SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE

REMnux®

Malware Analysis TIPS AND TRICKS

DFPS_FOR610_v1.4_0924

Uncovering the capabilities of malicious software allows security professionals to respond to incidents, fortify defenses, and derive threat intelligence. The malware analysis tips and tricks outlined in this poster act as a starting point and a reminder for the individuals looking to reverse-engineer and otherwise examine suspicious files such as compiled executables and potentially malicious documents.

What threat does the malicious or suspicious artifact pose? What do its mechanics reveal about the adversary's goals and capabilities? How effective are the company's security controls against such infections? What security measures can strengthen the infrastructure from future attacks of this nature? Malware analysis helps answer such questions critical to an organization's ability to handle malware threats and related incidents.

This poster brings together malware analysis resources related to:

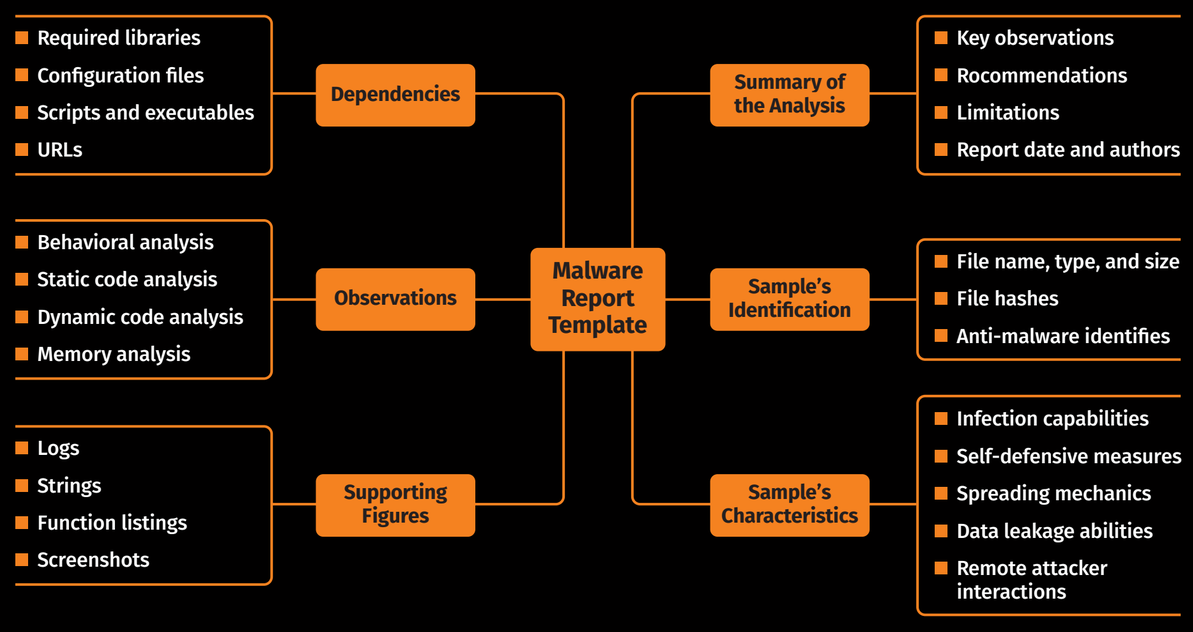
- **The overall process to examining malicious software in a controlled lab environment**
- **Using the REMnux® toolkit for analyzing malicious software using Linux-based tools**
- **Taking a closer look at malicious software by reversing it at the code level**
- **Analyzing malicious documents, including Microsoft Office and PDF files**

To learn more about this topic, consider the following SANS courses:

- FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques (sans.org/for610)
- FOR710: Reverse-Engineering Malware: Advanced Code Analysis (sans.org/for710)

For additional learning resources, follow along the practical malware analysis videos from SANS authors and instructors, available for free at for610.com/start-malware-analysis.

What to Include in Your Malware Analysis Report?

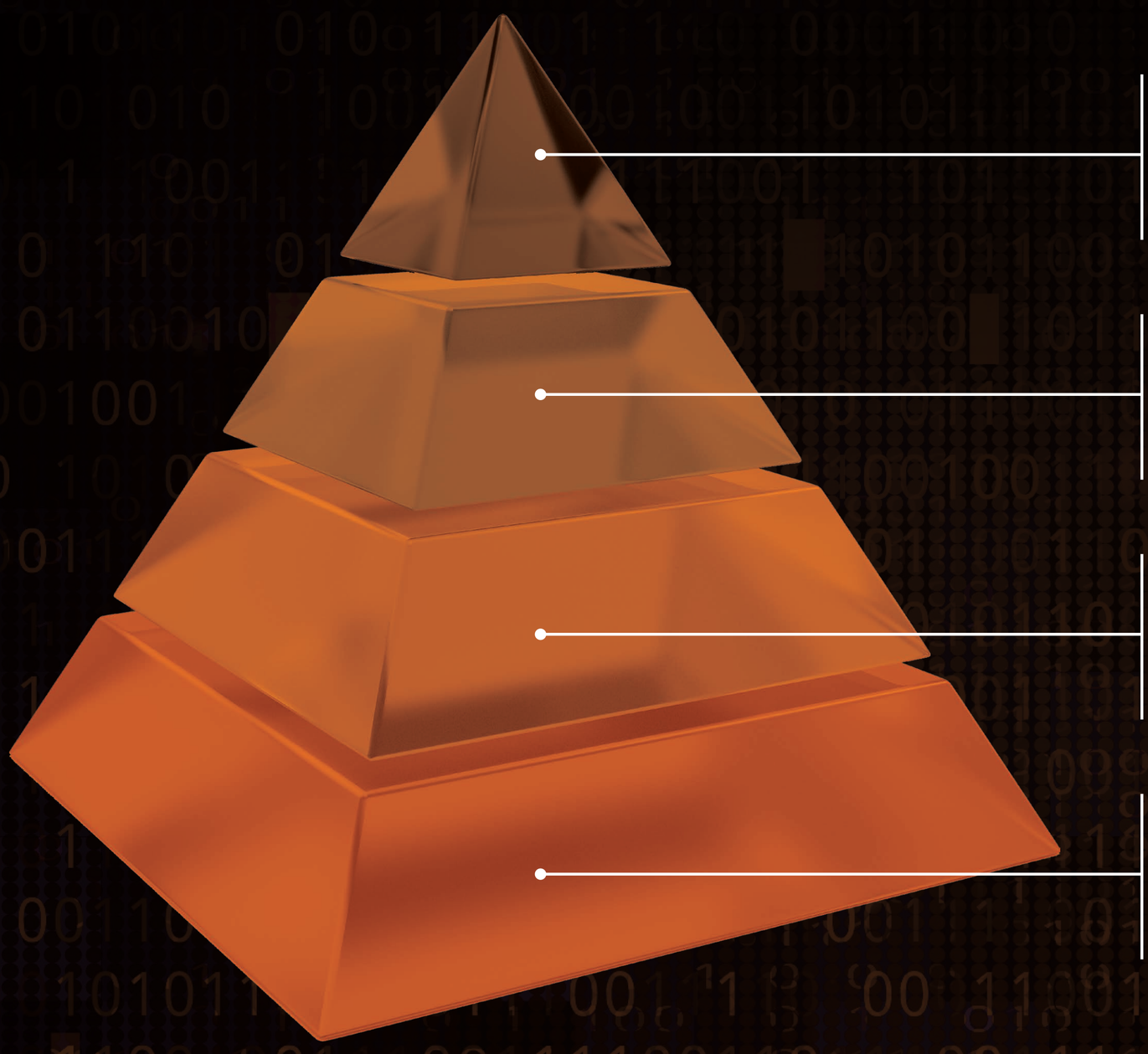


A typical malware analysis report covers the following areas:

- **Summary of the analysis:** Key takeaways the reader should get from the report regarding the specimen's nature, origin, capabilities, and other relevant characteristics
- **Identification:** The type of the file, its name, size, hashes (such as SHA256 and imphash), malware names (if known), current anti-virus detection capabilities
- **Characteristics:** The specimen's capabilities for infecting files, self-preservation, spreading, leaking data, interacting with the attacker, and so on; for a good reference of what characteristics you may need, take a look at the MAEC Malware Capabilities project or the alternative effort Malware Behavior Catalog (MBC)
- **Dependencies:** Files and network resources related to the specimen's functionality, such as supported OS versions and required initialization files, custom DLLs, executables, URLs, and scripts
- **Behavioral and code analysis findings:** Overview of the analyst's behavioral, as well as static and dynamic code analysis observations
- **Supporting figures:** Logs, screenshots, string excerpts, function listings, and other exhibits that support the investigators analysis
- **Incident recommendations:** Indicators for detecting the specimen on other systems and networks (a.k.a. indicators of compromise or IOCs), and possible for eradication steps

For downloadable malware analysis report templates, see for610.com/report-mindmap and for610.com/report-template.

Stages of Malware Analysis



Manual Code Reversing

Reverse-engineering the code that comprises the specimen can add valuable insights to the findings available after completing interactive behavior analysis. Some characteristics of the specimen are simply impractical to examine without examining the code. Code-level analysis often involves unpacking the specimen, deciding any data the malware author may have concealed, understanding the capabilities that didn't exhibit themselves during behavior analysis.

Interactive Behavior Analysis

Interactive behavior analysis involves examining how sample runs in the lab that's under the analyst's full control to go beyond what a fully automated approach might produce. This stage involves examining registry, filesystem, process, network, and memory activities. It is especially fruitful when the researcher interacts with the malicious program, rather than passively observing the specimen.

Static Properties Analysis

Analysts might proceed with examining the malware specimen by looking at its static properties, which are sometimes called metadata. This process entails examining the strings embedded into the file, its overall structure, and header data, without actually running the malicious program. This stage helps the analyst decide what aspects of the specimen, if any, are worth examining more closely.

Fully Automated Analysis

The easiest way to begin learning about a malware specimen is to examine it using fully automated tools. Sometimes called analysis sandboxes, these they're designed to assess what the specimen might do if it ran on a system. They might not provide as much insight as a human analyst would. However, they can handle vast amounts of malware, allowing the analyst to focus on the specimens that truly require her attention.

The process of analyzing malicious software involves several stages, which can be listed in the order of increasing complexity. Though it's convenient to group malware analysis tasks into discrete stages, the tasks are often intertwined, with the insights gathered in one stage informing efforts conducted in another. The pyramid diagram above, based in part on the experiences of security professional Alissa Torres, presents one such ranking.

SANS DFIR Cheat Sheet for Analyzing Malicious Software

This cheat sheet presents tips for analyzing and reverse-engineering malware. It outlines the steps for performing behavioral and code-level analysis of malicious software.

Overview of the Malware Analysis Process

1. Use automated analysis sandbox tools for an initial assessment of the suspicious file.
2. Set up a controlled, isolated laboratory in which to examine the malware specimen.
3. Examine static properties and meta-data of the specimen for triage and early theories.
4. Emulate code execution to identify malicious capabilities and contemplate next steps.
5. Perform behavioral analysis to examine the specimen's interactions with its environment.
6. Analyze relevant aspects of the code statically with a disassembler and decompiler.
7. Perform dynamic code analysis to understand the more difficult aspects of the code.
8. If necessary, unpack the specimen.
9. Repeat steps 4-8 above as necessary (the order may vary) until analysis objectives are met.
10. Augment your analysis using other methods, such as memory forensics and threat intel.
11. Document findings, save analysis artifacts and clean up the laboratory for future analysis.

Behavioral Analysis

Be ready to revert to good state via virtualization snapshots, Clonezilla, dd, FOG, PXE booting, etc.

Monitor local interactions (Process Explorer, Process Monitor, ProcDOT, Noriben).

Detect major local changes (RegShot, Autoruns).

Monitor network interactions (Wireshark, Fiddler).

Redirect network traffic (fakedns, accept-all-ips).

Activate services (INetSim or actual services) requested by malware and reinfect the system.

Adjust the runtime environment for the specimen as it requests additional local or network resources.

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**

Authored by Lenny Zeltser, who is the CISO at **Axonius** and Faculty Fellow at **SANS Institute**. You can find him at **twitter.com/lennyzeltser** and **zeltser.com**. Download this and other Lenny's security cheat sheets from **zeltser.com/cheat-sheets**.

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	**Go to Command prompt » SetBPX *API Name***
Highlight all occurrences of the keyword in disassembler	**h » Click on keyword**
Assemble instruction in place of selected one	**Select instruction » Spacebar**
Edit data in memory or instruction opcode	**Select data or instruction » 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 Detect It Easy, Exeinfo PE, Bytehist, peframe, etc.

To try unpacking the specimen quickly, infect the lab system and dump from memory using Scylla.

For more precision, find the Original Entry Point (OEP) in a debugger and dump with OllyDumpEx.

To find the OEP, anticipate the condition close to the end of the unpacker and set the breakpoint.

Try setting a memory breakpoint on the stack in the unpacker's beginning to catch it during cleanup.

To get closer to the OEP, set breakpoints on APIs such as LoadLibrary, VirtualAlloc, etc.

To intercept process injection set breakpoints on VirtualAllocEx, WriteProcessMemory, etc.

If cannot dump cleanly, examine the packed specimen via dynamic code analysis while it runs.

Rebuild imports and other aspects of the dumped file using Scylla, Imports Fixer, and pe_unmapper.

Bypassing Other Analysis Defenses

Decode obfuscated strings statically using FLOSS, xorsearch, Balbuzard, etc.

Decode data in a debugger by setting a breakpoint after the decoding function and examining results.

Conceal x64dbg/x32dbg via the ScyllaHide plugin.

To disable anti-analysis functionality, locate and patch the defensive code using a debugger.

Look out for tricky jumps via TLS, SEH, RET, CALL, etc. when stepping through the code in a debugger.

If analyzing shellcode, use scdbg and runsc.

Disable ASLR via setdllcharacteristics, CFF Explorer.

SANS DFIR REMnux Usage Tips for Malware Analysis on Linux

This cheat sheet outlines the tools and commands for analyzing malicious software on REMnux Linux distro.

Getting Started with REMnux

1. Get REMnux as a virtual appliance, install the distro on a dedicated system, or add it to an existing one.
2. Review REMnux documentation at docs.remnux.org.
3. Keep your system up to date by periodically running "remnux upgrade" and "remnux update".
4. Become familiar with REMnux malware analysis tools available as Docker images.
5. Know default logon credentials: remnux/malware

General Commands on REMnux

Shut down the system	`shutdown`
Reboot the system	`reboot`
Switch to a root shell	`sudo -s`
Renew DHCP lease	`renew-dhcp`
See current IP address	`myip`
Edit a text file	`code file`
View an image file	`feh file`
Start web server	`httpd start`
Start SSH server	`sshd start`

Analyze Windows Executables

Static Properties: manalyze, peframe, pefile, exiftool, clamscan, pescan, portex, bearcommander, pecheck

Strings and Deobfuscation: pestr, bbcrack, brxor.py, base64dump, xorsearch, flarestrings, floss, cyberchef

Code Emulation: binee, capa, vivbin

Disassemble/Decompile: ghidra, cutter, objdump, r2

Unpacking: bytehist, de4dot, upx

Reverse-Engineer Linux Binaries

Static Properties: trid, exiftool, pyew, readelf.py

Disassemble/Decompile: ghidra, cutter, objdump, r2

Debugging: edb, gdb

Behavior Analysis: ltrace, strace, frida, sysdig, unhide

Investigate Other Forms of Malicious Code

Android: apktool, droidlysis, androgui.py, baksmali, dex2jar

Java: cfr, procyon, jad, jd-gui, idx_parser.py

Python: pyinstxtractor.py, pycdc

JavaScript: js, js-file, objects.js, box-js

Shellcode: shellcode2exe.bat, scdbg, xorsearch

PowerShell: pwsh, base64dump

Flash: swfdump, flare, flasm, swf_mastah.py, xxxswf

Examine Suspicious Documents

Microsoft Office Files: vmonkey, pcodedmp, olevba, xlmdeobfuscator, oledump.py, msoffice-crypt, ssview

RTF Files: rtfobj, rtfdump

Email Messages: emldump, msgconvert

PDF Files: pdfid, pdfparser, pdfextract, pdfdecrypt, peepdf, pdftk, pdfresurrect, qpdf, pdfobjflow

General: base64dump, tesseract, exiftool

Explore Network Interactions

Monitoring: burpsuite, networkminer, polarproxy, mitmproxy, wireshark, tshark, ngrep, tcpxtract

Connecting: thug, nc, tor, wget, curl, irc, ssh, unfurl

Services: fakedns, fakemail, accept-all-ips, nc, httpd, inetsim, fakenet, sshd, myip

Gather and Analyze Data

Network: Automater.py, shodan, ipwhois_cli.py, pdnstool

Hashes: malwoverview.py, nsrllookup, Automater.py, vt, virustotal-search.py

Files: yara, scalpel, bulk_extractor, ioc_writer

Other: dexray, viper, time-decode.py

Other Analysis Tasks

Memory Forensics: vol.py, vol3, linux_mem_diff.py, aeskeyfind, rsakeyfind, bulk_extractor

File Editing: wxHexEditor, scite, code, xpdf, convert

File Extraction: 7z, unzip, unrar, cabextract

Use Docker Containers for Analysis

Thug Honeyclient: remnux/thug

JSDetox JavaScript Analysis: remnux/jsdetox

Rekall Memory Forensics: remnux/recall

RetDec Decompiler: remnux/retdec

Radare2 Reversing Framework: remnux/radare2

Ciphey Automatic Decrypter: remnux/ciphey

Viper Binary Analysis Framework: remnux/viper

REMnux in a Container: remnux/remnux-distro

Interact with Docker Images

List local images	`docker images`
Update local image	`docker pull image`
Delete local image	`docker rmi imageid`
Delete unused resources	`docker system prune`
Open a shell inside a transient container	`docker run --rm -it image bash`
Map a local TCP port 80 to container's port 80	`docker run --rm -it -p 80:80 image bash`
Map your current directory into container	`docker run --rm -it -v .:dir image bash`

Authored by Lenny Zeltser for REMnux v7. Lenny writes a security blog at **zeltser.com** and is active on Twitter at @lennyzeltser. Many REMnux tools and techniques are discussed in the Reverse-Engineering Malware course at SANS Institute, which Lenny co-authored.

SANS DFIR 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. Examine static properties of the Windows executable for initial assessment and triage.
2. Identify strings and API calls that highlight the program's suspicious or malicious capabilities.
3. Perform automated and manual behavioral analysis to gather additional details.
4. Emulate code execution to identify characteristics and areas for further analysis.
5. Use a disassembler and decompiler to statically examine code related to risky strings and APIs.
6. Use a debugger for dynamic analysis to examine how risky strings and API calls are used.
7. If appropriate, unpack the code and its artifacts.
8. As your understanding of the code increases, add comments, labels; rename functions, variables.
9. Progress to examine the code that references or depends upon the code you've already analyzed.
10. Repeat steps 5–9 above as necessary (the order may vary) until analysis objectives are met.

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

Understanding 64-Bit Registers

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

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

Decoding Conditional Jumps

`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.

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

Additional Code Analysis Tips

Be patient but persistent; focus on small, manageable code areas and expand from there.

Use dynamic code analysis (debugging) for code that's too difficult to understand statically.

Look at jumps and calls to assess how the specimen flows from "interesting" code block to the other.

If code analysis is taking too long, consider whether behavioral or memory analysis will achieve the goals.

When looking for API calls, know the official API names and the associated native APIs (Nt, Zw, Rtl).

Authored by Lenny Zeltser (**zeltser.com**) with feedback from **Anuj Soni**. Malicious code analysis and related topics are covered in the SANS Institute course **FOR610: Reverse-Engineering Malware**, which they've co-authored. For additional reversing, security and IT tips, visit **zeltser.com/cheat-sheets**.

SANS DFIR Cheat Sheet for Analyzing Malicious Documents

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

General Approach to Document Analysis

1. Examine the document for anomalies, such as risky tags, scripts, and embedded artifacts.
2. Locate embedded code, such as shellcode, macros, JavaScript, or other suspicious objects.
3. Extract suspicious code or objects from the file.
4. If relevant, deobfuscate and examine macros, JavaScript, or other embedded code.
5. If relevant, emulate, disassemble and/or debug shellcode that you extracted from the document.
6. Understand the next steps in the infection chain.

Microsoft Office Format Notes

Binary Microsoft Office document files (.doc, .xls, etc.) use the OLE2 (a.k.a. Structured Storage) format.

SRP streams in OLE2 documents sometimes store a cached version of earlier VBA macro code.

OOXML document files (.docx, .xlsm, etc.) supported by Microsoft Office are compressed zip archives.

VBA macros in OOXML documents are stored inside an OLE2 binary file, which is within the zip archive.

Excel supports XLM macros that are embedded as formulas in sheets without the OLE2 binary file.

RTF documents don't support macros but can contain malicious embedded files and objects.

Useful MS Office 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.
`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 to analyze them.
`evilclippy -uu file.ppt`	Remove the password prompt from macros in *file.ppt*.
`msoffcrypto-tool infile.docm outfile.docm -p`	Decrypt outfile.docm using specified password to create *outfile.docm*.
`pcodedmp file.doc`	Disassemble VBA-stomped p-code macro from *file.doc*.
`pcode2code file.doc`	Decompile VBA-stomped p-code macro from *file.doc*.
`rtfobj.py file.rtf`	Extract objects embedded into RTF *file.rtf*.
`rtfdump.py file.rtf`	List groups and structure of RTF file *file.rtf*.
`rtfdump.py file.rtf -O`	Examine objects in RTF file *file.rtf*.
`rtfdump.py file.rtf file.doc`	Extract hex contents from group in RTF file *file.rtf*.
`xlmdeobfuscator --file file.xlsm`	Deobfuscate XLM (Excel 4) macros in *file.xlsm*.

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. (**See examples**)

Useful PDF File Analysis Commands

`pdfid.py file.pdf -n`	Display risky keywords present 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*.

Shellcode and Other Analysis Commands

`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 characters in *file* to a string.

Additional Document Analysis Tools

SpiderMonkey, cscript, and box-js help deobfuscate JavaScript that you extract from document files.

Use the debugger built into Microsoft Office to deobfuscate macros in an isolated lab.

Use AMSIScriptContentRetrieval.ps1 to observe Microsoft Office execute macros in an isolated lab.

Some automated analysis sandboxes can analyze aspects of malicious document files.

REMnux distro includes many of the free document analysis tools mentioned above.

Authored by Lenny Zeltser (**zeltser.com**) with feedback from **Pedro Bueno** and **Didier Stevens**. Malicious document analysis and related topics are covered in the SANS Institute course **FOR610: Reverse-Engineering Malware**, which they've co-authored. More at **zeltser.com/cheat-sheets**.