

RegistryDecoder API Document
Version 1.1
11/1/2011

This document attempts to clearly explain the RegistryDecoder API for developing plugins. The API was designed for ease of use and readability for people with little or no programming (or Python) experience. Yes it is a bit verbose :) See the examples at the end of this document. For further clarification, see the plugins in the source distribution.

Intro:

A plugin file is a text file of Python code. At a minimum it must include the following 2 elements:

1. Definition of three plugin attributes in the global scope:

- *pluginname* - the string-formatted name of this plugin
- *description* - the short string-formatted description of this plugin
- *hive* - the string-formatted hive file this plugin runs on; must be exactly one of the elements of the set ("NTUSER", "SAM", "SECURITY", "SOFTWARE", "SYSTEM"); more advanced plugins which run on more than one hive type may use the attribute *hives*, with a list of any of the hive types listed above

2. Definition of a function named *run_me* which is called when the plugin is run by the system

```
e.g., def run_me():  
    pass
```

Plugins have access to all of standard Python, as well as access to the following API functions:

Registry key functions:

Given a string-formatted registry key path, returns the **key_object**. If the path does not exist, returns False. The string must begin with "\\".

reg_get_key(path)

Given a string-formatted registry key path, returns the **key_object**. If the path does not exist, the plugin terminates. Use for plugins which should just fail if the given path is not found. The string must begin with "\\".

reg_get_required_key(path)

Given a **key_object**, returns the string-formatted key name.

reg_get_key_name(key_object)

Given a *key_object*, returns string-formatted *lastwrite* time for the key.
reg_get_lastwrite(key_object)

Given a *key_object*, returns the (possibly empty) list of its child *key_objects*.
reg_get_subkeys(key_object)

Given a *key_object*, returns the (possibly empty) list of its *value_objects*.
reg_get_values(key_object)

Given a *key_object* and the string formatted name of a *value* contained in that key, return the data associated with that *value*. Returns *None* if the named *value* does not exist.
reg_get_data_for_value_name(key_object, name-string)

Registry value functions:

Given a *value_object*, returns its string-formatted name.
reg_get_value_name(value_object)

Given a *value_object*, returns its string-formatted data. The formatting is done automatically using the value's type (REG_SZ, etc.). This is not always correct.
reg_get_value_data(value_object)

Reporting functions:

Here "report" refers to the output generated by a plugin to be displayed in the GUI. Reports are generated dynamically as tables, row by row, in each plugin using the following functions. All reports consist of:

- an (auto-generated) header detailing the plugin name, and the file the plugin was run against
- an optional timestamp, set by ***reg_set_report_timestamp***
- optional table column headers, set by ***reg_set_report_header***
- zero or more of rows of data, normally generated one at a time by ***reg_report***

Set headers for columns for report generated by this plugin. *header* is a tuple of header names. E.g., to set column names of "name" and "data", e.g.

reg_set_report_header(("name", "data"))
reg_set_report_header(header)

The header of a plugin's output can optionally report a timestamp. This is timestamp is generally based on the *lastwrite* time of some registry key. Given a string-formatted date/time, this function sets the timestamp reported in the header.

reg_set_report_timestamp(timestamp)

Add to the reported plugin output the row data as a Python tuple. Following the example from `reg_set_report_header` above, a row of results can be added to the report as follows: `reg_report(("some_value_name", "that_value's_data"))`
reg_report(report_data)

Simplification function. Given a **key_object** key, add to the plugin's output report rows with all value names and data for key.
reg_report_values_name_data(key)

Same as **reg_report_values_name_data** above, except the output is filtered. Only values whose name appears in *name-list* will have their name / data reported.
reg_report_values_name_data_filtered(key, name_list)

Utility Functions

These functions are for operations commonly required by plugins.

Returns the string current control set number (e.g., "001"), or exits the plugin if the current control set cannot be determined. Note: always exits the plugin if called on a non-SYSTEM hive file.
reg_get_currentcontrolset()

Given a *unixtime* integer, returns the string-formatted date and time.
pretty_unixtime(unixtime)

Given a Windows 128-bit *datetime* object, returns the string-formatted date and time.
pretty_date128(date128)

Given a Rot-13 encoded string, returns the original string.
un_rot13(rot13_string)

Writing and Testing Plugins

The most efficient method to test plugins is from the command line. Users who are developing plugins from a source code checkout can simply place them in *templates/template_files*. Otherwise, a separate folder can be created and its path can be given on the command line following the "-d" parameter. For example, if you are testing with the Pyinstaller distributed application:

```
regdecoder.exe -d <path to your plugins directory>
```

Testing plugins from the commandline is fully supported, but less flexible than the GUI. The invocation to run a plugin from the command line is:

```
regdecoder.exe <case directory> <plugin name> <file id> <extra plugin directory  
(optional)>
```

Where "case directory" is the folder of the Registry Decoder created case. "Plugin name" is the name of the plugin as it appears in the plugin's *name* member. This is the same information that appears in the GUI drop down list of plugins. "Extra plugin directory" is the same as that can be passed as the *-d* parameter.

The "file id" is a little more complicated. To handle multiple files within a case, Registry decoder uses a file id per hive for tracking. To determine the file id of a file from a case you want to test, you need to examine the *evidence_database.db* within your case folder. This is Sqlite database, and can be viewed with the Sqlite3 command of any Linux distribution or the Windows binary package.

To see all filenames along with their file ids, please run this query from the Sqlite3 console:

```
select filename,id from evidence_sources;
```

The filename column contains the name of the registry file and the id column is the file id for the file. To ensure that file id handling does not become an issue, it is recommended to load only a few individual registry hives into Registry Decoder when developing and testing plugins.

Examples:

```
#####
```

```
# Windows 7 Word Wheel Query search terms list
```

```
# Required plugin attributes:
```

```
pluginname = "Word Wheel Query"  
description = "Lists Windows 7 user-entered search queries."  
hive = "NTUSER"
```

```
# Required function to be called by RegistryDecoder on execution:
```

```
def run_me():  
  
    regkey = reg_get_required_key("\\Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\WordWheelQuery")  
    reg_report_values_name_data(regkey)
```

```
#####
```

```
# Recently accessed user documents
```

```
# Required plugin attributes:
```

```
pluginname = "Recent Docs"  
description = "Displays files and folders recently accessed by this user."
```

```
hive = "NTUSER"
```

```
# Required function to called RegistryDecoder on plugin execution:
```

```
def run_me():
```

```
    regkey = reg_get_required_key("\Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs")  
    values = reg_get_values(regkey)
```

```
    for val in values:
```

```
        name = reg_get_value_name(val)
```

```
        data = reg_get_value_data(val)
```

```
        if name == "MRUListEx" and data:
```

```
            if len(data) == 1:
```

```
                data = ord(data)
```

```
    reg_report((name, data))
```

```
#####
```