Nandy Cyber Success

Cheat Sheet for Analyzing Malicious Software

This cheat sheet offers practical guidance for malware analysis and reverse-engineering, detailing the steps for conducting both behavioral and code-level examinations of malicious software.

Overview of the Malware Analysis Process

1. Perform initial assessment using automated sandbox analysis tools.
2. Set up a controlled, isolated environment for examining malware.
3. Inspect static properties and metadata to triage and develop early hypotheses.
4. Emulate code execution to uncover malicious functions and inform next steps.
5. Conduct behavioral analysis to observe specimen interactions with the environment.
6. Perform static code analysis using disassemblers and decompilers.
7. Conduct dynamic code analysis to examine complex behaviors.
8. Unpack the specimen, if necessary.
9. Repeat steps 4–8 as needed (order flexible) until analysis goals are met.
10. Supplement analysis with memory forensics and threat intelligence.
11. Document findings, save analysis artifacts, and reset the lab for future use.

Behavioral Analysis

- Prepare to revert systems to a clean state using virtualization snapshots, Clonezilla, dd, FOG, PXE boot, or similar methods.
- Monitor local system behavior (using Process Explorer, Process Monitor, ProcDOT, Noriben).
- Identify significant local system changes (w/RegShot&Autoruns).
- Monitor network traffic interactions (Wireshark, Fiddler).
- Redirect network traffic (via fakedns, accept-all-ips).
- Emulate services requested by malware (with INetSim or actual services) and reinfect as needed.
- Adapt the runtime environment to accommodate additional local or network resources required.

x64dbg/x32dbg for Dynamic Code Analysis

Run The Code F9
Step into/over instruction F7/F8
Execute until selected instruction F4
Execute until the next return ..... Ctrl+F9
Show previous/next executed instruction ..... -/+
Return to previous view ..... *
Go to specific expression ..... Ctrl+g
Insert comment/label ..... ;/:
Show current function as a graph ..... g
Find specific pattern ..... Ctrl+b

Set Software Breakpoint On Specific Instruction
..... Select Instruction » F2

Set Software Breakpoint On API
..... Command » SetBPX API Name

Highlight All Occurences Of The Keyword In Disassembler ..... h » Click On Keyword

Assemble Instruction In Place Of Selected One
..... Instruction » Spacebar

Edit Data In Memory Or Instruction Opcode
..... Select Data » Ctrl + e

Extract API call references
..... Right-click in disassembler » Search for » Current module » Intermodular calls

Unpacking Malicious Code

Determine whether the specimen is packed by using tools such as Detect It Easy, Exeinfo PE, Bytehist, or peframe. To quickly unpack the specimen, infect a lab system and extract it from memory using Scylla. For a more precise approach, identify the Original Entry Point (OEP) using a debugger, then perform a dump with OllyDumpEx.

To locate the OEP, anticipate conditions signaling the final stages of the unpacker, setting appropriate breakpoints. Additionally, placing memory breakpoints on the stack at the unpacker's startup can help capture cleanup routines. Breakpoints on APIs like LoadLibrary and VirtualAlloc can guide you closer to the OEP, while intercepting process injection can be accomplished by breakpointing functions such as VirtualAllocEx and WriteProcessMemory.

If a clean memory dump isn't possible, dynamically analyze the packed specimen during execution. Finally, rebuild the dumped file's imports and structure using tools such as Scylla, Imports Fixer, or pe_unmapper.

Ghidra for Static Code Analysis

Go to specific destination ..... g
Show references to instruction ..... Ctrl+Shift+f
Insert a comment ..... ;
Follow jump or call ..... Enter
Return to previous location ..... Alt+Left
Go to next location ..... Alt+Right
Undo ..... Ctrl+z
Define data type ..... t
Add a bookmark ..... Ctrl+d
Text search ..... Ctrl+Shift+e
Add or edit a label l
Disassemble values ..... d

Bypassing Other Analysis Defenses

Statically decode obfuscated strings with tools such as FLOSS, xorsearch, or Balbuzard. To decode data dynamically, set debugger breakpoints immediately after decoding routines and examine the results. Conceal debugging activity, especially when using x64dbg/x32dbg, with plugins like ScyllaHide.

Use a debugger to locate and disable anti-analysis mechanisms by identifying and patching defensive code. Be alert for complex jumps (such as TLS, SEH, RET, and CALL instructions) when stepping through code in a debugger.

For analyzing shellcode, leverage specialized tools like scdbg and runsc.

Additionally, disable ASLR using utilities like setdllcharacteristics or CFF Explorer to simplify the analysis process.

Scan Here
for more great resources

www.nandycybersuccess.com

Nandy Cyber Success

Cheat Sheet for Analyzing Malicious Software

Cheat sheet for reversing malicious Windows executables via static and dynamic code analysis.

Overview of the Code Analysis Process

1. Inspect static properties of the Windows executable for initial triage.
2. Identify suspicious or malicious capabilities through strings and API calls.
3. Conduct automated and manual behavioral analysis for additional insights.
4. Emulate execution to uncover behavior and guide further analysis.
5. Statically analyze code linked to suspicious strings and APIs with a disassembler or decompiler.
6. Perform dynamic analysis with a debugger to observe runtime use of risky APIs and strings.
7. Unpack the executable and related artifacts if necessary.
8. Annotate code by adding comments, labels, and renaming functions or variables as understanding deepens.
9. Progressively analyze code referencing previously examined sections.
10. Repeat steps 5–9 as needed, adjusting sequence, until analysis goals are achieved.

Some Risky Windows API Calls

Code injection:
CreateRemoteThread, OpenProcess, VirtualAllocEx, WriteProcessMemory, EnumProcesses

Dynamic DLL loading:
LoadLibrary, GetProcAddress

Memory scraping:
CreateToolhelp32Snapshot, OpenProcess, ReadProcessMemory, EnumProcesses

Data stealing:
GetClipboardData, GetWindowText

Keylogging:
GetAsyncKeyState, SetWindowsHookEx

Embedded resources:
FindResource, LockResource

Unpacking/self-injection:
VirtualAlloc, VirtualProtect

Query artifacts:
CreateMutex, CreateFile, FindWindow, GetModuleHandle, RegOpenKeyEx

Execute a program:
WinExec, ShellExecute, CreateProcess

Web interactions:
InternetOpen, HttpOpenRequest, HttpSendRequest, InternetReadFile

Common 32-Bit Registers and Uses

EAX	Addition, multiplication, function results
ECX	Counter; used by LOOP and others
EBP	Baseline/frame pointer for referencing function arguments (EBP+value) and local variables (EBP-value)
ESP	Points to the current "top" of the stack; changes via PUSH, POP, and others
EIP	Instruction pointer; points to the next instruction; shellcode gets it via call/pop
EFLAGS	Contains fl ags that store outcomes of computations (e.g., Zero and Carry flags)
FS	F segment register; FS[0] points to SEH chain, FS[0x30] points to the PEB.

Common 32-Bit Registers and Uses

EAX >> RAX, ECX >> RCX,
EBX >> RBX, ESP >> RSP, EIP >> RIP

Additional 64-bit registers are R8-R15. RSP is often used to access stack arguments and local variables, instead of EBP.

|||||||||||||||||||||||||||||||||||||||||||||||||| R8 (64 bits)
_____|||||||||||||||||||||||||||||||||||||||| R8D (32 bits)
_____ ||||||||||||||||||||||||||||||||| R8W (16 bits)
_____ |||||||| R8B (8 bits)

Passing Parameters to Functions

JA / JG	Jump if above/jump if greater.
JB / JL	Jump if below/jump if less.
JE / JZ	Jump if equal; same as jump if zero.
JNE / JNZ	Jump if not equal; same as jump if not zero.
JGE / JNL	Jump if greater or equal; same as jump if not less.

Additional Code Analysis Tips

Be patient yet persistent, starting your analysis with small, manageable sections of code and gradually expanding from there.

If static analysis proves overly challenging, consider employing dynamic code analysis through debugging.

Pay close attention to jumps and calls, as they reveal how execution flows between significant blocks of code. If code-level analysis becomes too time-consuming, evaluate whether behavioral or memory analysis might be more effective in achieving your goals.

Additionally, when examining API calls, ensure familiarity with both official API names and their corresponding native APIs, such as Nt, Zw, and Rtl.

Common x86 Assembly Instructions

mov EAX, 0xB8	Put the value 0xB8 in EAX.
push EAX	Put EAX contents on the stack.
pop EAX	Remove contents from top of stack and put them in EAX .
lea EAX, [EBP-4]	Put the address of variable EBP-4 in EAX.
call EAX	Call function whose address resides in the EAX register.
add esp, 8	Increase ESP by 8 to shrink stack by 2x 4-byte arguments.
sub esp, 0x54	Shift ESP by 0x54 to make room for local variable(s).
xor EAX, EAX	Set EAX contents to zero.
test EAX, EAX	Check EAX contains zero, set appropriate EFLAGS bits.
cmp EAX, 0xB8	Compare EAX to 0xB8, set the appropriate EFLAGS bits.

Passing Parameters to Functions

arg0	[EBP+8] on 32-bit, RCX on 64-bit
arg1	[EBP+0xC] on 32-bit, RDX on 64-bit
arg2	[EBP+0x10] on 32-bit, R8 on 64-bit
arg3	[EBP+0x14] on 32-bit, R9 on 64-bit

Scan Here
for more great resources

www.nandycybersuccess.com

Nandy Cyber Success

Cheat Sheet for Analyzing Malicious Documents

This cheat sheet outlines tips and tools for analyzing malicious documents, such as Microsoft Offi ce, RTF, and PDF files.

General Approach to Document Analysis

1. Inspect the document carefully for anomalies, such as risky tags, scripts, or embedded artifacts.
2. Identify embedded code, including shellcode, macros, JavaScript, or other suspicious objects.
3. Extract suspicious code or objects from the document.
4. Deobfuscate and analyze macros, JavaScript, or other embedded scripts as necessary.
5. Emulate, disassemble, or debug extracted shellcode when applicable.
6. Determine the next steps in the infection chain.

Microsoft Office Format Notes

- Binary Microsoft Office documents (.doc, .xls, etc.) utilize the OLE2 (Structured Storage) file format.
- OLE2 documents may contain SRP streams, which sometimes store cached versions of previous VBA macro code.
- OOXML Microsoft Office files (.docx, .xlsm, etc.) are compressed ZIP archives.
- VBA macros within OOXML files reside in an OLE2 binary file embedded inside the ZIP archive.
- Excel files support XLM macros, which are embedded directly as formulas within worksheets, without requiring an OLE2 binary container.
- RTF documents do not support macros but can include malicious embedded objects or files.

Additional Document Analysis Tools

xorsearch -W -d 3 file.bin

Locate shellcode patterns inside the binary file file.bin

scdbg /f *file.bin*

Emulate execution of shellcode in file.bin. Use "/off"" to specify offset.

runsc32 -f *file.bin* -n

Execute shellcode in file file.bin to observe behavior in an isolated lab.

base64dump.py *file.txt*

List Base64-encoded strings present in file file.txt.

numbers-to-string.py file

Convert numbers that represent charactersin file to a string.

Risky PDF Keywords

/OpenAction and /AA specify the script or action to run automatically.

/JavaScript, /JS, / AcroForm, and /XFA can specify JavaScript to run.

/URI accesses a URL, perhaps for phishing.

/SubmitForm and /GoToR can send data to URL.

/ObjStm can hide objects inside an object stream.

/XObject can embed an image for phishing.

Be mindful of obfuscation with hex codes, such as /JavaScript vs./J#61vaScript.

Useful PDF File Analysis Commands

pdfid.py *file.pdf* -n

Display risky keywords in file file.pdf

pdf-parser.py *file.pdf* -a

Show stats about keywords. Add "-O" to include object streams.

pdf-parser.py *file.pdf* -o id

Display contents of object id. Add "-d" to dump object's stream.

pdf-parser.py *file.pdf* -r id

Display objects that reference object id

qpdf --password=pass --decrypt infile.pdf outfile.pdf

Decrypt infile.pdf using password pass to create outfile.pdf

Useful MS Offi ce File Analysis Commands

zipdump.py *file.pptx*

Examine contents of OOXML file file.pptx.

zipdump.py *file.pptx -s 3 -d*

Extract file with index 3 from file.pptx to STDOUT.

olevba.py *file.xlsm*

Locate and extract macros from file.xlsm.

oledump.py *file.xls -s 3 -v*

Extract VBA source code from stream 3 in file.xls.

xmldump.py pretty

Format XML file supplied via STDIN for easier analysis.

rtfobj.py *file.rtf*

Extract objects embedded into RTF file.rtf.

rtfdump.py *file.rtf*

List groups and structure of RTF fi le file.rtf.

rtfdump.py *file.rtf -0*

Examine objects in RTF file file.rtf.

oledump.py *file.xls* -p plugin_http_heuristics

Find obfuscated URLs in file.xls macros.

vmonkey *file.doc*

Emulate the execution of macros in file.doc.

evilclippy -uu *file.ppt*

Remove the password prompt from macros

msoffcrypto-tool *infile.docm outfile.docm -p*

Decrypt outfile.docm using specified password to creat outfile.docm

pcodedmp *file.doc*

Disassemble VBA-stomped p-code macro for file.doc

pcode2code *file.doc*

Decompile VBA-stomped p-code macro from file.doc

rtfdump.py *file.rtf* *file.doc*

Extract hex contents from group in RTF file file.rtf.

Additional Document Analysis Tools

Deobfuscate JavaScript extracted from documents using tools like SpiderMonkey, cscript, or box-js. Use Microsoft Office's built-in debugger within an isolated lab environment to deobfuscate macros.

Monitor macro execution in Microsoft Office using AMSIScriptContentRetrieval.ps1 in an isolated lab. Certain automated sandboxes can analyze malicious document behaviors. The REMnux distribution includes many freely available tools for document analysis mentioned above.

Scan Here

for more great resources

www.nandycybersuccess.com