF/AnnualReport/2022_IC3Report.pdf.

[60] U.S. Department of Justice. *Lithuanian Man Sentenced to 5 Years in Prison for Theft of Over $100 Million in Fraudulent Business Email Compromise Scheme*. DOJ Press Release. Dec. 2019. url: https://www.justice.gov/usao-sdny/pr/lithuanian-man-sentenced-5-years-prison-theft-over-100-million-fraudulent-business.

[61] Scott Kitterman. *Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1*. RFC 7208. IETF, Apr. 2014. url: https://www.rfc-editor.org/rfc/rfc7208.

[62] Dave Crocker, Tony Hansen, and Murray S. Kucherawy. *DomainKeys Identified Mail (DKIM) Signatures*. RFC 6376. IETF, Sept. 2011. url: https://www.rfc-editor.org/rfc/rfc6376.

[63] Murray Kucherawy and Elizabeth Zwicky. *Domain-based Message Authentication, Reporting, and Conformance (DMARC)*. RFC 7489. IETF, Mar. 2015. url: https://www.rfc-editor.org/rfc/rfc7489.

[64] Elizabeth J. Feinler. *Oral History of Elizabeth (Jake) Feinler*. Computer History Museum. [CHM Reference number: X5671.2010]. Mar. 2010.

[65] M.D. Kudlick. *Host Names On-Line*. RFC 608. SRI-ARC, Jan. 1974. url: https://www.rfc-editor.org/rfc/rfc608.

[66] Paul Mockapetris. *Internet Hall of Fame Inductee*. Internet Society. 2012. url: https://www.internethalloffame.org/inductee/paul-mockapetris.

[67] Paul Mockapetris. *Domain Names - Concepts and Facilities*. RFC 882. [Obsoleted by RFC 1034]. USC/Information Sciences Institute, Nov. 1983. url: https://www.rfc-editor.org/rfc/rfc882.

[68] Paul Mockapetris. *Domain Names - Implementation and Specification*. RFC 883. [Obsoleted by RFC 1035]. USC/Information Sciences Institute, Nov. 1983. url: https://www.rfc-editor.org/rfc/rfc883.

[69] Paul Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034. USC/Information Sciences Institute, Nov. 1987. url: https://www.rfc-editor.org/rfc/rfc1034.

[70] Paul Mockapetris. *Domain Names - Implementation and Specification*. RFC 1035. USC/Information Sciences Institute, Nov. 1987. url: https://www.rfc-editor.org/rfc/rfc1035.

[71] ICANN. *There Are Not 13 Root Servers*. ICANN Blog. [Explains that 13 root server addresses map to hundreds of physical servers via anycast]. Nov. 2007. url: https://www.icann.org/en/blogs/details/there-are-not-13-root-servers-15-11-2007-en.

[72] Kim Zetter. *Dan Kaminsky, Internet Security Savior, Dies at 42*. The New York Times. Apr. 2021. url: https://www.nytimes.com/2021/04/27/technology/daniel-kaminsky-dead.html.

[73] MITRE Corporation. *CVE-2008-1447: DNS Cache Poisoning Vulnerability*. CVE. 2008. url: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1447.

[74] Erica Naone. The Flaw at the Heart of the Internet. *MIT Technology Review* (Oct. 2008). [In-depth account of Kaminsky DNS discovery and coordinated disclosure]. url: https://www.technologyreview.com/2008/10/20/126595/the-flaw-at-the-heart-of-the-internet/.

[75] CERT Coordination Center. *VU#800113: Multiple DNS implementations vulnerable to cache poisoning*. CERT Vulnerability Note. [Advisory on Kaminsky DNS cache poisoning vulnerability]. July 2008. url: https://www.kb.cert.org/vuls/id/800113.

[76] Matasano Security. *Kaminsky DNS Flaw Details Leaked*. Security advisory. [Matasano accidentally confirmed Halvar Flake's hypothesis on July 21, 2008; post removed within five minutes but already mirrored]. July 2008.

[77] Dan Kaminsky. It's the End of the Cache as We Know It. *Black Hat Briefings*. [Presentation on DNS cache poisoning vulnerability]. Las Vegas, NV, 2008.

[78] Donald Eastlake 3rd. *Domain Name System Security Extensions*. RFC 2535. [First comprehensive DNSSEC specification; obsoleted by RFC 4033-4035]. IBM, Mar. 1999. url: https://www.rfc-editor.org/rfc/rfc2535.

[79] Roy Arends et al. *DNS Security Introduction and Requirements*. RFC 4033. IETF, Mar. 2005. url: https://www.rfc-editor.org/rfc/rfc4033.

[80] IANA. *Root Zone Now Signed*. IANA News. [Root zone signed with DNSSEC on July 15, 2010]. July 2010. url: https://www.iana.org/news/2010/root-zone-now-signed.

[81] APNIC. *DNSSEC World Map*. APNIC Statistics. [Approximately 30% of DNS queries validated with DNSSEC globally]. 2024. url: https://stats.labs.apnic.net/dnssec.

[82] ISC. *CVE-2025-40780: Cache poisoning due to weak PRNG in BIND 9*. Internet Systems Consortium Security Advisory. [CVSS 8.6 HIGH. Affects BIND 9.16.0-9.16.50, 9.18.0-9.18.39, 9.20.0-9.20.13, 9.21.0-9.21.12. Cache poisoning vulnerability exploiting weak pseudo-random number generation, same class as Kaminsky's 2008 attack]. Oct. 2025.

[83] David L. Mills. A Brief History of NTP Time: Confessions of an Internet Timekeeper. *ACM SIGCOMM Computer Communication Review* **33**. (2003). [Mills's own account of NTP history from 1977 onwards], 9–22. url: https://www.eecis.udel.edu/~mills/database/papers/history.pdf.

[84] Jon Postel and Ken Harrenstien. *Time Protocol*. RFC 868. USC/Information Sciences Institute, SRI, May 1983. url: https://www.rfc-editor.org/rfc/rfc868.

[85] David L. Mills. *Network Time Protocol (NTP)*. RFC 958. University of Delaware, Sept. 1985. url: https://www.rfc-editor.org/rfc/rfc958.

[86] Harrison Smith. *David Mills, the internet's 'father time,' dies at 85*. The Washington Post. Jan. 2024. url: https://www.washingtonpost.com/obituaries/2024/01/26/david-mills-network-time-protocol-internet-obituary/.

[87] David L. Mills. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. RFC 1305. University of Delaware, Mar. 1992. url: https://www.rfc-editor.org/rfc/rfc1305.

[88] David L. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. University of Delaware, June 2010. url: https://www.rfc-editor.org/rfc/rfc5905.

[89] Matthew Prince. The DDoS That Knocked Spamhaus Offline (And How We Mitigated It). *Cloudflare Blog* (Mar. 2013). url: https://blog.cloudflare.com/the-ddos-that-knocked-spamhaus-offline-and-ho/.

[90] Brian Krebs. *The New Normal: 200-400 Gbps DDoS Attacks*. Krebs on Security. Feb. 2014. url: https://krebsonsecurity.com/2014/02/the-new-normal-200-400-gbps-ddos-attacks/.

[91] John Graham-Cumming. *Technical Details Behind a 400Gbps NTP Amplification DDoS Attack*. Cloudflare Blog. 2014. url: https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/.

[92] Octave Klaba. *NTP Amplification Attack Tweet*. Twitter/@oloesmovhcom. [Tweet reporting 350+ Gbps NTP amplification attack on OVH infrastructure]. Feb. 2014. url: https://twitter.com/oloesmovhcom.

[93] MITRE. *CVE-2013-5211: NTP monlist Amplification Vulnerability*. Common Vulnerabilities and Exposures. [NTP monlist command allows 556x traffic amplification]. 2013. url: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5211.

[94] US-CERT. *NTP Amplification Attacks Using CVE-2013-5211*. US-CERT Alert TA14-013A. Jan. 2014. url: https://www.cisa.gov/news-events/alerts/2014/01/13/ntp-amplification-attacks-using-cve-2013-5211.

[95] Aanchal Malhotra et al. Attacking the Network Time Protocol. *Network and Distributed System Security (NDSS) Symposium*. [Kiss-o'-Death (KoD) attack research from Boston University]. San Diego, CA, 2016. url: https://www.cs.bu.edu/~goldbe/papers/NTPattacks.html.

[96] Dieter Sibold, Stephen Roettger, and Kristof Teichel. *Network Time Security for the Network Time Protocol*. RFC 8915. IETF, Sept. 2020. url: https://www.rfc-editor.org/rfc/rfc8915.

[97] Dalia Mortada. *How a power outage in Colorado caused U.S. official time to be 4.8 microseconds off*. NPR. [NIST Boulder power outage December 17–21, 2025: windstorm caused 4-day power outage affecting atomic ensemble time scale]. Dec. 2025. url: https://www.npr.org/2025/12/21/nx-s1-5651317/colorado-us-official-time-microseconds-nist-clocks.

[98] Lindsay Clark. *NIST warns of NTP inaccuracy after blackouts across Colorado*. The Register. [Analysis of NIST Boulder power outage impact on UTC(NIST) and NTP timeservers]. Dec. 2025. url: https://www.theregister.com/2025/12/21/nist_ntp_outage_warning.

[99] Steve Bellovin. *Re: The List again :-)*. Usenet History Discussion. [Contains quote about shell scripts and protocol variants]. 1997. url: https://shikan.org/bjones/Usenet.Hist/Nethist/0011.html.

[100] Steven M. Bellovin. *Netnews: The Origin Story*. Department of Computer Science, Columbia University. [Historical account by one of Usenet's creators]. 2010. url: https://www.cs.columbia.edu/~smb/papers/netnews-hist.pdf.

[101] Steve Bellovin. *The Early History of Usenet, Part VI: The Public Announcement*. [Detailed account of January 1980 USENIX presentation in Boulder]. Nov. 2019. url: https://circleid.com/posts/20191127_the_early_history_of_usenet_part_vi_the_public_announcement.

[102] Rick Adams and Gene Spafford. *The Great Renaming*. Usenet. [Reorganization of Usenet newsgroups into the Big Seven hierarchies: comp, misc, news, rec, sci, soc, and talk]. 1987.

[103] Brian Kantor and Phil Lapsley. *Network News Transfer Protocol*. RFC 977. University of California San Diego, University of California Berkeley, Feb. 1986. url: https://www.rfc-editor.org/rfc/rfc977.

[104] Theodore Ts'o. *Draft NNTP ACL proposal*. IETF nntp mailing list. [Proposed authorization controls for NNTP, never implemented. "At the last IETF, I said I would put together a draft proposal for how to include authorization controls."]. Feb. 1993.

[105] Rich Salz. *Re: NNTP v2 Authentication Protocol*. IETF nntp mailing list. ["Nobody was going to go out and implement something that nobody else had, and that could change on you next month. Also, security and authentication are getting a lot of interest these days, and perhaps NNTP should just leave things to the more-qualified, viz. GSSAPI."]. Aug. 1993.

[106] Theodore Ts'o. *Re: Draft NNTP ACL proposal*. IETF nntp mailing list. ["No authentication system is such that you can trust it 100%; there are always possible failure modes... So trust is always relative; it is not black and white."]. Mar. 1993.

[107] Brad Templeton. *Reaction to the First Major Spam*. templetons.com. [Documentation of Canter and Siegel's green card spam]. 1994.

[108] Laurence A. Canter and Martha S. Siegel. *How to Make a Fortune on the Information Superhighway: Everyone's Guerrilla Guide to Marketing on the Internet and Other On-Line Services*. [Written by the perpetrators of the Green Card spam]. HarperCollins, 1994.

[109] Arnt Gulbrandsen. *Canter & Siegel: What actually happened*. Personal website. [First-person account of creating the cancelbot to combat Canter & Siegel spam in June 1994]. 1994. url: https://rant.gulbrandsen.priv.no/canter-siegel.

[110] Dave Fischer. *Re: What's the longest-running thread?* Usenet: alt.folklore.computers. [First use of term "September that never ended" (January 26, 1994)]. Jan. 1994.

[111] Juhoon Kim et al. Today's Usenet Usage: NNTP Traffic Characterization. *2010 IEEE Conference on Computer Communications Workshops*

(INFOCOM WKSHPS). [Analyzed NNTP traffic from 20,000+ residential DSL customers at a European ISP. Found yEnc-encoded binaries accounted for more than 99% of bytes transferred.]. IEEE, 2010, pp. 1–6.

[112] America Online. *AOL Discontinues Usenet Access*. AOL. [AOL discontinued Usenet access on June 25, 2005]. June 2005.

[113] New York Attorney General. *Internet Service Providers Agree to Block Access to Child Pornography*. NY Attorney General Press Release. [Verizon, Time Warner Cable, and Sprint agreed to block Usenet access; AOL had already discontinued in 2005]. June 2008.

[114] Duke University. *A Piece of Internet History*. [Announcement of Duke's Usenet server shutdown on May 20, 2010]. May 2010. url: https://today.duke.edu/2010/05/usenet.html.

[115] MIT News. *Looking back at Project Athena*. MIT News. [35th anniversary retrospective on Project Athena]. Nov. 2018. url: https://news.mit.edu/2018/mit-looking-back-project-athena-distributed-computing-for-students-1111.

[116] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. *USENIX Winter Conference*. [First published description of Kerberos protocol]. Dallas, TX, Feb. 1988, pp. 191–202.

[117] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine* **32**. (Sept. 1994). [Comprehensive overview of Kerberos v4 and v5], 33–38.

[118] John Kohl and B. Clifford Neuman. *The Kerberos Network Authentication Service (V5)*. RFC 1510. DEC, USC/ISI, Sept. 1993. url: https://www.rfc-editor.org/rfc/rfc1510.

[119] Clifford Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120. USC-ISI, MIT, July 2005. url: https://www.rfc-editor.org/rfc/rfc4120.

[120] Benjamin Delpy. *mimikatz*. GitHub. [Windows credential extraction tool, first released May 2011 to demonstrate WDigest credential stor-

age vulnerability]. May 2011. url: https://github.com/gentilkiwi/mimikatz.

[121] MITRE ATT&CK. *Steal or Forge Kerberos Tickets: Golden Ticket*. MITRE ATT&CK. [T1558.001: Golden Ticket technique reference]. 2024. url: https://attack.mitre.org/techniques/T1558/001/.

[122] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor*. FireEye Threat Research Blog. Dec. 2020. url: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[123] Cybersecurity and Infrastructure Security Agency. *Emergency Directive 21-01: Mitigate SolarWinds Orion Code Compromise*. CISA. Dec. 2020. url: https://cyber.dhs.gov/ed/21-01/.

[124] Shaked Reiner. *Golden SAML: Newly Discovered Attack Technique Forges Authentication to Cloud Apps*. CyberArk Blog. [First documentation of Golden SAML attack technique]. Nov. 2017. url: https://www.cyberark.com/resources/threat-research-blog/golden-saml-newly-discovered-attack-technique-forges-authentication-to-cloud-apps.

[125] Sygnia. *Detection And Hunting Of Golden SAML Attack*. Sygnia Threat Reports. [SolarWinds was first use of Golden SAML in the wild]. Dec. 2020. url: https://www.sygnia.co/threat-reports-and-advisories/golden-saml-attack/.

[126] National Security Agency. *Detecting Abuse of Authentication Mechanisms*. NSA Cybersecurity Advisory U/OO/198854-20. [Advisory on detecting forged SAML tokens in SolarWinds-related attacks]. Dec. 2020. url: https://media.defense.gov/2020/Dec/17/2002554125/-1/-1/0/AUTHENTICATION_MECHANISMS_CSA_U_OO_198854_20.PDF.

[127] B. Clifford Neuman. *Faculty Profile*. [Director of USC Center for Computer Systems Security, co-designer of Kerberos]. USC Viterbi School

of Engineering. 2024. url: https://viterbi.usc.edu/directory/faculty/Neuman/B (visited on 12/15/2024).

[128] Jarkko Oikarinen. *IRC History*. IRC.org. [Oikarinen's own account: "The birthday of IRC was in August 1988"]. 1988. url: http://www.irc.org/history_docs/jarkko.html.

[129] Daniel Stenberg. *History of IRC (Internet Relay Chat)*. daniel.haxx.se. [Comprehensive IRC history compiled by cURL creator Daniel "bagder" Stenberg]. 2023. url: https://daniel.haxx.se/irchistory.html.

[130] Jarkko Oikarinen and Darren Reed. *Internet Relay Chat Protocol*. RFC 1459. University of Oulu, May 1993. url: https://www.rfc-editor.org/rfc/rfc1459.

[131] Greg "wumpus" Lindahl. *EFnet Formation*. IRC history archives. ["Eris refused to remove that line, so I formed EFnet. It wasn't much of a fight."]. Aug. 1990.

[132] Multiple IRC users. *IRC logs from the beginning of the Gulf War*. Archived at ibiblio.org. [Real-time IRC coverage from January 16, 1991, with eyewitness reports from Israel and Finland]. Jan. 1991. url: https://www.ibiblio.org/pub/academic/communications/logs/Gulf-War/.

[133] Paul Barford and Vinod Yegneswaran. An Inside Look at Botnets. *Malware Detection*. Vol. 27. Advances in Information Security. [Academic analysis of botnet source code]. Springer, 2006, pp. 171–191.

[134] Symantec Security Response. *W32.Sdbot*. Symantec Threat Database. [First major C++ IRC botnet with publicly available source code]. 2002.

[135] Sophos. *Agobot author arrested in Germany*. Sophos News. [Axel Gembe arrested on May 7, 2004 in Waldshut, Germany]. May 2004.

[136] CNN. *Mafiaboy pleads guilty*. CNN Technology. [Coverage of Michael Calce's DDoS attacks on major websites]. Jan. 2001. url: https://edition.cnn.com/2001/TECH/internet/01/18/mafiaboy.plea/.

[137] Kaspersky Lab. *Workings of 30 Million Strong Bredolab Botnet Laid Bare*. Kaspersky Press Release. [Botnet owner made up to $139,000 per month from renting the botnet]. Oct. 2010. url: https://www.kaspersky.com/about/press-releases/2010_workings-of-30-million-strong-bredolab-botnet-laid-bare.

[138] Dutch National High Tech Crime Unit. *BredoLab Botnet Takedown*. Dutch Police Press Release. [Takedown of 30 million machine botnet]. Oct. 2010.

[139] The Register. *Armenia jails Bredolab botmaster for 4 years*. The Register. [Georgy Avanesov sentenced to 4 years for creating BredoLab]. May 2012. url: https://www.theregister.com/2012/05/23/bredolab_botmaster_jailed.

[140] The Register. *Freenode IRC staff resign en masse, unhappy about new management*. The Register. [14 volunteer staff members resigned following "hostile takeover" by Andrew Lee]. May 2021. url: https://www.theregister.com/2021/05/19/freenode_staff_resigns/.

[141] Jeffrey Davin et al. *Simple Gateway Monitoring Protocol*. RFC 1028. Proteon, University of Tennessee, NYSERNet, Rensselaer Polytechnic Institute, Nov. 1987. url: https://www.rfc-editor.org/rfc/rfc1028.

[142] Jeffrey Case et al. *A Simple Network Management Protocol (SNMP)*. RFC 1157. SNMP Research, Performance Systems International, MIT, May 1990. url: https://www.rfc-editor.org/rfc/rfc1157.

[143] Jeffrey Case et al. *Introduction to Community-based SNMPv2*. RFC 1901. SNMP Research, Cisco Systems, Dover Beach Consulting, International Network Services, Jan. 1996. url: https://www.rfc-editor.org/rfc/rfc1901.

[144] David Harrington, Randy Presuhn, and Bert Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. RFC 3411. Enterasys Networks, BMC Software, Lucent Technologies, Dec. 2002. url: https://www.rfc-editor.org/rfc/rfc3411.

[145] Cloudflare. *Inside Mirai the Infamous IoT Botnet: A Retrospective Analysis*. Cloudflare Blog. [Detailed analysis of Mirai botnet and October 21, 2016 Dyn DDoS attack]. 2017. url: https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/.

[146] Rapid7. *Disclosures: Exposure of Critical Information Via SNMP Public Community String*. Rapid7 Blog. [Research on SNMP public community string vulnerabilities]. May 2014. url: https://www.rapid7.com/blog/post/2014/05/16/r7-2014-01-r7-2014-02-r7-2014-03-disclosures-exposure-of-critical-information-via-snmp-public-community-string/.

[147] Imperva. *SNMP Reflection and Amplification DDoS Attack*. Imperva Learn. [SNMP amplification factors of 6.3x for basic queries to 650x for GETBULK]. 2024. url: https://www.imperva.com/learn/ddos/snmp-reflection/.

[148] Matthew Prince. *Technical Details Behind a 400Gbps NTP Amplification DDoS Attack*. Cloudflare Blog. [Analysis of February 2014 400 Gbps NTP amplification attack]. Feb. 2014. url: https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/.

[149] Cybersecurity and Infrastructure Security Agency. *Reducing the Risk of SNMP Abuse*. CISA Alert. [US-CERT advisory on SNMP security best practices]. June 2017. url: https://www.cisa.gov/news-events/alerts/2017/06/05/reducing-risk-snmp-abuse.

[150] SNMP Research. *Company History*. SNMP Research. [History of SNMP Research founded by Jeff Case at University of Tennessee]. 2024. url: https://snmp.com/company/history.shtml.

[151] Trend Micro Research. *Operation Zero Disco: SNMP Vulnerability Exploitation Campaign*. [Documented CVE-2025-20352 exploitation campaign targeting Cisco SNMP subsystems. Cisco estimated nearly 2 million devices running vulnerable software.]. Oct. 2025. url: https://www.trendmicro.com/en_us/research/25/j/operation-zero-disco-cisco-snmp-vulnerability-exploit.html.

[152] Internet Society. *Kirk Lougheed: Internet Hall of Fame Inductee*. Internet Hall of Fame. 2012. url: https://www.internethalloffame.org/inductee/kirk-lougheed/.

[153] Kirk Lougheed and Yakov Rekhter. *A Border Gateway Protocol (BGP)*. RFC 1105. IETF, June 1989. url: https://www.rfc-editor.org/rfc/rfc1105.

[154] Computer History Museum. *The Two-Napkin Protocol*. Computer History Museum Blog. [History of BGP's creation at the 12th IETF meeting in Austin, January 1989]. July 2019. url: https://computerhistory.org/blog/the-two-napkin-protocol/.

[155] Kirk Lougheed and Yakov Rekhter. *A Border Gateway Protocol (BGP)*. RFC 1163. [BGP-2]. IETF, June 1990. url: https://www.rfc-editor.org/rfc/rfc1163.

[156] Kirk Lougheed and Yakov Rekhter. *A Border Gateway Protocol 3 (BGP-3)*. RFC 1267. [BGP-3]. IETF, Oct. 1991. url: https://www.rfc-editor.org/rfc/rfc1267.

[157] Yakov Rekhter and Tony Li. *A Border Gateway Protocol 4 (BGP-4)*. RFC 1771. [Original BGP-4 specification with CIDR support]. IETF, Mar. 1995. url: https://www.rfc-editor.org/rfc/rfc1771.

[158] Yakov Rekhter, Tony Li, and Susan Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. IETF, Jan. 2006. url: https://www.rfc-editor.org/rfc/rfc4271.

[159] RIPE NCC. *YouTube Hijacking: A RIPE NCC RIS Case Study*. RIPE NCC News. [Analysis of the Pakistan Telecom/YouTube BGP hijack, February 24, 2008]. Feb. 2008. url: https://www.ripe.net/about-us/news/youtube-hijacking-a-ripe-ncc-ris-case-study/.

[160] Matt Lepinski and Stephen Kent. *An Infrastructure to Support Secure Internet Routing*. RFC 6480. BBN Technologies, Feb. 2012. url: https://www.rfc-editor.org/rfc/rfc6480.

[161] MANRS. *RPKI ROV Deployment Reaches Major Milestone*. MANRS Blog. [Announcement that 50% of IPv4 routes are covered by ROAs]. May

2024. url: https://manrs.org/2024/05/rpki-rov-deployment-reaches-major-milestone/.

[162] Doug Madory. *BGP Hijack of Twitter by Russian ISP*. Kentik Blog. [Analysis of March 28, 2022 Twitter BGP hijack by RTComm.RU]. Mar. 2022. url: https://www.kentik.com/analysis/bgp-hijack-of-twitter-by-russian-isp/.

[163] Internet Society. *What Happened? The Amazon Route 53 BGP Hijack to Take Over Ethereum Cryptocurrency Wallets*. Internet Society Blog. [Analysis of April 24, 2018 cryptocurrency theft via BGP hijack]. Apr. 2018. url: https://www.internetsociety.org/blog/2018/04/amazons-route-53-bgp-hijack/.

[164] MANRS. *Large European Routing Leak Sends Traffic Through China Telecom*. MANRS Blog. [Analysis of June 6, 2019 BGP route leak affecting European mobile traffic]. June 2019. url: https://manrs.org/2019/06/large-european-routing-leak-sends-traffic-through-china-telecom/.

[165] Mandiant. *APT1: Exposing One of China's Cyber Espionage Units*. Mandiant Intelligence Report. [First public attribution of Chinese state-sponsored hacking to a specific PLA unit]. Feb. 2013. url: https://www.mandiant.com/resources/apt1-exposing-one-of-chinas-cyber-espionage-units.

[166] U.S. Department of Justice. *U.S. Charges Five Chinese Military Hackers for Cyber Espionage Against U.S. Corporations and a Labor Organization for Commercial Advantage*. DOJ Press Release. May 2014. url: https://www.justice.gov/opa/pr/us-charges-five-chinese-military-hackers-cyber-espionage-against-us-corporations-and-labor.

[167] Chris C. Demchak and Yuval Shavitt. China's Maxim – Leave No Access Point Unexploited: The Hidden Story of China Telecom's BGP Hijacking. *Military Cyber Affairs* **3**. (2018). url: https://scholarcommons.usf.edu/mca/vol3/iss1/7.

[168] CIDR Report. *BGP Table Statistics*. CIDR Report. [Daily BGP routing table statistics; approximately 1,000,000 IPv4 prefixes and 78,000 ASNs as of late 2024]. 2024. url: https://www.cidr-report.org/as2.0/.

[169] Jay Hipps. Lucent Acquires Livingston Enterprises. *Hacienda Online* **5**. (Dec. 1997). [Contemporaneous trade article from the Hacienda Business Park newsletter, Pleasanton, California]. url: https://www.hacienda.org/news-events/hacienda-online/network/1997/december/lucent-acquires-livingston-enterprises.

[170] Carl Rigney et al. *Remote Authentication Dial In User Service (RADIUS)*. RFC 2138. [Original IETF standardization of RADIUS. Obsoleted by RFC 2865]. Livingston Enterprises, Merit Network, Daydreamer, Apr. 1997. url: https://www.rfc-editor.org/rfc/rfc2138.

[171] Carl Rigney et al. *Remote Authentication Dial In User Service (RADIUS)*. RFC 2865. Livingston Enterprises, Ascend Communications, Merit Network, Daydreamer, June 2000. url: https://www.rfc-editor.org/rfc/rfc2865.

[172] Sharon Goldberg et al. RADIUS/UDP Considered Harmful. *33rd USENIX Security Symposium*. [Blast-RADIUS attack (CVE-2024-3596). Disclosed July 9, 2024]. Philadelphia, PA, Aug. 2024. url: https://www.blastradius.fail/pdf/radius.pdf.

[173] MITRE. *CVE-2024-3596: RADIUS Protocol Forgery Vulnerability*. Common Vulnerabilities and Exposures. [CVSS 7.5, affects all RADIUS implementations using MD5]. 2024. url: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-3596.

[174] Xiaoyun Wang et al. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *CRYPTO 2004 Rump Session*. [First practical MD5 collision attack from Shandong University]. Santa Barbara, CA, Aug. 2004.

[175] Marc Stevens, Arjen Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. *Advances in Cryptology – EUROCRYPT 2007*. Vol. 4515. Lecture Notes in

Computer Science. [Demonstrated chosen-prefix collision attacks on MD5]. Berlin, Heidelberg: Springer, 2007, pp. 1–22.

[176] FreeRADIUS Project. *About FreeRADIUS*. FreeRADIUS. [FreeRADIUS founded June 1999 by Miquel van Smoorenburg and Alan DeKok]. 2024. url: https://www.freeradius.org/about/.

[177] Stefan Winter et al. *Transport Layer Security (TLS) Encryption for RADIUS*. RFC 6614. [RadSec: RADIUS over TLS]. RESTENA, OSC, Cisco Systems, May 2012. url: https://www.rfc-editor.org/rfc/rfc6614.

[178] Usenet Archives. *Duplicate IP Address Discussions*. comp.dcom.lans.ethernet. [Archived discussions of IP collisions and spreadsheet management]. 1991.

[179] Bill Croft and John Gilmore. *Bootstrap Protocol (BOOTP)*. RFC 951. Stanford University, Sun Microsystems, Sept. 1985. url: https://www.rfc-editor.org/rfc/rfc951.

[180] Ralph Droms. *Dynamic Host Configuration Protocol*. RFC 1531. [Obsoleted by RFC 1541, then RFC 2131]. Bucknell University, Oct. 1993. url: https://www.rfc-editor.org/rfc/rfc1531.

[181] Ralph Droms. *Dynamic Host Configuration Protocol*. RFC 2131. Bucknell University, Mar. 1997. url: https://www.rfc-editor.org/rfc/rfc2131.

[182] Alfredo Andrés and David Barroso. *Yersinia: A Framework for Layer 2 Attacks*. Black Hat Europe 2005. 2005. url: https://blackhat.com/presentations/bh-europe-05/BH_EU_05-Berrueta_Andres/BH_EU_05_Berrueta_Andres.pdf.

[183] MITRE ATT&CK. *Adversary-in-the-Middle: DHCP Spoofing*. MITRE ATT&CK Technique T1557.003. 2024. url: https://attack.mitre.org/techniques/T1557/003/.

[184] Tatu Ylönen. *The Story of Getting SSH Port 22*. SSH.com Blog. [Ylonen's account of obtaining port 22 from IANA in July 1995]. 2017. url: https://www.ssh.com/academy/ssh/port.

[185] SSH Communications Security. *SSH History*. SSH.com. [Company founded December 31, 1995]. 2023. url: https://www.ssh.com/about/history/.

[186] OpenBSD Project. *OpenSSH: Project History*. OpenSSH Website. [Official history: forked from SSH 1.2.12, first release with OpenBSD 2.6 in December 1999]. 1999. url: https://www.openssh.com/history.html.

[187] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. SSH Communications Security, Cisco Systems, Jan. 2006. url: https://www.rfc-editor.org/rfc/rfc4252.

[188] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. SSH Communications Security, Cisco Systems, Jan. 2006. url: https://www.rfc-editor.org/rfc/rfc4254.

[189] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. SSH Communications Security, Cisco Systems, Jan. 2006. url: https://www.rfc-editor.org/rfc/rfc4251.

[190] Debian Security Team. *DSA-1571-1 openssl – predictable random number generator*. Debian Security Advisory. May 2008. url: https://www.debian.org/security/2008/dsa-1571.

[191] NIST. *CVE-2008-0166: OpenSSL Predictable Random Number Generator*. National Vulnerability Database. [CVSS Score 7.8: OpenSSL 0.9.8c-1 through 0.9.8g-9 on Debian-based systems]. 2008. url: https://nvd.nist.gov/vuln/detail/cve-2008-0166.

[192] GitHub. *We updated our RSA SSH host key*. GitHub Blog. Mar. 2023. url: https://github.blog/2023-03-23-we-updated-our-rsa-ssh-host-key/.

[193] Qualys Threat Research Unit. *regreSSHion: RCE in OpenSSH's server, on glibc-based Linux systems*. Qualys Security Advisory. [CVE-2024-6387: Race condition in signal handling, CVSS 8.1. Bug fixed in 2006 (CVE-2006-5051), reintroduced in OpenSSH 8.5p1 (October 2020)]. July 2024. url: https://www.qualys.com/2024/07/01/cve-2024-6387/regresshion.txt.

[194] Columbia University. *Henning G. Schulzrinne*. Columbia Engineering Faculty Profile. [Schulzrinne joined Columbia as professor in August 1996]. 2024. url: https://www.engineering.columbia.edu/faculty-staff/directory/henning-g-schulzrinne.

[195] International Telecommunication Union. *H.323: Packet-based multimedia communications systems*. ITU-T Recommendation. [First version published November 1996 for voice/video over packet networks]. Nov. 1996. url: https://www.itu.int/rec/T-REC-H.323.

[196] Mark Handley et al. *SIP: Session Initiation Protocol*. RFC 2543. IETF, Mar. 1999. url: https://www.rfc-editor.org/rfc/rfc2543.

[197] Jonathan Rosenberg et al. *SIP: Session Initiation Protocol*. RFC 3261. IETF, June 2002. url: https://www.rfc-editor.org/rfc/rfc3261.

[198] Henning Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. Columbia University, Packet Design, July 2003. url: https://www.rfc-editor.org/rfc/rfc3550.

[199] Mark Handley, Van Jacobson, and Colin Perkins. *SDP: Session Description Protocol*. RFC 4566. UCL, PARC, University of Glasgow, July 2006. url: https://www.rfc-editor.org/rfc/rfc4566.

[200] Mark Baugher et al. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. [Provides encryption and authentication for RTP media streams]. Cisco Systems, Ericsson Research, Mar. 2004. url: https://www.rfc-editor.org/rfc/rfc3711.

[201] YouMail. *Robocall Index: 2021 Full Year Report*. YouMail. [Approximately 50 billion robocalls in the United States in 2021]. 2022.

[202] U.S. Department of Justice. *California Man Sentenced In Deadly Wichita Swatting Case*. U.S. Attorney's Office, District of Kansas. [Tyler Barriss sentenced to 20 years for 51 charges including one count of making a false report resulting in death]. Mar. 2019. url: https://www.justice.gov/usao-ks/pr/california-man-sentenced-deadly-wichita-swatting-case.

[203] U.S. Department of Justice. *Ohio Gamer Sentenced In Deadly Swatting Case*. U.S. Attorney's Office, District of Kansas. [Casey Viner sentenced

to 15 months for conspiracy]. Sept. 2019. url: https://www.justice.gov/usao-ks/pr/ohio-gamer-sentenced-deadly-swatting-case.

[204] Federal Bureau of Investigation. *The Crime of "Swatting": Fake 9-1-1 Calls Have Real Consequences*. FBI Stories. [Documents first federal swatting case in 2007 led by FBI Dallas]. 2008. url: https://www.fbi.gov/news/stories/the-crime-of-swatting-fake-9-1-1-calls-have-real-consequences1.

[205] Brian Krebs. *The World Has No Room For Cowards*. Krebs on Security. [Krebs describes being swatted on March 14, 2013]. Mar. 2013. url: https://krebsonsecurity.com/2013/03/the-world-has-no-room-for-cowards/.

[206] U.S. Congress. *Truth in Caller ID Act of 2009*. Public Law 111-331. [Signed December 22, 2010, prohibits transmitting misleading caller ID information]. 2010.

[207] Jon Peterson et al. *Authenticated Identity Management in the Session Initiation Protocol (SIP)*. RFC 8224. [STIR standard for cryptographic caller identity verification]. NeuStar, Cisco, RTFM Inc., Comcast, Feb. 2018. url: https://www.rfc-editor.org/rfc/rfc8224.

[208] Federal Communications Commission. *STIR/SHAKEN Caller ID Authentication*. FCC. [STIR/SHAKEN mandate effective June 30, 2021]. June 2021. url: https://www.fcc.gov/call-authentication.

[209] Tim Berners-Lee. *Information Management: A Proposal*. CERN Internal Document. Mar. 1989. url: https://www.w3.org/History/1989/proposal.html.

[210] Vannevar Bush. As We May Think. *The Atlantic Monthly* **176**. (July 1945), 101–108.

[211] Theodor H. Nelson. Complex Information Processing: A File Structure for the Complex, The Changing and the Indeterminate. *Proceedings of the 20th ACM National Conference*. [Paper in which Nelson coined the term "hypertext"]. ACM, 1965, pp. 84–100.

[212] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. HarperCollins, 1999.

[213] CERN. *The World's First Website*. CERN. [First website published December 20, 1990 at info.cern.ch on Tim Berners-Lee's NeXT computer]. Dec. 1990. url: http://info.cern.ch.

[214] Tim Berners-Lee. *WorldWideWeb: Summary*. Usenet: alt.hypertext. [First public announcement of the World Wide Web, August 6, 1991]. Aug. 1991.

[215] National Center for Supercomputing Applications. *NCSA Mosaic*. NCSA, University of Illinois. [First graphical web browser released February 1993 by Marc Andreessen and Eric Bina]. Feb. 1993.

[216] Matthew Gray. *Measuring the Growth of the Web*. MIT. [Early web growth statistics: 2,738 websites by mid-1994; 10,000 by end of 1994]. 1994. url: https://www.mit.edu/people/mkgray/growth/.

[217] Mosaic Communications Corporation. *Mosaic Communications Corporation Founded*. Mountain View, California. [Founded April 4, 1994; renamed Netscape Communications November 1994]. Apr. 1994.

[218] Adam Barth. *HTTP State Management Mechanism*. RFC 6265. IETF, Apr. 2011. url: https://www.rfc-editor.org/rfc/rfc6265.

[219] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. MIT/LCS, UC Irvine, W3C, May 1996. url: https://www.rfc-editor.org/rfc/rfc1945.

[220] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. IETF, June 1999. url: https://www.rfc-editor.org/rfc/rfc2616.

[221] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. IETF, May 2015. url: https://www.rfc-editor.org/rfc/rfc7540.

[222] Mike Bishop. *HTTP/3*. RFC 9114. IETF, June 2022. url: https://www.rfc-editor.org/rfc/rfc9114.

[223] Eric Butler. *Firesheep*. ToorCon San Diego. [Firefox extension demonstrating HTTP session hijacking]. 2010.

[224] Twitter. *HTTPS now in beta*. Twitter Blog. [Twitter made HTTPS available January 2011 in response to Firesheep]. Jan. 2011.

[225] Facebook. *A Continued Commitment to Security*. Facebook Blog. [Facebook added always-on HTTPS option January 2011 in response to Firesheep]. Jan. 2011.

[226] Internet Security Research Group. *Let's Encrypt Has Issued 1 Million Certificates*. Let's Encrypt Blog. [Let's Encrypt public beta launched December 2015; issued millionth certificate March 2016]. Mar. 2016. url: https://letsencrypt.org/2016/03/08/our-millionth-cert.html.

[227] Google. *HTTPS encryption on the web*. Google Transparency Report. [Over 95% of Chrome traffic uses HTTPS as of 2023]. 2023. url: https://transparencyreport.google.com/https/overview.

[228] Apache Software Foundation. *CVE-2021-41773: Path Traversal and Remote Code Execution in Apache HTTP Server 2.4.49*. Apache Security Advisory. [Path traversal vulnerability allowing attackers to read files outside the document root; actively exploited in the wild]. Oct. 2021. url: https://httpd.apache.org/security/vulnerabilities_24.html.

[229] Apache Software Foundation. *CVE-2021-42013: Path Traversal and Remote Code Execution in Apache HTTP Server 2.4.49 and 2.4.50*. Apache Security Advisory. [Incomplete fix for CVE-2021-41773; both vulnerabilities exploited by ransomware operators within days of disclosure]. Oct. 2021. url: https://httpd.apache.org/security/vulnerabilities_24.html.

[230] Taher Elgamal. *Oral History of Taher Elgamal*. Computer History Museum. [Elgamal developed SSL at Netscape in 1994-1995]. 2015.

[231] Taher Elgamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *Advances in Cryptology - CRYPTO '84*. [Original ElGamal signature scheme paper; basis for DSA]. Springer, 1985, pp. 10–18.

[232] Paul C. Kocher, Alan O. Freier, and Philip Karlton. *The SSL Protocol Version 3.0*. Netscape Communications Corporation. [SSL 3.0 specification authored by Kocher (independent consultant) and Netscape

engineers]. Nov. 1996. url: https://www.paulkocher.com/doc/SSLv3.pdf.

[233] The Marconi Society. *Paul Kocher and Taher Elgamal: 2019 Marconi Prize Recipients*. Marconi Society. [Kocher and Elgamal jointly awarded for "development of SSL/TLS and other contributions to the security of communications"]. 2019. url: https://marconisociety.org/fellow-bio/paul-kocher/.

[234] Tim Dierks and Christopher Allen. *The TLS Protocol Version 1.0*. RFC 2246. Certicom, Jan. 1999. url: https://www.rfc-editor.org/rfc/rfc2246.

[235] Electronic Frontier Foundation. *The Clipper Chip*. EFF Archive. [EFF documentation of the Clipper Chip controversy]. 1993. url: https://www.eff.org/issues/clipper-chip.

[236] Phil Zimmermann. *Why I Wrote PGP*. PGP User's Guide. [Zimmermann's original essay on creating Pretty Good Privacy]. 1991. url: https://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html.

[237] Matt Blaze. Protocol Failure in the Escrowed Encryption Standard. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. [Blaze's paper demonstrating critical flaws in the Clipper Chip]. 1994, pp. 59–67.

[238] Electronic Frontier Foundation. *Bernstein v. United States Department of Justice*. EFF Case Archive. [Ninth Circuit ruling that source code is protected speech]. 1999. url: https://www.eff.org/cases/bernstein-v-us-dept-justice.

[239] National Institute of Standards and Technology. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Special Publication 800-90A. [Original standard including Dual_EC_DRBG]. NIST, 2006.

[240] Dan Shumow and Niels Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng. *Crypto 2007 Rump Session*. [First public presentation of potential Dual_EC_DRBG backdoor]. Aug. 2007.

[241] Joseph Menn. Exclusive: Secret contract tied NSA and security industry pioneer. *Reuters* (Dec. 2013). [Report on NSA's $10M payment to RSA Security to use Dual_EC_DRBG]. url: https://www.reuters.com/article/us-usa-security-rsa-idUSBRE9BJ1C220131220.

[242] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* **22**. (Nov. 1976), 644–654.

[243] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Mozilla, Aug. 2018. url: https://www.rfc-editor.org/rfc/rfc8446.

[244] Eric Rescorla. *TLS 1.3 Published: in Firefox Today*. Mozilla Security Blog. [Announcement of RFC 8446 publication; Rescorla was editor and primary author]. Aug. 2018. url: https://blog.mozilla.org/security/2018/08/13/tls-1-3-published-in-firefox-today/.

[245] Google Chrome Team. *Chromium Blog: Evolving the Security Indicators in Chrome*. Google Chrome Blog. [Chrome removed HTTPS lock icon in September 2023]. 2023.

[246] Google Chrome Team. *An update on attempted man-in-the-middle attacks*. Google Security Blog. [Chrome's certificate pinning detected fraudulent Google certificate; revoked DigiNotar trust]. Aug. 2011. url: https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html.

[247] Fox-IT. *Black Tulip: Report of the investigation into the DigiNotar Certificate Authority breach*. Tech. rep. [Official investigation report on DigiNotar breach; identified 300,000+ Iranian OCSP requests indicating MITM attacks]. Fox-IT, Aug. 2012. url: https://roselabs.nl/files/audit_reports/Fox-IT_-_DigiNotar.pdf.

[248] RTL Nieuws. *Onderzoeker DigiNotar: 'Er zijn mensen gedood om een andere mening'*. RTL News. [Dutch prosecutor Hans Hoogstraaten's statement on DigiNotar investigation]. Sept. 2011.

[249] DigiNotar BV. *DigiNotar Certificate Authority Breach*. VASCO Security. [DigiNotar parent company VASCO announced bankruptcy September 20, 2011; 531 fraudulent certificates identified]. Sept. 2011.

[250] Codenomicon and Google Security. *The Heartbleed Bug*. https://heartbleed.com. [CVE-2014-0160: OpenSSL TLS heartbeat extension vulnerability]. 2014.

[251] MITRE Corporation. *CVE-2014-0160: OpenSSL Heartbleed Vulnerability*. CVE. [Buffer over-read vulnerability in OpenSSL's heartbeat extension]. 2014. url: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[252] Bruce Schneier. *Heartbleed*. Schneier on Security blog. Apr. 2014. url: https://www.schneier.com/blog/archives/2014/04/heartbleed.html.

[253] Google. *HTTPS encryption on the web*. Google Transparency Report. [Percentage of web traffic using HTTPS; over 95% of Chrome traffic uses HTTPS as of 2024]. 2024. url: https://transparencyreport.google.com/https/overview.

[254] Let's Encrypt. *Let's Encrypt Statistics*. Let's Encrypt. [Let's Encrypt serves over 400 million active certificates as of 2024]. 2024. url: https://letsencrypt.org/stats/.

[255] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. Google, June 2013. url: https://www.rfc-editor.org/rfc/rfc6962.

[256] Stephan Friedl et al. *DNS Certification Authority Authorization (CAA) Resource Record*. RFC 8659. [CAA records allow domain owners to specify which CAs may issue certificates]. Cisco Systems, Apple Inc., Google, Nov. 2019. url: https://www.rfc-editor.org/rfc/rfc8659.

[257] Engineering and Technology History Wiki. *Vic Hayes*. ETHW. [Hayes joined NCR in 1974, approached IEEE in 1988, led 802.11 working group from 1990-2000]. 2023. url: https://ethw.org/Vic_Hayes.

[258] IEEE. *IEEE Leadership Award: Victor Hayes*. IEEE 802. [For 10 years of leadership as chairman of IEEE 802.11 Wireless LAN Working Group]. 2000.

[259] IEEE. *IEEE Standard 802.11-1997: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Standards Association. [Original 802.11 standard with WEP, 40-bit encryption due to U.S. export restrictions]. 1997.

[260] National Research Council. *Cryptography's Role in Securing the Information Society*. [Study of U.S. encryption export policy and 40-bit key limitations]. National Academy Press, 2000.

[261] Ronald L. Rivest. *RC4 Stream Cipher*. RSA Data Security. [Designed by Ron Rivest in 1987; trade secret until leaked in 1994]. 1987.

[262] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. *Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001*. Vol. 2259. Lecture Notes in Computer Science. Toronto, Ontario, Canada: Springer, Aug. 2001, pp. 1–24.

[263] Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. Breaking 104 Bit WEP in Less Than 60 Seconds. *Information Security Applications: 8th International Workshop, WISA 2007*. Vol. 4867. Lecture Notes in Computer Science. [PTW attack: 40,000 frames for 50% success, 85,000 for 95%]. Jeju Island, Korea: Springer, Aug. 2007, pp. 188–202.

[264] Wi-Fi Alliance. *Wi-Fi Protected Access*. Wi-Fi Alliance. [WPA with TKIP released in 2003 as interim security measure before 802.11i]. 2003.

[265] IEEE. *IEEE 802.11i-2004: Amendment 6: Medium Access Control (MAC) Security Enhancements*. IEEE Standards Association. [Ratified June 24, 2004; marketed as WPA2; introduced AES/CCMP]. June 2004.

[266] Joseph Pereira. How Credit-Card Data Went Out Wireless Door. *The Wall Street Journal* (May 2007). [Detailed account of TJX breach via WEP vulnerability at Marshalls stores]. url: https://www.wsj.com/articles/SB117824446226991797.

[267] U.S. Department of Justice. *International Hacker Pleads Guilty for Massive Hacks of U.S. Retail Networks*. DOJ Press Release 09-948. Sept. 2009.

url: https://www.justice.gov/archives/opa/pr/international-hacker-pleads-guilty-massive-hacks-us-retail-networks.

[268] TJX Companies Inc. *Form 8-K: Current Report*. U.S. Securities and Exchange Commission. [Disclosure of data breach affecting 45.6 million credit and debit card numbers]. Jan. 2007.

[269] U.S. Department of Justice. *Leader of Hacking Ring Sentenced for Massive Identity Thefts from Payment Processor and U.S. Retail Networks*. DOJ Press Release 10-282. [Albert Gonzalez sentenced to 20 years in prison by Judge Patti B. Saris on March 25, 2010]. Mar. 2010. url: https://www.justice.gov/archives/opa/pr/leader-hacking-ring-sentenced-massive-identity-thefts-payment-processor-and-us-retail.

[270] Berger Montague. *TJX Companies Retail Security Breach Litigation*. Class action settlement documentation. [Total costs exceeded $250 million including forensics, credit monitoring, legal fees]. 2009. url: https://bergermontague.com/cases/tjx-companies-retail-security-breach-litigation/.

[271] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. [Discovered by Mathy Vanhoef at KU Leuven, Belgium]. Dallas, TX, USA: ACM, 2017, pp. 1313–1328.

[272] CERT/CC. *VU#228519: Wi-Fi Protected Access II (WPA2) handshake traffic can be manipulated to induce nonce and session key reuse*. CERT Vulnerability Note. [Multiple CVEs: CVE-2017-13077 through CVE-2017-13088]. Oct. 2017. url: https://www.kb.cert.org/vuls/id/228519.

[273] Wi-Fi Alliance. *Wi-Fi Alliance® introduces Wi-Fi CERTIFIED WPA3™ security*. Wi-Fi Alliance Press Release. [Official launch of WPA3 certification program on June 25, 2018]. Austin, TX, June 2018. url: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-wpa3-security.

[274] Dan Harkins. *Dragonfly Key Exchange*. RFC 7664. [Defines password-authenticated key exchange used in WPA3-SAE]. IETF, Nov. 2015. url: https://www.rfc-editor.org/rfc/rfc7664.

[275] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. [Discovered side-channel and downgrade attacks against WPA3-SAE within months of release]. San Francisco, CA: IEEE, 2019, pp. 517–533. url: https://wpa3.mathyvanhoef.com/.

[276] Mathy Vanhoef. Fragment and Forge: Breaking Wi-Fi Through Frame Aggregation and Fragmentation. *Proceedings of the 30th USENIX Security Symposium*. [Design flaws in 802.11 frame aggregation and fragmentation dating to the original 1997 standard; CVE-2020-24586 through CVE-2020-26145]. USENIX Association, 2021, pp. 161–178. url: https://www.fragattacks.com/.

[277] The Radicati Group. *Email Statistics Report, 2024-2028*. Radicati Group Market Research. [Industry statistics on email volume; approximately 361 billion emails sent daily in 2024, projected 376 billion by 2025; 4.4 billion email users worldwide]. 2024. url: https://www.radicati.com/.

[278] Cloudflare. *How Cloudflare analyzes 1M DNS queries per second*. Cloudflare Blog. [Cloudflare's 1.1.1.1 resolver processes approximately 1 million DNS queries per second]. 2023. url: https://blog.cloudflare.com/how-cloudflare-analyzes-1m-dns-queries-per-second/.

[279] NS1. *Analysis of 7.5 Trillion DNS Queries*. CircleID. [Analysis revealing approximately 1 million DNS queries per second globally; public resolvers handle 60% of recursive DNS]. 2023. url: https://circleid.com/posts/20230316-analysis-of-7.5-trillion-dns-queries-reveals-public-resolvers-dominate-the-internet.

[280] Geoff Huston. BGP in 2024. *APNIC Blog* (Jan. 2025). [Annual analysis of BGP routing table growth; IPv4 table grew by 6% (53,000 entries) in 2024]. url: https://blog.apnic.net/2025/01/06/bgp-in-2024/.

[281] NIST. *National Vulnerability Database Dashboard*. https://nvd.nist.gov/general/nvd-dashboard. [25,000+ CVEs awaiting processing as of January 2026]. 2025.

[282] Axios. *CISA Workforce Reduction: Agency at 35% Capacity*. https://www.axios.com/2025/12/cisa-workforce-reduction. [CISA reduced from 3,400 to 900 staff (74% reduction) in 2025]. 2025.

[283] MITRE Corporation. *CVE Program Contract Extension*. https://cve.mitre.org/news/. [11-month extension executed April 16, 2025; expires March 16, 2026]. 2025.

[284] FIRST. *FIRST Member Teams Directory*. https://www.first.org/members/teams/. [808 member teams in 110+ countries; 143 national CERTs globally]. 2025.

[285] International Telecommunication Union. *ITU Facts and Figures 2025*. ITU Statistics. [Global internet usage statistics; 5.53 billion internet users as of 2025]. 2025. url: https://www.itu.int/itu-d/reports/statistics/facts-figures/.

[286] Engineering and Technology History Wiki. *Stephen Deering*. ETHW. [Biography of IPv6 and IP Multicast inventor]. 2024. url: https://ethw.org/Stephen_Deering.

[287] Frank Solensky. *Continued Internet Growth*. IETF 18 Proceedings. [First serious study predicting IPv4 address exhaustion, p. 53]. Aug. 1990.

[288] Scott Bradner and Allison Mankin. *The Recommendation for the IP Next Generation Protocol*. RFC 1752. IETF, Jan. 1995. url: https://www.rfc-editor.org/rfc/rfc1752.

[289] Stephen Deering and Robert Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 1883. IETF, Dec. 1995. url: https://www.rfc-editor.org/rfc/rfc1883.

[290] Number Resource Organization. *Free Pool of IPv4 Address Space Depleted*. NRO Press Release. [IANA allocated the last five /8 blocks to RIRs on February 3, 2011]. Feb. 2011. url: https://www.nro.net/ipv4-free-pool-depleted/.

[291] Internet Society. *World IPv6 Day: June 8, 2011*. Internet Society. [24-hour test of IPv6 compatibility by major websites]. 2011. url: https://www.worldipv6launch.org/.

[292] Internet Society. *World IPv6 Launch: June 6, 2012*. Internet Society. [Permanent IPv6 enablement by Google, Facebook, Yahoo, and others]. 2012. url: https://www.worldipv6launch.org/.

[293] Google. *IPv6 Adoption Statistics*. Google. [Real-time IPv6 adoption statistics from Google]. 2024. url: https://www.google.com/intl/en/ipv6/statistics.html.

[294] Jim Roskind. QUIC: Quick UDP Internet Connections. *Google Technical Talk*. Mountain View, CA, 2013.

[295] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. Fastly, Mozilla, May 2021. url: https://www.rfc-editor.org/rfc/rfc9000.

[296] Ed Jankiewicz, John Loughney, and Thomas Narten. *IPv6 Node Requirements*. RFC 6434. IETF, Dec. 2011. url: https://www.rfc-editor.org/rfc/rfc6434.

[297] Pekka Savola and Christophe Patel. *Security Considerations for 6to4*. RFC 3964. IETF, Dec. 2004. url: https://www.rfc-editor.org/rfc/rfc3964.

[298] Jim Hoagland. *The Teredo Protocol: Tunneling Past Network Security*. Black Hat USA 2007. [Security analysis of Teredo tunneling vulnerabilities]. 2007. url: https://blackhat.com/presentations/bh-usa-07/Hoagland/Whitepaper/bh-usa-07-hoagland-WP.pdf.

[299] Ole Troan and Brian Carpenter. *Deprecating the Anycast Prefix for 6to4 Relay Routers*. RFC 7526. IETF, May 2015. url: https://www.rfc-editor.org/rfc/rfc7526.

[300] Antonios Atlasis. *Security Impacts of Abusing IPv6 Extension Headers*. Black Hat Abu Dhabi 2012. [Catalog of IPv6 extension header attacks affecting firewalls and IDS]. 2012. url: https://media.blackhat.com/ad-12/Atlasis/bh-ad-12-security-impacts-atlasis-wp.pdf.

[301] Alessandro Ghedini. *Even faster connection establishment with QUIC 0-RTT resumption*. Cloudflare Blog. [Explanation of 0-RTT replay risks and mitigations]. 2018. url: https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/.

[302] National Cyber Security Hall of Fame. *The Cyber Security Hall of Fame Celebrates the 2024 Honorees*. Cyber Security Hall of Fame. [Roskind inducted as Vice President and Distinguished Engineer at Amazon; cited for inventing QUIC at Google and contributions at Netscape]. Dec. 2024. url: https://www.cybersecurityhalloffame.org/.

[303] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor*. FireEye Threat Research. Dec. 2020. url: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[304] Sergey Ulasen. *Interview: The Man Who Found Stuxnet*. Wired. [Ulasen's account of discovering Stuxnet at VirusBlokAda]. 2011.

[305] David E. Sanger. Obama Order Sped Up Wave of Cyberattacks Against Iran. *The New York Times* (June 2012). [First detailed public account of Operation Olympic Games]. url: https://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html.

[306] Huib Modderkolk. Dutch intelligence helped the US-Israeli Stuxnet cyberattack against Iran. *Yahoo News* (Sept. 2019). [Investigation revealing Dutch AIVD role and Erik van Sabben]. url: https://news.yahoo.com/revealed-how-a-secret-dutch-mole-aided-the-us-israeli-stuxnet-cyber-attack-on-iran-160026018.html.

[307] Kevin Mandia. *FireEye Shares Details of Recent Cyber Attack, Actions to Protect Community*. FireEye Blog. Dec. 2020. url: https://www.fireeye.com/blog/products-and-services/2020/12/fireeye-shares-details-of-recent-cyber-attack-actions-to-protect-community.html.

[308] CrowdStrike. *SUNSPOT: An Implant in the Build Process*. CrowdStrike Blog. Jan. 2021. url: https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/.

[309] Mandiant. *SUNBURST Additional Technical Details*. Mandiant Threat Intelligence. Dec. 2020. url: https://www.mandiant.com/resources/blog/sunburst-additional-technical-details.

[310] SolarWinds Corporation. *Form 8-K: Report of Unscheduled Material Events*. Securities and Exchange Commission. Dec. 2020. url: https://www.sec.gov/Archives/edgar/data/1739942/000162828020017451/swi-20201214.htm.

[311] SolarWinds Corporation. *An Investigative Update of the Cyberattack*. SolarWinds Blog. [SolarWinds investigative update: initial access September 4, 2019; fewer than 100 customers actively exploited]. May 2021. url: https://www.solarwinds.com/blog/an-investigative-update-of-the-cyberattack.

[312] Federal Bureau of Investigation. *Understanding and Responding to the SolarWinds Supply Chain Attack: The Federal Perspective*. FBI Congressional Testimony. [FBI testimony: 9 federal agencies and fewer than 100 non-government entities actively compromised]. Feb. 2021. url: https://www.fbi.gov/news/speeches-and-testimony/understanding-and-responding-to-the-solar-winds-supply-chain-attack-the-federal-perspective.

[313] The White House. *Fact Sheet: Imposing Costs for Harmful Foreign Activities by the Russian Government*. White House Statements and Releases. [Official US attribution of SolarWinds to SVR]. Apr. 2021. url: https://www.whitehouse.gov/briefing-room/statements-releases/2021/04/15/fact-sheet-imposing-costs-for-harmful-foreign-activities-by-the-russian-government/.

[314] Microsoft Threat Intelligence Center. *Deep dive into the Solorigate second-stage activation: From SUNBURST to TEARDROP and Raindrop*. Microsoft Security Blog. [Technical details on SUNBURST dormancy period (12-14 days) and second-stage malware]. Jan. 2021. url: https://www.microsoft.com/en-us/security/blog/2021/01/20/deep-

dive-into-the-solorigate-second-stage-activation-from-sunburst-to-teardrop-and-raindrop/.

[315] Executive Office of the President. *Executive Order 14028: Improving the Nation's Cybersecurity*. Federal Register. May 2021. url: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity.

[316] CISA. *Known Exploited Vulnerabilities Catalog Statistics 2025*. https://www.cisa.gov/known-exploited-vulnerabilities-catalog. [Catalog grew 20% in 2025 while CISA workforce shrank 74%]. 2025.

[317] MITRE Corporation. *CVE Program Funding Status*. https://cve.mitre.org/news/. [CISA contract expires March 16, 2026. Near-shutdown April 2025 averted with 17 hours notice]. 2025.

[318] Matthew Gray. *World Wide Web Wanderer*. MIT. [First autonomous web crawler, deployed June 1993]. 1993. url: https://www.mit.edu/~mkgray/bio.html.

[319] Martijn Koster. *ALIWEB - Archie-Like Indexing for the WEB*. Nexor Ltd. [Announced November 30, 1993]. Nov. 1993. url: https://archives.iw3c2.org/www1/PdfWWW94/aliweb.pdf.

[320] Martijn Koster. *A Standard for Robot Exclusion*. Web Robots Pages. [Original robots.txt proposal]. 1994. url: https://www.robotstxt.org/orig.html.

[321] Digital Equipment Corporation. *AltaVista Search Engine*. DEC. Dec. 1995.

[322] Martijn Koster et al. *Robots Exclusion Protocol*. RFC 9309. IETF, Sept. 2022. url: https://www.rfc-editor.org/rfc/rfc9309.

[323] Common Crawl Foundation. *Common Crawl: Open Repository of Web Crawl Data*. Common Crawl. [Nonprofit 501(c)(3) archiving the web since 2008]. 2008. url: https://commoncrawl.org/.

[324] OpenAI. *ChatGPT: Optimizing Language Models for Dialogue*. OpenAI Blog. [Launched November 30, 2022]. Nov. 2022. url: https://openai.com/blog/chatgpt.

[325] UBS. *ChatGPT May Be the Fastest Growing App in History*. UBS Research. Feb. 2023.

[326] OpenAI. *GPTBot*. OpenAI Documentation. [Documentation for OpenAI's web crawler, announced August 7, 2023]. 2023. url: https://platform.openai.com/docs/gptbot.

[327] Google. *Google-Extended Crawler*. Google Search Central. [Introduced September 28, 2023]. Sept. 2023. url: https://developers.google.com/search/docs/crawling-indexing/google-common-crawlers.

[328] The New York Times. *The Times Sues OpenAI and Microsoft Over A.I. Use of Copyrighted Work*. The New York Times. Dec. 2023. url: https://www.nytimes.com/2023/12/27/business/media/new-york-times-open-ai-microsoft-lawsuit.html.

[329] Dhruv Mehrotra and Tim Marchman. *Perplexity Is a Bullshit Machine*. WIRED. June 2024. url: https://www.wired.com/story/perplexity-is-a-bullshit-machine/.

[330] Robb Knight. *Perplexity AI, Robots.txt, and Other Questions*. rknight.me. June 2024. url: https://rknight.me/blog/perplexity-ai-robotstxt-and-other-questions/.

[331] Cloudflare. *Perplexity is using stealth, undeclared crawlers*. Cloudflare Blog. Aug. 2025. url: https://blog.cloudflare.com/perplexity-is-using-stealth-undeclared-crawlers-to-evade-website-no-crawl-directives/.

[332] Aravind Srinivas. *Perplexity CEO Interview*. Fast Company. June 2024. url: https://www.fastcompany.com/91144894/perplexity-ai-ceo-aravind-srinivas-on-plagiarism-accusations.

[333] W3C TDMRep Community Group. *TDM Reservation Protocol (TDMRep)*. W3C Community Group Report. Feb. 2024. url: https://www.w3.org/community/reports/tdmrep/CG-FINAL-tdmrep-20240202/.

[334] Leonard Kleinrock. *The Day the Infant Internet Uttered its First Words*. UCLA Computer Science Department. [Kleinrock's account of October 29, 1969, 10:30pm: "lo" message sent by Charley Kline]. 2009. url: https://www.lk.cs.ucla.edu/internet_first_words.html.

[335] Fox-IT. *DigiNotar Certificate Authority Breach: Operation Black Tulip*. Interim Report. [Forensic analysis of the DigiNotar CA compromise]. 2012.

[336] Federal Bureau of Investigation. *Business Email Compromise: The $55 Billion Scam*. Internet Crime Complaint Center PSA. [BEC cost US and global organizations $55.5 billion between October 2013 and December 2023]. Sept. 2024. url: https://www.ic3.gov/PSA/2024/PSA240911.

[337] Linus Torvalds. *Free minix-like kernel sources for 386-AT*. Usenet: comp.os.minix. [August 25, 1991: "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu)"]. Aug. 1991.

[338] Vint Cerf. *I Remember IANA*. RFC 2468. [Memorial RFC for Jon Postel]. MCI, Oct. 1998. url: https://www.rfc-editor.org/rfc/rfc2468.

[339] The Washington Post. Jon Postel, Internet Pioneer, Dies at 55 After Heart Surgery. *The Washington Post* (Oct. 1998). url: https://www.washingtonpost.com/archive/local/1998/10/18/jon-postel-internet-pioneer-dies-at-55-after-heart-surgery/.

[340] Cybersecurity and Infrastructure Security Agency. *PRC State-Sponsored Actors Compromise and Maintain Persistent Access to U.S. Critical Infrastructure*. CISA Joint Cybersecurity Advisory. [Advisory on Volt Typhoon's pre-positioning in critical infrastructure]. Feb. 2024. url: https://www.cisa.gov/news-events/cybersecurity-advisories/aa24-038a.

[341] Dustin Volz et al. China's 'Salt Typhoon' Hack Shows Critical U.S. Telecom Vulnerabilities. *The Wall Street Journal* (Oct. 2024). [Investigation of Salt Typhoon compromise of 9 US telecom carriers]. url: https://www.wsj.com/politics/national-security/china-salt-typhoon-hack-us-telecom-wiretapping-6efb4bb6.

[342] Jon Postel. *File Transfer Protocol*. RFC 765. USC/Information Sciences Institute, June 1980. url: https://www.rfc-editor.org/rfc/rfc765.

[343] M. Horowitz and S. Lunt. *FTP Security Extensions*. RFC 2228. Cygnus Solutions, Bellcore, Oct. 1997. url: https://www.rfc-editor.org/rfc/rfc2228.

[344] Ulf Blumenthal and Bert Wijnen. *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*. RFC 3414. Lucent Technologies, Dec. 2002. url: https://www.rfc-editor.org/rfc/rfc3414.

[345] Cybersecurity and Infrastructure Security Agency. *Heightened DDoS Threat Posed by Mirai and Other Botnets*. CISA Alert TA16-288A. Oct. 2016. url: https://www.cisa.gov/news-events/alerts/2016/10/14/heightened-ddos-threat-posed-mirai-and-other-botnets.

[346] Wichita Police Department. *Officer-Involved Shooting Investigation*. Wichita, Kansas. [Andrew Finch killed December 28, 2017 in SWATting incident]. Dec. 2017.

[347] Kim Zetter. *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon*. Crown, 2014.

[348] Andy Greenberg. The Untold Story of NotPetya, the Most Devastating Cyberattack in History. *Wired* (Aug. 2018). url: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

[349] Codecov. *Bash Uploader Security Update*. Codecov Blog. [Disclosure of CI/CD supply chain compromise]. Apr. 2021. url: https://about.codecov.io/security-update/.

[350] Cybersecurity and Infrastructure Security Agency. *CISA-FBI Guidance for MSPs and their Customers Affected by the Kaseya VSA Supply-Chain Ransomware Attack*. CISA Alert. July 2021. url: https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-188a.

The internet was designed for a world that no longer exists.
It works in the world that does.

Every protocol in this book has been exploited. Email spoofing. DNS poisoning. BGP hijacking. Certificate fraud. The assumptions the engineers made—trust the sender, believe the address, accept the packet—were wrong.

And yet. Five billion people use the internet daily. The packets arrive. The names resolve. The connections encrypt. A system built on violated assumptions keeps running—patched and hardened but never redesigned.

A bathroom in Boelter Hall. Seventy-five cents. Christmas in San Diego. Lunch in Austin. Two hours of darkness. Faces in the sidebar. The 911 call. Thirty million machines. The machines stopped asking.

Each phrase is a story. Each story is a protocol. Each protocol is still running. This book traces twenty-three arcs from creation to exploitation—and explains why fixing them would break everything.

ABOUT THE AUTHOR

Michael J Bommarito II is a researcher and entrepreneur in AI, law, and finance. His research has been published in scientific journals, law reviews, and mainstream media, including Science, Physica A, and the New York Times. He holds degrees from the University of Michigan in mathematics, political science, and financial engineering.

rfcbook.com

ISBN 979-8-9943457-7-1