PyREbox Documentation

Release 1.0

Xabier Ugarte-Pedrero

Nov 19, 2018

Contents


PyREBox

CHAPTER 1

Fetching PyREBox

You should always download PyREBox from either the *master* or *dev* branches in GitHub. Please clone the repository using *git clone* instead of downloading the zip/tar package, as the git metadata contains important information for downloading and installing required submodules during the installation process.

CHAPTER 2

Building PyREBox

Installing dependencies for Debian based distributions:

```
apt-get install build-essential zlib1g-dev pkg-config libglib2.0-dev binutils-dev␣
↪libboost-all-dev autoconf libtool libssl-dev libpixman-1-dev libpython-dev python-
↪pip python-capstone virtualenv
```

Installing dependencies for CentOS 7:

```
yum groupinstall 'Development Tools'
yum install zlib-devel.x86_64 glib2-devel.x86_64 binutils-devel.x86_64 boost-devel.
↪x86_64 autoconf.noarch libtool.x86_64 openssl-devel.x86_64 pixman-devel.x86_64␣
↪python-devel.x86_64 libfdt-devel
yum install epel-release
yum install python-virtualenv python34-pip.noarch python2-pip.noarch
```

For RHEL/Fedora:

```
dnf install make automake gcc gcc-c++ kernel-devel zlib-devel pkgconf-pkg-config␣
↪glib2-devel binutils-devel boost-devel autoconf libtool openssl-devel pixman-devel␣
↪python2-devel python2-pip python2-virtualenv capstone-python
```

Required python packages (see the next paragraph for installation instructions):

```
ipython>=5,<6 sphinx sphinx-autobuild prettytable pefile capstone distorm3 pycrypto␣
↪pytz
```

We strongly recommend to use a virtual env to install your python dependencies. If you have a local installation of volatility, it will intefere with the volatility package used by PyREBox. Create the virtual env:

```
virtualenv pyrebox_venv
```

Once it has been created, activate it in order to install your python dependencies:

```
source pyrebox_venv/bin/activate
```

To install the python dependencies you can use pip:

```
pip install -r requirements.txt
```

Do not forget to activate your virtual env every time you want to start PyREBox!

```
source pyrebox_venv/bin/activate
```

Project configuration and building

```
./build.sh
```

CHAPTER 3

Installing PyREBox

PyREBox package installation is not yet supported.

CHAPTER 4

Creating a VM image for PyREBox

At this moment, PyREBox supports any Windows image (32 and 64 bit) that is supported by Volatility.

You can create your own image using KVM. In order to avoid compatibility problems, use the pyrebox binaries instead of your system installation qemu binaries:

```
qemu-img create -f qcow2 -o compat=0.10 images/xpsp3.qcow2 4G
./pyrebox-i386 -m 256 -monitor stdio -usb -drive file=images/xpsp3.qcow2,index=0,
↪media=disk,format=qcow2,cache=unsafe -cdrom images/WinXP.iso -boot d -enable-kvm
```

Proceed with installation, and then boot with network (don't use -net none) and usb support (-usb), and plug in a usb (see Loading a USB image). Let the system install all the drivers

```
./pyrebox-i386 -m 256 -monitor stdio -usb -drive file=images/xpsp3.qcow,index=0,
↪media=disk,format=qcow2,cache=unsafe -netdev user,id=network0 -device rtl8139,
↪netdev=network0
```

Basic QEMU usage documentation: ——————-

PyREBox is based on QEMU, so in order to start a VM within PyREBox, you need to run it exactly as if you were booting up a QEMU VM. A couple of example scripts are provided: `start_i386.sh`, `start_x86_64.sh`, you can use them as an example.

The only QEMU monitor option supported currently is *stdio* (`-monitor stdio`).

Some useful QEMU parameters are the following:

Memory, in megabytes

```
-m 256
```

Start a prompt on standard input/output in order to interact with the qemu monitor

```
-monitor stdio
```

Enable usb support

```
-usb
```

If the host(local) mouse pointer isn't properly synchronized with guest(remote) mouse pointer, add

```
-device usb-tablet
```

You can specify main image file with unsafe caching. Unsafe caching will make snapshoting much faster

```
-drive file=images/xpsp3.qcow,index=0,media=disk,format=qcow2,cache=unsafe
```

Disable networking interfaces. See QEMU documentation for other configuration options

```
-net none
```

Start vm at its first snapshot

```
-loadvm 1
```

Once you start a VM, you will have a QEMU prompt in which you can run all the QEMU commands, plus those implemented in PyREBox.

4.1 Snapshots

You can load an snapshot when starting a VM by using the -loadvm [snapshot] argument, where [snapshot] is the snapshot number or descriptor. Snapshots taken when running with KVM are not compatible with snapshots taken when running the whole system emulation approach (no KVM). So, in order to take a snapshot that can be loaded with pyrebox, you should not enable KVM for it. Booting up the operating system will be slower, but hopefully you will only need to do this once.

List snapshots

```
(qemu)info snapshots
```

Creating an snapshot

```
(qemu)savevm init
```

Loading an snapshot

```
(qemu)loadvm init
(qemu)loadvm 1
```

4.2 Networking

Refer to QEMU documentation. By default, the option `-net none` disables networking.

User-mode networking interfaces

```
-netdev user,id=network0 -device rtl8139,netdev=network0
```

4.3 Loading a usb image (with files)

Create a usb image template

```
qemu-img create -f raw usb_image_template.img 256M
```

Boot QEMU/PyREBox, with usb support `-usb`, and run the following commands:

```
(qemu) drive_add 0 if=none,id=stick,file=/path/to/usb_image.img,format=raw
(qemu) device_add usb-storage,id=stick,drive=stick
```

On your guest system, partition and format the usb drive. Finally, umount it (safe extract).

Remove the USB drive from QEMU/PyREBox

```
(qemu) device_del stick
```

If you are not sure about which USB drive to remove, you can use the command `info usb`.

Keep the file, because it can be useful as an empty USB drive template.

Copy the image template (usb_image_template.img) to a new file, and then mount and modify it

```
mount -o loop,offset=32256 usb_image.img /mnt/location
```

Copy files to /mnt/location

Unmount

```
umount /mnt/location
```

Finally, plug usb image in the machine, and use it!

```
(qemu)usb_add disk:/path/to/usb/image
```

4.4 Sharing a host directory

Check out existing documentation for sharing a host directory with the guest via SAMBA.

CHAPTER 5

Basic PyREBox usage

Once you start a VM, you will have a (qemu) prompt in which you can run all the QEMU commands.

PyREBox will first read its configuration file (pyrebox.conf).

```
[MODULES]
scripts.script_example.py: True
scripts.volatility_example: False

[VOL]
profile: WinXPSP3x86

[AGENT]
name: win_agent_32.exe
conf: win_agent_32.exe.conf

[SYMBOL_CACHE]
path: symbols.WinXPSP3x86
```

The [MODULES] section contains a list of python modules (packages and subpackages can be specified using standard python notation (using dots)). You can enable or disable scripts on demand. These scripts will be automatically loaded.

The [VOL] section contains the volatility configuration. You will need to adjust the profile according to your operating system version.

The [AGENT] section allows you to configure the name of the agent binary (see documentation related to the agent), and the configuration file for that binary.

The [SYMBOL_CACHE] section, allows you to speficy the path for a json file that will be used by PyREBox to preserve resolved symbols between different sessions. This path should be unique for each qemu image you have, and improves significantly the performance once it is loaded with data on the first execution of the system.

There are PyREBox commands that will allow you to load/unload scripts:

Import a module and initialize it

```
(qemu) import_module scripts.my_plugin
```

List loaded modules

```
(qemu) list_modules
```

Reload a module, by module handle (you can obtain this handle by listing loaded modules)

```
(qemu) reload_module 1
```

Unload a module, by module handle (you can obtain this handle by listing loaded modules)

```
(qemu) unload_module 1
```

Start the PyREBox shell

```
(qemu) sh
```

CHAPTER 6

PyREBox Shell

NOTE: At this moment the only supported option for monitor is `-monitor stdio`.

6.1 Shell environment basics

PyREBox offers a command line that allows to inspect and manipulate the VM at runtime, in different ways. We can distinguish 2 different shell environments: QEMU and PyREBox.

- When PyREBox (QEMU) is started, we will see the QEMU prompt. This shell allows to execute any QEMU command.
- If we type the `sh` command, the machine will be paused and we will enter into the PyREBox shell. This mode allows to run any PyREBox command in an IPython shell environment. PyREBox also exposes its complete API so that you can use it programmatically in this shell using python.

Here, we can obtain a full list of commands by typing the following commands:

```
%list_commands
%list_vol_commands
```

The first command will provide us a list of PyREBox commands, as well as all the custom commands defined in dynamically loaded scripts. The second command will list all the available volatility commands. All the commands can be optionally preceded by the `%` character.

Scripts can define new **custom** commands. **For running custom commands**, we will need to type `%custom` followed by the command name and the parameters for the command.

Additionally, PyREBox exposes the volatility framework both to the PyREBox shell and the scripting engine. **For running a volatility command**, we will need to type `%vol` followed by the volatility command name and its parameters. The list of volatility plugins is generated automatically, so any plugin added to the corresponding volatility directory located in our PyREBox directory will available be for us.

```
[4] pyrebox> %list_vol_commands

        Supported volatility commands:

                amcache         Print AmCache information
                apihooks        Detect API hooks in process and kernel memory
                atoms           Print session and window station atom tables
                atomscan        Pool scanner for atom tables
                auditpol        Prints out the Audit Policies from HKLM\SECURITY\Policy\PolAdtEv
                bigpools        Dump the big page pools using BigPagePoolScanner
                bioskbd         Reads the keyboard buffer from Real Mode memory
                cachedump       Dumps cached domain hashes from memory
                callbacks       Print system-wide notification routines
                clipboard       Extract the contents of the windows clipboard
                cmdline         Display process command-line arguments
                cmdscan         Extract command history by scanning for _COMMAND_HISTORY
                connections     Print list of open connections [Windows XP and 2003 Only]
                connscan        Pool scanner for tcp connections
                consoles        Extract command history by scanning for _CONSOLE_INFORMATION
                crashinfo       Dump crash-dump information
                deskscan        Poolscaner for tagDESKTOP (desktops)
                devicetree      Show device tree
                dlldump         Dump DLLs from a process address space
                dlllist         Print list of loaded dlls for each process
                driverirp       Driver IRP hook detection
                drivermodule    Associate driver objects to kernel modules
                driverscan      Pool scanner for driver objects
                dumpcerts       Dump RSA private and public SSL keys
                dumpfiles       Extract memory mapped and cached files
                dumpregistry    Dumps registry files out to disk
                editbox         Displays information about Edit controls. (Listbox experimental.)
                envars          Display process environment variables
```

```
[13] pyrebox(1f0)> vol pslist
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------------------
0x817caa00 System                    4      0     72      150 ------      0
0x817d9b28 smss.exe                312      4      3       19 ------      0 2017-06-02 20:31:44 UTC+0000
0x81621020 csrss.exe               408    312     11      305      0      0 2017-06-02 20:31:45 UTC+0000
0x81620ad0 winlogon.exe            432    312     21      503      0      0 2017-06-02 20:31:46 UTC+0000
0x81609a00 services.exe            484    432     16      246      0      0 2017-06-02 20:31:48 UTC+0000
0x81604da0 lsass.exe               496    432     23      325      0      0 2017-06-02 20:31:48 UTC+0000
0x8179f020 svchost.exe             652    484     19      193      0      0 2017-06-02 20:32:00 UTC+0000
0x815f0998 svchost.exe             760    484      9      208      0      0 2017-06-02 20:32:29 UTC+0000
0x815ee488 svchost.exe            1064    484     58     1028      0      0 2017-06-02 20:33:55 UTC+0000
0x81605a08 svchost.exe            1084    484      5       58      0      0 2017-06-02 20:33:57 UTC+0000
0x8160e838 svchost.exe            1148    484     13      178      0      0 2017-06-02 20:34:03 UTC+0000
0x817427a0 explorer.exe           1156   1120     11      305      0      0 2017-06-02 20:34:04 UTC+0000
0x8161c880 spoolsv.exe            1304    484     15      122      0      0 2017-06-02 20:34:11 UTC+0000
0x81594298 alg.exe                1768    484      7      100      0      0 2017-06-02 20:34:30 UTC+0000
0x8158f020 wscntfy.exe            1796   1064      1       26      0      0 2017-06-02 20:34:33 UTC+0000
0x815eb020 wuauclt.exe             112   1064      8      176      0      0 2017-06-02 20:35:04 UTC+0000
```

In order to exit the PyREBox shell, we can type `quit`, `q`, `c`, or `continue`, or just ctrl-d.

6.2 Some IPython features

- **Tab completion**. You can use tab completion in order to inspect the objects and commands during your VM inspection sessions.
- **The ‘‘cpu‘‘ object**. When the *sh* command starts a new PyREBox shell, the state of the CPU is exposed as an object named cpu, which will allow you to retrieve the status for every register in the CPU.
- **Python code in PyREBox commands**. You can embedd python code when running a PyREBox command by using the IPython variable expansion features, such as $ and {}. Example: `bpr {hex(cpu.ESP - 0x10)}:0x20`. The previous command will place a memory read breakpoint in the memory sorrounding the current stack pointer (ESP). The expression between {} symbols will be interpreted as python code.

- **Listing available objects**. Apart from the `cpu` object, whenever you create a variable in the IPython interpreter, it will be kept for the following session. You can list the available variables by typing the command `%who`.
- **Obtaining help for a command or API function**. The PyREBox shell will also allow you to obtain information about the different available commands, as well as the API which is exposed to both the PyREBox shell and the scriting engine. You just need to type help(*command*|*api_funcion*), or *command*? | *api_function*? in order to obtain the object's documentation.

6.3 QEMU monitor commands

PyREBox adds four commands on top of QEMU:

Commmand	Description
`sh`	Start PyREBox shell
`import_module`	Load a PyREBox script (module)
`list_modules`	List loaded PyREBox scripts (modules)
`unload_module`	Unload a PyREBox script (module)
`reload_module`	Reload a PyREBox script (module)

6.4 PyREBox shell commands

Once you enter the PyREBox shell, you can run any PyREBox command. You can use the `help()` python function or the `?` suffix to obtain a description of the command and its parameters. Many of the commands follow the same naming as Windbg commands, but not necessarily the same syntax.

Conventions:

- In order to use a virtual address, just use the hex address: `0x7c871235`. You can specify physical addresses in the following way: `p0x00100000`
- The commands that involve referencing virtual addresses require to work on a process context. Use the `proc` command to set the process context to one specific running process. You can specify either the process name (or part if it), its PGD, or its PID.

Shell operation commands:

Commmand	Description
`list_commands`	List the PyREBox commands available.
`list_vol_commands`	List the Volatility commands available.
`vol comm`	Enter a Volatility command.
`custom comm`	Enter a custom command
`proc`	Specify working context to a given process. You can specify its PID, CR3, or (part of) the name. Subsequent virtual address will refer to that process.
`mon`	Monitor a process. Many events are triggered only for monitored processes.
`unmon`	Stop monitoring a process.
`set_cpu`	Set the CPU number to operate on (when QEMU is configured to have several CPUs).
`q`	Exit PyREBox shell and continue execution.
`c`	Exit PyREBox shell and continue execution.
`continue`	Exit PyREBox shell and continue execution.
`quit`	Exit PyREBox shell and continue execution.
`ctrl-d`	Exit PyREBox shell and continue execution.
`help(*api*)`	Obtain help for an API function or constant.
`help(*comm*)`	Obtain help for a PyREBox command.
`*api*?`	Obtain help for an API function.
`*comm*?`	Obtain help for a PyREBox command.

Breakpoints. A PyREBox shell is started when a breakpoint is hit.

Command	Description
`bd`	Disable breakpoint
`bl`	List breakpoints
`be`	Enable existing breakpoint
`bp`	Set a breakpoint at an address. It accepts virtual and physical addresses, as well as API names. It also accepts memory ranges (e.g., break whenever an instruction in some memory range is executed)
`bpr`	Set a breakpoint at a memory read. Similar to bp, but triggers when the address/range is read.
`bpw`	Set a breakpoint at a memory write. Similar to bp, but triggers when the address/range is written.

Introspection

Command	Description
`ps`	List running processes.
`lm`	List modules for a process. Specify process by pid, name, or cr3. Specify '0', 'System' or 'kernel' in order to list kernel modules

Machine state inspection and manipulation

Commmand	Description
`print_cpu`	Show cpu.
`dis`	Dissassemble at PC on currently running process.
`u`	Disassemble at a given address.
`db,dw,dd,dq`	Display byte,word,dword,qword of data in memory.
`dump`	Display any size of data in memory.
`eb,ew,ed,eq`	Overwrite byte,word,dword,qword of data in memory. Accepts HEX, ANSI strings and unicode strings.
`write`	Overwrite any size of data in memory. Accepts HEX, ANSI strings and unicode strings.
`r`	Display and/or manipulate a register. See help.
`ior[b|w|d]`	Read IO port address (byte, word, dword).
`iow[b|w|d]`	Write IO port address (byte, word, dword).

Symbols

Command	Description
`ln`	List nearest symbols (APIs) to a given address.
`x`	List address of a symbol (API). You can use substrings and wildcards. Format is *modulename!api*, you can specify one of them or both.

Other commands

Command	Description
`strings`	Display strings in a given memory region.
`s`	Search pattern (hex, ASCII or unicode string) in a given memory region.
`savevm`	Save a vm snapshot. Specify snapshot name that can be a number or string with no apostrophes. E.g.: savevm 1, savevm my_snapshot
`loadvm`	Load a vm snapshot. Specify snapshot name that can be a number or string with no apostrophes. E.g.: loadvm 1, loadvm my_snapshot

CHAPTER 7

Scripting in PyREBox

PyREBox scripts (see examples under `scripts/` folder) allow to:

- Define new commands for the PyREBox environment.
- Define callbacks (functions that will be called on different events for each of the monitored processes).
- Assign triggers to callbacks.
- **Use the python API exported by PyREBox, allowing:**
  - To query processes, modules.
  - To query symbols (API name resolution).
  - To read and manipulate registers and memory.
  - To start a PyREBox shell.
  - To make use of the volatility library.

7.1 Examples

You can find self-documented examples in the scripts folder. New scripts and contributions will be added there.

7.2 Script life-cycle

PyREBox scripts can be loaded and unloaded dynamically at any moment during the execution of the VM. The configuration file pyrebox.conf allows to specify a list of scripts that should be loaded at startup.

After this moment, you can load or unload scripts using the `import_module` and `unload_module` commands on QEMU's prompt.

Additionally, if a script has been loaded but you modified its code, you can reload it using the `reload_module` command.

In the `scripts/` directory you can find a good self-documented example PyREBox script.

Any PyREBox script needs to implement 2 basic functions: `initialize_callbacks` and `clean`. The first one will be called when the script is loaded, while the second one will be called when it is unloaded. The first one can register the callbacks you want to listen to, while the latter should unregister them. The clean() function of the CallbackManager class will help you to unregister all the active callbacks previously registered for a CallbackManager instance.

A script can optionally have one additional member named `requirements`, which consists of a list of additional scripts (in python module notation), that should get loaded before the script can be loaded.

Once the initialization function has been executed, the script will only be executed if:

- One of the custom commands implemented in your script is executed.
- Some callback is triggered by an event in the system.

Another important aspect is the concept of *monitored processes*. In general, the callbacks will only be triggered for those processes being monitored. This limits the number of events that trigger the execution of our script. You can set and unset which processes you want to monitor both from the python API and from the interactive shell. Also, at any point you will be able to see which processes are being monitored running the `ps` command in the PyREBox shell environment.

There are some exceptions to these rules:

- **Block begin / Instruction begin**. If an address and pgd are specified for the callback (see examples), this callback will be triggered only for that address and process, no matter if it is monitored or not.
- **Keystroke callback**. It will be called for all the processes in the system and in the context of any process in the system.
- **NIC send/receive**. It will be called for all the processes in the system and in the context of any process in the system.
- **Opcode range callback**. It will be called at the instruction end for instructions with the specified opcodes, only for the monitored processes.
- **Triggers**. Triggers are C/C++ compiled shared objects that are associated to a given callback. This code will be executed before the python callback function is called, and can decide whether the callback should be delivered to the python function or not. This approach allows to improve the overall performance by setting arbitrary callback conditions. When a trigger is attached to a callback, the trigger will be executed for every event (no matter if the process is being monitored or not), and it is the responsibility of the developer to check that the callback happened in the appropiate context (usually checking the PGD, that determines the current address space).

7.3 Defining a new command

In order to define a new command, you just need to declare a function in your script with the following prototype:

```
def do_command(self,line):
```

Where `command` is the command name you want to create, and `line` is the argument containing all the command arguments. When the script is loaded, the command will be available in the PyREBox shell. In order to use the command, you will need to use the `custom` keyword, followed by the command name and its parameters. You can document your command using the standard python docstring based documentation, that will be automatically loaded by PyREBox.

7.4 Callback types

This section lists the different callback types, together with a description of the callback and the parameters provided to the function.

7.4.1 Old style vs new style callback functions

PyREBox supports 2 different styles for callback functions. The old style requires each function to accept a variable number of positional arguments. The number and meaning of each positional argument depends on the callback type.

In the new style, in contrast, callback functions have one single argument: a dictionary. In this dictionary, the keys are strings representing the parameter name, and the value represents the value associated to such parameter. This style simplifies the callback interfacem and will allow to add callback parameters in the future without breaking backwards compatibility.

At this moment, PyREBox defaults to old-style in order to preserve compatibility. Nevertheless, whenever the user loads a script using old-style parameters, a warning is shown informing that the style is deprecated and will be removed in the future. New-style parameters can be enabled by providing the argument new_style = True when either CallbackManager or the BP class are instantiated. Once a script is new_style compliant, the warning message will not be shown again.

There are some common data types used in many callbacks.

Parameter	Description
cpu	Object representing the CPU state. It will contain one member (field) for every register in the CPU
tb	Tuple containing information about the translation block (set of instructions translated at one time) by QEMU, similar in concept to a basic block. The tuple contains 3 values: (pc,size,icount), where pc is the program counter of the first instruction, size is the size of the block, and icount the number of instructions in it. **Translation blocks may not necessarily match basic blocks. The QEMU emulator will disassemble instruction by instruction until it finds either a control flow instruction, or a point where the next address cannot be guessed statically. All these instructions conform a translation block. Note that in some cases (e.g. special instructions), translation blocks may not necessarily match basic blocks.**

7.4.2 Block begin

The callback is triggered for every executed translation block in the context of the monitored processes, at the beginning of the translation block. It is useful for tracing translation blocks. It allows to specify an address and PGD. In such a case, it will be triggered only for that address and process address space, no matter if the process is monitored or not.

Callback type: `CallbackManager.BLOCK_BEGIN_CB`

Example:

```
cm.add_callback(CallbackManager.BLOCK_BEGIN_CB, my_function)

cm.add_callback(CallbackManager.BLOCK_BEGIN_CB, my_function, address=address, pgd=pgd)
```

Old-style callback interface:

```
def my_function(cpu_index, cpu, tb):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "cpu": ...,
 "tb": ...}
```

7.4.3 Block end

The callback is triggered for every executed translation block in the context of the monitored processes, at the end of the translation block. It is useful for tracing translation blocks. The `cur_pc` parameter represents the current instruction pointer, while `next_pc` represents the next instruction to execute. When the callback is triggered, the emulated cpu is already at the start of the next instruction.

Callback type: `CallbackManager.BLOCK_END_CB`

Example:

```
cm.add_callback(CallbackManager.BLOCK_END_CB, my_function)
```

Old-style callback interface:

```
def my_function(cpu_index, cpu, tb, cur_pc, next_pc):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "cpu": ...,
 "tb": ...,
 "cur_pc": ...,
 "next_pc": ...}
```

7.4.4 Instruction begin

Similar to previous callbacks, but at instruction level. Useful to trace single instructions. It allows to specify an address and pgd. In such a case, it will be triggered only for that address no matter if the process is monitored or not.

Callback type: `CallbackManager.INSN_BEGIN_CB`

Example:

```
cm.add_callback(CallbackManager.INSN_BEGIN_CB, my_function)

cm.add_callback(CallbackManager.INSN_BEGIN_CB, my_function, addr=addr, pgd=pgd)
```

Old-style callback interface:

```
def my_function(cpu_index, cpu):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "cpu": ...}
```

7.4.5 Instruction end

Similar to previous callbacks, but at instruction level. Useful to trace single instructions. When the callback is triggered, the emulated cpu is already at the start of the next instruction.

Callback type: `CallbackManager.INSN_END_CB`

Example:

```
cm.add_callback(CallbackManager.INSN_END_CB, my_function)
```

Old-style callback interface:

```
def my_function(cpu_index, cpu):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "cpu": ...}
```

7.4.6 Memory read

Triggered whenever any memory address is read in any of the processes monitored. The parameter `vaddr` represents the modified virtual address. `haddr` is the corresponding physical address, and `size` is the size of the modification.

Callback type: `CallbackManager.MEM_READ_CB`

Example:

```
cm.add_callback(CallbackManager.MEM_READ_CB, my_function)
```

Old-style callback interface:

```
def my_function(cpu_index, vaddr, size, haddr):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "vaddr": ...,
 "size": ...,
 "haddr": ...}
```

7.4.7 Memory write

Triggered whenever any memory address is written in any of the processes monitored. The parameter `vaddr` represents the modified virtual address. `haddr` is the corresponding physical address, and `size` is the size of the modification. The callback is called *after* the memory has been written. The `data` parameter contains the written memory value.

Callback type: `CallbackManager.MEM_WRITE_CB`

Example:

```
cm.add_callback(CallbackManager.MEM_WRITE_CB, my_function)
```

Old-style callback interface:

```
def my_function(cpu_index, vaddr, size, haddr, data):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "vaddr": ...,
 "size": ...,
 "haddr": ...,
 "data": ...}
```

7.4.8 Keystroke event

Triggered whenever a key is pressed into the system.

Callback type: `CallbackManager.KEYSTROKE_CB`

Example:

```
cm.add_callback(CallbackManager.KEYSTROKE_CB, my_function)
```

Old-style callback interface:

```
def my_function(keycode):
    ...
```

New-style callback parameters:

```
{"keycode": ...}
```

7.4.9 NIC send

Triggered whenever data is sent through the network interface. This event requires the network card to be configured in this way:

```
-device ne2k_pci,netdev=network0
```

The parameter `addr` represents the address of the buffer, `size` represents its size, and buffer is the content being sent.

Callback type: `CallbackManager.NIC_SEND_CB`

Example:

```
cm.add_callback(CallbackManager.NIC_SEND_CB, my_function)
```

Old-style callback interface:

```
def my_function(addr,size,buf):
    ...
```

New-style callback parameters:

```
{"vaddr": ...,
 "size": ...,
 "buf": ...}
```

7.4.10 NIC receive

Triggered whenever data is received through the network interface. This event requires the network card to be configured in this way:

```
-device ne2k_pci,netdev=network0
```

The parameter `size` represents its size, and `buffer` is the content being sent.

Callback type: `CallbackManager.NIC_REC_CB`

Example:

```
cm.add_callback(CallbackManager.NIC_REC_CB, my_function)
```

Old-style callback interface:

```
def my_function(buf, size, cur_pos, start, stop):
    ...
```

New-style callback parameters:

```
{"buf": ...,
 "size": ...,
 "cur_pos": ...,
 "start": ...,
 "stop": ...}
```

7.4.11 Opcode range callback

Triggered whenever an instruction with an opcode in the specified range is executed. E.g.: trigger for all call instructions, for the monitored processes. This callback presents some particularities:

- The callback is called after the instruction has been executed. The cpu parameter corresponds to this new state. Interrupt instructions are an exception. In those cases, it happens at instruccion beginning.
- The `pc` parameter corresponds to the PC where the involved instruction was located.
- The `next_pc` parameter corresponds to the next instruction. It might be 0 if the address is not provided in the instruction (e.g.: interrupts or return instructions).

Callback type: `CallbackManager.OPCODE_RANGE_CB`

Example:

```
cm.add_callback(CallbackManager.OPCODE_RANGE_CB, my_function, start_opcode=0xE8, end_
↪opcode=0xE9)
```

Old-style callback interface:

```
def my_function(cpu_index, cpu,pc, next_pc):
    ...
```

New-style callback parameters:

```
{"cpu_index": ...,
 "cpu": ...,
 "pc": ...,
 "next_pc": ...}
```

7.4.12 TLB callback

Triggered for every TLB flush callback.

Callback type: `CallbackManager.TLB_EXEC_CB`

Example:

```
cm.add_callback(CallbackManager.TLB_EXEC_CB, my_function)
```

Old-style callback interface:

```
def my_function(cpu, vaddr):
    ...
```

New-style callback parameters:

```
{"cpu": ...,
 "vaddr": ...}
```

7.4.13 Context change

Triggered for every context change.

Callback type: `CallbackManager.CONTEXTCHANGE_CB`

Example:

```
cm.add_callback(CallbackManager.CONTEXTCHANGE_CB, my_function)
```

Old-style callback interface:

```
def my_function(old_pgd, new_pgd):
    ...
```

New-style callback parameters:

```
{"old_pgd": ...,
 "new_pgd": ...}
```

7.4.14 Create process

Triggered whenever a new process is created in the system. Parameters are self-descriptive.

Callback type: `CallbackManager.CREATEPROC_CB`

Example:

```
cm.add_callback(CallbackManager.VMI_CREATEPROC_CB, my_function)
```

Old-style callback interface:

```
def my_function(pid, pgd, name):
    ...
```

New-style callback parameters:

```
{"pid": ...,
 "pgd": ...,
 "name": ...}
```

7.4.15 Remove process

Triggered whenever a new process is killed in the system. Parameters are self-descriptive.

Callback type: `CallbackManager.REMOVEPROC_CB`

Example:

```
cm.add_callback(CallbackManager.REMOVEPROC_CB, my_function)
```

Old-style callback interface:

```
def my_function(pid, pgd, name):
    ...
```

New-style callback parameters:

```
{"pid": ...,
 "pgd": ...,
 "name": ...}
```

7.4.16 Module load

Triggered whenever a library or a driver is loaded in the address space of a process. Parameters are self-descriptive.

Callback type: `CallbackManager.LOADMODULE_CB`

Example:

```
cm.add_callback(CallbackManager.LOADMODULE_CB, my_function, pgd = cpu.CR3)
```

Old-style callback interface:

```
def my_function(pid, pgd, base, size, name, fullname):
    ...
```

New-style callback parameters:

```
{"pid": ...,
 "pgd": ...,
 "base": ...,
 "size": ...,
 "name": ...,
 "fullname": ...}
```

7.4.17 Module remove

Triggered whenever a library or a driver is removed from the address space of a process. Parameters are self-descriptive.

Callback type: `CallbackManager.REMOVEPROC_CB`

Example:

```
cm.add_callback(CallbackManager.REMOVEMODULE_CB, my_function, pgd = cpu.CR3)
```

Old-style callback interface:

```
def my_function(pid, pgd, base, size, name, fullname):
    ...
```

New-style callback parameters:

```
{"pid": ...,
 "pgd": ...,
 "base": ...,
 "size": ...,
 "name": ...,
 "fullname": ...}
```

7.5 Triggers

Triggers are libraries developed in C/C++ that are compiled into native code and loaded at runtime. These triggers define a function named `trigger` that can perform any necessary computation and use the API offered by `qemu_glue.h`. This function will then decide if the attached python callback should be executed or not. If the function returns 1, the python callback will be executed. If the function returns 0, the python callback is not executed.

When a trigger is added to a callback, it will be called for every event happening in any process context (not only monitored processes). Note that this is different from the default behavior in certain callback types. For instance, if we add a block begin callback and attach a trigger to it, the trigger will be called every time a block is executed in any process on the system. The trigger should then decide whether the event must be followed by a python callback function call, or be ignored, by checking the process context, or any other relevant value.

Triggers can access variables associated to the callback (trigger variables), which can be set in the python script once the trigger has been loaded.

You can find several examples of triggers under directory `triggers/`.

Each trigger has to implement 3 functions (using the extern "C" clause): `get_type`, `trigger`, and `clean`.

- **get_type** should return the callback type it can be loaded into. The system will not allow us to load a trigger into an incomptible callback type.

- **trigger** should return 1 if the callback should be executed, and 0 otherwise.
- **clean** should clean all the variables (and deallocate memory), and it will be called only once, when the trigger is unloaded.

These triggers allow us to:

- Precompute some condition and decide whether to call the python callback (reduce run-time overhead).
- Precompute some value efficiently and store it in some variable that can be read afterwards from python.

In order to access variables, we need to use the functions `get_var()`, and `set_var()`.

```
void* get_var(callback_handle_t handle, const char* key_str);
void set_var(callback_handle_t handle, const char* key_str,void* val);
```

The value is a pointer in all cases. When a variable is created, you should allocate some memory and pass to the function the address of your allocated memory. If we call set_var() for an already existing variable, it will deallocate the memory pointed by the previous variable by calling free() over the pointer.

Be careful with using complex data structures, because the set_var() will only call free over the pointed chunck. It is your responsibility to avoid memory leaks when using these variables.

In order to create variables in a trigger accesible from python code (in its triggered python callback), see the provided examples and be careful with reference counting and garbage collection (scripts/getset_var_example.py).

Bellow you can find the definition of the callback_params_t type

```
typedef struct block_begin_params {
    int cpu_index;
    qemu_cpu_opaque_t cpu;
    qemu_tb_opaque_t tb;
} block_begin_params_t;

typedef struct block_end_params {
    int cpu_index;
    qemu_cpu_opaque_t cpu;
    qemu_tb_opaque_t tb;
    pyrebox_target_ulong cur_pc;
    pyrebox_target_ulong next_pc;
} block_end_params_t;

typedef struct insn_begin_params {
    int cpu_index;
    qemu_cpu_opaque_t cpu;
} insn_begin_params_t;

typedef struct insn_end_params {
    int cpu_index;
    qemu_cpu_opaque_t cpu;
} insn_end_params_t;

typedef struct mem_read_params {
    int cpu_index;
    pyrebox_target_ulong vaddr;
    pyrebox_target_ulong paddr;
    pyrebox_target_ulong size;
} mem_read_params_t;

typedef struct mem_write_params {
```

(continues on next page)

(continued from previous page)

```
    int cpu_index;
    pyrebox_target_ulong vaddr;
    pyrebox_target_ulong paddr;
    pyrebox_target_ulong size;
} mem_write_params_t;

typedef struct keystroke_params {
    unsigned int keycode;
} keystroke_params_t;

typedef struct nic_rec_params {
    unsigned char* buf;
    uint64_t size;
    uint64_t cur_pos;
    uint64_t start;
    uint64_t stop;
} nic_rec_params_t;

typedef struct nic_send_params {
    unsigned char* buf;
    uint64_t size;
    uint64_t address;
} nic_send_params_t;

typedef struct opcode_range_params {
    int cpu_index;
    qemu_cpu_opaque_t cpu;
    pyrebox_target_ulong cur_pc;
    pyrebox_target_ulong next_pc;
    uint16_t opcode;
} opcode_range_params_t;

typedef struct tlb_exec_params {
    qemu_cpu_opaque_t cpu;
    pyrebox_target_ulong vaddr;
} tlb_exec_params_t;

typedef struct vmi_create_proc_params {
    pyrebox_target_ulong pid;
    pyrebox_target_ulong pgd;
    char* name;
} vmi_create_proc_params_t;

typedef struct vmi_remove_proc_params {
    pyrebox_target_ulong pid;
    pyrebox_target_ulong pgd;
    char* name;
} vmi_remove_proc_params_t;

typedef struct vmi_context_change_params {
    pyrebox_target_ulong old_pgd;
    pyrebox_target_ulong new_pgd;
} vmi_context_change_params_t;

//Params for the qemu->pyrebox callback (native)
typedef struct callback_params {
   union {
```

(continues on next page)

(continued from previous page)

```
        block_begin_params_t block_begin_params;
        block_end_params_t block_end_params;
        insn_begin_params_t insn_begin_params;
        insn_end_params_t insn_end_params;
        mem_read_params_t mem_read_params;
        mem_write_params_t mem_write_params;
        keystroke_params_t keystroke_params;
        nic_rec_params_t nic_rec_params;
        nic_send_params_t nic_send_params;
        opcode_range_params_t opcode_range_params;
        tlb_exec_params_t tlb_exec_params;
        vmi_create_proc_params_t vmi_create_proc_params;
        vmi_remove_proc_params_t vmi_remove_proc_params;
        vmi_context_change_params_t vmi_context_change_params;
   };
} callback_params_t;
```

In order to test if a trigger compiles correctly, cd to the PyREBox directory and run the following command. Adjust the target architecture and name of the plugin depending on your needs.

```
make triggers/trigger_template-i386-softmmu.so
```

CHAPTER 8

API Documentation

class api.**BP**(*addr*, *pgd*, *size=0*, *typ=0*, *func=None*, *new_style=False*)

Class used to create execution, memory read, and memory write breakpoints

__init__(*addr*, *pgd*, *size=0*, *typ=0*, *func=None*, *new_style=False*)

Constructor for a BreakPoint

Parameters

- **addr** (*int*) – The (start) address where we want to put the breakpoint. If a str is provided, it will search for a symbol and put the breakpoint there. The syntax is module!symbol, and it does not require to specify the full module or symbol name as long as there is no ambiguity.
- **pgd** (*int*) – The PGD or address space where we want to put the breakpoint. Irrelevant for physical address breakpoints.
- **size** (*int*) – Optional. The size of the area we want to put a breakpoint on. We can put the BP on a single address or a memory range.
- **typ** (*int*) – The type of breakpoint: BP.EXECUTION, BP.MEM_READ, BP.MEM_WRITE, BP.MEM_READ_PHYS, BP.MEM_WRITE_PHYS
- **func** (*function*) – Optional. The function that will be called as callback for the breakpoint. The parameters for the function should be the ones corresponding to the INSN_BEGIN_CB callback for execution breakpoints, and MEM_READ_CB or MEM_WRITE_CB for memory read/write breakpoints. If no function is specified, a shell is started when the breakpoint is hit.
- **new_style** (*bool*) – Defines whether the function *func* optionally passed as parameter uses old or new callback calling convention. See documentation for further reference. Defaults to False.

Returns An instance of class BP for the inserted breakpoint

Return type *BP*

disable()

Disable a breakpoint

Returns None

Return type None

enable()
Enable a breakpoint

Returns None

Return type None

enabled()
Return whether the breakpoint is enabled or not

Returns Whether the breakpoint is enabled or not

Return type bool

get_addr()
Get the address where the breakpoint is registered

Returns The address

Return type int

get_pgd()
Get the PGD of the process where the breakpoint is registered

Returns The PGD of the process where the breakpoint is registered

Return type int

get_size()
Get the size of the breakpoint

Returns The size of the breakpoint

Return type int

get_type()
Get the type of the breakpoint

Returns The type of the breakpoint: BP.EXECUTION, BP.MEM_READ, BP.MEM_WRITE

Return type int

class api.**CallbackManager**(*module_hdl*, *new_style=False*)
Class that abstracts callback management,optionally associating names to callbacks, and registering the list of added callbacks so that we can remove them all with a single call to "clean()" after we are done.

__init__(*module_hdl*, *new_style=False*)
Constructor of the class

Parameters

- **module_hdl** (*int*) – The module handle provided to the script as parameter to the initialize_callbacks function. Use 0 if it doesn't apply.
- **new_style** (*bool*) – Enables the new-style callback parameter format. New-style callback functions accept a single parameter (dictionary), with a key (str) per parameter, and a value (value of the parameter), instead of positional arguments.

add_callback(*callback_type*, *func*, *name=None*, *addr=None*, *pgd=None*, *start_opcode=None*, *end_opcode=None*, *new_style=None*)
Add a callback to the module, given a name, so that we can refer to it later.

If the name is repeated, it will provide back a new name based on the one passed as argument, that can be used later for removing it or attaching triggers to it.

Parameters

- **name** (*str*) – The name of the callback
- **callback_type** (*int*) – The callback type to insert. One of INSN_BEGIN_CB, BLOCK_BEGIN_CB, etc... See help(api) from a pyrebox shell to get a complete listing of constants ending in _CB
- **func** (*function*) – The callback function (python function)
- **addr** (*int*) – Optional. The address where we want to place the callback. Only applies eo INSN_BEGIN_CB, BLOCK_BEGIN_CB
- **pgd** (*int*) – Optional. The PGD (addr space) where we want to place the callback. Only applies to INSN_BEGIN_CB, BLOCK_BEGIN_CB
- **new_style** (*bool*) – Optional. Enables the new-style callback parameter format. New-style callback functions accept a single parameter (dictionary), with a key (str) per parameter, and a value (value of the parameter), instead of positional arguments. This parameter overrides the class-wide new_style parameter in the CallbackManager __init__ function.

Returns The actual inserted callback name. If the callback name indicated already existed, this name will be updated to make it unique. This name can be used as a handle to the callback

Return type str

add_trigger(*name*, *trigger_path*)

Add trigger to a callback.

Adds a trigger to a given callback. If the trigger is not compiled or the binary is outdated, it will force a compilation of the trigger before loading it.

Parameters

- **name** (*str*) – The callback name to which we want to add the trigger
- **trigger_path** (*str*) – The path to the trigger.

Returns None

Return type None

call_trigger_function(*name*, *function_name*)

Call a trigger function associated to callback (name) with function name function_name

param name The callback name

type name str

param function_name The function name

type function_name str

return The value, if it exists, None otherwise

rtype str or int

callback_exists(*name*)

Determine if a callback exists or not, given its name

Parameters **name** (*str*) – The callback name to check

Returns True if the callback already exists

Return type bool

clean()
Clean all the inserted callbacks.

Clean all the inserted callbacks. Will remove all the callbacks registered within this manager.

Returns None

Return type None

generate_callback_name(*name*)
Generates a unique callback name given an initial name

Parameters **name** (*str*) – The initial name

Returns The new generated name

Return type str

get_module_handle()
Returns the module handle associated to this callback manager

Returns The handle of the module this callback manager is bound to.

Return type int

get_trigger_var(*name*, *var_name*)
Get a trigger variable associated to callback (name) with variable name var_name

param name The callback name

type name str

param var_name The variable name

type var_name str

return The value, if it exists, None otherwise

rtype str or int

rm_callback(*name*)
Remove a callback given its name. Associated triggers will get unloaded too.

Parameters **name** (*str*) – The name of the callback to remove

Returns None

Return type None

rm_trigger(*name*)
Remove the trigger from the callback specified as parameter

Parameters **name** (*str*) – The callback name from which we want to remove the trigger

Returns None

Return type None

set_trigger_var(*name*, *var_name*, *val*)
Add a trigger variable with name var_name and value val, to the callback with the given name

param name Name of the callback

type name str

param var_name Name of the variable to set

type var_name str

param val Value of the variable to set

type val unsigned int or str

return None

rtype None

class api.**GuestFile**(*filesystem_index*, *file_handle*, *size*, *name*)

Class used to manage guest files residing on the guest file system

__init__(*filesystem_index*, *file_handle*, *size*, *name*)

get_name()

Returns the name of the file

Returns The name of the file

Return type str or unicode

get_offset()

Returns the current offset

Returns The offset of the file

Return type int

get_size()

Returns the file size

Returns The file size

Return type int

read(*size=None*, *offset=None*)

Reads data at the current offset, or the specified offset.

Parameters

- **size** (*int*) – The size to read
- **offset** (*int*) – Optional. The offset to read at. It will not change the current file pointer.

Returns The data read

Return type str

seek(*offset*)

Sets the offset to read the file

Parameters **offset** (*int*) – The offset to set

Returns None

Return type None

api.**get_filesystems**()

Returns a list of filesystems to open.

Returns A list of dictionaries, each dictionary containing the keys: "index", "type" and "size", and their respective values.

Return type list

api.**get_loaded_modules**()

Returns a dictionary of modules loaded in pyrebox.

Returns Dictionary with the keys: “module_handle”, “module_name”, “is_loaded”

Return type dict

api.**get_module_list**(*pgd*)
Return list of modules for a given PGD

Parameters **pgd** (*int*) – The PGD of the process for which we want to extract the modules, or 0 to extract kernel modules

Returns List of modules, each element is a dictionary with keys: “name”, “fullname”, base”, “size”, and “symbols_resolved”

Return type list

api.**get_num_cpus**()
Returns the number of CPUs on the emulated system

Returns The number of CPUs on the emulated system

Return type int

api.**get_os_bits**()
Return the bitness of the system / O.S. being emulated

Returns The bitness of the system / O.S. being emualated

Return type int

api.**get_process_list**()
Return list of processes.

Returns List of processes. List of dictionaries with keys: “pid”, “pgd”, “name”, “kaddr”, where kaddr stands for the kernel address representing the process (e.g.: EPROCESS)

Return type list

api.**get_running_process**(*cpu_index=0*)
Returns the PGD or address space of the process that is being executed at this moment

Parameters **cpu_index** (*int*) – CPU index that we want to query. Each CPU might be executing a different address space

Returns The PGD or address space for the process that is executing on the indicated CPU

Return type int

api.**get_symbol_list**(*pgd=None*)
Return list of symbols

Parameters **pgd** (*int*) – The pgd to obtain the symbols from. 0 to get kernel symbols

Returns List of symbols, each element is a dictionary with keys: “mod”, “mod_fullname”, “name”, and “addr”

Return type list

api.**get_system_time**()
Retrieve the system time for the running guest.

Returns The system time for the running system.

Return type datetime.datetime

api.**import_module**(*module_name*)
Import a module given its name (e.g. scripts.script_example)

Parameters **module_name** (*str*) – The module name following python notation. E.g.: scripts.script_example

Returns None

Return type None

api.**is_kernel_running**(*cpu_index=0*)
Returns True if the corresponding CPU is executing in Ring 0

Parameters **cpu_index** (*int*) – CPU index that we want to query. Each CPU might be executing a different address space

Returns True if the corresponding CPU is executing in Ring 0, False otherwise

Return type bool

api.**is_monitored_process**(*pgd*)
Returns true of a given process is being monitored. Process-wide callbacks will be called for every process that is being monitored

param pgd PGD, or address space of the process to monitor

type pgd int

return True of the process is being monitored, False otherwise

rtype bool

api.**load_vm**(*name*)
Load a previously saved snapshot of the virtual machine.

Parameters **name** (*str*) – Name of the snapshot to load

Returns None

Return type None

api.**open_guest_path**(*filesystem_index*, *path*)
Open a file or directory in a given file system.

Parameters

- **filesystem_index** (*int*) – The index of the filesystem to open
- **path** (*str*) – The path to open (either a file or directory).

Returns A list of files (if the path is a directory), an instance of GuestFile (if the path is a file), or None

Return type list, *GuestFile*, or None

api.**r_cpu**(*cpu_index=0*)
Read CPU register values :param cpu_index: The CPU index to read. 0 by default. :type cpu_index: int

Returns The CPU

Return type X64CPU | X86CPU | ...

api.**r_ioport**(*address*, *size*)
Read I/O port

Parameters

- **address** (*int*) – The port address to read, from 0 to 65536
- **size** (*int*) – The size to read (1, 2, or 4)

Returns The value read

Return type int

api.**r_pa**(*addr*, *length*)
Read physical address

Parameters

- **addr** (*int*) – The address to read
- **length** (*int*) – The length to read

Returns The read content

Return type str

api.**r_va**(*pgd*, *addr*, *length*, *use_filesystem=False*)
Read virtual address

Parameters

- **pgd** (*int*) – The PGD (address space) to read from
- **addr** (*int*) – The address to read
- **length** (*int*) – The length to read
- **use_filesystem** (*bool*) – Optional. Default: False. If set to True, PyREBox will use The Sleuthkit to inspect the file system and obtain this data from the file backing the memory page: The referenced file if it is memory mapped, or the pagefile.sys in case it has been paged out.

Returns The read content

Return type str

api.**reload_module**(*module_handle*)
Reload a module given its handle.

Parameters **module_handle** (*int*) – The module handle.

Returns None

Return type None

api.**save_vm**(*name*)
Save the state of the virtual machine so that it can be restored later

Parameters **name** (*str*) – Name of the snapshot to save

Returns None

Return type None

api.**start_monitoring_process**(*pgd*)
Start monitoring a process. Process-wide callbacks will be called for every process that is being monitored

Parameters **pgd** (*int*) – PGD, or address space of the process to check

Returns None

Return type None

api.**stop_monitoring_process**(*pgd*, *force=False*)
Start monitoring a process. Process-wide callbacks will be called for every process that is being monitored

Parameters **pgd** (*int*) – PGD, or address space of the process to stop monitoring

Returns None

Return type None

api.**sym_to_va**(*pgd*, *mod_name*, *func_name*)
Resolve an address given a symbol name

Parameters

- **pgd** (*int*) – The PGD or address space for the process for which we want to search the symbol
- **mod_name** (*str*) – The module name that contains the symbol
- **func_name** (*str*) – The function name to resolve

Returns The address, or None if the symbol is not found

Return type str

api.**unload_module**(*module_handle*)
Unload a module given its handle.

Parameters **module_handle** – The module handle.

Returns None

Return type None

api.**va_to_pa**(*pgd*, *addr*)
Virtual to physical address.

Parameters

- **pgd** – PGD, or address space of the address to translate
- **addr** (*int*) – Virtual address to translate

Returns The translated physical address

Return type int

api.**va_to_sym**(*pgd*, *addr*)
Find symbols for a particular virtual address

Parameters

- **pgd** (*int*) – The PGD or address space for the process for which we want to search the symbol
- **addr** (*int*) – The virtual address to search

Returns A tuple containing the module name and the function name, None if nothing found

Return type tuple

api.**w_ioport**(*address*, *size*, *value*)
Write I/O port

Parameters

- **address** (*int*) – The port address to write, from 0 to 65536
- **size** (*int*) – The size to read (1, 2, or 4)

Returns The value written

Return type int

api.**w_pa**(*addr*, *buff*, *length=None*)
Write physical address

Parameters

- **addr** (*int*) – The address to write
- **buff** – The buffer to write

Returns None

Return type None

api.**w_r**(*cpu_index*, *regname*, *val*)
Write register

Parameters

- **cpu_index** (*int*) – CPU index of the register to write
- **regname** (*str*) – Name of the register to write
- **val** (*int*) – Value to write

Returns None

Return type None

api.**w_sr**(*cpu_index*, *regname*, *selector*, *base*, *limit*, *flags*)
Write segment register. Only applies to x86 / x86-64

Parameters

- **cpu_index** (*int*) – CPU index of the register to write
- **regname** (*str*) – Name of the register to write
- **selector** (*int*) – Value (selector) to write
- **base** – Value (base) to write
- **limit** – Value (limit) to write

Returns None

Return type None

api.**w_va**(*pgd*, *addr*, *buff*, *length=None*)
Write virtual address

Parameters

- **pgd** (*int*) – The PGD (address space) to write to.
- **addr** (*int*) – The address to write
- **buff** – The buffer to write

Returns None

Return type None

class plugins.guest_agent.**GuestAgentPlugin**(*cb*, *printer*)
This plugin deals with a host-guest interface through unused opcodes. It facilitates getting samples into the guest VM, starting them, ...

How to use this plugin:

1. For scripts: - Add *plugins.guest_agent: True* to your pyrebox.conf

or

- Add a member to your module named "requirements" containing a list of required plugins/scripts. E.g.:

  requirements = ["plugins.guest_agent"]

- Import the plugin with *from plugins.guest_agent import guest_agent* in your script.
- Interact with the guest agent using the public interface of this class (agent is a singleton instance of GuestAgentPlugin).

2. In the ipython shell: - If no loaded script is loading the guest_agent plugin, you will need to make sure it

   gets loaded by adding *plugins.guest_agent: True* to your pyrebox.conf

   - Interact with the guest agent using the global member agent that is a singleton instance of GuestAgentPlugin.

__init__(*cb*, *printer*)
Create a new instance of the GuestAgentPlugin.

Parameters

- **cb** – The callback manager.
- **printer** – The printer where logs should go to.

copy_file(*source_path*, *destiny_path*, *callback=None*)
Copy file from host machine to guest VM

Parameters

- **source_path** (*str*) – The path (on the host) of the file to copy
- **destiny_path** – The path (on the guest) of the file to copy
- **callback** (*func*) – A python function that will be called when the command is requested by the guest (just before it is executed).

execute_file(*path*, *args=[]*, *env={}*, *exit_afterwards=False*, *callback=None*)
Execute file on the guest VM and terminate the agent

Parameters

- **path** (*str*) – The path of the file to execute.
- **args** (*list*) – The list of arguments to execute the file. (list of str)
- **env** (*dict*) – A dictionary with environment variables to set for the file to be executed.
- **callback** (*func*) – A python function that will be called when the command is requested by the guest (just before it is executed).

exit_agent()
Forces the agent to exit. Nevertheless, if a new agent is spawned in the guest, it will be up and running again.

print_command_list()
Prints the list of commands in the queue.

remove_command(*cmd_number*)
Removes a command from the queue of commands to execute

Parameters **cmd_number** (*int*) – The command number to remove (obtained from print_command_list())

stop_agent()
Forces the agent to exit, and stops listening to agent creation, so guest agent interaction is disabled forever.

plugins.guest_agent.**clean**()
Clean up everything.

CHAPTER 9

Guest Agent

9.1 Windows

You can find the windows guest agent under the guest/win directory.

9.1.1 Compiling guest agent

You may need to install mingw-w64 packages. For example, on Ubuntu, or Debian:

```
apt-get install gcc-mingw-w64-i686 g++-mingw-w64-i686 mingw-w64-i686-dev mingw-w64-
↪tools gcc-mingw-w64-x86-64 mingw-w64-x86-64-dev g++-mingw-w64-x86-64
```

Just compile with `make`. It will produce 2 files: win_agent_32.exe and win_agent_64.exe, for 32 and 64 bit windows guests respectively.

9.1.2 Compiling test files

In order to compile the test files, just use the provided Makefile as follows:

```
make test_32
make test_64
```

9.2 Linux

You can find the linux guest agent under the guest/linux path.

9.2.1 Compiling guest agent

You may need to install the following packages. For example, on Ubuntu, or Debian:

```
apt-get install libc6-dev-i386
```

Just compile with `make`. It will produce 2 files: linux_agent_32 and linux_agent_64.exe, for 32 and 64 bit linux guests respectively.

9.2.2 Compiling test files

In order to compile the test files, just use the provided Makefile as follows:

```
make 32bit_test
make 64bit_test
```

9.3 General usage

9.3.1 Configuring guest agent

- Add *plugins.guest_agent: True* to your pyrebox.conf
- (Optionally) modify your guest agent file name.
- Add the agent configuration to your pyrebox.conf
- Adjust the configuration appropriately (if you changed the agent file name).
- Make sure the agent conf file exists and is up to date. This file is automatically generated by the compilation process.
- Copy the corresponding guest agent (32 or 64 bit version) to the guest VM, and make sure it follows the same name as declared in the configuration name.
- Start the agent (you can configure the VM to start the agent on every system start-up).
- Once the agent is started, you can take a snapshot.

Example agent configuration in pyrebox.conf:

```
[AGENT]
name: win_agent_64.exe
conf: win_agent_64.exe.conf
```

9.3.2 Using the guest agent

In scripts:

- Add `plugins.guest_agent: True` to your pyrebox.conf, or:
- Add a member to your module named "requirements" containing a list of required plugins/scripts. E.g.: `requirements = ["plugins.guest_agent"]`
- Import the plugin in your script: `from plugins.guest_agent import guest_agent`.

- Interact with the guest agent using the public interface of this class (`guest_agent` is a singleton instance of GuestAgentPlugin). See API.

In the IPython shell:

- If no script loads the guest_agent plugin, you will need to make sure it gets loaded by adding `plugins.guest_agent: True` to your pyrebox.conf.
- Interact with the guest agent using the global member `agent` that is a singleton instance of GuestAgentPlugin. See API.

9.3.3 Examples

The example script provides a custom command that documents how to use the agent in a PyREBox script. The `requirements: ["plugins.guest_agent"]` member allows you specificy that the scripts needs the guest agent plugin to be loaded in order to work.

CHAPTER 10

Malware Monitor

Malware monitor is a set of PyREBox scripts for automatically extracting useful information during malware analysis. Moreover, it tries to help the analyst in the first phase, by providing insights about how a given malware sample deploys its main payload (i.e., unpacking, process injection, process hollowing, file dropping, file downloading...). Also, it collects various types of information that can be imported into IDA to enrich the IDB database. Malware monitor consists of several modules that can be activated/deactivated and configured by editing a json file. Each module produces several logs in different formats.

The **api tracer** module allows to trace Windows API function calls, and to automatically extract the input and output parameters. An IDA Python script allows to import and visualize this information in IDA.

The **dumper** module allows to dump the memory of a process during its execution. This module is configurable by the user, who can choose the best moment to trigger the memory dump.

The **coverage** module collects an execution trace that can be used to colorize basic blocks in IDA. This features provides the user information about which code paths get executed, and which do not.

Finally, the **memory monitor** module (refered to as *interproc* in the scripts), monitors different memory-related operations and events, and also allows to monitor process interaction events, like new processes created, memory injection to existing processes, and so on. This last module is orthogonal to the other three. Since it monitors process creation and opening, it allows to monitor not only the initial process, but all those related to it. For example, if *api tracer* is turned on, and the *memory monitor* detects that the first process creates a second process, *api tracer* will start monitoring this new process and will generate an API call trace for it as well.

10.1 Configuration files

Malware monitor has two different configuration files:

10.1.1 mw_monitor.conf

Each of the four modules generates several log files. The names of the logs can be configured in this configuraiton file, that must be accessible from the directory where PyREBox is started. A common option is to place it in the same folder as the pyrebox.conf file.

This file allows to configure the path and file names for the logs generated by the different modules. It also allows to determine the file name of the results *bundle*, which is .tar.gz file containing all the collected results. You can find a self-explanatory configuration file under the config_examples directory.

interproc

- **bin_log_name.** This file is a binary log (serialized data) of the data collected during memory operation monitoring.
- **text_log_name.** This file is a text log of all the events related to memory monitoring captured during the execution.
- **basic_stats_name.** This file is a structured text summary of the data collected.

dumper

- **path.** This option allows to choose the path where we want to dump the memory of the process, loaded dlls, and the rest of the VAD regions that do not overlap the main process memory or any DLL.

coverage

- **cov_log_name.** This file is a binary log of the instruction trace collected. This log can be imported into IDA with a corresponding script.
- **cov_text_name.** This file is a text log that summarizes the instruction trace collected. Each line in this log represents a transition from one VAD region to a different VAD region, and includes both the origin and destiny address.

api_tracer

- **text_log_name.** This file is a text log containing the recoded API calls, with or without their parameters (depends on configuration).
- **bin_log_name.** This file is a binary log containing the same information as the text log. This file can be imported into IDA with a corresponding script.

10.1.2 mw_monitor_run.json

This json file allows to turn on/off each of the modules separately, under the *modules* section. It also allows to configure different parameters for each module. Malware monitor also provides sample execution automation, which can be configured in this json file.

general

- **files_bundle.** The path, in the host system, of a zip file containing several files (typically a .exe and .dll dependencies). The files inside this zip container will be copied into the guest system, under the *files_path* path.
- **files_path.** The directory in the guest system where the files will be copied at startup.
- **main_executable.** The name of the file to execute, among the ones in the zip file **files_bundle.**
- **api_database.** The path the API database generated with Deviare and the MSDN crawler.

interproc

- **basic_stats.** Boolean value that turns on/off the *basic_stats* report generation, which contains a summary of the observed memory operations.
- **bin_log** Boolean value that turns on/off the binary log generation of this module.
- **text_log** Boolean value that turns on/off the text log generation of this module. This log contains a trace of all the memory operations monitored.

dumper

- **dump_on_exit.** Boolean that determines if the process memory should be dumped when it exits.
- **dump_at.** Value that allows to configure when to dump the process memory. It accepts 3 possible formats: an address, a symbol, and a symbol followed by an address. In the first case, the process memory will be dumped when the control flow reaches a given address under the context of the process. In the second case, the process memory will be dumped when the control flow reaches the symbol specified (generally, an specific API call). The third option will dump the process memory when the process calls an API function, specifically from a given address.

coverage

- **procs.** A list of strings that specifies the process names of the processes which should be traced in order to generate a coverage file. If a none value or an empty list are specified, all the monitored processes (the initial one, and any related process) will be recorded.

api_tracer

- **bin_log.** Boolean value that allows to turn on/off the generation of the binary log.
- **text_log.** Boolean value that allows to turn on/off the generation of the text log.
- **light_mode.** Boolean value that allows to turn on/off the light mode. Under light mode, function call arguments are not dereferenced, resulting in an slightly faster execution of the guest system.
- **exclude_apis.** A list of API functions to exclude from being logged.
- **exclude_modules.** List of module names to exclude from being traced. Any call to a function in a module in this list will not be logged.
- **exclude_origin_modules.** List of module names to exclude from being traced. Any call originating from a module in this list, will not be logged.
- **include_apis.** A list of API functions to include in the trace, even if the module where it is located is in some exclusion list. This finer-granularity option overrides any exclusion rule.
- **procs.** A list of strings that specifies the process names of the processes which should be traced in order to generate a coverage file. If a none value or an empty list are specified, all the monitored processes (the initial one, and any related process) will be recorded.

10.2 IDA Python scripts

We provide IDA Python scripts under the ida_scripts directory. There are 2 main scripts:

- **mw_monitor_coverage.py**. Allows to read the coverage binary log and to colorize the basic blocks that have been executed.
- **mw_monitor_ida_functions_rename.py**. Opens a new tab in IDA that allows to load the api tracer binary log and to visualize the API calls traced, as well as their origin and destiny addresses and parameters.

In order to run these scripts, you will need to copy the entire mw_monitor directory to a path that must be accessible from your IDA setup. These IDA scripts have several dependencies under the mw_monitor/ directory of this project.

10.3 API tracer database

The API tracer relies on an sqlite database in order to inspect automatically API parameters. This database can be generated with a combination of the Deviare project, the MSDN crawler published by Zynamics, and a custom script that allows to integrate both data sources into the sqlite database that malware monitor uses.

In order to generate the database, you will first need to clone the Deviare project (https://github.com/nektra/deviare2) and slighly modify the DbGenerator subproject to produce an sqlite database. See the readme file for information about which files must be patched. Then, run the DbGenerator project for the corresponding version (32 or 64 bit) windows machine, to generate the initial sqlite database.

This database still lacks information about which parameters are input parameters, and which are output parameters. This information can be obtained from the MSDN. In order to parse the MSDN, use the provided script. This script is based on the msdn_crawler script published by Zynamics. This modified script will produce an xml file with information for each API documented in the MSDN.

Finally, the last step involves running the populate_db.py script, in order to populate the sqlite database with the information extracted with the MSDN crawler.

10.4 Documentation

This documentation is also hosted toguether with the main PyREBox documentation at readthedocs.io.

10.5 Bugs, questions and support

If you think you've found a bug, please report it here.

Before creating a new issue, please go through the questions opened by other users before.

This program is provided "AS IS", and no support is guaranteed. That said, in order to help us solve your issues, please include as much information as possible in order to reproduce the bug:

- Operating system used to compile and run PyREBox.
- The specific operating system version and emulation target you are using.
- Shell command / script / task you were trying to run.
- Any information about the error such as error messages, Python (or IPython) stack trace, or QEMU stack trace.
- Any other relevant information

PyREBox is a Python scriptable Reverse Engineering sandbox. It is based on QEMU, and its goal is to aid reverse engineering by providing dynamic analysis and debugging capabilities from a different perspective. PyREBox allows to inspect a running QEMU VM, modify its memory or registers, and to instrument its execution, by creating

simple scripts in python to automate any kind of analysis. QEMU (when working as a whole-system-emulator) emulates a complete system (CPU, memory, devices...). By using VMI techniques, it does not require to perform any modification into the guest operating system, as it transparently retrieves information from its memory at run-time.

Several academic projects such as DECAF, PANDA, S2E, or AVATAR, have previously leveraged QEMU based instrumentation to overcome reverse engineering tasks. These projects allow to write plugins in C/C++, and implement several advanced features such as dynamic taint analysis, symbolic execution, or even record and replay of execution traces. With PyREBox, we aim to apply this technology focusing on keeping the design simple, and on the usability of the system for threat analysts.

CHAPTER 11

Goals

- Provide a whole system emulation platform with a simple interface for inspecting the emulated guest system.
  - Fine grained instrumentation of system events.
  - Integrated Virtual Machine Introspection (VMI), based on volatility. No agent or driver needs to be installed into the guest.
  - An IPython based shell interface.
  - A Python based scripting engine, that allows to integrate into the scripts any of the security tools based on this language (one of the biggest ecosystems).
- Have a clean design, de-coupled from QEMU. Many projects that are built over QEMU do not evolve when QEMU gets upgraded, missing new features and optimizations, as well as security updates. In order to achieve this, PyREBox is implemented as an independent module that can be compiled together with QEMU requiring a minimal set of modifications.
- Support for different architectures. Currently, PyREBox only supports Windows for x86 and x86-64 bit architectures, but its design allows to support other architectures such as ARM, MIPS, or PowerPC, and other operating systems as well.

CHAPTER 12

IPython shell

Starting a PyREBox shell is as easy as typing the `sh` command on QEMU's monitor. It will immediately start an IPython shell. This shell records the command history as well as the defined variables. For instance, you can save a value and recover it later at a different point of the execution, when you start the shell again. PyREBox takes advantage of all the available features in IPython such as auto-completion, command history, multi-line editing, and automated command help generation.

PyREBox will allow you to debug the system (or a process) in a fairly stealthy way. Unlike traditional debuggers which stay in the system being debugged (even modifying the memory of the debugged process to insert breakpoints), PyREBox stays completely outside the inspected system, and it does not require the installation of any driver or component into the guest.

PyREBox offers a complete set of commands to inspect and modify the state of the running VM. Just type `list_commands` to obtain a complete list. You can run any volatility plugin just by typing `vol` and the corresponding volatility command. For a complete list of available volatility plugins, you can type `list_vol_commands`. This list is generated automatically, so it will also show any volatility plugin you install on PyREBox's `volatility/` path.

You can also define your own commands! It is as simple as declaring a function in a script, and loading it.

If you need something more expressive than a command, you can write a Python snippet leveraging the API. For a detailed description of the API, see *corresponding documentation* or type `help(api)` in the shell.

CHAPTER 13

Scripting

PyREBox allows to dynamically load scripts that can register callback functions that are called when certain events occur, like instructions executed, memory read/written, processes created/destroyed, and so on.

Given that PyREBox is integrated with Volatility, it will let you take advantage of all the volatility plugins for memory forensics in your python scripts. Many of the most famous reverse engineering tools are implemented in Python or at least have Python bindings. Our approach allows to integrate any of these tools into a script.

Finally, given that python callbacks can introduce a performance penalty on frequent events such as instructions executed, it is also possible to create *triggers*. *Triggers* are native-code plug-in's (developed in C/C++) that can be inserted dynamically at run-time on any event just before the Python callback is executed. This allows to limit the number of events that hit the python code, as well as to precompute values in native code.

In this repository you will find example scripts that can help you to write your own code. Contributions are welcome!

CHAPTER 14

Install

A build script is provided. For specific details about dependencies, please see the *quickstart guide*. We also provide a Dockerfile.

CHAPTER 15

Acknowledgement

First of all, PyREBox would not be possible without QEMU and Volatility. We thank to their developers and mantainers for such a great work.

PyREBox is inspired by several academic projects, such as DECAF, or PANDA. In fact, many of the callbacks supported by PyREBox are equivalent to those found in DECAF, and the concepts behind the instrumentation are based on these works.

PyREBox benefits from third-party code, which can be found under the directory pyrebox/third_party. For each third-party project, we include an indication of its original license, the original source code files taken from the project, as well as the modified versions of the source code files (if applicable), used by PyREBox.

CHAPTER 16

Bugs and support

If you think you've found a bug, please report it here.

This program is provided "AS IS", and no support is guaranteed. That said, in order to help us solve your issues, please include as much information as possible in order to reproduce the bug:

- Operating system used to compile and run PyREBox.
- The specific operating system version and emulation target you are using.
- Shell command / script / task you were trying to run.
- Any information about the error such as error messages, Python (or IPython) stack trace, or QEMU stack trace.
- Any other relevant information

Python Module Index

a

p

Index

Symbols

A

B

C

D

E

G

I

L

O

P

R

S

U

V

W