Do you know what's happening in your <put application title here>?

Felix Leder
Honeynet Project

Daniel Plohmann
Fraunhofer FKIE

University of Bonn

A hypothetical scenario

ROOTKITS = MONITORING TOOLS

Levels of root-kits

Ring 3: User-space modifications
- Observe in debugger – use breakpoints
- (API) hook relevant functions

\+ Allows to observe almost everything the program is doing
- Can easily be detected

Ring 0: Kernel-space modifications
- System Service Descriptor Table (**SSDT**) hooking
- IDT hooking
- IRP handler hooking
- MSR, callbacks, …

\+ Is very stealthy
- Only interaction with kernel is observed

Levels of root-kits

Ring -1: Virtual Machine Introspection
- Insert handlers during code translation process

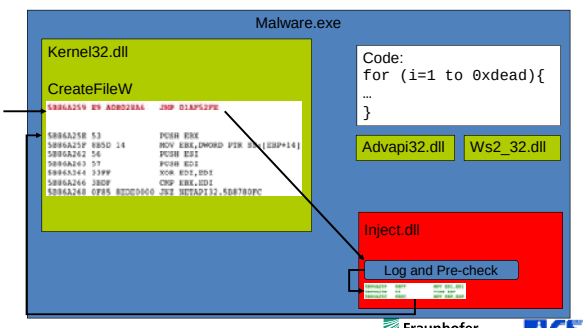
\+ No chance to identify any changes to the system or process
- Virtual machine detection is possible [Rutkowska, …]
- Performance
- Depends on OS (version)

Ring -2, -3: Direct hardware interaction
- ACPI, Firmware, Intel AMT

\+ This level is usually not investigated
\+ Full memory access (sometimes CPU)
- Convenience
- Hard to monitor specific events (performance)
- Limited space – only low complexity

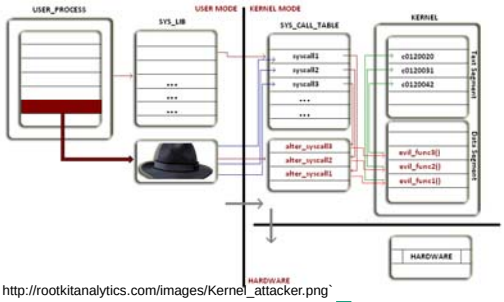
API Hooking Explained (User-Space)

- Used by malware and malware monitoring tools



SSDT Hooking (Kernel-Space)

- General concept published by Russinovich, Schreiber, … (e.g. Undocumented Windows 2000 secrets - http://undocumented.rawol.com/)



http://rootkitanalytics.com/images/Kernel_attacker.png`

Virtual Machine Introspection

- In VMs original (assembly) code is split into blocks
- Each block is in-time translated into new “VM emulated” code
- Custom code can be inserted during translation:
  - Logging
  - Modification of current state
  - Firewalling / prevention of certain functions

```
push    ebx
push    ebx
push    ebx
push    ebx
lea     eax, [ebp+22]
push    eax
call    ds:InternetOpenA
```

Translation

```
push    ebx
push    ebx
push    ebx
push    ebx
lea     eax, [ebp+22]
push    eax

if processname == “malw.exe”
    call VM_log_InternetOpenA

call    VM_InternetOpenA
```


Stay flexible – Stay in user-space

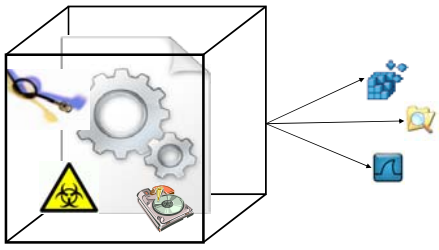
- Run on bare metal or VM
- Run on different Windows versions
- Run on different hardware (limited)
- See everything for real:
  - Network data before encryption
  - Internal functions (encoding, interpreter, …)

Sandbox

Framework

Python

for creating sandboxes



PyBox - Use case examples

- **Forensics** (live system)
  - Monitoring running processes
  - Fake harddrive
  - Fake (interceptable) network connection (e.g. by patching certificate verification function)
- **Malware investigation**
  - Automatic behavior extraction
  - Infection mechanisms, persistency
  - Propagation
- **Root-kit research**
  - Creating arbitrary user mode root-kits

Existing Sandbox approaches

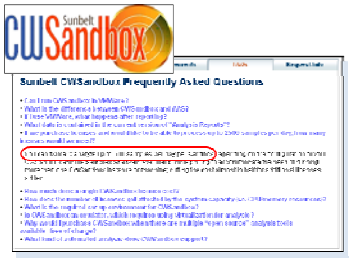
Powerful but …

- Hard-coded monitoring capabilities
- Influence on performance (by irrelevant monitoring)
- No reconfiguration at run-time
- No process internals (e.g. scripting interpreter, encryption, …)

➔ Hard to adopt to new types of use-cases

Scalability Problem Illustrated

- Annual Reports 2009:
  ~ 55.000 new samples per day (PandaLabs)
  ~ 90.000 unique ZeuS binaries (Symantec)
  2,895,802 new malware signatures (Symantec)



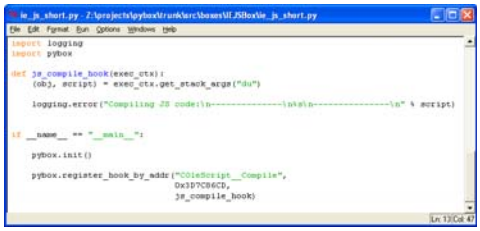
- 500 analyzed samples / day ⇔ 2-5 minutes / samples
- 55k samples/day
  = 1 new sample per 1.5 sec

PyBox Overview

- Framework for creating sandboxes in python
- Flexible
  - Only monitor what is required
  - Reconfiguration at run-time
  - Arbitrary hooking
    (even of functions with unknown/changing signature)
- Ease of use
  - High-level :Python - no need for ASM or even C
    (not as high as UML , yet ;))
  - Script
  - Fully exchangeable at run-time
- Open Source

Python Integration

- Idea: Inject Python interpreter into remote process
- High-level:
  - Scripting (no compilation)
  - Reconfiguration
  - In Python almost everything can be done in 10 LoC
- Low-level:
  - Full memory access
  - Full register access
  - Full function parameters
- Ctypes is awesome ;)

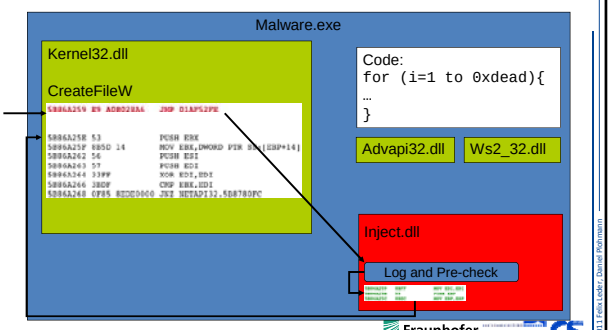


Demo

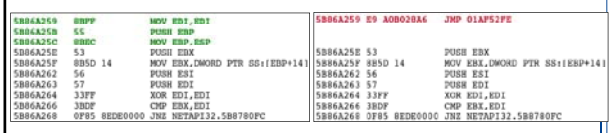
IE JAVASCRIPT

API Hooking Explained

- PyBox relies on API hooking

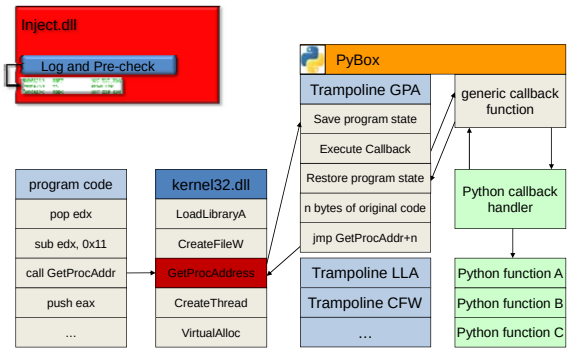


Hooking Challenges



- Jmp (or call) needs 5 bytes
- Intel x86 instructions have variable length
- Make sure, there is enough space
- Make sure, no instruction gets split

The Trampoline



Behind the curtains

!!! Don't care about this – PyBox does it for you !!!

- Trampolines are automatically created:
  - Individually for each hook
  - Heap
- Generic Callback with parameters in each trampoline

PUSHF PUSHAD	Store environment (flags, registers)
PUSH 1 PUSH 3D7C86CD	Callback Parameters (Individual, hook address)
CALL pybox.00F810E0	Call PyBox
POPAD POPF	Restore environment

MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
	Original code

JMP jscript.3D7C86D2	

Convenience

Hooking by …

- Dll-name + function name

```
register_hook("kernel32.dll“,
              "LoadLibraryExW",
               ll_hook)
```

- Address

```
register_hook_by_addr("COleScript__Compile",
                      0x3D7C86CD,
                      js_compile_hook)
```

- On return to original code

```
register_return_hook("create_object_rethook",
                     exec_ctx,
                     co_return_hook)
```


High Level but still advanced …

Keep manipulation and investigation options:

- Full access to registers
  (manipulation due to trampoline)

```
object_addr = exec_ctx.regs.EAX
```

- Full access to memory
  - "Safety" layer, searching, ascii, unicode, dereferencing

```
memorymanager.find_regex(start, end, "[A-F0-9]{32}")
```

- Possibility to modify original arguments

```
exec_ctx.set_arg(1, 0xh0tbabe)
```

- Modify result (hiding)

Dynamic Sandboxing – Mass Hooking

- Mass hooking
  - Normal hooking requires info about signature
  - (calling convention + parameters)
  - A lot of effort for hooking every function

```
VOID WINAPI TimedSleep(DWORD dwMilliseconds) {
    … pre-analysis
    TrueSleep(dwMilliseconds);
    … post-analysis
}
```

- Better: don't interfere with stack
- PyBox:
  - Pre-Analysis ➔ Original State ➔ Post-Analysis
  - Parameters can be extracted from ExecutionContext
    (if known)

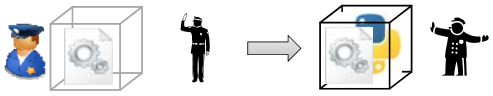
Dynamic Sandboxing – Hook Manager

- Book-keeping of hooks
  (differentiate own hooks from real root-kits)
- Check for removed hooks
  (anti / evasion)
- Hook removal
  - Improve performance by removing un-needed hooks
  - Keep manipulation up only as long as needed (detection)

Getting the Monitoring Going

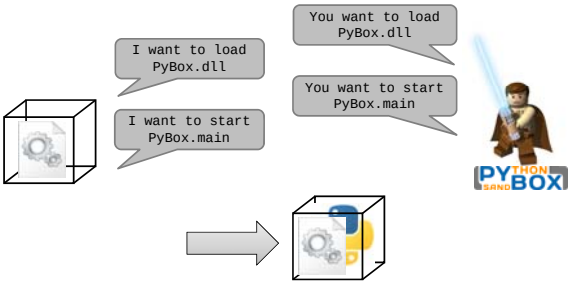
- Inject PyBox.dll into process
- Set up Python environment
- Execute configured script
  - Perform hooking
  - …

Option A: Injection during start



Getting the Monitoring Going

- Option B:
  Injection into running process



Demo

FIREFOX DEMO

Sandbox Features

ALL INCLUSIVE

Standard Sandbox

- Monitor File Accesses
- Monitor Registry Accesses
- Monitor Network Activity
- Process Activity (Termination, Creation, …)
- Memory allocation / deallocation
- Windows Services interface

```
def register_hooks():
    """Register all hooks"""

    hooks_executable.init()
    hooks_file.init()
    hooks_memory.init()
    hooks_misc.init()
    hooks_network.init()
    hooks_registry.init()
    hooks_services.init()
    hooks_synchronisation.init

    return
```


Process Tracking

Malware often

- Drops other executables (and executes)
- Injects itself into other prcesses (just like PyBox)
- Makes tracing difficult

Process Tracking

- 1 LoC and be happy:

```
pybox.proctrack.init()
```

- Track:
  - Process creation (CreateProcess)
  - Thread creation (into other process)
- Process Rigger – the man
  - same script (std.) or new script for other process
  - Set Python environment
  - Everything prepared for _you_

Demo

SPY EYE

What about…

ADOBE ACROBAT?

Acrobat Internals

- Different versions – different API
- Exploits often related to embedded javascript
- Monitoring activities
- Malicious activities start internally
- Regular sandbox allows monitoring of exploit consequences, but not exploit itself = functions involved in Acrobat

Plugins

- In "\Reader\plug_ins*.api"
- Spelling, File formats, Forms, Mailer, **Scripting Interpreter**
- We are interested in monitoring the scripting interpreter
- Regular Dlls loaded with LoadLibraryExW
- Only one exported function -> PlugInMain()
- Hooks cannot be installed at program start
- Wait until document requires javascript plugin

Waiting for the plugin

```
pybox.register_hook("kernel32.dll",
                    "LoadLibraryExW",
                    plugin_hook)

def plugin_hook(exec_ctx):
    path = exec_ctx.get_stack_args("u")[0]
    filename = pluginpath.lower()
    if filename.endswith("escript.api"):
          # do something
```

Inside the Javascript

- Different (js) functions have been vulnerable in the past
- Want to monitor **ALL** functions / methods (locate the problem)
- Long way: RE all available names + addresses
- Quicker – hook…
  - Register_new_object
  - Register_new_method
  - Register_new_property
  - Register_new_function

  … and hooked the function names + address parameters

Monitoring Method Registration

10 Lines of Code:

```
(parent, name, callback) =
      exec_ctx.get_stack_args("dad")

parent_name = "unknown (0x%x)" % parent
if OBJECT_DICT.has_key(parent):
     parent_name = OBJECT_DICT[parent]

fullname = "%s.%s" % (parent_name, name)

if not pybox.HOOK_MANAGER.is_hooked(callback):
      pybox.register_hook_by_addr(fullname,
                                  callback,
                                  generic_hooker,
                                  fullname)
```


eval(), unescape(), …

- JS often obfuscated
- Common strategies
  - unescape, decrypt
  - Eval
  - document.write with new JS
- Common defense: Use javascript framework to follow (e.g. spidermonkey)
- Our approach: Wait for JS to compile

Demo

ACROBAT

Future Work

For us:

- Limited to Windows, yet ;)
- Limited to 32-Bit, yet ;)
- Limited to Intel x86, yet ;)

For you:

- Evasion (we are not powering our opponents)
- Other types of logging (we don't do XML ;))

Interested in More?

- !! We are interested in Feedback !! – Open Source

http://code.google.com/p/pyboxed

felix.leder@googlemail.com
daniel.plohmann@googlemail.com