In-Memory Malware Analysis

PV204 Laboratory of security and applied cryptography II

Course handouts and notes. // Václav Lorenc

In-Memory Malware Analysis

PV204 Laboratory of security and applied cryptography II

Before we start…

A short introduction, how a common attack (let's assume we are talking about targeted attacks) is usually performed:

1. Reconnaissance
2. Weaponization
3. Delivery
4. **Exploitation**
5. **C2**
6. **Exfiltration**

Malware phases/stages:

1. System Infection / Exploit
2. Dropper / Downloader (multi-stated)
3. Callbacks
4. Configuration / Updates
5. Removal

How to use in-memory analysis?

1. Run malware / acquire memory.
2. Analyze memory, have fun.
3. ???
4. Profit!

How to setup your lab system?

- Oracle VirtualBox (or another virtualization technology capable of making snapshots and memory dumps).
- Windows XP/7/8 license or trial
- Memory dumping tool
- Adobe Reader and any other vulnerable software you consider as important.

- *(if possible, disable swap)*

- *Debuggers*
- *Disassembles*
- *Text editors*

Reverse Engineering for Beginners (x86)

Registers

8bit	16bit	32bit	64bit
AL	AX	EAX	RAX
BL	BX	EBX	RBX
CL	CX	ECX	RCX
DL	DX	EDX	RDX
	SI	ESI	RSI
	DI	EDI	RDI
	BP	EBP	RBP
	SP	ESP	RSP

And of course, instructions pointer (`IP/EIP/RIP`) and flags (`flags/rflags`), segment registers (`CS, SS, DS, ES, GS, FS`), FPU registers, SSE, SSE2, …

BTW: `BX` register can be used for loops (like many other registers), but `LOOP` instruction works with `(E)CX`.

Instructions "families"

Evergreen	The past	The future
push	aaa	crc32
mov	xlat	aesenc
call	verr	pcmpistrm
retn	smsw	vfmsubadd132ps
jmp	lsl	movbe

Function Entry/Exit

```
push ebp
mov ebp, esp
sub esp, X
```

```
mov esp, ebp
pop ebp
ret X ; sizeof(x) + sizeof(y) + sizeof(z)
```

More Information

Very nice PDF published by Dennis Yurichev with introduction into Reverse Engineering, assembly and some more advanced topics: http://yurichev.com/non-wiki-files/RE_for_beginners-en.pdf

Other interesting/necessary topics

Memory addressing

- Segmentation/Paging/Virtualization
- Process vs. Kernel space and addresses; **DLL/process injection**

Executable formats

- Legacy DOS formats (DOS, EXE)
- Portable Executable (PE), symbol imports, DLL loading

Non-documented instructions/behavior

- Intel vs. AMD; Virtual Machines

Calling conventions

- cdecl
- stdcall (most common on Windows)
- fastcall

Anti-debugging tricks

- Exceptions, interrupts, time checking, debuggers detection, PE header manipulation, PEB manipulation…

Anti-VM tricks

- looking for uncommon sequences and/or behavior (CPUID instruction, e.g.); BIOS analysis
- Windows registry keys presence

Code obfuscation/packing

- Virtualization (own instruction interpreters)
- UPX/ASPack – the most famous packers
- Usually combined with the previous techniques

Getting current IP (instructions pointer)

- near/far calls or jumps, useful for shellcodes

Win32 API Calls

- API hash strings, trampolines, … (to prevent detection)

Crypto Algorithms

- xor, aes, rc4, rsa

Data Execution Prevention & others

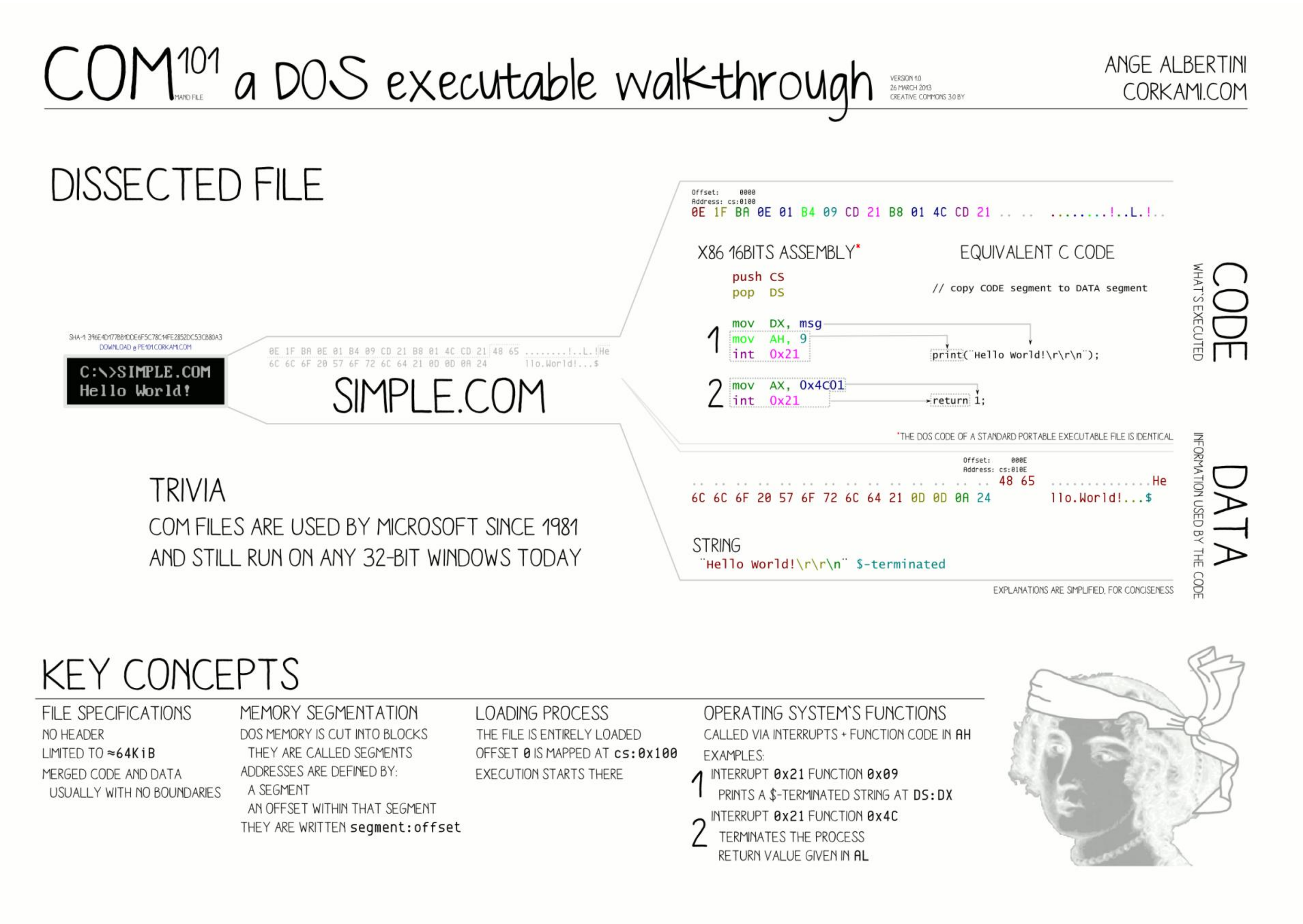
- DEP (Data Execution Prevention)
- ASLR (Address Space Layout Randomization)
- The previous techniques can (of course) be bypassed: ROP (Return Oriented Programming)

Images

All the images were taken from Corkami Project webpage (https://code.google.com/p/corkami/).

Index of images:

- COM file, DOS executable walkthrough
- PE, Portable Executable walkthrough



COM^{101} a DOS executable walkthrough

MAND FILE

VERSION 1.0
26 MARCH 2013
CREATIVE COMMONS 3.0 BY

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE

SHA-1: 396E4D177BB1DDE6F5C78C14FE2852DC53CB80A3
DOWNLOAD @ PE101.CORKAMI.COM

```
C:\>SIMPLE.COM
Hello World!
```

```
0E 1F BA 0E 01 B4 09 CD 21 B8 01 4C CD 21 48 65  ........!..L.!He
6C 6C 6F 20 57 6F 72 6C 64 21 0D 0D 0A 24        llo.World!...$
```

SIMPLE.COM

CODE

WHAT'S EXECUTED

```
Offset:     0000
Address: cs:0100
0E 1F BA 0E 01 B4 09 CD 21 B8 01 4C CD 21 .. ..  ........!..L.!..
```

X86 16BITS ASSEMBLY* | EQUIVALENT C CODE

```
  push CS                          // copy CODE segment to DATA segment
  pop  DS

1 mov  DX, msg
  mov  AH, 9
  int  0x21                        print("Hello World!\r\r\n");

2 mov  AX, 0x4C01
  int  0x21                        return 1;
```

*THE DOS CODE OF A STANDARD PORTABLE EXECUTABLE FILE IS IDENTICAL

DATA

INFORMATION USED BY THE CODE

```
                                   Offset:     000E
                                   Address: cs:010E
.. .. .. .. .. .. .. .. .. .. .. .. .. .. 48 65  ..............He
6C 6C 6F 20 57 6F 72 6C 64 21 0D 0D 0A 24        llo.World!...$
```

STRING
"Hello World!\r\r\n" $-terminated

EXPLANATIONS ARE SIMPLIFIED, FOR CONCISENESS

TRIVIA

COM FILES ARE USED BY MICROSOFT SINCE 1981
AND STILL RUN ON ANY 32-BIT WINDOWS TODAY

KEY CONCEPTS

FILE SPECIFICATIONS

NO HEADER
LIMITED TO ≈64KiB
MERGED CODE AND DATA
USUALLY WITH NO BOUNDARIES

MEMORY SEGMENTATION

DOS MEMORY IS CUT INTO BLOCKS
THEY ARE CALLED SEGMENTS
ADDRESSES ARE DEFINED BY:
A SEGMENT
AN OFFSET WITHIN THAT SEGMENT
THEY ARE WRITTEN `segment:offset`

LOADING PROCESS

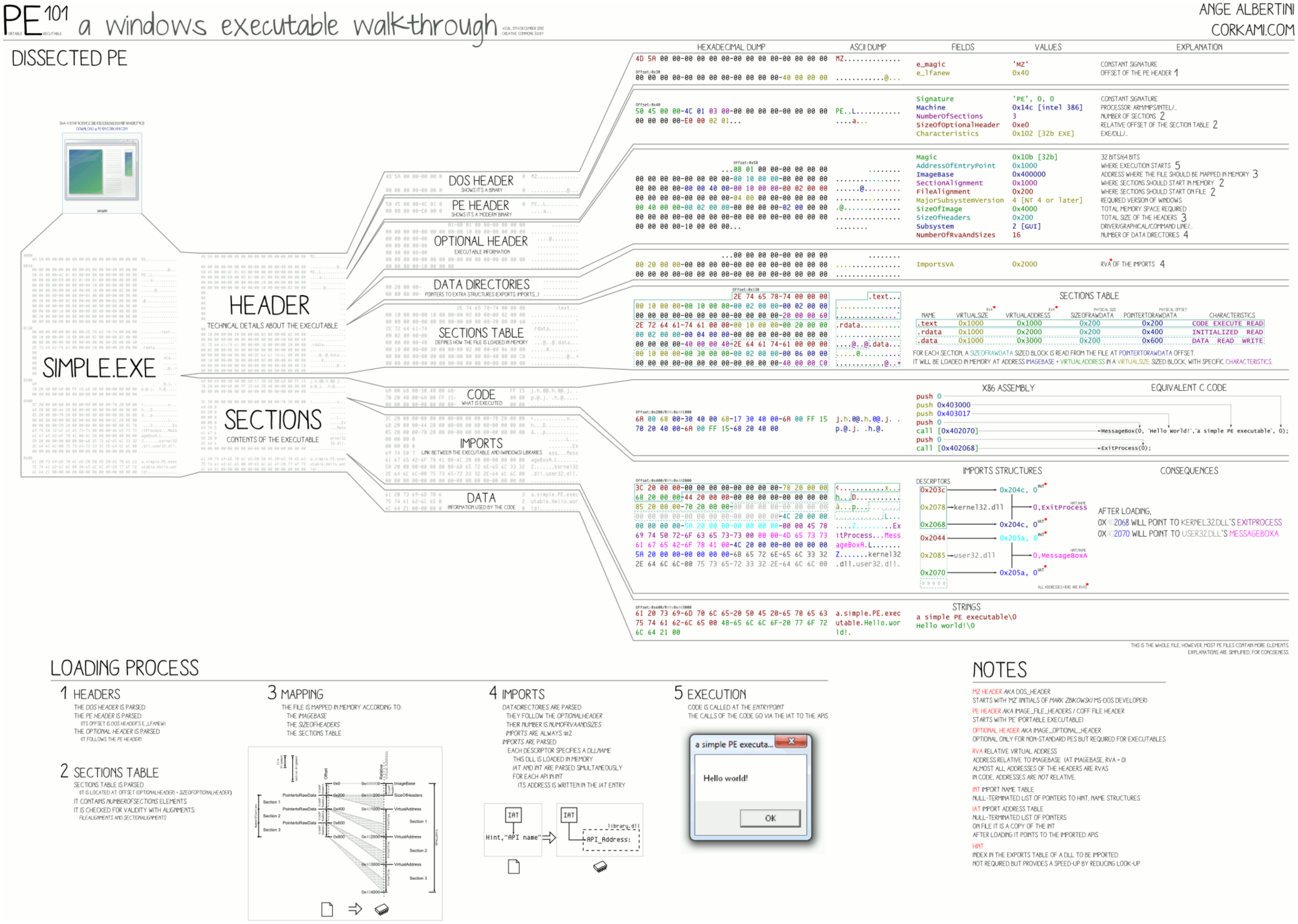
THE FILE IS ENTIRELY LOADED
OFFSET `0` IS MAPPED AT `cs:0x100`
EXECUTION STARTS THERE

OPERATING SYSTEM'S FUNCTIONS

CALLED VIA INTERRUPTS + FUNCTION CODE IN `AH`
EXAMPLES:

1 INTERRUPT `0x21` FUNCTION `0x09`
PRINTS A $-TERMINATED STRING AT `DS:DX`

2 INTERRUPT `0x21` FUNCTION `0x4C`
TERMINATES THE PROCESS
RETURN VALUE GIVEN IN `AL`



PE 101 a windows executable walkthrough

PORTABLE EXECUTABLE
V2.0L 5TH DECEMBER 2012
CREATIVE COMMONS 3.0 BY
ANGE ALBERTINI
CORKAMI.COM

DISSECTED PE

SHA-1 B7AF4C851CE3BE43E030656E826FBFAB4080F9C8
DOWNLOAD @ PE101.CORKAMI.COM
simple
SIMPLE.EXE
HEADER
TECHNICAL DETAILS ABOUT THE EXECUTABLE
SECTIONS
CONTENTS OF THE EXECUTABLE
DOS HEADER
SHOWS IT'S A BINARY
PE HEADER
SHOWS IT'S A MODERN BINARY
OPTIONAL HEADER
EXECUTABLE INFORMATION
DATA DIRECTORIES
POINTERS TO EXTRA STRUCTURES (EXPORTS, IMPORTS,...)
SECTIONS TABLE
DEFINES HOW THE FILE IS LOADED IN MEMORY
CODE
WHAT IS EXECUTED
IMPORTS
LINK BETWEEN THE EXECUTABLE AND (WINDOWS) LIBRARIES
DATA
INFORMATION USED BY THE CODE

HEXADECIMAL DUMP / ASCII DUMP

```
4D 5A 00 00-00 00 00 00-00 00 00 00-00 00 00 00   MZ..............
Offset:0x30
00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 00   ............@...

Offset:0x40
50 45 00 00-4C 01 03 00-00 00 00 00-00 00 00 00   PE..L...........
00 00 00 00-E0 00 02 01...                        ....a...

Offset:0x58
            ...0B 01 00 00-00 00 00 00                    ........
00 00 00 00-00 00 00 00-00 10 00 00-00 00 00 00   ................
00 00 00 00-00 00 40 00-00 10 00 00-00 02 00 00   ......@.........
00 00 00 00-00 00 00 00-04 00 00 00-00 00 00 00   ................
00 40 00 00-00 02 00 00-00 00 00 00-02 00 00 00   .@..............
00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00   ................
00 00 00 00-10 00 00 00...                        ........

            ...00 00 00 00-00 00 00 00                    ........
00 20 00 00-00 00 00 00-00 00 00 00-00 00 00 00   ................
00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00   ................

Offset:0x138
            2E 74 65 78-74 00 00 00                   .text...
00 10 00 00-00 10 00 00-00 02 00 00-00 02 00 00   ................
00 00 00 00-00 00 00 00-00 00 00 00-20 00 00 60   ...............`
2E 72 64 61-74 61 00 00-00 10 00 00-00 20 00 00   .rdata..........
00 02 00 00-00 04 00 00-00 00 00 00-00 00 00 00   ................
00 00 00 00-40 00 00 40-2E 64 61 74-61 00 00 00   ....@..@.data...
00 10 00 00-00 30 00 00-00 02 00 00-00 06 00 00   .....0..........
00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0   ............@..+
```

FIELDS	VALUES	EXPLANATION
e_magic	'MZ'	CONSTANT SIGNATURE
e_lfanew	0x40	OFFSET OF THE PE HEADER 1
Signature	'PE', 0, 0	CONSTANT SIGNATURE
Machine	0x14c [intel 386]	PROCESSOR: ARM/MIPS/INTEL/...
NumberOfSections	3	NUMBER OF SECTIONS 2
SizeOfOptionalHeader	0xe0	RELATIVE OFFSET OF THE SECTION TABLE 2
Characteristics	0x102 [32b EXE]	EXE/DLL/...
Magic	0x10b [32b]	32 BITS/64 BITS
AddressOfEntryPoint	0x1000	WHERE EXECUTION STARTS 5
ImageBase	0x400000	ADDRESS WHERE THE FILE SHOULD BE MAPPED IN MEMORY 3
SectionAlignment	0x1000	WHERE SECTIONS SHOULD START IN MEMORY 2
FileAlignment	0x200	WHERE SECTIONS SHOULD START ON FILE 2
MajorSubsystemVersion	4 [NT 4 or later]	REQUIRED VERSION OF WINDOWS
SizeOfImage	0x4000	TOTAL MEMORY SPACE REQUIRED
SizeOfHeaders	0x200	TOTAL SIZE OF THE HEADERS 3
Subsystem	2 [GUI]	DRIVER/GRAPHICAL/COMMAND LINE/...
NumberOfRvaAndSizes	16	NUMBER OF DATA DIRECTORIES 4
ImportsVA	0x2000	RVA* OF THE IMPORTS 4

SECTIONS TABLE

NAME	VIRTUALSIZE (RVA*)	VIRTUALADDRESS (RVA*)	SIZEOFRAWDATA (PHYSICAL SIZE)	POINTERTORAWDATA (PHYSICAL OFFSET)	CHARACTERISTICS
.text	0x1000	0x1000	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	DATA READ WRITE

FOR EACH SECTION, A SIZEOFRAWDATA SIZED BLOCK IS READ FROM THE FILE AT POINTERTORAWDATA OFFSET.
IT WILL BE LOADED IN MEMORY AT ADDRESS IMAGEBASE + VIRTUALADDRESS IN A VIRTUALSIZE SIZED BLOCK, WITH SPECIFIC CHARACTERISTICS.

```
Offset:0x200/RVA:0x401000
6A 00 68 00-30 40 00 68-17 30 40 00-6A 00 FF 15   j.h.0@.h.0@.j. .
70 20 40 00-6A 00 FF 15-68 20 40 00               p.@.j. .h.@.
```

X86 ASSEMBLY / EQUIVALENT C CODE

```
push 0
push 0x403000
push 0x403017
push 0
call [0x402070]        MessageBox(0, "Hello world!","a simple PE executable", 0);
push 0
call [0x402068]        ExitProcess(0);
```

```
Offset:0x400/RVA:0x402000
3C 20 00 00-00 00 00 00-00 00 00 00-78 20 00 00   <...........x...
68 20 00 00-44 20 00 00-00 00 00 00-00 00 00 00   h...D...........
85 20 00 00-70 20 00 00-00 00 00 00-00 00 00 00   à...p...........
00 00 00 00-00 00 00 00-00 00 00 00-4C 20 00 00   ............L...
00 00 00 00-5A 20 00 00-00 00 00 00-00 00 45 78   ....Z.........Ex
69 74 50 72-6F 63 65 73-73 00 00 00-4D 65 73 73   itProcess...Mess
61 67 65 42-6F 78 41 00-4C 20 00 00-00 00 00 00   ageBoxA.L.......
5A 20 00 00-00 00 00 00-6B 65 72 6E-65 6C 33 32   Z.......kernel32
2E 64 6C 6C-00 75 73 65-72 33 32 2E-64 6C 6C 00   .dll.user32.dll.
```

IMPORTS STRUCTURES

DESCRIPTORS
0x203c → 0x204c, 0 INT*
0x2078 → kernel32.dll → 0, ExitProcess (HINT/NAME)
0x2068 → 0x204c, 0 IAT*
0x2044 → 0x205a, 0 INT*
0x2085 → user32.dll → 0, MessageBoxA (HINT/NAME)
0x2070 → 0x205a, 0 IAT*
0 0 0 0 0
ALL ADDRESSES HERE ARE RVAS*

CONSEQUENCES

AFTER LOADING,
0X402068 WILL POINT TO KERNEL32.DLL'S EXITPROCESS
0X402070 WILL POINT TO USER32.DLL'S MESSAGEBOXA

```
Offset:0x600/RVA:0x403000
61 20 73 69-6D 70 6C 65-20 50 45 20-65 78 65 63   a.simple.PE.exec
75 74 61 62-6C 65 00 48-65 6C 6C 6F-20 77 6F 72   utable.Hello.wor
6C 64 21 00                                       ld!.
```

STRINGS

```
a simple PE executable\0
Hello world!\0
```

THIS IS THE WHOLE FILE, HOWEVER, MOST PE FILES CONTAIN MORE ELEMENTS.
EXPLANATIONS ARE SIMPLIFIED, FOR CONCISENESS.

LOADING PROCESS

1 HEADERS

THE *DOS HEADER* IS PARSED
THE *PE HEADER* IS PARSED
(ITS OFFSET IS *DOS HEADER'S E_LFANEW*)
THE *OPTIONAL HEADER* IS PARSED
(IT FOLLOWS THE *PE HEADER*)

2 SECTIONS TABLE

SECTIONS TABLE IS PARSED
(IT IS LOCATED AT: OFFSET (*OPTIONALHEADER*) + *SIZEOFOPTIONALHEADER*)
IT CONTAINS *NUMBEROFSECTIONS* ELEMENTS
IT IS CHECKED FOR VALIDITY WITH ALIGNMENTS:
FILEALIGNMENTS AND *SECTIONALIGNMENTS*

3 MAPPING

THE FILE IS MAPPED IN MEMORY ACCORDING TO:
THE *IMAGEBASE*
THE *SIZEOFHEADERS*
THE SECTIONS TABLE

4 IMPORTS

DATADIRECTORIES ARE PARSED
THEY FOLLOW THE *OPTIONALHEADER*
THEIR NUMBER IS *NUMOFRVAANDSIZES*
IMPORTS ARE ALWAYS #2
IMPORTS ARE PARSED
EACH DESCRIPTOR SPECIFIES A *DLLNAME*
THIS DLL IS LOADED IN MEMORY
IAT AND *INT* ARE PARSED SIMULTANEOUSLY
FOR EACH API IN *INT*
ITS ADDRESS IS WRITTEN IN THE *IAT* ENTRY

5 EXECUTION

CODE IS CALLED AT THE *ENTRYPOINT*
THE CALLS OF THE CODE GO VIA THE IAT TO THE APIS

NOTES

MZ HEADER *AKA* DOS_HEADER
STARTS WITH 'MZ' (INITIALS OF *MARK ZBIKOWSKI* MS-DOS DEVELOPER)

PE HEADER *AKA* IMAGE_FILE_HEADERS / COFF FILE HEADER
STARTS WITH 'PE' (PORTABLE EXECUTABLE)

OPTIONAL HEADER *AKA* IMAGE_OPTIONAL_HEADER
OPTIONAL ONLY FOR NON-STANDARD PES BUT REQUIRED FOR EXECUTABLES

RVA RELATIVE VIRTUAL ADDRESS
ADDRESS RELATIVE TO IMAGEBASE (AT IMAGEBASE, RVA = 0)
ALMOST ALL ADDRESSES OF THE HEADERS ARE RVAS
IN CODE, ADDRESSES ARE *NOT* RELATIVE.

INT IMPORT NAME TABLE
NULL-TERMINATED LIST OF POINTERS TO HINT, NAME STRUCTURES

IAT IMPORT ADDRESS TABLE
NULL-TERMINATED LIST OF POINTERS
ON FILE IT IS A COPY OF THE INT
AFTER LOADING IT POINTS TO THE IMPORTED APIS

HINT
INDEX IN THE EXPORTS TABLE OF A DLL TO BE IMPORTED
NOT REQUIRED BUT PROVIDES A SPEED-UP BY REDUCING LOOK-UP

In-Memory Analysis

Even though doing an advanced reverse engineering could be life changing experience, analyzing malware in this depth usually needs some good clarification. In many security teams, one of the first steps in incident classification is a triage.

For that reason, in-memory analysis of a running malware might be beneficial.

However, there are many other good reasons why a security engineer should do in-memory analysis first. You should use it when…

- Doing a rapid threat assessment – very efficient method.
- Infected host is online and available for the analysis, not restarted yet.
- There is a chance that the original binary is gone (transient infections).
- There was no original binary stored on the infected host.
- You cannot read JavaScript/Java or any other exploited application's code and vulnerability triggered.
- Messing with packers and obfuscated code can be annoying. Indeed.
- Even though you can pass many defenses put in place by attackers, after many hours you can find one that cannot be broken because of missing DLL/configuration file.
- Some data are being exfiltrated memory can contain important evidence – attackers steps, passwords, tools used, …
- …

Recommended Tools

There are some publicly available tools that can be used for memory analysis. They differ in capabilities, supported systems and licensing.

1. Volatility Framework
2. HBGary Responder
3. Mandiant Redline
4. F-Response
5. …
6. grep/strings :)

My preferred tools are Volatility and HBGary Responder (Pro Edition). The first one is great for its broad scope of system that can be analyzed, the second for graphical debugger.

Memory Acquisition Tools and Techniques (Windows OS)

1. Virtual machine memory dump
    - Not applicable for many hosts (laptops, servers).
    - Super useful for malware analysis when the malware doesn't do any anti-VM tricks.
        - Don't forget to configure as little memory as possible for the running system; it will significantly speed up your analysis.
    - VirtualBox, VMWare can do this, VMWare more convenient.
2. FastDump (Pro)
    - HBGary solution, small footprint, one of the best tools available.
    - Cannot be obtained easily, official pages don't work well; Pro version is expensive.
    - **Can acquire memory that is currently swapped-out!**
        - This can be critical for non-VM systems!
3. Memoryze
    - Free tool by Mandiant.
    - Quite big footprint, XML files, installer.
4. Win32dd.exe
    - Not available anymore, replaced by MoonSols Windows Memory Toolkit.
5. MoonSols Windows Memory Toolkit
    - Community Edition available.
    - Single binary, small footprint.
6. Forensic tools (EnCase, Mandiant, Access Data, …)
    - Remote acquisitions (over the network), compressed images, using already-installed drivers, thus no tampering with the system memory.
    - Corporate tools -- inhuman expensive ;)

Memory Analysis Tools

Mandiant Redline

- http://www.mandiant.com/resources/download/redline
- For free, available for Windows XP, Vista and 7 (32-bit and 64-bit).
- Malware risk scoring index.
  - Helps to assess system quickly.
  - List with system and well-known good apps, suspicious mutexes, etc.

HBGary Responder (CE/Pro)

- http://www.hbgary.com/hbgary-releases-responder-ce
- Community Edition is available against registration.
- Available for Windows XP, Vista and 7.
- Nothing really awesome unless you use Pro Edition.
  - Simple disassembler with graphing features, priceless.
  - Digital DNA – very good process/memory scoring system.
- Can be extended by C# plug-ins.

Volatility Framework

- https://code.google.com/p/volatility/
- Open source!
- Capable of memory analysis of Windows, Linux (Android) and MacOS systems.
- Extensible, written in Python.
- No GUI yet.

What Can Be Found in Memory?

Almost everything!

- **Malware :) (rootkits included)**
- System information
    - Hardware and software
- Processes and threads
    - Loaded DLLs
- Network sockets, URLs, IP addresses
- Open files (and pipes)
- Mutexes/Handles
- User-generated content
    - Passwords, clipboards, caches
- Encryption keys!
    - E.g. for TrueCrypt (can be automated)
- Registry hives
- Event logs
- *(Screen preview!)*

Also, you can search for system inconsistencies – hidden processes, hidden drivers, non-system handlers (interrupts handled by non-system processes).

What to Search For?

- IRP (**I/O Request Packets)**
  - Mostly focused at NTFS, DISK, FAT, TCPIP, NDIS and KBDCLASS drivers.
  - Look for a single hook IRP_MJ_DEVICE_CONTROL.
  - Use your brain and Google.
- SSDT (**System Service Dispatch Table)**
  - Used by legit systems (HIPS, malware protection) and malware.
  - Differentiating is not easy; use your brain and Google.
    - Hooking to unresolved drivers is suspicious.
    - Unsigned drivers are suspicious.
    - Only a few hooks in place can be suspicious.
- IDT (**Interrupt Descriptor Table**)
  - It allows attackers to subvert memory manager, keyboard events, system calls.
  - Not much used, any IDT hook is suspicious :)
- Hidden processes, DLLs, drivers, …
- Process injections
- Process path/user inconsistencies
  - E.g. `svchost.exe` executed from `c:\windows\sytem32\dllcache\`
  - Running as a non-standard user.
- Open sockets, network connections
- Mutexes/Handles
- URLs (URL-like strings)
  - URL-like strings can also be interesting! (e.g. `http://%d%S/config.html`)
- Anything suspicious!
  - **Malware can be digitally signed!**

Memory Injection

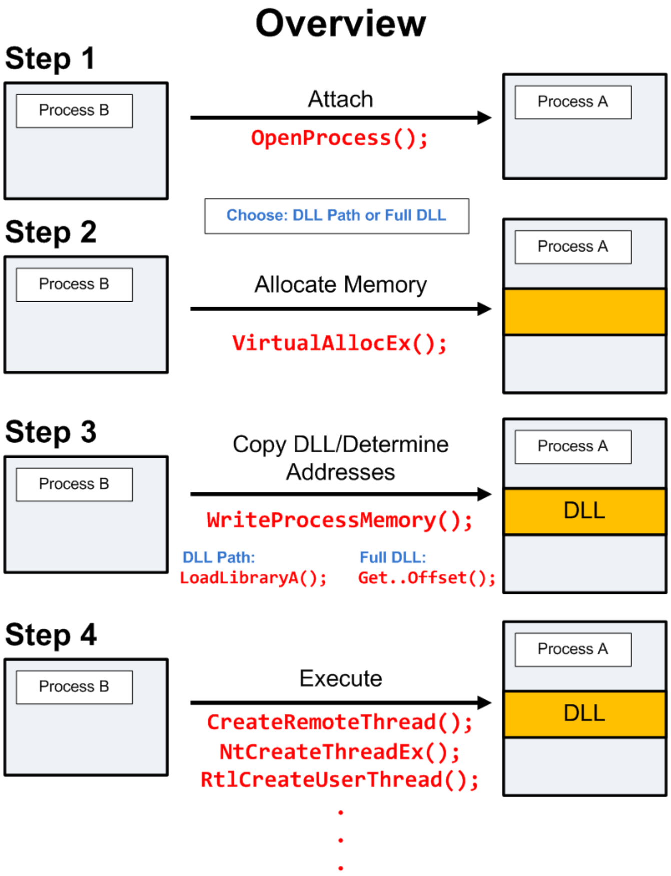
What is it and why is it so important?

A very simple definition: a mechanism of inserting dynamic library / malicious code in the process of confidence.

Why? After successful injection the malware can use all the benefits of the original process. Thus, if the malware injected the process of Internet Explorer, it can now bypass Windows Firewall and run its code in any port. With some care, malware can also spy on the original process; can re-define some of the original functions and/or event handlers.

Technical details and possible ways how the code can be injected into a running process can be found at:

1. http://resources.infosecinstitute.com/code-injection-techniques/
2. http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces
3. http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html (source of the following image)



Well-known Suspicious Mutexes

Virus/Tool Name	Mutex
Conficker	`.*-7` and `.*-99`
Sality.AA	`Op1mutx9`
Flystud.??	`Hacker.com.cn_MUTEX`
NetSky	`'D'r'o'p'p'e'd'S'k'y'N'e't'` `YY99knPY`
Sality.W	`u_joker_v3.06`
Poison Ivy	`)!VoqA.I4`
koobface	`35fsdfsdfgfd5339`

Expected Paths

Process	Expected Paths
lsass.exe	\windows\system32
services.exe	\windows\system32
csrss.exe	\windows\system32
explorer.exe	\windows
spoolsv.exe	\windows\system32
smss.exe	\windows\system32
svchost.exe	\windows\system32
iexplore.exe	\program files \program files (x86)
winlogon.exe	\windows\system32

Suspicious Imports

Scenario #1

- `GetProcAddress`
- `LoadLibrary`
  - What the binary can do with these calls?
  - What if the binary doesn't contain any other import?

Scenario #2

- `CreateToolhelp32Snapshot`
- `Process32Next`
- `Process32First`
  - What the binary/process can do with these calls?

Scenario #3

- `Ws2_32.dll / msock32.dll`
- `Wininet.dll`
- `Netapi32.dll`
  - Network-related imports; can they be present in `calc.exe`? And in `svchost.exe`?

Volatility Cheat Sheet

Source: https://blogs.sans.org/computer-forensics/files/2012/04/Memory-Forensics-Cheat-Sheet-v1_2.pdf

- `vol.py -h / vol.py plugin -h / vol.py plugin --info` (help)
- `vol.py -f image.file imageinfo` (info about the image, useful for further steps)
- `vol.py -f image.file --profile=profile plugin` (sample command line)
    - `export VOLATILITY_LOCATION=image.file`
    - `export VOLATILITY_PROFILE=WinXPSP3x86`
        - can save you some typing
- `vol.py -f image.file` *`-profile=profile`* `<plugin>`
    - `psxview` (look for hidden processes)
    - `apihooks`
    - `driverscan`
    - `ssdt / driverirp / idt`
    - `connections / connscan` (WinXP, list of open TCP connections / all TCP connections)
    - `netscan` (Win7, scan for connections and sockets)
    - `pslist / psscan` (high-level process list vs. scan for EPROCESS blocks)
    - `malfind / ldrmodules` (find injected code, dump sections / detect unlinked DLLs)
    - `hivelist` (find and list available registry hives) / `hashdump`
    - `handles / dlllist / filescan` (list of open handles / DLL files / FILE_OBJECT handles)
    - `cmdscan / consoles` (find the history of cmd.exe / console buffer)
    - `shimcache` (application compatibility info)
    - `memdump / procmemdump / procexedump`

Homework

The task is simple. Analyze the given memory image, find all irregularities there, prepare a formal report and document your findings. The more details and information you provide, the more points you get.

Please note that you have to present me not only your results, but also the tools you used in the whole process described in your report. If you use Redline for one part of the analysis and Volatility for another please document it. I have to be able to follow your investigation in order to validate the steps you took.

File: homework.zip

Hints:

- should be similar to one of the lecturing samples,
- watch for suspicious connections,
- do not forget to check execution times,
- find suspicious URLs, if there are any,
- can you find the source of the infection? And any technical details? (Google is your friend)

References and further reading

1. https://code.google.com/p/volatility/wiki/PublicMemoryImages

2. https://www.mandiant.com/blog/precalculated-string-hashes-reverse-engineering-shellcode/

3. https://github.com/iagox86/nbtool/blob/master/samples/shellcode-win32/hash.py

4. http://blog.spiderlabs.com/2013/04/basic-packers-easy-as-pie.html (simple unpacking, not UPX)

5. http://zeltser.com/remnux/ -- REMnux. All you need to do a reverse engineering.

6. http://www.deer-run.com/~hal/Detect_Malware_w_Memory_Forensics.pdf

7. http://downloads.ninjacon.net/downloads/proceedings/2011/Michael_J_Graven-Finding_Evil_in_Live_Memory.pdf

8. http://www.skullsecurity.org/blog/2013/ropasaurusrex-a-primer-on-return-oriented-programming (Practical introduction into Return Oriented Programming)