Shellcode Analysis

Shellcode is Different from Regular Code
Loading Shellcode
Position-Independent Code
Identifying Execution Location
Manual Symbol Resolution
Finding `kernel32.dll`
Finding Functions
Hashing Names
Shellcode is Different from Regular Code

- Name comes from propensity of exploits to want to run a *shell* process.
- Usually runs along with an exploit or is injected by an exploit.
- Has no access to windows loader.
- Must work harder to implement interesting behaviors.
Loading Shellcode

- Since shellcode cannot run independently, debugging it can be difficult—there's no process to debug.
- Shellcode is typically just a sequence of bytes of executable code and data.
- The program shellcode_launcher.exe will be used to load and execute shellcode.
- Loading into IDA is easy but there is not executable file format. Must use the *Intel 80x86 processors:metapc* processor type and select *32-bit disassembly*. IDA does not perform any automatic analysis.
Position-Independent Code - 1

- Any code that does not depend on its location in memory is referred to as *position-independent* code (PIC).
- Shellcode is nearly always PIC because it may not know exactly where in memory it will appear.
- Some instructions are position-independent, but others are not. This is determined by the op-code. For example
  - **CALL opcode E8** is a near relative call. E8 DD CC BB AA does a transfer to relative address AABBCCDD (which is calculated by adding 5 + AABBCCDD to location of the call instruction).
  - **CALL opcode 9A** is a far absolute call. 9A DD CC BB AA does a transfer to absolute address AABBCCDD.
- The near relative call can be used in PIC, but not the far absolute call.
Position-Independent Code - 2

- It is easy to insure PIC for control flow since most control flow instructions are already PIC.

- Data references are trickier. Consider
  - `mov eax, dword_407030 - 8B 05 30 70 40 00`
    This is not IPC. It takes data from an absolute memory location and loads it into `eax`.
  - `mov eax, [ebp-4] - 8B 45 FC`
    This is IPC. It loads data from the address 4 bytes lower than the base pointer. (Normally this would be a local variable of the current function.)

- The Intel Instruction Chart from can come in handy here:
  [http://sparksandflames.com/files/x86InstructionChart.html](http://sparksandflames.com/files/x86InstructionChart.html)
Identifying Execution Location

- The `eip` register contains the shellcode location, but `eip`-relative addressing is not supported by Intel. Shellcode may load the `eip` into a register.

- There is no instruction to directly move `eip` into a register, like `mov eax, eip`. Two other approaches can be taken:
  - `call/pop`
    A call forces `eip` (after the call instruction) into the stack. The `eip` can then be popped into any appropriate register.
  - `fnstenv`
    Stores state of FPU to memory. Offset 12 in this state is the address of the last CPU instruction that used the FPU. This can be forced by performing `fldz` (push zero onto FPU stack).
Manual Symbol Resolution – 1
Finding *kernel32.dll*

- Shellcode must find addresses of `LoadLibraryA` and `GetProcAddress`.
- These are found in *kernel32.dll* by using undocumented structures. Documentation can be found at [http://undocumented.ntinternals.net/](http://undocumented.ntinternals.net/)
- Important to realize that
  - Links fields are 8 bytes and contain `FLINK` and `BLINK`
  - Pointers (`InLoadOrderLinks`, `InMemoryOrderLinks`, `InInitializationOrderLinks`) point to the corresponding link in the referenced structure, NOT the beginning of the structure.
Manual Symbol Resolution – 2
Finding Functions

• PE code uses *relative virtual addresses*. To convert to actual address, add the PE image base address.

• Must find RVA to the PE IMAGE_EXPORT_DIRECTORY at the end of the IMAGE_OPTIONAL_HEADER. In 32-bit machines this is at offset 120.

• The Export Directory maps Ordinals of functions to their Names. Search the AddressOfNames list for the symbol of interest and the corresponding entry AddressOfNameOrdinals is its ordinal. (Names are sorted.)

• Find the ordinal's entry in the AddressofFunctions array. Add PE base address this RVA to get a pointer to the function of interest (usually LoadLibraryA).
Manual Symbol Resolution – 3 Hashing Names

• Using strcmp against the full string name may use lots of storage. MSDN functions have loooong names.

• Hashing the names with a short function can save storage which is at a premium in shellcode as there may be limits on the length of code a specific exploit supports.

• Hash only needs to distinguish between all the functions in any of the dlls being inspected.

• Text shows example of 32-bit rotate-right-additive has used by Metasploit.
Next Time

• More of PMA Chapter 19 (Shellcode Analysis)
• More exercises