PyREBox

Making Dynamic Instrumentation
Great Again

Talos
Deep Packer Inspector
https://packerinspector.github.io
https://packerinspector.com
WHY IS THIS GUY TALKING ABOUT
ABOUT PACKER INSPECTOR??
Many instrumentation frameworks...

PIN
DynamoRIO
WinAppDbg
PyKD
Unicorn
PyDbg
S2E
PANDA
Avatar
DynInst
TEMU/DECAF
Frida
PyREBox

- Motivation
- Principles
- Design / architecture
- Features
- Malware monitor
- Future work
Technical aspects
- Single process/binary, or whole system?
- What events does it hook / instrument?
- Transparency?

Practical aspects
- How ‘easy’ is it to use?
- Programming languages?

Other aspects
- How often is it ‘updated’?
- Community?
- Is the project even alive?
Frameworks based on emulation

- Full system emulator (vs. user-mode)
- QEMU!
- Emulate CPU, BIOS, memory, devices
  - Boot and fully emulate unmodified O.S.
  - (Linux, Solaris, Windows, DOS, BSD…)
- Different guest architectures on different host architectures (TCG)
QEMU

- “Transparent” instrumentation
  - Emulated memory is not modified
- No agent needed
- Full system == ...
  - Allows to monitor inter-process interaction
  - Allows to instrument / inspect kernel
Some shortcomings...

- PANDA, DECAF, etc…
  - Plugins are coded in C/C++
  - I prefer **python**!
    - Faster development
    - Great libraries
- Complex QEMU modifications
  - Risk of not updating frequently as QEMU evolves
So, what is PyREBox?

Yet another dynamic instrumentation engine

- Interactive analysis
  - Allows inspecting memory/registers
  - Useful built-in commands
  - IPython
So, what is PyREBox?

- **Scripting** (python)
- Callback types...
  - Instruction/block begin/end
  - Memory read/write
  - Specific opcode execution
  - Process create/remove
  - Module load/unload
  - TLB flush / context change
- Extend shell with **new commands**
QEMU

- Full system emulator (QEMU)
  What about **hardware assisted virtualization**?

- E.g.: KVM
  - Target & host arch. must be the same
  - Host O.S. dependent
    - (e.g.: KVM won’t run on Windows)
WON VOLATILY PLUGIN CONTEST

NOT A VOLATILY PLUGIN

meme generator.net
So, what is PyREBox?

- Leverages Volatility for memory introspection
- It is free!! (as in freedom)

General Public License
Design
Some principles...

- Interaction and scripts based in **python**
  - Tradeoff: high overhead

- KISS: Keep *Instrumentation Simple* Stupid
  - **Minimal modifications** to QEMU
  - Core of the framework **de-coupled** from QEMU
  - **Easier to upgrade** to new QEMU versions
  - Tradeoff: advanced features
    - Taint analysis, record replay…
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility (VMI)

Python Core (PyREBox)

Python run-time

Interactive shell
Python-based API
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility
(VMI)

Python run-time

Python Core
(PyREBox)

API

Initialization

Script1.py
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility (VMI)

Python run-time

Python Core (PyREBox)

API

Script1.py

Register callbacks
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility (VMI)

Python Core (PyREBox)

Python run-time

API

Script1.py
Glue

QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

E.g.: Inspect or manipulate memory

Volatility (VMI)

Python run-time

Python Core (PyREBox)

API

Script1.py

Python run-time

PyREBox (C/C++)

Volatility (VMI)

QEMU
(600 LoC of modifications)
We see the system as a *raw* CPU!!

- Only memory, registers, devices
- Sequence of instructions
- Processes, threads, handles, libraries...
  - Abstractions of the O.S.

**Virtual Machine Introspection**
- Understand these abstractions
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility (VMI)

Python run-time

Python Core (PyREBox)

API

+ Basic VMI

Script1.py
VMI

- Support for **Windows and Linux, 32 and 64 bit**
  - Process enumeration
  - Module (DLL / shared library) enumeration
  - Symbol resolution (exported symbols)
- Deliver certain callbacks
Triggers

- Python can be **prohibitively expensive**
  - Instruction begin, memory read...

- Triggers
  - C/C++ snippets
  - Compiled as shared libraries (.so)
  - Loaded at runtime
  - Returns 0 if callback should not be delivered, 1 otherwise.

```c
int trigger(callback_handle_t handle, callback_params_t params){
    return should_deliver;
}
```
QEMU
(600 LoC of modifications)

Glue

PyREBox (C/C++)

Volatility (VMI)  Python Core (PyREBox)

Python run-time

Callback

API

Trigger (plugin) gets callback notification. Decides whether it must be delivered or not

Script1.py

? .so
PyREBox usage
Easy to compile, install

- Compiles and runs (tested):
  - Linux
  - Windows (thanks to linux subsystem)
  - Docker is supported
Easy to compile, install

- Easy to compile, install
- Starting PyREBox is like starting any QEMU session.
- QEMU options via command line arguments (Check QEMU docs)
- Example scripts provided
- PyREBox configuration file
- Complete PyREBox documentation
  https://pyrebox.readthedocs.io/en/latest/
PyREBox shell
PyREBox shell

**QEMU monitor**
- Regular QEMU commands
- Snapshot management
- PyREBox script management

**PyREBox shell**
- **Pauses the guest**
- Inspect regs/mem
- Modify regs/mem
- Run built-in commands
- Run volatility commands
- Run custom commands
- Run python code (ipython)
- Autocompletion, syntax

$sh
Scripting
Scripting

- Loaded/unloaded/reloaded
  - Startup script
  - QEMU command

- Can start a shell at any time
  >>> start_shell()

- Can import and use any python library
def initialize_callbacks(module_hdl, printer):
    """
    Initialize callbacks for this module. This function will be triggered whenever import_module command is triggered.
    """
    global cm
    global pyrebox_print
    from api import CallbackManager
    # Initialize printer
    pyrebox_print = printer
    pyrebox_print("[*] Initializing callbacks")
    cm = CallbackManager(module_hdl)
    cm.add_callback(CallbackManager.CREATEPROC_CB, new_proc, name="vmi_new_proc")
    cm.add_callback(CallbackManager.REMOVEPROC_CB, remove_proc, name="vmi_remove_proc")
    pyrebox_print("[*] Initialized callbacks")
def clean():
    
    Clean up everything. At least you need to place this clean() call to the callback manager, that will unregister all the registered callbacks.
    
global cm
pyrebox_print("[*] Cleaning module")
cm.clean()
pyrebox_print("[*] Cleaned module")
Script life-cycle

Script requirements:
  > requirements = ["plugins.guest_agent"]

Once it is initialized, it will be executed when:
  – An installed callback is triggered
  – A defined command is executed

> def do_command(line):
def new_proc(pid, pgd, name):
    global cm
    pyrebox_print("Process %s: PID:%x PGD:%x" % (name, pid, pgd))

def remove_proc(pid, pgd, name):
    global cm
    pyrebox_print("Removed process %s: PID:%x CR3:%x" % (name, pid, pgd))
Scripting

- Key concepts
  - Processes are identified by their address space (PGD / CR3)
  - Callbacks have different behavior
    - Check docs!

- Monitored process
  - Certain callbacks are only triggered for monitored processes
  - From shell: mon/unmon
  - From script:
    api.start_monitoring_process
Scripting

- Several scripts provided as **examples**
  - Automatically running a binary and starting a shell on entry point
  - Monitoring memory write + memory execution (unpacked code detection)
  - Tests for every callback type
  - Usage of triggers

- Complete API documentation provided
Agent
Agent

- File transfer and execution

- Process running on the guest that communicates with host via invalid opcodes

- Windows and Linux guests supported, 32 & 64 bits

- From shell or scripts:
  
  ```python
  > agent.copy_file(src_path, dest_path)
  > agent.execute_file(path, args=[], env={}, exit_afterwards=False)
  ```
Malware Monitor
Malware Monitor

- 4 different modules, configurable (json)
- **API tracer**
  - Text log
  - Binary log (import in IDA)
  - Optionally, can extract parameters
- **Memory dumper**
  - Automatically dump under certain conditions
- **Code coverage**
  - Binary log (colorize B.B.s in IDA)
  - Text log (identify jumps between VAD regions)
Malware Monitor

- Memory event logger (interproc)
  - Events monitored:
    - Memory allocation / deallocation
    - Process creation, process handle opening
    - Remote memory writes / memory sharing
    - File reading/writing. File mapping
    - Memory permission changes

- Useful to track injections, droppers, downloaders
- Outputs a condensed text-based report
  - + A log of events
Future work
What’s next?

- Support for additional architectures (ARM / MIPS)
- Support for other Operating Systems
- Debugging backend for IDA or r2
- Integration into PyREBox of other tools
- Support for other backends (PANDA?)
Questions?