Chapter 18

Packing and Unpacking
Packer

- Used to shrink and encrypt malware to thwart detection by antivirus
- When packed, only access to the packed file
  - Prevents static analysis since malware must be unpacked and decrypted before analysis
  - Original executable transformed to a unique self-extracting one via compression, encryption, and obfuscation
  - Difficult to reverse-engineer
  - Employs anti-disassembly, anti-debugging, and anti-VM techniques to make analysis difficult
Packer Anatomy

- **Unpacking Stub**
  - Small piece of code loaded by the OS just as a normal program
  - Unpacking stub **loads original program**
  - Can be viewed by the analyst

- **Steps**
  - Unpacking original executable into memory
    - Loader reads PE header and copies packed sections into allocated memory normally
    - Unpacking stub unpacks code and copies it into the space allocated
  - Resolving imports of original executable
    - Windows loader cannot read packed info—resolve imports by unpacking stub
    - Loader reads PE header to find library functions to import and their addresses
    - Unless packed code's imports included in unpacking code's import section, unpacker must resolve imports manually using `LoadLibrary` and `GetProcAddress`
    - Other approaches also available—discussed in Chapter 19 (no import is needed, stealthier)
  - Transfer of execution to original execution point
    - Tail jump to entry point—use ret or call function to obscure
Unpacking Illustrated

Original Executable Prior to Packing

After unpacked and loaded into memory
Entry point still points to the unpacking stub

Fully unpacked program
Entry point back to the original entry point
Identifying Packed Programs

- Indicators of a packed program
  - Program with few imports and imports are `LoadLibrary` and `GetProcAddress`
  - IDA Pro recognizes a small amount of code by automatic analysis
  - Warning from OllyDbg – maybe packed
  - Presence of UPX0 section (a specific packer)
  - Abnormal section sizes - .text section with Size of Raw Data of 0 and Virtual Size of nonzero
  - Used by tools such as PEiD to determine if code is packed

- Entropy calculation
  - Disorder in a program much larger in encrypted and compressed payloads
  - Compressed or encrypted data more closely resembles random data – high entropy
  - Tool – Mandiant Red Curtain (calculates a threat score for any executable using entropy)
Unpacking Options

- Three options:
  1. Automatic unpacking
  2. Manual dynamic unpacking
     - Prefer automatic unpacking but not always work
     - Create a new binary does the same thing as original

- Automatic unpacking
  - Decompress and decrypt executable to restore original code – fastest, best method (if it works), not running executable
  - Specific to a single packer (prior knowledge of which packer is used)
  - PE Explorer
    - Supports NSPack, UPack, and UPX
  - Must determine where the unpacking stub ends and the original executables begins. (may easily fail)
Manual Dynamic Unpacking

- **Option #1:** Discover packing algorithm and write a program to run it in reverse
  - Drawbacks: inefficient, malware specific to individual packing program

- **Option #2:** Run packed program so unpacking stub does the work
  - P389-390, find the OEP – first instruction before packed (difficult task)
  - OllyDbg -> Find OEP by Section Hop (breakpoint before the OEP executes)
  - Then Dump Debugged Process -> dump everything from process memory onto disk should be done automatically by OllyDump (plugins)
Manual Unpacking

- Rebuild Import Table
  - OllDbg may fail to rebuild the original import table – use Import Reconstructor to perform these steps (p. 390)

- Find OEP via automated tools
  - Doesn’t work if calls doesn’t return
  - Try both step-over and step-in methods

- Find OEP manually
  - Look for tail jump – jumps from unpacking stub to OEP
  - Some malware make it a ret instruction to hide from detection
  - Listing 18-1: two features to identify the tail jump
    - At the end of the code
    - Links to an address far away
Identify Tail Jump

00416C31   PUSH EDI
00416C32   CALL EBP
00416C34   POP EAX
00416C35   POPAD
00416C36   LEA EAX,DWORD PTR SS:[ESP-80]
00416C3A   PUSH 0
00416C3C   CMP ESP,EAX
00416C3E   JNZ SHORT Sample84.00416C3A
00416C40   SUB ESP,-80
00416C43   JMP Sample84.00401000
00416C48   DB 00
00416C49   DB 00
00416C4A   DB 00
00416C4B   DB 00
00416C4C   DB 00
00416C4D   DB 00
00416C4E   DB 00
Common Packers

- **UPX (Ultimate Packer for eXecutables)**
  - Open-source
  - Designed for performance not security
  - OllyDump – Find OEP by Section Hop Feature

- **PECompact**
  - Similar to UPX, can be difficult to unpack (includes anti-debugging exceptions and obfuscated code)
  - Uses a tail jump `jmp eax` followed by many 0x00 bytes.

- **ASPack**
  - Uses self-modifying code to thwart analysis
  - Breakpoints make the program terminate prematurely
Common Packers

- **Petite**
  - Similar to ASPack
  - Uses single-step exceptions to break into debugger

- **WinUpack**
  - Uses PUSH followed by RET for tail jump
  - Placed in the middle of stub (Listing 18-5, p. 399)- hard to find

- **Themida**
  - Secure packer employing anti-debugging, anti-analysis, and anti-VM techniques
  - Contains a kernel component making it difficult to follow (kernel code has less restrictions, analysis in user space)
  - Runs code continuously
  - Use `ProcDump` to dump memory without attaching debugger