SANS FOR610 WORK NOTES

Reverse-Engineering Malware: Malware Analysis Tools and Techniques

This document is FAR from being a replacement of the official SANS materials but I made it to prepare myself for the GREM Certification with the important topics I wanted to focus on.

The page number references are from the official SANS PDF materials version FOR610_2_G01_05

Hope it helps 😊

Created by Martin Bayard in 04/2023 - http://prfalken.org

Table of Content

610.1 : Malware Analysis Fundamentals

Introduction to Malware Analysis (Page 4)

Pyramid of Complexity (Low to High) (Page 9)

- Fully automated analysis
- Static Properties Analysis
- Interactive behavior analysis
- Manual Code Reversing

Typical Report (Page 11)

- Summary of the analysis : Key takeaways
- Identification : Files, hashes, strains ...
- Characteristics : Capabilities for infection, spreading ...
- Dependencies : Files and network resources ...
- Behavioral and code findings
- Supporting figures : Logs, screenshots, strings ...
- Incident Recommendations : Indicators for detecting and eradication

Pivoting : Approach to look for associations between known attributes of the malicious program with new characteristics.

Investigation Tips

- Hide your origin (Page 19)
  - TOR : but exit nodes are well known
  - Commercial VPN : cheap and fast but exit nodes are well known
  - Custom VPN : Setup OpenVPN on a public cloud is the best solution to keep investigations undetected
  - Warning of DNS leakage : make sure DNS traffic goes through VPN
- Using public services :
  - Don't upload samples to 3rd party unless you're sure why
  - Using public tools like urlQuery or vURL can reveal the investigation

Malware Analysis Lab (Page 23)

- Virtual
  - Isolation
    - "host-only" network or "custom virtual network" (no host virtual adapter connected)
    - Avoid using "Shared Folders" or USB Device Mapping
    - Keep your host and hypervisor up to date with latest patch to avoid VM escapes
    - VM Tools can be detected easily and change the behavior of the malware
  - Restore with Snapshot is very convenient
- Physical
  - Restore with Disk Cloning Solutions : CloneZilla, FOG, dd
  - Restore with PXE booting
  - Restore with Software mimics snapshots (not 100%)

- Static Properties Analysis : PeStudio, strings, CFF Explorer, peframe, Detect It Easy ...
- Behavioral Analysis : Process Hacker, Process Monitor, RegShot, Wireshark, fakedns, INetSim ...
- Code Analysis : Ghidra, x32dbg/x64dbg, OllyDumpEx, Scylla, runsc32 or scdbg for shellcodes ...

Static Properties Analysis (Page 37)

Static Properties (Page 42)

- File and Section hashes
- Packer Identification
- Embedded Resources
- Imports and Exports
- Crypto References
- Digital Certificates
- "Interesting" strings

Tools

- pestr
- strings -a (once for ASCII, then --encoding=l for Unicode)
- peframe (REMnux) and PeStudio (Windows) both give a good overview of the PE
- Detect It Easy and Exeinfo PE both give a good overview of the PE Header (compiler, packer, entropy, hashes ...)

Behavioral Analysis Essentials (Page 51)

Tools

- Process Hacker
  - Advanced Task Manager
  - Network Tab is limited to active connections, no historical view (check TcpLogView instead if needed)
- Process Monitor
  - Records processes, registry, network, file activity
  - Use filters to remove noise (unwanted / OS related events)
- RegShot : Take a shot before detonation, a shot after detonation, and find modifications on files and registry keys
- ProcDot : Use Process Monitor output to create a graphical view
- Wireshark
  - Sniff and analyze network packets (http, dns ...)
  - Run it on REMnux with Windows Default GW defined with REMnux IP
  - Right Click -> "Follow" -> "TCP Stream" shows the complete HTTP Request
- fakedns : On REMnux, will respond to DNS queries
- httpd : On REMnux, will respond to HTTP requests
- INetSim : On REMnux, will respond to DNS, HTTP, HTTPS queries, will send fake files to malware ...

Approach by "resource starvation" : give the malware only what he needs -> if enabling DNS responses too early, we could miss tries to other domains.

Code Analysis Essentials (Page 78)

- Static Code-level analysis :
  - Disassembler gives assembly
  - Decompiler gives pseudo code.
- Dynamic code-level analysis :
  - Debugger runs the code step by step (in assembly or higher level language)
- Emulating
  - Speakeasy, capa, binee, Qiling, Vivisect
  - Useful to examine API activity
  - Can be confused by unfamiliar instructions or API calls

Speakeasy (Page 83)

- run_speakeasy.py -t malware.exe -o malware.json 2> malware.txt
- jq ".entry_points[].apis[].api_name" malware.json | more

Capa (Page 86)

- capa -vv malware.exe | more
- Show ATT&CK and MBC (Malware Behaviors Catalog) matches

x64dbg (Page 89)

- SetBPX / bpx / bp ReadFile (case sensitive API)
- Run (F9)
- Step Into (F7)
- Step Over (F8)
- Execute till Return (Ctrl-F9)
- Run to User Code (Alt-F9)

API Monitor (Page 101)

- Select the API you want to monitor
- Select the process to observe
- Check the returned values (decrypted buffer for example)

Exploring Network Interactions (Page 111)

Example with brbbot.exe

- Launch web server on REMnux : httpd start
- Create a file : echo "cexe c:\windows\notepad.exe" > /var/www/html/ads.php
- Start brbbot.exe on Windows and observe it launching notepad after sometime

INetSim (Page 119)

- Support a lot of services
- Configuration of services in /etc/inetsim/inetsim.conf
- Logs stored in /var/log/inetsim/service.log
- Files stored in /var/lib/inetsim

Fiddler (Page 123)

- Intercept traffic on client side
- Enable rules in AutoResponse
- Make sure https decryption is enabled
- Review the Inspectors -> Raw tab for details

IP Redirection (Page 126)

- Enable
  - iptables -t nat -A PREROUTING -i eth0 -j REDIRECT
  - accept-all-ips start
- Disable
  - iptables -t nat -D PREROUTING -i eth0 -j REDIRECT
  - accept-all-ips stop

610.2 : Reversing Malicious Code

Core Reversing (Page 3)

.text : Executable Code
.rdata : Read-only data
.data : Data
.reloc : Relocation data to fix up addresses in the file if it is not loaded at its preferred address

Ghidra

- Green Arrow -> Path if the condition is met
- Red Arrow -> Path if the condition is NOT met
- Blue Arrow -> Code Block ends with an unconditional JUMP
- To resize columns : Browser Field Formatter
- To see imported functions :
    - Window -> Symbol References (+ filter by Imported)
    - Focus on Access : "Call" in the right panel
- Rename variables and functions to make it more clear

Variables

- Local : Only accessible in the function that allocates it
- Global : Accessible from anywhere within the program and display with an address (DAT_XXXXXXXX)
- Static : Only accessible in the function that allocates it but not marked for reuse

Static and Global are indistinguishable in assembly.

Registers (Page 25)

- EAX : Addition, Multiplication, Return Values
- EBX/EDX : Generic Registers
- ECX : Counter
- EBP :
    - Arguments/Parameters (EBP + Value)
    - Local variables (EBP - Value)
- ESP : Last Item on the Stack
- ESI/EDI : Memory Transfer
- EIP : Address of the next instruction to execute

"A pointer is simply a variable that contains the address of some location in memory"

Addressing Memory (Pages 31-33)

- Direct : "Brackets mean fetch data at the specified address (dereference)" -> MOV EAX, [0x410230]
- Indirect : Base + (Index * Scale) + Displacement
  - [EAX] : base
  - [EBP +0x10] : base + displacement / example : access data on the stack
  - [EAX + EBX * 8] : base + (index * scale) / example : access an array with 8-byte structures
  - [EAX + EBX + 0xC] : base + index + displacement / example : access fields in a 2-dimensional array of structures

Conditional Jumps (Page 42)

- CMP is an implied SUB
- TEST in an implied AND

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE JNA	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA JNBE	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL JNGE	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE JNL	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE JNG	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG JNLE	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP JPE	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP JPO	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ JECXZ	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

Unconditional Jumps

- JMP XXX
- CALL XXX : PUSH EIP, JUMP TO XXX
- RET : POP EIP

Reversing Functions (Page 61)

Prologue (Page 67)

- Allocates space for variables
- Saves registers that will be reused in the function body
- Example :
  - PUSH EBP (save the current EBP -> saved Frame Pointer (SFP))
  - MOV EBP, ESP (save ESP in EBP to be used as unchanging reference to part of the stack because ESP may vary in the function body)
  - PUSH ECX (Allocate space for local variable, which decrement ESP by 4 with shorter space than a SUB ESP, 4)

Epilogue (Page 67)

- Cleans up the stack
- Restore registers
- Example :
  - MOV ESP, EBP (restore ESP)
  - POP EBP (restore EBP)
  - RET (POP EIP -> move the return address into EIP register to jump back to the caller)
- LEAVE opcode is equivalent to (MOV ESP, EBP + POP EBP)
- Setting up the return value is NOT part of the EPILOGUE, but we consider it is when it's surrounded by other epilogue instructions

Function Example (Page 69)

1	Prologue	PUSH EBP	Save current EBP
2		MOV EBP, ESP	Save ESP in EBP
3		SUB ESP, 0x04	Allocate space for local variable 1
4	Function Body	MOV EAX, [EBP + 0x08]	Save Parameter 1 in EAX
5		ADD EAX, [EBP + 0x0C]	Add Parameter 2 to EAX
6		ADD EAX, [EBP + 0x10]	Add Parameter 3 to EAX
7		MOV [EBP - 0x04], EAX	Save EAX in local variable 1
8		MOV EAX, [EBP - 0x04]	Set return value in EAX
9	Epilogue	MOV ESP, EBP	Restore ESP
10		POP EBP	Restore EBP
11		RET	Restore EIP

The Stack (Page 68)

- The stack is LIFO (Last In First Out) and grows toward lower addresses
- Function arguments are pushed onto the stack in reverse order (from RIGHT to LEFT)
- The role of EBP is to have access to Parameters and Variables in a fixed manner in case the function also manipulates the stack (ESP will change)

Stack Example (Page 70)

In the body of the function above, the stack will look like :

Lower Addresses	Local Variable 1	[EBP – 0x04]
	Saved EBP (also called SFP)	[EBP]
	Return Address	[EBP + 0x04]
	Parameter 1	[EBP + 0x08]
	Parameter 2	[EBP + 0x0C]
	Parameter 3	[EBP + 0x10]
Higher Addresses	(Stack values before function is called)	

Calling Conventions (Pages 72-73)

- cdecl (most common) : Arguments on the stack (right -> left), return value in EAX, caller cleans up the stack
- stdcall (WIN32 APIS) : Similar to cdecl but callee cleans up the stack
- fastcall : Arguments in registers (ECX + EDX) + stack (if needed), callee cleans up the stack (if needed)
- thiscall (Used in C++) :
    - Microsoft Compilers : ECX holds "this" pointer, callee cleans up arguments on the stack
    - GNU Compilers : "this" is pushed last onto the stack, caller cleans up the stack

In some optimization cases of stdcall, arguments are MOV to the stack (ex MOV dword ptr [ESP+local_8], 0x0), but as the callee did the clean-up (add X bytes to ESP before returning), the compiler needs to compensate by undoing the clean-up (sub X bytes to ESP)

CONTROL FLOW IN-DEPTH (Page 97)

- Loops have 5 major components
  - Control Variable : Used to determine if a loop exists
  - Loop Initialization : The value of the control variable is initialized (outside the body loop)
  - Loop Body : Code Block executed
  - Loop Update: The value of the control variable is modified
  - Stopping Condition : Used to determine if the loop should exit
- Simple Expressions evaluate only a single condition
  - if (x<4) { block of code }
- Compound Expressions evaluate multiple conditions
  if (x<4) AND (x>1) { block of code }
- AND (Page 124)
  - Invert the logic of the condition
  - Jump to end of block if true

Pseudo Code	Assembly

If ((a<4) && (a>1)) {
 	
  Code Block
}
Next Code Block
	CMP EAX, 4
	JNL END
	CMP EAX, 1
	JNG END
	BLOCK1:
	  Code Block
	END:
	  Next Code Block

- OR (Page 125)
  - Test each condition
  - Jump to code block if a condition is met
  - Negate logic of last condition and jump to end if true

Pseudo Code	Assembly

If ((a<4) || (a>10)) {
 	
  Code Block
}
Next Code Block
	CMP EAX, 4
	JNL BLOCK1
	CMP EAX, 10
	JNG END
	BLOCK1:
	  Code Block
	END:
	  Next Code Block

API Patterns in Malware (Page 135)

Resources

- FindResourceW : Determine Location of a resource
- SizeofResource : Obtain the size of a resource
- LockResource : Obtain a pointer to the resource
- CreateFileA + WriteFile : Save the resource into a file

Frequently used by droppers

Mutex/Mutant

- CreateMutexA : creates or opens a mutex object

Frequently used to detect if malware has already infected the system

Stealers

- GetKeyState / GetAsyncKeyState : Determine is a specific key has been pressed
- GetWindowText : Retrieves text from a windows's title bar
- OpenClipboard / GetClipboardData / CloseClipboard : Gather data from user's clipboard

Internet

- InternetOpen / InternetConnect : Creates and HTTP Connection
- HttpOpenRequest / HttpAddRequestHeaders : Build the HTTP Request
- HttpSendRequest : Send the HTTP Request
- InternetReadFile : Read the response to the HTTP Request

Frequently used by downloaders

64-BIT Code Analysis (Page 147)

- Registers are now 64 bits (EAX -> RAX, EBX -> RBX, EIP -> RIP …)
- 8 new general-purpose registers r8 -> r15
- 32-bit code running on 64-bit Windows use the WoW64 subsystem
- Calling convention resembles fastcall (RCX, RDX, R8, R9 (from left to right), then on the stack (from right to left))
- R9D is the lower 32 bits part of R9
- R9W is the lower 16 bits part of R9
- R9B is the lower 8 bits part of R9 (there is no equivalent of AH to get the higher 8 bits part)

610.3 : Analyzing Malicious Documents

Malicious PDF File Analysis (Page 3)

Risky Keywords

- Executing Embedded JavaScript : /JS, /JavaScript, /AcroForm, XFA
- Launching programs : /Launch, /EmbeddedFiles
- Action when file is opened : /OpenAction, /AA
- Interact with Websites : /URI, /SubmitForm
- Images : /XObject (to confirm with the /Subtype), /Xform
- Associate clickable link with an image : /Annots
- Object Stream (stream that contains other objects) : /ObjStm

- Filter FlateDecode : zlib/deflate decompression method

Tools

- pdf-parser.py (Didier Stevens) (Page 9) :
  - -a : show a summary of the file (pdf-parser.py doc.pdf -a)
  - -s : select objects with a specific keyword (pdf-parser.py doc.pdf -s /URI)
  - -k : shows the values for the given key (pdf-parser.py doc.pdf -k /URI)
  - -o : examine a specific object (pdf-parser.py doc.pdf -o 6)
  - -d : dump a specific object (pdf-parser.py doc.pdf -o 6 -d object6.jpg)
  - *-f : apply filters (FlateDecode,ASCIIHexDecode,ASCII85Decode,LZWDecode,RunLengthDecode)*
  - *-w : raw*
  - -r : shows objects that reference the specified object number (pdf-parser.py doc.pdf -r 6)
  - -O : specifies pdf-parser to look into Object Streams if any
    - pdf-parser.py doc.pdf -O -a
    - pdf-parser.py doc.pdf -O -k /URI
    - pdf-parser.py doc.pdf -O -r 39
  - ++
- Fiddler (Page 20)
  - Proxy tool to examine HTTP(S) requests, possibility to extract file that was downloaded in the stream

VBA Macros in Microsoft Office Documents (Page 36)

Formats

- Before 2007 : Object Linking Embedding 2 (OLE2)
  also called
  - Structured Storage (SS)
  - Compound File Binary Format (CFBF)
  - Composite Document File V2 (CDFV2)
- After 2007 : Office Open XML (OOXML)
  - ZIP and XML based
  - Macros are saved in a OLE2 file inside the ZIP

Tools

- zipdump.py (Didier Stevens)
  - Extract a specific file (zipdump.py doc.doc -s 5 -d > image.jpg)

- olevba (oletools from Philippe Lagadec) :
  - Analyze the doc : olevba doc.doc > doc.olevba

- oledump.py (Didier Stevens)
  - -i : Shows information (oledump.py doc.doc -i)
  - "M" = Macro present
  - "m" = No meaningful macro code
  - "!" = Anomaly (VBA Stomping, look for pcode and source code sizes)
  - xxx+yyy = size_of_pcode + size_of_compressed_source_code
  - -s a : Select all streams that have VBA code
  - -v : Decompress VBA (oledump.py doc.doc -s a -v)
  - -p : use plugin (oledump.py doc.doc -p plugin_http_heuristics)

- re-search.py (Didier Stevens)
  - Extract from input stings that match commonly used RE
  - -n str-u : matches strings enclosed within quotation
  - -n url-domain : matches URLs
  - -n all : matches all known RE

- sets.py (Didier Stevens)
  - Perform operations on lines of text
  - join "" : will join all lines into a single one

- numbers-to-string.py (Didier Stevens) (Page 57)
  - Extract decimal numbers from input into a string
  - -j : join all lines

- xor-kpa.py (Didier Stevens) (Page 69)
  - -x : XOR the 1st string with the 2nd string
  - prefix with #h# to specify it's hexadecimal (xor-kpa.py -x '#h#2B07372B185D480C222A1C3B3204' '#h#66546F')

- base64dump.py (Didier Stevens) (Page 88)
  - Find base64 strings from the input (oledump.py doc.doc -s 7 -d | base64dump.py)
  - View the first base64 string from the input as a hexadecimal viewer (oledump.py doc.doc -s 7 -d | base64dump.py -s 1 -a)
  - View the first base64 string from the input as a readable string (oledump.py doc.doc -s 7 -d | base64dump.py -s 1 -t utf16)

- translate.py (Didier Stevens) (Page 93)
  - Translate the input using the specified Python expression (base64dump.py file.ps1 -n 10 -s 2 -d | translate.py "byte ^ 35" > shellcode.bin)

- scdbgc (Page 94, 114)
  - Emulate a shellcode (scdbgc /f shellcode.bin /s -1)
  - Emulate a shellcode at offset 0x3B (scdbgc /f shellcode.bin /s -1 /foff 3B)
  - Emulate a shellcode at offset 0x3B with an handle on a document (scdbgc /f shellcode.bin /s -1 /foff 3B /fopen doc.doc)

- runsc32 (Page 116)
  - Run a shellcode at offset 0x3B with an handle on a document (runsc32 -f shellcode.bin -o 0x3B -d doc.doc -n)

- yara-rules.py (Page 95)
  - Run all the yara configured on a file (yara-rules shellcode.bin)

- 1768.py (Didier Stevens) (Page 95)
  - Extract info from a CobalStrike payload (1778.py shellcode.bin)

- vmonkey (ViperMonkey from Philippe Lagadec) :
  - vmonkey doc.doc > doc.vmonkey
  - Not always work but worth trying

- Evilclippy :
  - Remove the password on the macro (Macros are not crypted)
  - evilclippy -uu doc.doc

VBA Stomping

- Source code is removed from the file, only the p-code is present, can only work on a compatible Office version.
- Look for "!" in the output of oledump.py doc.docm -I
- oledump.py doc.docm -s A3 -v : Error unable to decompress
    - oledump.py doc.docm -s A3s -A : Shows the source code like a hexadecimal viewer, most probably 00ed
    - oledump.py doc.docm -s A3c -A : Shows the compiled code like a hexadecimal viewer, look for strings
- pcodedump.py doc.docm > doc.pcodedump : shows p-code in assembly-like language specific to VBA, not really usable
- pcode2code doc.docm : shows p-code in VBA which can be reused in a new document + the debugger to analyze it

VBA Purging

- P-code is removed from the file, source can still be extracted by oledump.py

Tips

- If OLE2 file that contains VBA macros includes streams with "SRP" -> cached compiled copy which can be examined with oledump.py and strings for example (Page 53)

- Can check for "themeFontLang" value inside the "word/settings.xml" (zipump.py doc.doc -s 9 -d | xmldump.py pretty | grep -i "themeFontLang") (Page 54)

- If the macro is referring to other parts of the document (UserForm1.CheckBox1.ControlTipText for example), look for the strings in the other streams that are related to this element (oledump.py doc.docm -i -> with UserForm1 in the name for example) -> "oledump.py doc.docm -s 5 -S" (Page 86)

Examining Malicious RTF Files (Page 99)

Tools

- rtfdump.py (Didier Stevens) (Page 102)
  - Shows information (rtfdump.py doc.doc)
  - Shows information about embedded objects (rtfdump.py doc.doc -O)
  - Extract embedded object (rtdump.py doc.doc -O -s 1 -d > doc.object), the extracted object can be an OLE2 -> use oledump.py to see inside.
  - Look for unexpected/unknown characters in the "u=" tag (Page 108)
  - Look for the group with a lot of hex characters at the deepest nesting level in the "h=" tag
  - Examine the content of a group (rtfdump.py doc.doc -s 5) and look for NOP sleds (909090…)
  - Extract the content of a group (rtfdump.py doc.doc -s 5 -H -d > doc.bin) (Page 110)

- format-bytes.py (Didier Stevens) (Page 107)
  - Parse binary data and format it according a format string supported by Python struct module.
  - Predefined format for Equation Editor (oledump.py doc.object -s 4 -d | format-bytes.py -f name=eqn1)

- xorsearch
  - Check if a file contains a shellcode by looking for well-known operations (xorsearch -W -d 3 doc.bin) (Page 111)

Shellcode

- CALL / POP (can be some JMP around) to get EIP
- Access the PEB to locate kernel32.dll in the memory of the process. A pointer to the PEB is at offset 0x30 of the TIB, which is at address contained in the FS register
  -> "MOV EAX, DWORD PTR FS:[30h]"
- Tools : scdbgc and runsc32

610.4 : In-depth Malware Analysis

Deobfuscating Malicious JavaScript (Page 3)

Keywords

- eval : Run the parameter as JavaScript
- document.write / document.body.appendChild / document.parentNode.insertBefore (Page 6) -> Add other JavaScript
- arguments.callee (ex var M1FDAB=arguments.callee.toString()) : allow a function to examine his own body -> checks if code has been modified! (Page 18)

REMnux

- js-beautify : can beautify JavaScript code
- extractscripts.py : extract JavaScript from a webpage
- SpiderMonkey
  - JavaScript interpreter from Mozilla/FF (js -f script.js)
  - JavaScript interpreter with other definitions and modified eval function (js -f /usr/share/remnux/objects.js -f script.js)
- box-js : For scripts designed to run outside browser

Windows

- Cscript (VBS and JS)
  - AMSI Trace (Page 10)
    - logman start AMSITrace -p Microsoft-Antimalware-Scan-Interface Event1 -o AMSITrace.etl -ets
    - cscript.exe script.js
    - logman stop AMSITrace -ets
    - AMSIScriptContentRetrieval > script-output.txt
  - Redefine the "eval" function to add a "WScript.Echo(XXX)" before eval(XXX) (Page 12)

Recognizing Packed Malware (Page 19)

Indicators

- Few readable strings
- Limited Imports
- High Entropy
- Section Names
- RawSize of section = 0 but VirtualSize > 0 (ie where the code will be unpacked)

Tools

- PEStudio
- Detect It Easy
- Exeinfo PE
- Bytehist (for bytes distribution : not uniform -> not packed, uniform -> packed)

Getting Started with Unpacking (Page 32)

Approach

- Disable ASLR (Modify the flag in the DllCharacteristics field of PE Header or use "setdllcharacteristics -d file.exe")
- Run and let it unpack itself
- Search for strings with Process Hacker in the running process to confirm it's unpacked
- Run Scylla.exe and attach to the process
    - Dump
    - IAT Autosearch
    - Get Imports
    - Fix Dump

While "dirty", this approach is suitable for static analysis and have an overview of the malware. However, since we didn't fix the OEP, the dumped executable won't run.

Using Debugger for Dumping (Page 45)

Approach

- Disable ASLR
- Load packed exe into x64dbg
- Find the end of the unpacker code
- Trace until OEP
    - Confirm by "Search for" -> "Current Region" -> "String References" to find some useful strings
    - Confirm by "Search for" -> "Current Region" -> "Intermodular calls" to find some useful API calls
- Dump process with "Plugins" -> "OllyDumpEx" -> "Dump Process"
    - Get RIP as OEP (because we are on the first instruction of the unpacked code)
    - Set section UPX1 with MEM_WRITE attribute (otherwise crash)
    - Dump
- Rebuilt IAT with "Plugins" -> "Scylla"
    - IAT Autosearch
    - Get Imports
    - Fix Dump (select the dump made with OllyDumpEx)

With this modified UPXed binary it's easy to find the end of the unpacking code, but in real life, it may be more complicated.

Debugging Packed Malware (Page 61)

Approach

- Disable ASLR
- Load packed exe into x64dbg and run it (F9)
- In Memory Map, look for memory segments with "E" (execute) flag in the Protection column
- Right click and "Follow in Disassembler"
- Confirm by "Search for" -> "Current Region" -> "Intermodular calls" to find some useful API calls
- Pick an interesting API and "Follow in Disassembler"
- Set a hardware breakpoint after the call to the API
- Restart the program (Ctrl-F2) and run it (F9)
- Continue debugging to get what you're looking for (decrypted configuration files …)

Analyzing Multi-Technology Malware (Page 73)

Tools

- reg_export : Useful to export a registry key which contains values that would have not have been exported properly by Regedit
- PowerShell ISE :
  - Set a breakpoint when the shellcode is ready but not yet injected
  - Save the shellcode into a file ([io.file]::WriteAllBytes('sc32.bin',$sc32))
- scdbg
  - Emulates the shellcode
  - Passing the shellcode address as parameter to the shellcode implies modifying the stack -> complicated
- runsc32
  - Runs the shellcode (runsc32 -f sc32.bin)
  - Shellcode address is passed as parameter by runsc32 (-a to disable this)

Approach

- Runs the shellcode
- Attach runsc32 process to x32dbg
- Set a breakpoint at address displayed by runsc32 (beginning of shellcode)
- Continue the process in runsc32 window
- Come back to x32dbg and see in the stack that the address was given as parameter
- Breakpoint on some interesting APIs
  - Use the Call Stack to find the code that called the API
  - Go to the code that called the API
  - Next instruction -> "Run until selection"
- Scroll and find other interesting API (VirtualAlloc for example)
- Breakpoint on VirtualAlloc -> Ctrl-F9 -> F8
- Right-click on EAX -> "Follow in dump" -> Dump1
- Continue the process and repeat for each VirtualAlloc call, check every Dump window for interesting stuff
- When a dump window contains a MZ header, it's interesting to save it to disk
  - Right-click -> "Follow in memory map"
  - Dump memory to File

In this case the shellcode doesn't contain any GetEIP pattern but expects his own address to be passed as argument by the PowerShell script.
We could continue the analysis by analysing the MZ file we dumped from the shellcode memory.

Code Injection and API Hooking (Page 125)

Code Injection

API Examples :

- CreateToolsHelp32Snapshot or EnumProcesses or CreateProcess
- OpenProcess
- VirtualAllocEx
- WriteProcessMemory
- CreateRemoteThread

Malware may use lower level calls to remain undetected (names start with Nt/Zw/Rtl).

Native API Examples :

- NtQuerySystemInformation
- NtOpenProcess or ZwOpenProcess
- NtAllocateVirtualMemory or ZwAllocateVirtualMemory
- NtWriteVirtualMemory or ZwWriteVirtualMemory
- NtCreateThreadEx or ZwCreateThreadEx

Methodology :

- Open the file in Ghidra
- Look for CreateRemoteThread in Windows -> Symbol References
- Look around for other API calls related to Code Injection :
    - In Windows -> Function Call Trees
    - In Windows -> Function Call Graph
- Note that VirtualAllocEx takes 0x40 as parameter (PAGE_EXECUTE_READWRITE)

DLL Injection (Page 142)

The goal is to execute LoadLibrary as a thread inside the victim process (to load a malicious DLL)

- Open the victim process (OpenProcess)
- Get some memory in the victim process (VirtualAllocEx)
- Write the DLL name in the previously allocated memory of the victim process (WriteProcessMemory)
- Call GetModuleHandle on kernel32.dll and GetProcAddress on LoadLibrary to locate the address of the LoadLibrary function
- Call CreateRemoteThread asking the victim process to run the LoadLibrary function in a new thread with the DLL name as parameter

Hooking (Page 143)

Inline hooks : patch the beginning of targeted functions to jump into the rootkit

Methodology :

- ReadProcessMemory : Read the 1st few bytes of the targeted function so they can be backed up for future use
- VirtualProtect : If needed to make the region writable
- WriteProcessMemory : Overwrite the start of the targeted function with opcodes to jump into the rootkit
    - JMP (0xE9)
    - PUSH (0x68) / RET (0xC3) : less visible than a classic jump

610.5 : Examining Self-Defending Malware

Debugger Detection and Data Protection (Page 3)

Debugger Detection

Well known detection techniques

- IsDebuggerPresent : 0 means no debugger
- CheckRemoteDebuggerPresent
- NtQueryInformationProcess (Example with Debugger port on Page 39) / ZwQueryInformationProcess
- OutputDebugString (valid value is being debugged)
- Check BeingDebugged field in PEB (Page 12)
    - MOV EAX, FS:[30h] ; Get address of the PEB
    - MOV EAX, [EAX+2] ; Get field value
    - TEST EAX, EAX   ; Test field value
- Check if NtGlobalFlag field in PEB is 0x70 (Page 13)
  FLG_HEAP_ENABLE_TAIL_CHECK (0x10) + FLG_HEAP_ENABLE_FREE_CHECK (0x20) + FLG_HEAP_VALIDATE_PARAMETERS (0X40)
    - MOV EAX, DWORD PTR FS:[30]
    - TEST BYTE PTR DS:[EAX+68], 70
- GetTickCount (Check if execution is too slow -> debugged)
- GetLocalTime, GetSystemTime
- RDTSC (Read Time-Stamp Counter) (Page 14)

Bypass

- Patch the return value manually
- Patch the program itself manually (Page 10)
- Easier way : Use ScyllaHide Plugin and check the 1st column

Data Protection

XOR

- Looking for a specific pattern like "http:" (Page 17) : xorsearch -i -s getdown.exe http:
- Looking for English words (Page 18) : brxor.py hubert.dll
- Using multiple transformation types and get ranked results : bbrack -l 1 hubert.dll

Stack Strings

- In Ghidra we can define a shortcut to convert to Char via the Keybindings
- Extract Stack Strings (Page 24) : strdeobg.pl 9.exe
- Extract Stack Strings and other types (Page 25) :
    - floss 9.exe
    - floss 9.exe --no-static-strings
    - -x to see address of the obfuscation method detected by Floss

Discovered strings can indicate risky API calls that we can breakpoint on (example RtlDecompressBuffer (Page 30))

Unpacking Process Hollowing (Page 43)

Pattern (Page 52)

- CreateProcess (or variations) in suspended mode
- NtUnmapViewOfSection (or ZwUnmapViewOfSection) to deallocate virtual memory of the process
- VirtualAllocEx (or variations) to allocate memory for new code
- WriteProcessMemory (or variations) to write the new code
- GetThreadContext
- SetThreadContext
- ResumeThread to awake the suspended process and run the new code

Using capa -vv, find an indicator of process hollowing -> "Create Process::Create Suspended Process"

Using Ghidra

- Confirm that dwCreationFlags=0x4 when CreateProcessA is called (Page 49)
- Look around of other typical API calls related to process hollowing

Example (Page 54)

- Disable ASLR / Load in x32dbg
- Breakpoint on WriteProcessMemory
- Follow the lpBuffer parameter in a Dump Window
- Follow the Dump in Memory Map
- Dump memory to File
- Examine dumped file with PEStudio

Detecting the Analysis Toolkit (Page 61)

Sandbox and LAB detection

- Hardware characteristics
  - MAC addresses
  - Video Controllers
  - Numbers of CPU cores
  - Large disk
  - Mouse Moving (via Hook)
- Installed programs
  - Registry Keys
  - Files
  - Processes
- File Path assigned to malware detonation
- Username
- System
  - Name
  - IP address scheme
  - Uptime
  - Empty clipboard

Example with vbprop.exe (Page 65)

- Load in Ghidra
- Search for calls to SetWindowsHookExA
- Installs a hook on WH_MOUSE_LL (0xe) to a function of its own process (Page 68)
- Checks for WM_MOUSEMOVE (0x200) but does nothing (Calls CallNextHookEx)
- Checks for WM_LBUTTONDOWN (0x201) but does nothing (Calls CallNextHookEx)
- Checks for WM_LBUTTONUP (0x202) then calls UnhookWindowsHookEx then malware is triggered (Page 70)

Example with raas.exe (Page 72)

- Disable ASLR / Load in x32dbg
- Close to Entry Point we see that process exits if 402BD6 returns != 0
- Step in 402BD6 to see a lot of checks :
    - BlockInput to prevent analyst to work -> patch the value from 1 to 0 (Page 75)
    - GetModuleHandleW on avghookx.dll (Page 76)
    - FindWindow on OLLYDBG and others debuggers (Page 77)
    - Reads from 7FFE02D4 : if != 0 -> Kernel-mode debugger is present (Page 78)
    - IsDebuggerPresent (Page 79)
    - GetModuleHandleW to find unwanted modules to be loaded (Page 80)
    - CreateToolhelp32Snapshot, Process32FirstW, Process32NextW to find unwanted process (Page 83)
    - RegOpenKeyExW, RegOpenValue to find the type of the hard disk (Vmware disk device…)
- We note that all the checks are within 402BD6, to bypass : (Page 89)
    - Modify EAX after the call
    - Patch the file

Handling Misdirection Techniques (Page 93)

SEH : Structured Exception Handling (Page 97)

SEH Chain

- Linked list of SEH records
- Stored on the stack
- The address first SEH record of the chain is located at the beginning of the TIB structure -> FS:[0] (For 64-bit program the TIB is pointed by the GS register)
- SEH Tab in x32dbg shows the SEH Chain, also visible by adding a Watch on "fs:[0]"

SEH record

- 32-bit programs : frame-based mechanisms
    - The _EXCEPTION_REGISTRATION structure has 2 elements :
        - Pointer to the next (previously defined) SEH record
        - Pointer to the exception-handling mechanism
- 64-bit programs : table-based approach

The exception handler may be used as a nonstandard branching mechanism. When an exception happens, the flow of execution continues within the exception handler.

Example with want.exe (PECompact) (Page 100)

- Load in x32dbg / Disable Exceptions Catching
- A new SEH record is pushed on the stack and an exception is raised to continue execution of code within the handler
- Breakpoint on the new SEH Handler and run
- To get close to the OEP, we look when the packer will remove the SEH record from the stack (or reuse this space)
    - Set a hardware-based breakpoint on the top of the stack (which contains currently the address of the handler)
    - Run a few times until it breaks out of ntdll.dll AND that we have some strings visible in the region
- Once we break on a JMP EAX, step Over (F8) and see what looks like a function prologue
- Confirm by "Search for" -> "Current Region" -> "String References" to find some useful strings
- Dump the process with OllyDumpEx (Click Get EIP as OEP)
- Rebuilds IAT with Scylla (IAT Autosearch + Get Imports + Fix Dump)
- Load the dumped executable into PEStudio and see strings / imports

TLS Callbacks (Page 114)

Description

- Designed to store different values for a global variable for each thread
- Run before code at entry point
- Visible in PEStudio / Ghidra (in Symbol Tree -> Exports)
- x32dbg usually breaks on TLS callbacks if there are some (check the box)

Example with lansrv.exe (Page 116)

- Disable ASLR / Load in x32dbg / Disable Exceptions Catching
- The TLS function contains a XOR loop for decryption
- Set a hardware breakpoint after the loop (after because the loop will overwrite the values, including the CC) (Page 121)
- We now see a IsDebuggerPresent call that will store 0x0B at 401015 if TRUE, or 0x00 if FALSE (Page 123)
- Continue tracing until after the entry point and set another breakpoint after the visible loop (Page 125)
- Malware assigns FS register value to GS in order to manipulate SEH via GS:[0] sneakily (instead of FS:[0]) (Page 127)
- Malware modifies the value of the 1st SEH record instead of creating a new one (Page 134)
- Malware sleep a bit to complicate analysis, and make a division by the variable defined in the TLS (at 401015) (Page 138)
    - If debugger is not detected, value is 0x00, exception is raised, malicious SEH handler is called
    - If debugger is detected, value is 0x0B, exception is NOT raised, other code branch is taken, crashes or exits

Unpacking by Anticipating Actions (Page 143)

Example with yep.exe (Trickbot) (Page 145)

As there few imports, we expect yep.exe to call LoadLibraryA runtime.

- Load in x32dbg / Disable Exceptions Catching
- Set a breakpoint on LoadLibraryA
- Run until specimen loads msvcrt.dll (which was not in the imports)
- We should be close to OEP, check with xAnalyzer -> Analyze Function and see that functions address are being called from the stack (Page 151)
- Notice VirtualProtect, set a breakpoint on VirtualProtect and continue running (Page 153)
- Once reached, follow the 1st parameter in a dump Window, we see a MZ structure at 0x400000, but not sure unpacking is finished at this point (Page 154)
- Continue debugging until we break on VirtualProtect with 0x20 as parameter (PAGE_EXECUTE_READ) on region 0x401000
- Follow the region in a dump window and set a Breakpoint -> Memory, Execute -> Singleshoot (after the VirtualProtect is executed) (Page 159)
- Once reached, it's good time to dump, Follow 0x401000 in memory map, the segment will start at 0x400000, dump memory to file (Page 161)
- Use pe_unmapper on the dumped file to rebase it and other tweaks (Page 162)
- Use Scylla to rebuild imports and fix the dump previously fixed by pe_unmapper (Page 163)
- Load the dumped / fixed executable into PEStudio and see strings / imports

Note that some malwares can check if we have a breakpoint on LoadLibrary by checking if the address starts with 0xCC (int3), we can use hardware breakpoint (using registers) instead to avoid being detected.