Reverse Engineering Malware Dynamic Analysis of Binary Malware II

F-Secure®

Advanced dynamic analysis

- Debugger scripting
- Hooking and library injection
- Instrumentation frameworks
- Emulators and virtualization
- Memory forensics

F-Secure

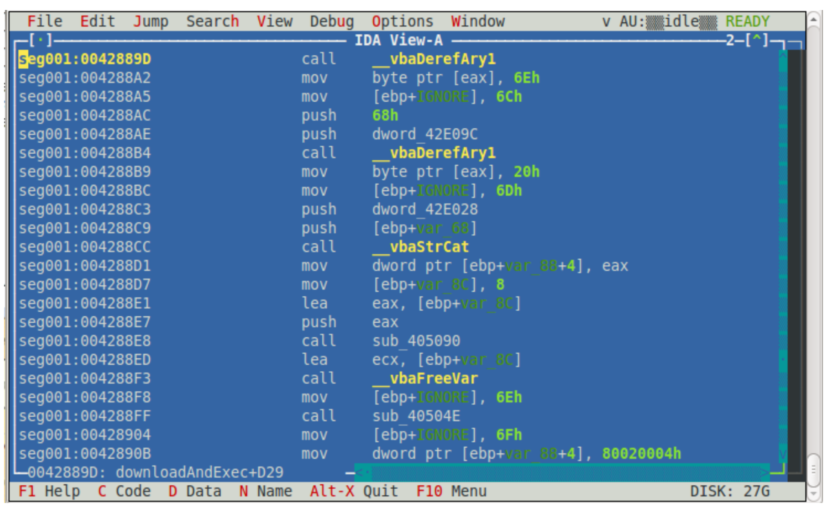
Debugger automation with scripting

- Debuggers can be extended with flexible scripting languages like python
- Any debugging task can be automated: unpacking, decrypting strings, etc.
- Debuggers that support python scripting:
  - Immunity debugger
  - GDB
  - IDA debugger
- Python debugger module for Windows:
  - PaiMei, reverse engineering framework includes ”PyDbg” module
  - F-secure proprietary python Win32 debugger using ctypes

Example debugger script: Sober.Y URL's

- Sober was a family of email-worms, written in Visual Basic
- It updated itself using a set of dynamically generated URL's
- Reversing the URL generation algorithm was very challenging
- Developing an automated debugging script was much faster

DEMO: Case Sober.Y



F-Secure

Hooking and Library Injection

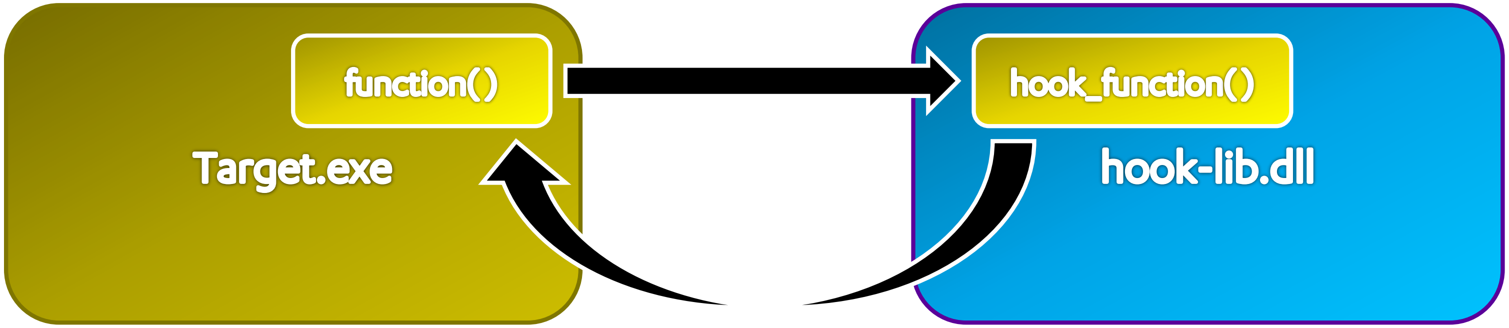
- Basic tracing and troubleshooting techniques
- Can be used in dynamic analysis and reverse engineering very effectively
- Hooks are not using debug API:
  - Fast execution
  - Not confused by anti-debugging tricks
- Intrusive (will modify the target address space)
- Tools can be quite complicated to use (notable exception: Frida)
- Executed on real hardware!
- Example use: Trace file I/O, registry and networking for analyzing program functionality
- Example use: Detect dynamically generated code sections for unpacking

Inline hooking

- Simple way to instrument binaries dynamically, without need to recompile programs
- Usually done by inserting branches to the hooked (target) functions
- Hooking function gets the same parameters as target function
- After the analysis, target function gets back the control
- Analysis code in the hooking function needs memory space
- Position-independent shellcode can be inserted anywhere in the memory space, but more common method is to provide code by injecting dynamically loaded library

Library injection

- Target process is forced to load extra module containing the instrumentation code
- Interesting functions in the target process are hooked
- Hooking module does the necessary processing and returns back to hooked function



F-Secure

PE IAT hooking

- Basic idea: hook by replacing the pointer in import address table (IAT) to the hooking function
- IAT can be easily parsed from the PE headers
- Hooking function is inserted in the address space by library injection

Example hooking library: Detours

- Microsoft Research hooking/injection library for x86/amd64/ia64 Windows
- Uses flexible inline hooking technique
- Understands native functions and managed code (MSIL)
- Detours DLL is loaded with library injection:
  - Dynamically with library injection
  - Statically, by modifying the target import table for loading the Detours DLL before target entry point executes (**DetourCreateProcessWithDLL()**)
- Hooks are inserted conveniently with **DetourAttach()** and removed with **DetourDetach()**

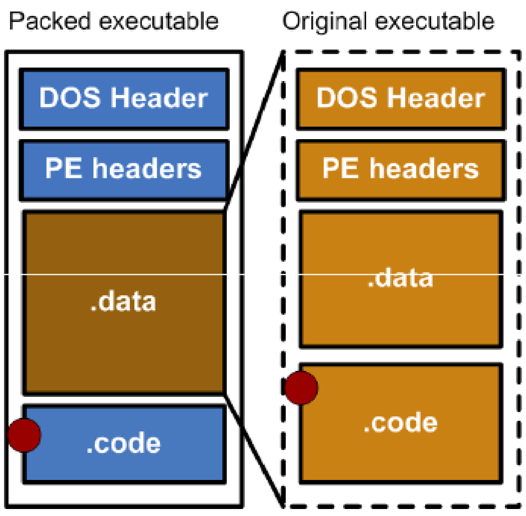
F-Secure.

Example hooking library: Frida

- Hook library functions on Windows, OSX, iOS, Linux and Android
- Injected code written in JavaScript
- Internally, Frida injects V8 engine and builds transparent code transitions from native to JS and back

Example tool: FIST

- FIST: F-secure Interactive System Trace
- Proprietary tool for generic unpacking on x86 Windows
- Hooks most kernel32.dll, advapi32.dll, msvcrt.dll, shell32.dll and user32.dll functions
- Hook functions compare the code in return address to the disk image
- If the return address was modified, the code is possibly near the original entry point
- Based on the fact that most non-trivial programs need to use Win32 API's



F-Secure

Instrumentation frameworks

- Dynamic manipulation of programs using binary instrumentation
- Good for profiling and debugging, but also writing reverse engineering tools
- More flexible than function hooking:
  - Instrument instructions, basic blocks
  - Instrument system calls
  - Inspect memory read/write
  - ... And much more
- Major frameworks: DynamoRIO, Pin, Valgrind
- Problems with some packed files!

Using emulators for tracing and instrumentation

- Emulators can be used for tracing by instrumenting code outside the OS
- By definition, it is non-intrusive
- Target executed on emulated hardware, more safe than debugging and hooking
- Instrumentation API:
  - Interface for hooking up instructions, exceptions etc.
  - Example: bochs instrumentation API
- Debugging API:
  - Emulator can export standard debugger API, such as GDB
  - Example: qemu GDB stub

Emulator types

- Hardware virtualization
  - Emulator is sharing the hardware resources with the host machine
  - CPU instructions run directly on real CPU
  - Good performance
  - Examples: VMWare, VirtualBox, Xen, KVM (Linux kernel VM)
- Software
  - Emulator implemented purely using software
  - CPU instructions are interpreted or translated dynamically
  - Can be quite slow
  - Examples: Bochs, Qemu

The HW/SW distinction is not really that clear, for example all HW virtualization solutions will fallback to software emulation in certain situations, like real-mode. Also Xen and KVM use qemu for hardware emulation.

Emulator example: Bochs instrumentation API

- Bochs: open-source PC (x86/amd64) emulator
- Uses interpretation for emulating the instructions
- Interpretation makes Bochs very portable, it runs on any C++ environment
- Supports powerful instrumentation with C++
- Callbacks for
  - CPU events, like interrupts and exceptions
  - CPU instructions
- Support functions, such as memory I/O

Emulator example: Using the Qemu GDB stub

- Qemu: open-source multi-platform emulator
- Uses dynamic code translation for speeding up the emulation
- Supports debugging via the built-in GDB stub
- Qemu GDB stub features:
  - Non-intrusive
  - Breakpoints are implemented in the stub (”hardware”)
  - VM time stops when the stub is waiting for input
- GDB supports python scripting
- Flexible system-level tracing tools

Attacking emulators

- Malware has a lot of ways to detect emulators, roughly categorized as:
  - Timing attacks
  - OS implementation
  - Hardware implementation
  - Emulator-related software inside the OS, for example VMWare tools
- Emulators can also be attacked with denial of service attack:
  - Execute massive amount of instructions
  - Emulators in AV engines cannot give too much clock cycles for the emulator
- Most dangerous attack on emulators is to escape from the emulated environment by using a bug in the emulator sofware

Detecting emulators: OS implementation

- If the emulator is not running full-blown OS, its API emulation can be easily detected
- Windows has huge amount of documented API's and undocumented, still quite solid API's
- Emulators try to return something even for unsupported API's, just to keep execution ongoing
- Current malware uses a lot of API-related tricks to detect emulators
- Some examples:
  - Call API's with bogus or unsupported parameters, verify return values
  - Use of callback functions in the API's for doing something useful
  - Observe side-effects of API's (register values, traces in stack etc.)

Detecting emulators: Hardware

- Implementing a CPU emulator is a very complicated task:
  - Intel x86/amd64 instruction set consists of ~500 instructions
  - Paging and exception handling is complicated
- Full-blown PC emulator needs to implement a fair amount of hardware devices to be convincing
- Some examples for detecting emulator hardware:
  - Detect missing CPUID information or inconsistencies (*)
  - Check implementation of complicated instructions, like CMPXCHG8B (*)
  - Check non-zero Local Descriptor Table (LDT) to detect VMWare (*)
  - Detect VMWare devices, for example ”VMWare PCI Express Root Port”

(*) Peter Ferrie: Attacks on More Virtual Machine Emulators (http://pferrie.tripod.com/papers/attacks2.pdf)

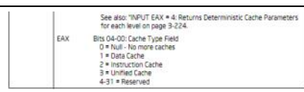
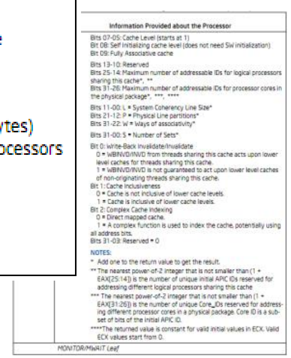
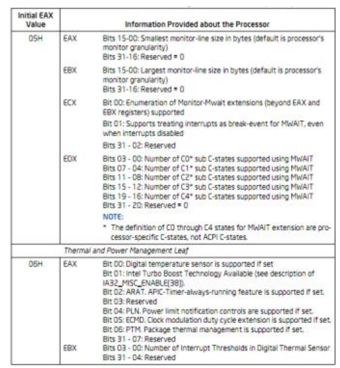
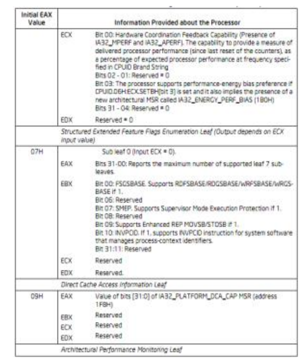
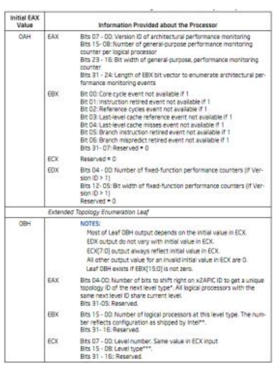
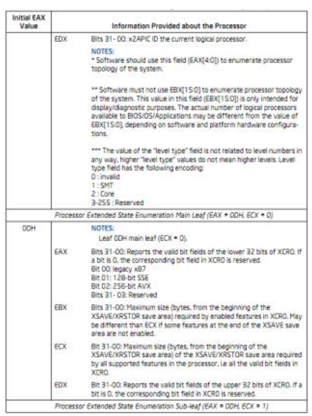
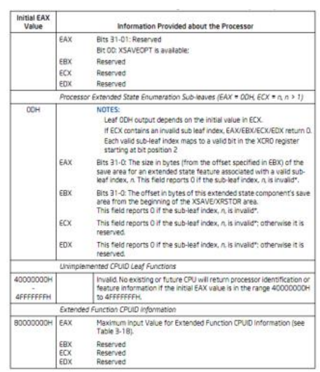
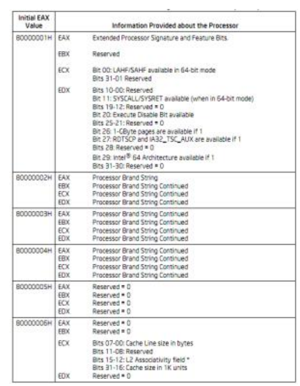
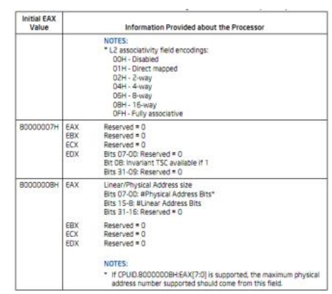
Emulator detection example: CPUID instruction

- CPUID is used to get the processor information:
  - Vendor identification string, for example ”GenuineIntel”
  - CPU type, family, model and stepping
  - Supported instruction sets
  - Other features, such as thermal and power management
- Software emulator needs to be consistent in CPUID return values and features it can emulate
- Attacker can also check if such a CPU is really available in reality
- Almost all software emulators fail to be consistent

CPUID continued

From Intel 64 and IA-32 Architectures Software Developer's Manual:

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information (see Table 3-18)
	EBX	"Genu"
	ECX	"ntel"
	EDX	"ineI"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5)
	EBX	Bits 07-00: Brand Index Bits 15-08: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID
	ECX	Feature Information (see Figure 3-6 and Table 3-20)
	EDX	Feature Information (see Figure 3-7 and Table 3-21)

Memory forensics

- Idea: let the malware run freely and analyze the memory
- Why is it effective:
  - All interesting is in memory (executable images, file I/O buffers, OS structures and objects etc.)
  - Analysis is done outside the running environment
- Possible drawbacks:
  - Slowness
  - When to dump?
  - On physical machines, dump software can be tampered

Volatility

- Open-source framework for physical memory analysis
- Support for 32 –and 64-bit Windows, Linux and OSX
- Gather information from processes, virtual memory, OS structures, OS objects and more
- Dump images and memory pages on disk
- Lots of useful plugins, like ”malfind”
- Typical workflow: pause a VM, give volatility the physical memory as a flat file

DEMO

Volatility example: Equation/GRAYFISH

```
> volatility -f grayfish.vmem malfind

Process: services.exe Pid: 716 Address: 0xb90000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 8, PrivateMemory: 1, Protection: 6

0x00b90000  68 00 00 00 00 68 17 00 b9 00 68 d5 1f 82 7c 68   h....h....h...|h
0x00b90010  fa 13 b0 00 ff 24 24 8b c5 83 c0 11 c7 00 29 16   .....$$.......).
0x00b90020  80 7c 81 c0 ad ff ff ff bf 6a 15 b0 00 33 ed ff   .|.......j...3..
0x00b90030  64 24 fc 90 90 90 90 90 90 90 90 90 90 90 90 90   d$..............

0xb90000 6800000000       PUSH DWORD 0x0
0xb90005 681700b900       PUSH DWORD 0xb90017
0xb9000a 68d51f827c       PUSH DWORD 0x7c821fd5
0xb9000f 68fa13b000       PUSH DWORD 0xb013fa
0xb90014 ff2424           JMP DWORD [ESP]
```

Equation: The Death Star of Malware Galaxy - Kaspersky Labs' Global Research & Analysis Team

Equation/GRAYFISH

```
> volatility -f grayfish.vmem vaddump –pid 716
```

```
> hexdump -C services.exe.1f64550.0x00b00000-0x00b0afff.dmp

00000000  0f 5e 30 00 61 00 00 00  2c 00 00 00 35 35 00 00  |.^0.a...,...55..|
00000010  e8 00 00 00 00 00 00 00  c0 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 50 00 00 00  |............P...|
00000040  1a 95 7e 1a 00 bc 23 8f  2b e8 cb 44 8f 2b 9c 78  |..~...#.+..D.+.x|
00000050  43 31 60 d0 66 05 ad 66  eb 6f 60 81 eb 3a 3a 05  |C1`.f..f.o`..::.|
00000060  fc 60 b6 17 60 66 c7 3a  60 43 3a 60 ec a5 d1 60  |.`..`f.:`C:`...`|
00000070  6f 05 4c 17 7a 4f 4f ee  8c 00 00 00 00 00 00 00  |o.L.zOO.........|
00000080  99 ac eb 4a 85 97 e5 8f  85 97 e5 8f 85 97 e5 8f  |...J............|
00000090  a4 73 cb 8f ef 97 e5 8f  80 3e f7 8f 59 97 e5 8f  |.s.......>..Y...|
000000a0  80 3e b9 8f ef 97 e5 8f  13 a1 f7 8f 62 97 e5 8f  |.>..........b...|
000000b0  a4 ef 06 8f 24 97 e5 8f  85 97 1a 8f 7d 97 e5 8f  |....$.......}...|
000000c0  13 a1 2c 8f 97 97 e5 8f  b0 37 b9 8f ba 97 e5 8f  |..,......7......|
000000d0  06 43 81 78 85 97 e5 8f  00 00 00 00 00 00 00 00  |.C.x............|
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  70 b7 00 00 44 cb 2c 00  e5 16 93 4d 00 00 00 00  |p...D.,....M....|
```

F-Secure

Equation/GRAYFISH

```
> volatility -f grayfish.vmem vaddump –pid 716
```

```
> hexdump -C services.exe.1f64550.0x00b00000-0x00b0afff.dmp

00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 f0 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  ab 84 61 9e ef e5 0f cd  ef e5 0f cd ef e5 0f cd  |..a.............|
00000090  6c f9 01 cd ed e5 0f cd  80 fa 05 cd eb e5 0f cd  |l...............|
000000a0  80 fa 0b cd ed e5 0f cd  d9 c3 05 cd e6 e5 0f cd  |................|
000000b0  6c ed 52 cd ec e5 0f cd  ef e5 0e cd d7 e5 0f cd  |l.R.............|
000000c0  d9 c3 04 cd e5 e5 0f cd  10 c5 0b cd ee e5 0f cd  |................|
000000d0  52 69 63 68 ef e5 0f cd  00 00 00 00 00 00 00 00  |Rich............|
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  50 45 00 00 4c 01 04 00  0f 82 59 47 00 00 00 00  |PE..L.....YG....|
```

F-Secure

KAN: Forensics memory tracing

- Memory tracing engine KAN:
  - Build on top of KVM, the Linux kernel VM
  - Presented in Recon 2014
- Instead of a single memory snapshot, take a series of snapshots
- Create a coherent overall picture of system behavior (much like debugging/system tracing)
- Capture transient memory data, like
  - Obfuscated code and data
  - Self-modifying code
  - Crypto keys and buffers
  - Short-lived data, like URL's, networking buffers, configuration data, etc.
- More information coming from Endre Bangerter, Security Engineering Lab, Bern University of Applied Sciences

Summing it all: Cuckoo sandbox

- Cuckoo has it all: instrumentation with hooks, emulators, memory forensics, network traffic analysis, automated reporting
- Can utilize several different VM platforms: VMWare, KVM, VirtualBox, or use a custom platform, for example real HW
- Support of volatility analysis after sample run
- Nice reporting
- Used by VirusTotal and many others

SWITCH ON FREEDOM