MALWARE ANALYSIS CHEAT SHEET

The analysis and reversing tips behind this reference are covered in the SANS Institute course FOR610: Reverse-Engineering Malware.

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. Perform behavioral analysis to examine the specimen's interactions with its environment.
5. Perform static code analysis to further understand the specimen's inner-workings.
6. Perform dynamic code analysis to understand the more difficult aspects of the code.
7. If necessary, unpack the specimen.
8. Perform memory forensics of the infected lab system to supplement the other findings.
9. Repeat steps 4-8 above as necessary (the order may vary) until analysis objectives are met.
10. 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, FakeNet-NG).

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.

IDA Pro for Static Code Analysis

Text search	`Alt+t`
Show the operand as a character	`r`
Insert repeatable comment	`;`
Follow jump or call in view	`Enter`
Return to previous view	`Esc`
Go to next view	`Ctrl+Enter`
Toggle between text and graph views	`Spacebar`
Display a diagram of function calls	`Ctrl+F12`
List program's entry point(s)	`Ctrl+e`
Go to specific address	`g`
Rename a variable or function	`n`
Show cross-references to selected function	`Select function name » x`

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, UIF, pe_unmapper.

Bypassing Other Analysis Defenses

Decode obfuscated strings statically using FLARE, 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 jmp2it.

Disable ASLR via setdllcharacteristics, CFF Explorer.