Cuckoo Monitor Documentation

Release 1.3

Jurriaan Bremer

Oct 03, 2017

Contents

The **Cuckoo Monitor** is the *new* user-mode monitor for Cuckoo Sandbox. It has been rewritten from scratch with clean and easily extendable code in mind.

This document is meant as a guide to start customizing the monitor to your own needs.

Contents:

CHAPTER 1

Requirements

Building the Cuckoo Monitor is easiest on a Ubuntu machine. It should also be possible on other operating systems, but this has not been tested.

CHAPTER 2

Required packages

To quickly get started the following commands will help setup all requirements.

```
sudo apt-get install mingw-w64 python-pip nasm
sudo pip install sphinx docutils pyyaml
```

CHAPTER 3

Compilation

To compile the Monitor on a machine that has been *configured to feature all requirements* all one has to do is run `make`. The various *Components* will be automatically processed and stitched together.

CHAPTER 4

Components

The Monitor exists of a few different components which as a whole represent the project.

- *C Framework*
- *Assembly snippets*
- *Hook Definitions*
- (*Python pre-processor script(s)*)

C Framework

The majority of the Monitor has been implemented in C as this allows the most flexibility at runtime. The code can be found in `src/` and all related headers in `inc/`.

The C Framework includes the following functionality (and more):

- Hooking:

  ```
  Wrapper around the assembly snippets and creation of original function
  stub.
  ```

- Dropped Files:

  ```
  Special handling for file operations in order to automatically dump files
  that are dropped by a particular sample.
  ```

- Sleep Skip:

  ```
  Ability to (force) skip long sleeps which tend to make Cuckoo run into the
  analysis timeout.
  ```

- Unhook Detection:

```
Thread that regularly checks whether all function hooks are still as we
left them. This feature can detect samples that attempt to unhook or
overwrite the Monitor's hooks.
```

- Filtering:

```
Basic filtering of common windows-related filepaths that are generally
not interesting to Cuckoo.
```

Assembly snippets

In order to correctly handle API hooking at runtime a few layers of indirection are being used. Listed in order of execution:

- *Hook Trampoline*
- *Hook Guide*
- *Hook Cleanup*

Hook Trampoline

When a sample injected with the Monitor calls an API that has been hooked then it'll be redirected to our `trampoline` right away (through a jump.)

The `trampoline` then goes through the following steps:

- Check whether we're already inside another hook through a thread-specific counter, and if so, ignore this hook (and all the following steps):

```
For example, system() calls CreateProcessInternalW() internally. However,
as we already log the call to system() we do not have to log the call to
CreateProcessInternalW(), as that would only give us duplicate data.
```

- Increase the hook counter:

```
So that any further calls during this hook will not be logged.
```

- Save the Last Error:

```
Our hook handler, of which we have one per function, may call any number
of other API functions before calling the original function. In order to
restore the Last Error right before calling the original function the
trampoline saves the Last Error before calling the original hook handler.
```

- Save the original Return Address and replace it with one of ours:

```
In order to do some cleanup at the very end of this API call (namely right
before it returns to the caller) we save the last error just like the Last
Error and replace it with one that points to the Assembly Cleanup snippet.
```

- Jump to the hook handler:

```
At this point the hook has been setup as required and the Monitor jumps
to our hook handler. From here on the hook handler can log and modify
parameters as well as call the original function (or not call it at all,
of course.)
```

Hook Guide

In most cases the hook handler will call the `original` function. This is the point where the `guide` comes to play. The guide performs the following steps:

- Restore the Last Error:

```
At this point the Monitor restores the Last Error that had been saved by
the trampoline. Optionally the hook handler is able to overwrite the saved
Last Error before calling the original function, but in general this is
not desired - this would be more useful when modifying parameters or
return values.
```

- Save the Return Address and replace it with one of ours:

```
Just as the Monitor saved the return address in the trampoline it does the
same here. The guide replaces the return address with another address in
the guide where execution will now go to after the original function
returns.
```

- Execute the *Original Function Stub*.
- Save the Last Error:

```
We're now back in the guide right after having executed the original
function. As the original function will likely have modified the Last
Error, and we don't want the hook handler to mess it up, we save it again
here.
```

- Fetch and jump to the Return Address:

```
Finally the guide fetches the return address that was stored in the first
part of the guide.
```

So basically the `guide` does not do much special. It's one and only purpose is to ensure the Last Error is preserved correctly around the original function. Execution now continues in the hook handler which will at some point return after which we get into the `Hook Cleanup`.

Hook Cleanup

Finally the `hook cleanup` snippet performs the following tasks:

- Restore the Last Error:

```
Restore the Last Error that was saved in the guide. This is usually the
Last Error as it was right after calling the original function.
```

- Decrease the hook counter:

```
Having finished handling this function hook any further API calls should
be logged again and thus we decrease the hook counter.
```

- Fetch and jump to the Original Return Address:

```
This is the last step of our hooking mechanism - the cleanup snippet
fetches the return address as stored by the trampoline and jumps to it.
```

Original Function Stub

As the first few bytes of the original function have been overwritten by our hook we can't jump there anymore. Instead of calling the original function the hook handler will actually call a stub which contains the original instructions and a jump to the original function plus the offset to which point the stub has covered the instructions:

```
Let's assume that, like most WINAPI functions, the function prolog of a
function X looks like the following.

    mov edi, edi
    push ebp
    mov ebp, esp
    sub esp, 24
    ...

In this case the first three instructions represent five bytes together.
Effectively this means that the function would look like the following
after being hooked by the Monitor.

    jmp hook-trampoline
    sub esp, 24

Now in order to call the original function the stub will look like the
following.

    mov edi, edi
    push ebp
    mov ebp, esp
    jmp original_function+5

And that's all..
```

Hook Definitions

The Monitor features a unique and dynamic templating engine to *create API hooks <hook-create>*. These API hooks are based on a simple to use text format and are then translated into equivalent C code.

All of the API hooks can be found in the `sigs/` ("signatures") directory.

Python pre-processor script(s)

As of now there is only one Python script. This Python script takes all of the signature files and translates them into a few files in the `object/code/` directory:

- hooks.c - hook `code`.
- hooks.h - hook `prototypes`.
- explain.c - strings related to `logging` hooked API calls.
- tables.c - table containing all `hook entries` to hook.

These generated C files are compiled and used by the C Framework as a sort of data feed.

CHAPTER 5

Creating Hooks

Creating new hooks is as simple as understanding the signature format, knowing the correct return value and arguments, finding the correct signature file to put it in, and optionally know the somewhat more advanced features.

Signature Format

The signature format is a very basic reStructuredText-based way to describe an API signature. This signature is pre-processed by utils/process.py to emit C code which is in turn compiled into the Monitor.

Following is an example signature of system():

```
system
======

Signature::

    * Calling convention: WINAPI
    * Category: process
    * Is success: ret == 0
    * Library: msvcrt
    * Return value: int

Parameters::

    ** const char *command
```

The Signature block describes meta-information about the API function. The Parameters block has a list of all parameters to the function. There are also Pre, Prelog, Logging, and Post blocks. Following are all blocks described with their syntax and features - in their prefered order.

General Syntax

Each API signature starts with a small header containing the API name.

```
FunctionName
============
```

Followed is at least one block - the *Signature Block*. Each block has the block name followed by two colons, followed by one or more indented lines according to the blocks' syntax.

```
FunctionName
============

Block::

    Line #1
    Line #2
    Line #3
    ...

Block2::

    Line #1
    Line #2
```

It is recommended to keep the signatures clean and standard:

- One blank line between the API name header and the first block.
- One blank line between a block name and its contents.
- One blank line between blocks.
- Two lines between each API signature.

Available Blocks

Signature Block

The signature block describes meta-information for each API signature. The syntax for each line in the signature block is as follows.

```
Signature::

    * key: value
```

The key is converted to lowercase and spaces are replaced by underscores, the value is kept as-is.

Available keys:

- Calling Convention:

  The calling convention of this API function. This value should be be set to WINAPI.
- Category:

  The category of this API signature, e.g., file, process, or crypto.
- Library:

The DLL name of the library, e.g., kernel32 or ntdll.

- Return value:

  The return value of this function. To determine whether a function call was successful (the "is-success" flag) there are definitions for most common data types. However, some functions return an int or DWORD - these have to be handled on a per-API basis.

- Is success:

  This key is only required for non-standard return values. E.g., if an API function returns an int or DWORD then it's really up to the function to describe when it's return success or failure. However, most cases can be generalized.

  Following is a list of all available data types which have a pre-defined is-success definition.

```
BOOL = ret != FALSE
HANDLE = ret != NULL && ret != INVALID_HANDLE_VALUE
NTSTATUS = NT_SUCCESS(ret) != FALSE
HRESULT = ret == S_OK
HHOOK = ret != NULL
HINTERNET = ret != NULL
DNS_STATUS = ret == DNS_RCODE_NOERROR
SC_HANDLE = ret != NULL
void = 1
HWND = ret != NULL
SOCKET = ret != INVALID_SOCKET
HRSRC = ret != NULL
HGLOBAL = ret != NULL
PCCERT_CONTEXT = ret != NULL
HCERTSTORE = ret != NULL
NET_API_STATUS = ret == NERR_Success
SECURITY_STATUS = ret == SEC_E_OK
```

- Special:

  Mark this API signature as `special`. Special API signatures are always executed, also when the monitor is already *inside* another hook. E.g., when executing the `system()` function we still want to follow the `CreateProcessInternalW()` function calls in order to catch the process identifier(s) of the child process(es), allowing the monitor to inject into said child process(es).

Parameters Block

The parameter block describes each parameter that the function accepts - one per line. Its syntax is either of the following:

```
*  DataType VariableName
** DataType VariableName
** DataType VariableName variable_name
```

One asterisks indicates this parameter should not be logged. Two asterisks indicate that this variable should be logged. Finally, if a third argument is given, then this indicates the `alias`. In the reports you'll see the `alias`, or the `VariableName` if no alias was given, as key. Due to consistency it is recommended to use the original variable names as described in the API prototypes and to use lowercase aliases as Cuckoo-specific names.

Flags Block

This block describes values which are flags and that we would like the string representation. Its syntax should be as follows:

```
<name> <value> <flag-type>
```

The `name` should be either :

- `variable_name` (or `VariableName` if no alias was given) (see *Parameters Block*). In this case, you will get meaning of the specified parameter arg as described in flag file. (See *Flag Format*) `value` and `flag-type` will be overwritten as follows:
  - `value` = `VariableName`
  - `flag-type` = `<apiname>_<VariableName>`
- **A unique name alias. Here it's mandatory to provide :**
  - `value` : any C expression which is a flag
  - `flag-type` : Flag type block in flag file. (See *Flag Format*)

Ensure Block

The ensure block describes which parameters should never be null pointers. As an example, the `ReadFile` function has the `lpNumberOfBytesRead` parameter as optional. However, in order to make sure we know exactly how many bytes have been read we'd like to have this value at all times. This is where the ensure block makes sure the `lpNumberOfBytesRead` is not NULL.

Its syntax is a line for each parameter's VariableName:

```
Ensure::

    lpNumberOfBytesRead
```

Pre Block

The pre block allows one to execute code before any other code in the hook handler. For example, when a file is deleted using the `DeleteFile` function, the Monitor will first want to notify Cuckoo in order to make sure it can make a backup of the file before it is being deleted (also known as `dropped files` in Cuckoo reports.)

There is no special syntax for pre blocks - its lines are directly included as C code in the generated C hooks source.

As an example, a stripped down example of `DeleteFileA`'s pre block:

```
Pre::

    pipe("FILE_NEW:%z", lpFileName);
```

Prelog Block

The prelog block allows buffers to be logged before calling the original function. In functions that encrypt data possibly into the original buffer this is useful to be able to log the plaintext buffer rather than the encrypted buffer. (See for example the signature for `CryptProtectData`.)

The prelog block has the same syntax as the *Logging Block* except for the fact that at the moment only **one** `buffer` line is supported. (Mostly because there has been no need for other data types or multiple buffers yet.)

Middle Block

The middle block executes arbitrary C code after the original function has been called but before the function call has been logged. Its syntax is equal to the *Pre Block*.

Replace Block

The replace block allows one to replace the parameters used when calling the original function. This is useful when a particular argument has to be swapped out for another parameter.

Logging Block

The logging block describes data that should be logged after the original function has been called but that is not really possible to explain in the *Parameters Block*. For example, many functions such as `ReadFile` and `WriteFile` pass around buffers which are described by a length parameter and a parameter with a pointer to the buffer.

Each line in the logging block should be as follows:

```
Logging::

    <format-specifier> <parameter-alias> <the-value>
```

The `format specifier` should be one of the values as described in `inc/pipe.h`. The alias is much like the aliases from *Parameters Block*. The value is any C expression that will get the correct value.

Following are some examples (with stripped down API signatures):

```
ReadFile
========

Logging::

    B buffer lpNumberOfBytesRead, lpBuffer


CreateProcessInternalW
======================

Ensure::

    lpProcessInformation

Logging::

    i process_identifier lpProcessInformation->dwProcessId
    i thread_identifier lpProcessInformation->dwThreadId
```

Post Block

The post block executes arbitrary C code after the original function has been called and after the function call has been logged. Its syntax is equal to the *Pre Block*.

Logging API

In order to easily log the hundreds of parameters that the various API signatures feature a standardized logging format string has been developed that supports all currently-used data types.

The `log_api()` function accepts such format strings. However, one does not have to call this function as the calls to `log_api()` is automatically generated by the Python pre-processor script. (In fact, this would currently result in undefined behavior, so don't do it.)

Logging Format Specifier

Following is a list of all currently supported format specifiers:

- `s`: zero-terminated ascii string
- `S`: ascii string with length
- `u`: zero-terminated unicode string
- `U`: unicode string with length in characters
- `b`: buffer pointer with length
- `B`: buffer pointer with pointer to length
- `i`: 32-bit integer
- `l`: 32-bit or 64-bit long
- `p`: pointer address
- `P`: pointer to pointer address
- `o`: pointer to `ANSI_STRING`
- `O`: pointer to `UNICODE_STRING`
- `x`: pointer to `OBJECT_ATTRIBUTES`
- `a`: array of zero-terminated ascii strings
- `A`: array of zero-terminated unicode strings
- `r`: registry stuff - to be updated
- `R`: registry stuff - to be updated
- `q`: 64-bit integer
- `Q`: pointer to 64-bit integer (e.g., pointer to `LARGE_INTEGER`)
- `z`: bson object
- `c`: REFCLSID object

Flag Format

The `flag format` is a very basic `reStructuredText`-based way to describe meaning of bit flag argument in Windows API. It is found in `flags/` This flag is pre-processed by `utils/process.py` to emit C code which is in turn compiled into the Monitor.

General Syntax

Each flag starts with a small header containing the flag type.

```
FlagType
========
```

Followed is at least one block. Each block has the block name followed by two colons, followed by one or more indented lines according to the blocks' syntax.

```
FlagType
========

Block::

    <value>
    <value1>
    <value2>
    ...

Block2::

    <value1>
    <value2>
```

It is recommended to keep flags clean and standard:

- One blank line between the Flag type header and the first block.
- One blank line between a block name and its contents.
- One blank line between blocks.
- Two lines between each flag type.

Available Blocks

Inherits Block

Value Block

This block defines possible values when only one flag could be set.

Enum block

This block defines possible values in a bitwise manner.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search