TIPS FOR REVERSE-ENGINEERING MALICIOUS CODE

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+offset) and local variables (EBP-offset)
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 Function on Windows

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 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. This cheat sheet, version 1.1, is released under the Creative Commons v3 "Attribution" License. For additional reversing, security and IT tips, visit zeltser.com/cheat-sheets.