Analyzing Malware with Hooks, Stomps, and Return-addresses

arashparsa.com/catching-a-malware-with-no-name/

Arash's Security Thoughts n Stuff March 12, 2022

Mar 12, 2022

Table of Contents

Introduction

This is the second post in my series on developing robust malware and their relevant detection's. This post will focus on an interesting observation I made when creating my heap encryption and how this could be leveraged to detect arbitrary shell-code as well as tools like cobalt strike, how those detections could be bypassed and even newer detections can be made.

EDITED: Forgot the POC! Here it is https://github.com/waldo-irc/MalMemDetect

The First Detection

If you recall in the first post, our method at targeting Cobalt Strikes heap allocations was to hook the process space and manage all allocations made by essentially what was a module with no name. Here is the code we had used as a refresher:

```
#include <intrin.h>
#pragma intrinsic(_ReturnAddress)

GlobalThreadId = GetCurrentThreadId(); We get the thread Id of our dropper!

HookedHeapAlloc (Arg1, Arg2, Arg3) {
    LPVOID pointerToEncrypt = OldHeapAlloc(Arg1, Arg2, Arg3);
    if (GlobalThreadId == GetCurrentThreadId()) { // If the calling ThreadId matches
our initial thread id then continue

        HMODULE hModule;
        char lpBaseName[256];

                if (::GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
(LPCSTR)_ReturnAddress(), &hModule) == 1) {
                ::GetModuleBaseNameA(GetCurrentProcess(), hModule, lpBaseName,
sizeof(lpBaseName));
         }

        std::string modName = lpBaseName;
        std::transform(modName.begin(), modName.end(), modName.begin(),
                [](unsigned char c) { return std::tolower(c); });
        if (modName.find("dll") == std::string::npos && modName.find("exe") ==
std::string::npos) {
                     // Insert pointerToEncrypt variable into a list
        }
    }
}
```

The magic lines lie here:

```
                if (::GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
(LPCSTR)_ReturnAddress(), &hModule) == 1) {
                ::GetModuleBaseNameA(GetCurrentProcess(), hModule, lpBaseName,
sizeof(lpBaseName));
         }
```

What we are trying to do here is take the current address our function will be returning to and attempting to resolve it to a module name using the function GetModuleHandleExA with the argument GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS. With this flag the implication is the address we are passing is: "an address in the module" (https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandleexa). The module name will get returned and stored in the lpBaseName variable.

With the case of our thread targeted heap encryption this function actually returns 0, as it cannot resolve the return address to a module! This also means lpBaseName ends up containing nothing.

As always, let's see what this looks like in our debugger. First, we'll start with a legitimate call. I've gone ahead and hooked HeapAlloc using MinHook (https://github.com/TsudaKageyu/minhook) and am tracing the return address of all callers. Let's see who the first function to call our hooked malloc is:

```
    LPCSTR data = (LPCSTR)_ReturnAddress();
    if (::GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS, data, &hModule) == 1) {
        ::GetModuleBaseNameA(GetCurrentProcess(), hModule, lpBaseName, sizeof(lpBaseName));
    }
    else {
        if (threadMonitor == NULL) {
            threadMonitor = callerId;
        }
        snprintf(log, 255, "Suspicious Malloc() from thread with id:%d LPVOID:%p Heap Handle:%p Siz
        LogDetected(&log);
```

fig 1. Usage of _ReturnAddress intrinsic

Here we can see within our code we use the Visual C++ _ReturnAddress() intrinsic (https://docs.microsoft.com/en-us/cpp/intrinsics/returnaddress?view=msvc-160) and store the value in a variable named "data". We then pass this variable to GetModuleHandleExA in order to resolve the module name we will be returning to.

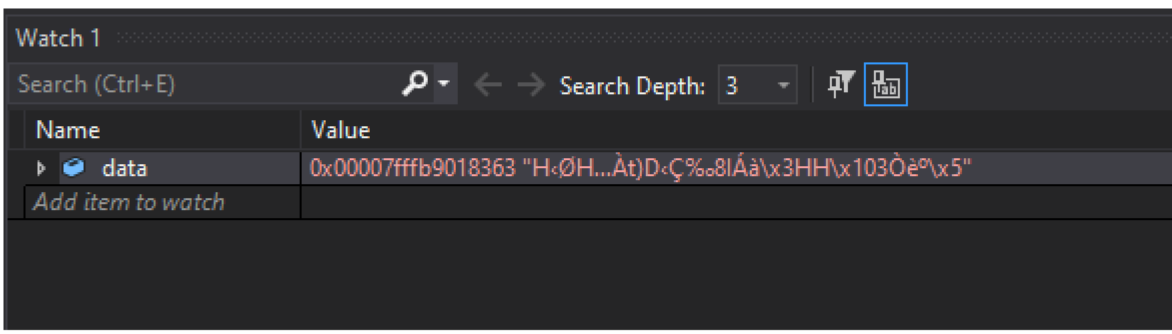


fig 2.

Return address value

Taking a look at data we can see it seems to have stored a valid address. Now let's look at this address in our disassembler.

```
LdrpGetNewTlsVector:
00007FFFB901832C 48 89 5C 24 08       mov         qword ptr [rsp+8],rbx
00007FFFB9018331 48 89 74 24 10       mov         qword ptr [rsp+10h],rsi
00007FFFB9018336 57                   push        rdi
00007FFFB9018337 48 83 EC 20          sub         rsp,20h
00007FFFB901833B 8B 15 EF 21 12 00    mov         edx,dword ptr [NtdllBaseTag (07FFFB913A530h)]
00007FFFB9018341 8B F9                mov         edi,ecx
00007FFFB9018343 81 C2 00 00 0C 00    add         edx,0C0000h
00007FFFB9018349 65 48 8B 0C 25 60 00 00 00 mov         rcx,qword ptr gs:[60h]
00007FFFB9018352 4C 8D 04 FD 10 00 00 00 lea         r8,[rdi*8+10h]
00007FFFB901835A 48 8B 49 30          mov         rcx,qword ptr [rcx+30h]
00007FFFB901835E E8 3D 26 FE FF       call        RtlAllocateHeap (07FFFB8FFA9A0h)
00007FFFB9018363 48 8B D8             mov         rbx,rax
```

fig 3. Return address location

As you can see we are right at that "mov rbx,rax" instruction at the end of the screenshot based on the address. That means when our hooked function completes this is where it will return, and we can further validate this as the correct assembly instruction we will return to as right before this is a call to RtlAllocateHeap, our hooked function! Using this we now know we are in the function LdrpGetNewTlsVector, that our hooked RtlAllocateHeap was just ran, and on completion it'll continue within LdrpGetNewTlsVector right after the call as usual. If we attempt to identify what module this function comes from we can clearly see it is from ntdll.dll.

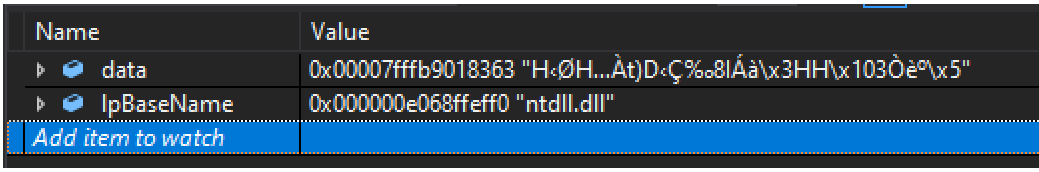


fig 4. Return address module resolved

This works because the function maps to a DLL we appear to have loaded from disk. Because of this, Windows knows how to identify what module the function comes from. What about our shellcode though? Let's see what that looks like.

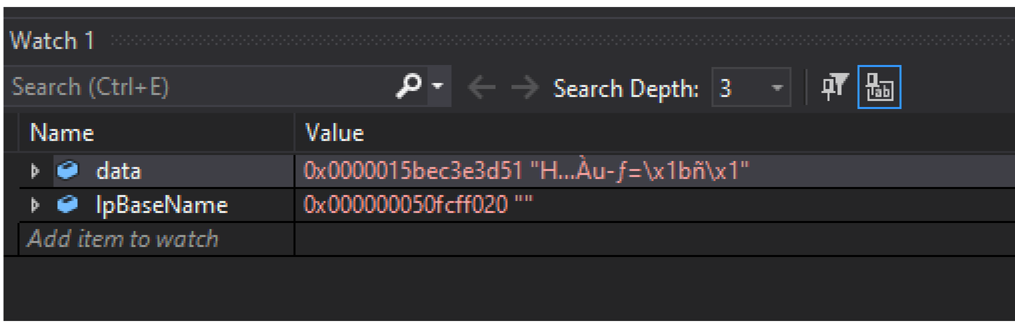


fig 5. Shellcode return address and failed resolution

So our base name is empty because the function fails to resolve the address to a module. Lets see what that address looks like in the disassembler:

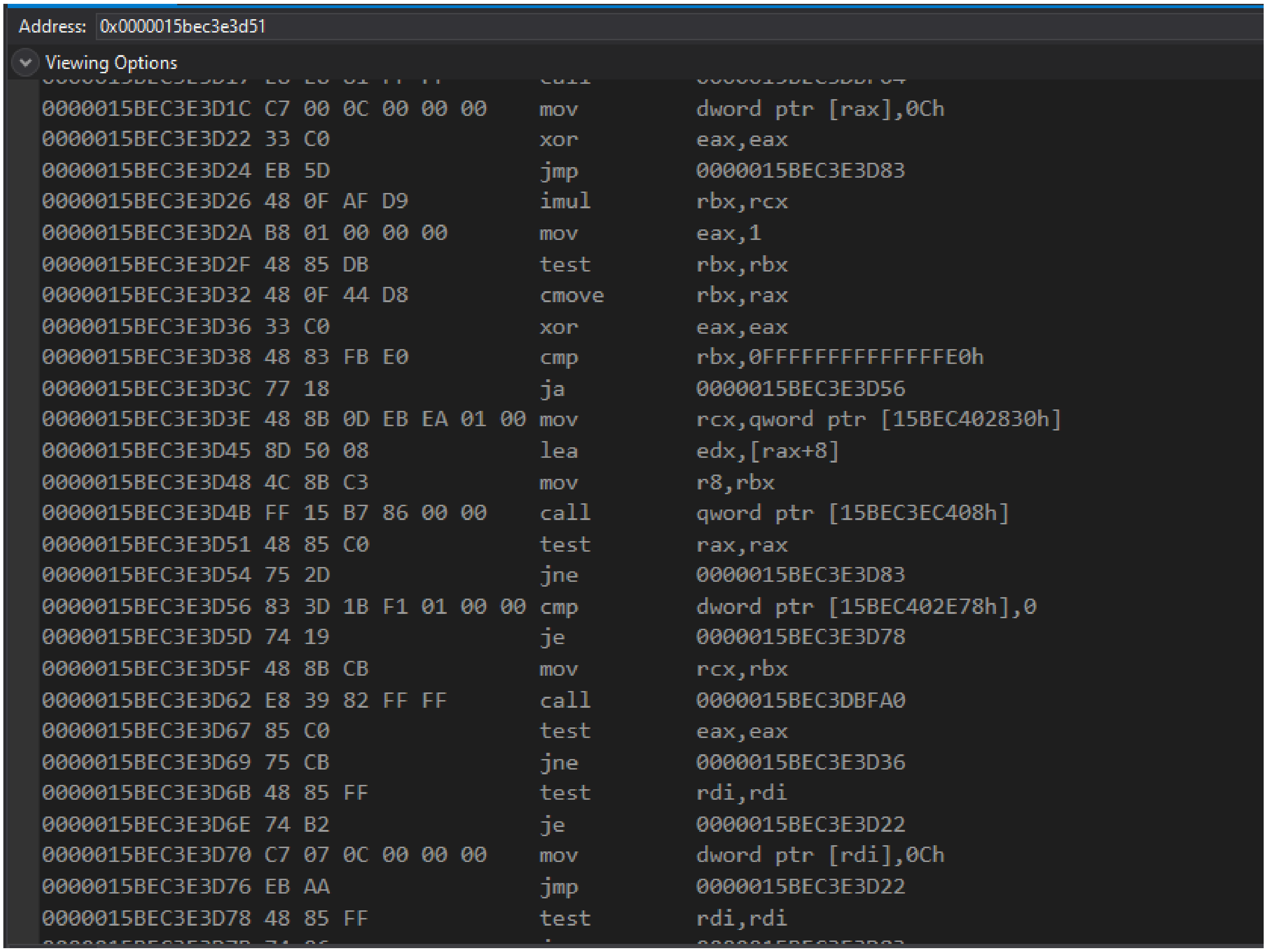
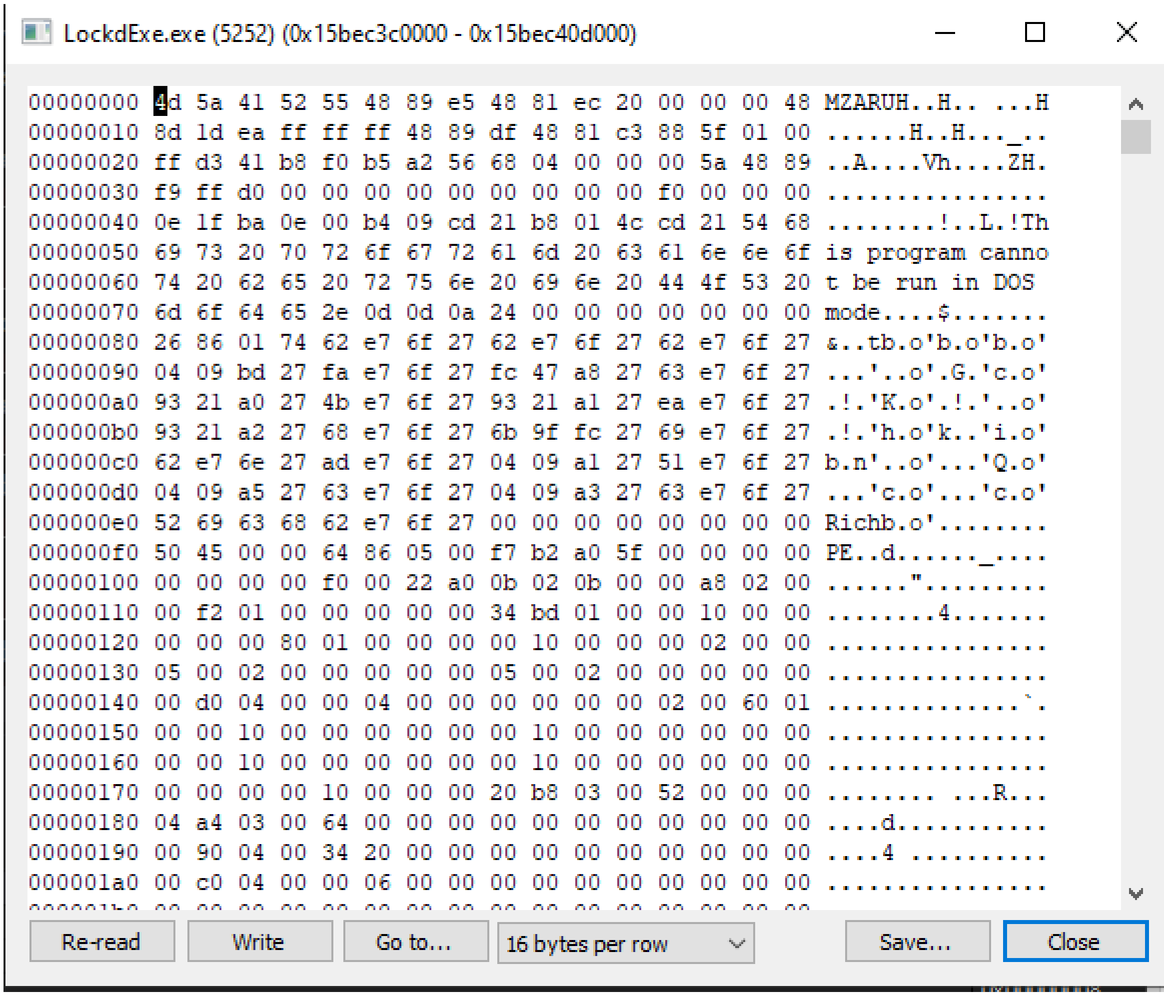


fig 6. Shellcode return address location

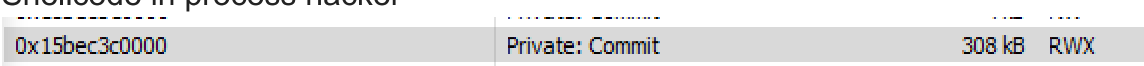
There's our address at "test rax,rax". We actually know this is our shell-code based on the address:




```
00000000 4d 5a 41 52 55 48 89 e5 48 81 ec 20 00 00 00 48 MZARUH..H.. ...H
00000010 8d 1d ea ff ff ff 48 89 df 48 81 c3 88 5f 01 00 ......H..H..._..
00000020 ff d3 41 b8 f0 b5 a2 56 68 04 00 00 00 5a 48 89 ..A....Vh....ZH.
00000030 f9 ff d0 00 00 00 00 00 00 00 00 00 f0 00 00 00 ................
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
00000080 26 86 01 74 62 e7 6f 27 62 e7 6f 27 62 e7 6f 27 &..tb.o'b.o'b.o'
00000090 04 09 bd 27 fa e7 6f 27 fc 47 a8 27 63 e7 6f 27 ...'..o'.G.'c.o'
000000a0 93 21 a0 27 4b e7 6f 27 93 21 a1 27 ea e7 6f 27 .!.'K.o'.!.'..o'
000000b0 93 21 a2 27 68 e7 6f 27 6b 9f fc 27 69 e7 6f 27 .!.'h.o'k..'i.o'
000000c0 62 e7 6e 27 ad e7 6f 27 04 09 a1 27 51 e7 6f 27 b.n'..o'...'Q.o'
000000d0 04 09 a5 27 63 e7 6f 27 04 09 a3 27 63 e7 6f 27 ...'c.o'...'c.o'
000000e0 52 69 63 68 62 e7 6f 27 00 00 00 00 00 00 00 00 Richb.o'........
000000f0 50 45 00 00 64 86 05 00 f7 b2 a0 5f 00 00 00 00 PE..d......_....
00000100 00 00 00 00 f0 00 22 a0 0b 02 0b 00 00 a8 02 00 ......".........
00000110 00 f2 01 00 00 00 00 00 34 bd 01 00 00 10 00 00 ........4.......
00000120 00 00 00 80 01 00 00 00 00 10 00 00 00 02 00 00 ................
00000130 05 00 02 00 00 00 00 00 05 00 02 00 00 00 00 00 ................
00000140 00 d0 04 00 00 04 00 00 00 00 00 00 02 00 60 01 ..............`.
00000150 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
00000160 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00 ................
00000170 00 00 00 00 10 00 00 00 20 b8 03 00 52 00 00 00 ........ ...R...
00000180 04 a4 03 00 64 00 00 00 00 00 00 00 00 00 00 00 ....d...........
00000190 00 90 04 00 34 20 00 00 00 00 00 00 00 00 00 00 ....4 ..........
000001a0 00 c0 04 00 00 06 00 00 00 00 00 00 00 00 00 00 ................
```

fig 7.

Shellcode in process hacker

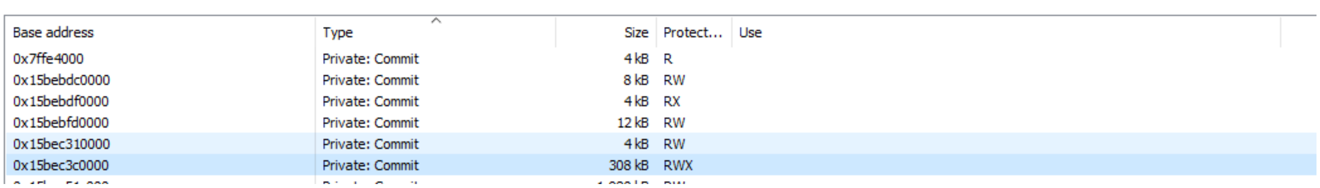


0x15bec3c0000	Private: Commit	308 kB	RWX

fig 8.

Shellcode region in process hacker

Within process hacker we can see our MZ header and that the location we are returning to is within the address space of our shellcode. We can also see unlike other modules like ntdll.dll, in ProcessHacker the "use" column is empty for our shellcode:



Base address	Type	Size	Protect...	Use
0x7ffe4000	Private: Commit	4 kB	R	
0x15bebdc0000	Private: Commit	8 kB	RW	
0x15bebdf0000	Private: Commit	4 kB	RX	
0x15bebfd0000	Private: Commit	12 kB	RW	
0x15bec310000	Private: Commit	4 kB	RW	
0x15bec3c0000	Private: Commit	308 kB	RWX	

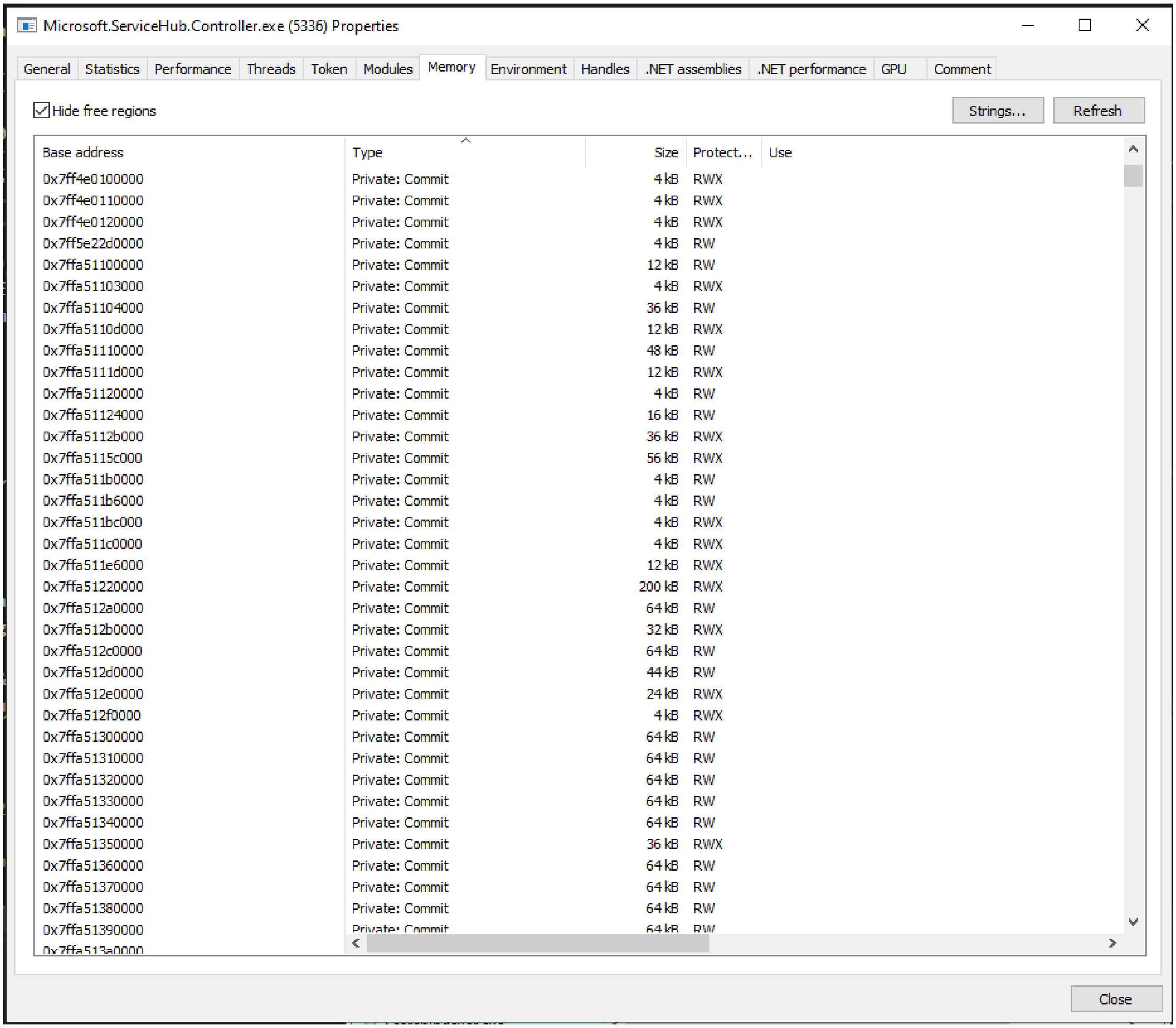
fig 9. Use section for shellcode is empty

Base address	Type	Size	Protect...	Use
0x7fffb8d38000	Image: Commit	216 kB	R	C:\Windows\System32\advapi32.dll
0x7fffb8d6e000	Image: Commit	4 kB	RW	C:\Windows\System32\advapi32.dll
0x7fffb8d6f000	Image: Commit	4 kB	WC	C:\Windows\System32\advapi32.dll
0x7fffb8d70000	Image: Commit	8 kB	RW	C:\Windows\System32\advapi32.dll
0x7fffb8d72000	Image: Commit	4 kB	WC	C:\Windows\System32\advapi32.dll
0x7fffb8d73000	Image: Commit	36 kB	R	C:\Windows\System32\advapi32.dll
0x7fffb8ed0000	Image: Commit	4 kB	R	C:\Windows\System32\kernel32.dll
0x7fffb8ed1000	Image: Commit	508 kB	RX	C:\Windows\System32\kernel32.dll
0x7fffb8f50000	Image: Commit	204 kB	R	C:\Windows\System32\kernel32.dll
0x7fffb8f83000	Image: Commit	8 kB	RW	C:\Windows\System32\kernel32.dll
0x7fffb8f85000	Image: Commit	36 kB	R	C:\Windows\System32\kernel32.dll
0x7fffb8fd0000	Image: Commit	4 kB	R	C:\Windows\System32\ntdll.dll
0x7fffb8fd1000	Image: Commit	1,132 kB	RX	C:\Windows\System32\ntdll.dll
0x7fffb90ec000	Image: Commit	288 kB	R	C:\Windows\System32\ntdll.dll
0x7fffb9134000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll
0x7fffb9135000	Image: Commit	8 kB	WC	C:\Windows\System32\ntdll.dll
0x7fffb9137000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll
0x7fffb9140000	Image: Commit	532 kB	R	C:\Windows\System32\ntdll.dll

fig 10. Use section for DLL's is filled

This is because our arbitrarily allocated memory does not map to anything on disk. Because of this, when we attempt to resolve the return address to a module we get nothing returned as a result.

That being said, we can see instances of RWX memory that don't map to disk in processes that use JIT compilers such as C# and browser processes as well. You can see in stage 3 of the Managed Execution Process (https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process) that an additional compiler takes the C# code a user creates and turns it into native code (which means our C# IL now becomes native assembly). For this process to take place a RWX region needs to be allocated for it to be able to write the new code and also be able to execute it. We can see these RWX regions in C# processes with ProcessHacker.



Base address	Type	Size	Protect...	Use
0x7ff4e0100000	Private: Commit	4 kB	RWX	
0x7ff4e0110000	Private: Commit	4 kB	RWX	
0x7ff4e0120000	Private: Commit	4 kB	RWX	
0x7ff5e22d0000	Private: Commit	4 kB	RW	
0x7ffa51100000	Private: Commit	12 kB	RW	
0x7ffa51103000	Private: Commit	4 kB	RWX	
0x7ffa51104000	Private: Commit	36 kB	RW	
0x7ffa5110d000	Private: Commit	12 kB	RWX	
0x7ffa51110000	Private: Commit	48 kB	RW	
0x7ffa5111d000	Private: Commit	12 kB	RWX	
0x7ffa51120000	Private: Commit	4 kB	RW	
0x7ffa51124000	Private: Commit	16 kB	RW	
0x7ffa5112b000	Private: Commit	36 kB	RWX	
0x7ffa5115c000	Private: Commit	56 kB	RWX	
0x7ffa511b0000	Private: Commit	4 kB	RW	
0x7ffa511b6000	Private: Commit	4 kB	RW	
0x7ffa511bc000	Private: Commit	4 kB	RWX	
0x7ffa511c0000	Private: Commit	4 kB	RWX	
0x7ffa511e6000	Private: Commit	12 kB	RWX	
0x7ffa51220000	Private: Commit	200 kB	RWX	
0x7ffa512a0000	Private: Commit	64 kB	RW	
0x7ffa512b0000	Private: Commit	32 kB	RWX	
0x7ffa512c0000	Private: Commit	64 kB	RW	
0x7ffa512d0000	Private: Commit	44 kB	RW	
0x7ffa512e0000	Private: Commit	24 kB	RWX	
0x7ffa512f0000	Private: Commit	4 kB	RWX	
0x7ffa51300000	Private: Commit	64 kB	RW	
0x7ffa51310000	Private: Commit	64 kB	RW	
0x7ffa51320000	Private: Commit	64 kB	RW	
0x7ffa51330000	Private: Commit	64 kB	RW	
0x7ffa51340000	Private: Commit	64 kB	RW	
0x7ffa51350000	Private: Commit	36 kB	RWX	
0x7ffa51360000	Private: Commit	64 kB	RW	
0x7ffa51370000	Private: Commit	64 kB	RW	
0x7ffa51380000	Private: Commit	64 kB	RW	
0x7ffa51390000	Private: Commit	64 kB	RW	
0x7ffa513a0000				

fig 11. JIIT Compiler RWX sections

Above you can see a small sample of these RWX sections within my Microsoft.ServiceHug.Controller.exe process. This means in theory we could see false positives from JIT compiler based languages that run any of our hooked functions from these memory regions. Additionally, this means these sorts of processes can also be great spaces to hide your RWX malware, as Private Commit RWX regions are otherwise considered suspicious (as we have executable memory that doesn't map to anything on disk).

Outside of blending in with JIT processes though, let's discuss another simple bypass to this, one that exists within Cobalt Strikes own C2 profile even.

The Module Stomp Bypass

If we think back to the original detection, we were able to observe exectuable memory calling our hooked functions that couldn't resolve to any module name. A first thought may be "what is a mechanism to bypass this" as one must exist. Several exist in fact, but we can start with a simple one, a mechanism called "Module Stomping" (https://www.forrestorr.net/post/malicious-memory-artifacts-part-i-dll-hollowing as well as https://www.ired.team/offensive-security/code-injection-process-injection/modulestomping-dll-hollowing-shellcode-injection).

What this technique effectively does is load a DLL that our process doesn't currently have loaded and hollow out its memory regions to instead contain the data for a malicious DLL of ours instead. This would make it so all our calls now appear to be coming from this legitimate module!

The section in your malleable C2 profile (for Cobalt Strike) that you would have to edit is the following:

```
set allocator "VirtualAlloc";  # HeapAlloc,MapViewOfFile, and VirtualAlloc.

# Ask the x86 ReflectiveLoader to load the specified library and overwrite
#  its space instead of allocating memory with VirtualAlloc.
# Only works with VirtualAlloc
set module_x86 "xpsservices.dll";
set module_x64 "xpsservices.dll";
```

These settings can be observed in the old reference profile here: https://github.com/rsmudge/Malleable-C2-Profiles/blob/master/normal/reference.profile. By changing your allocator to "VirtualAlloc" and enabling the set module_x86 and x64 settings you can now allocate your Cobalt Strike payload to arbitrary modules you load instead of arbitrarily allocated exectuable memory space.

Let's change the setting and see what this looks like. We will simply run an unstaged Cobalt Strike EXE and observe for this experiment.

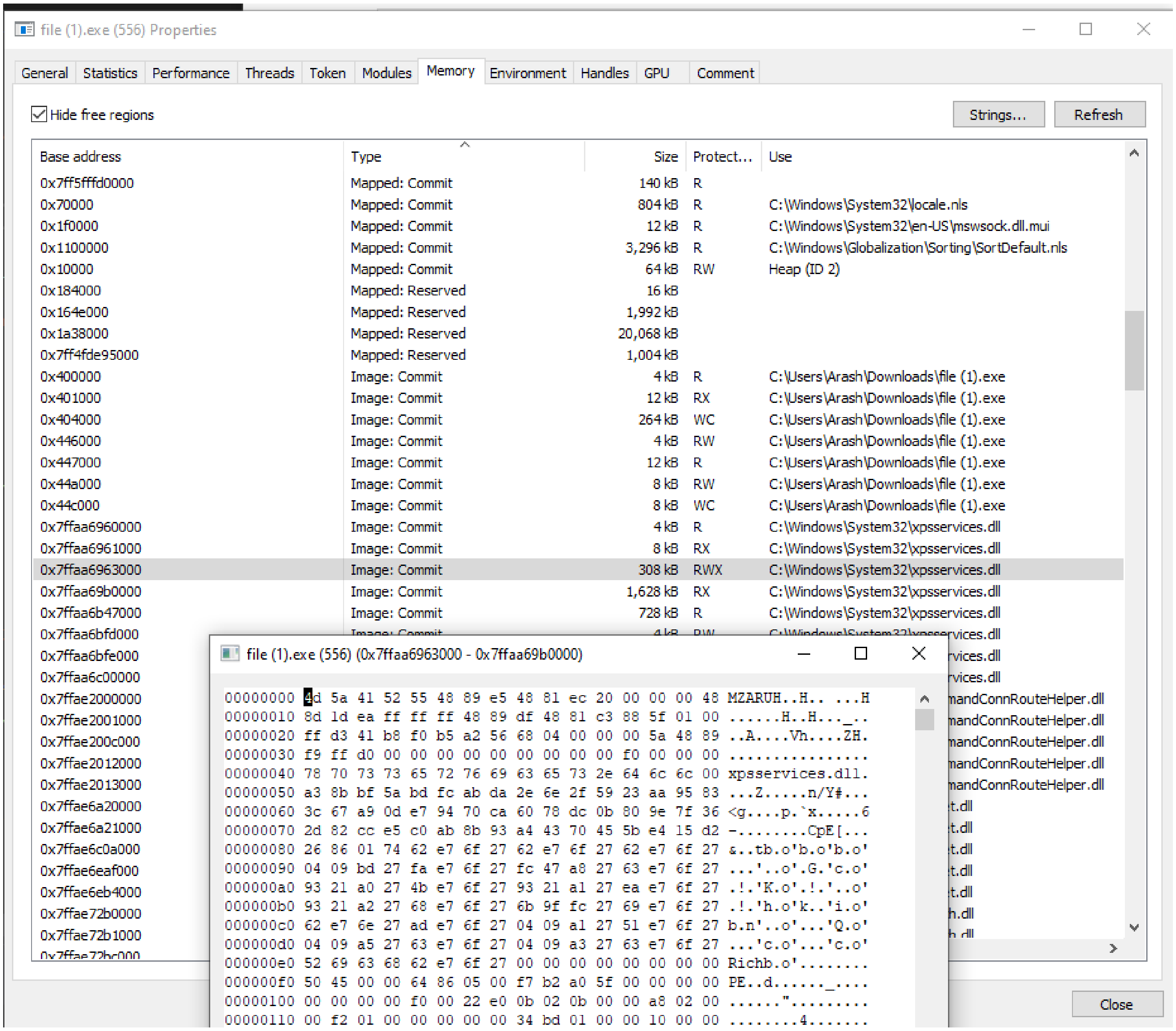
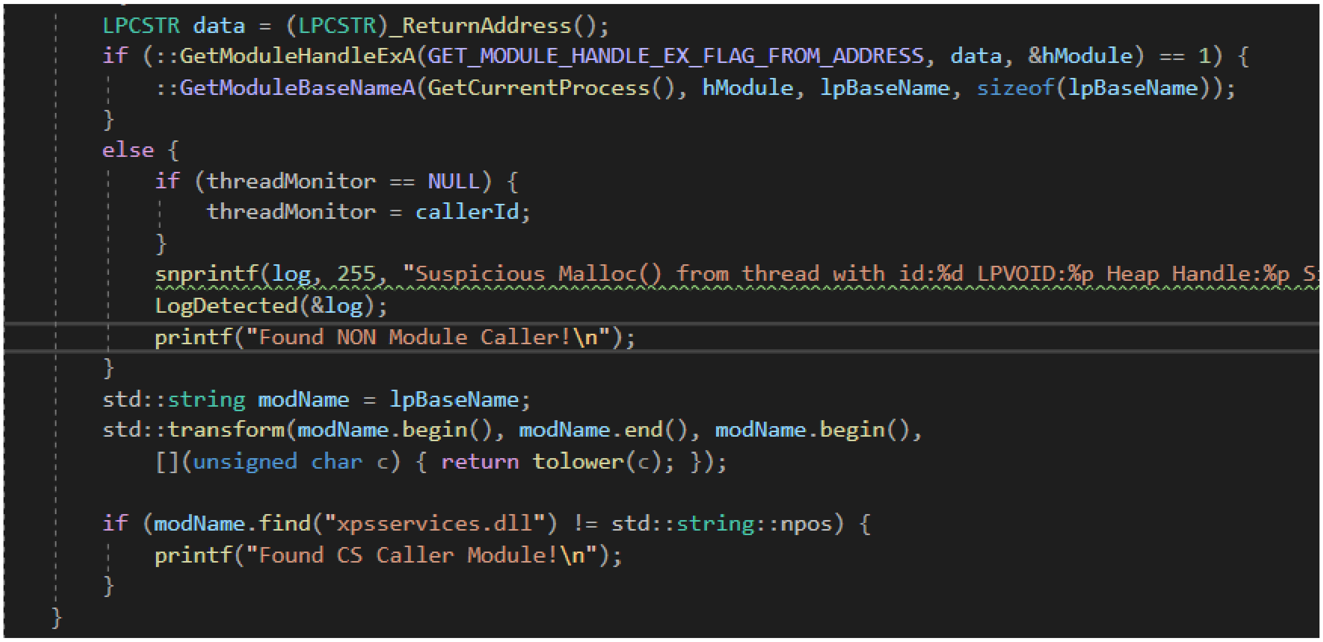


fig 12. Cobalt Strike module stomp

Let's go ahead and run this with our module name resolver and see what it looks like. Since the name should always resolve now we will change the logic a bit to monitor only xpsservices.dll.



```
    LPCSTR data = (LPCSTR)_ReturnAddress();
    if (::GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS, data, &hModule) == 1) {
        ::GetModuleBaseNameA(GetCurrentProcess(), hModule, lpBaseName, sizeof(lpBaseName));
    }
    else {
        if (threadMonitor == NULL) {
            threadMonitor = callerId;
        }
        snprintf(log, 255, "Suspicious Malloc() from thread with id:%d LPVOID:%p Heap Handle:%p S
        LogDetected(&log);
        printf("Found NON Module Caller!\n");
    }
    std::string modName = lpBaseName;
    std::transform(modName.begin(), modName.end(), modName.begin(),
        [](unsigned char c) { return tolower(c); });

    if (modName.find("xpsservices.dll") != std::string::npos) {
        printf("Found CS Caller Module!\n");
    }
}
```

fig 13. New code to monitor xpsservices

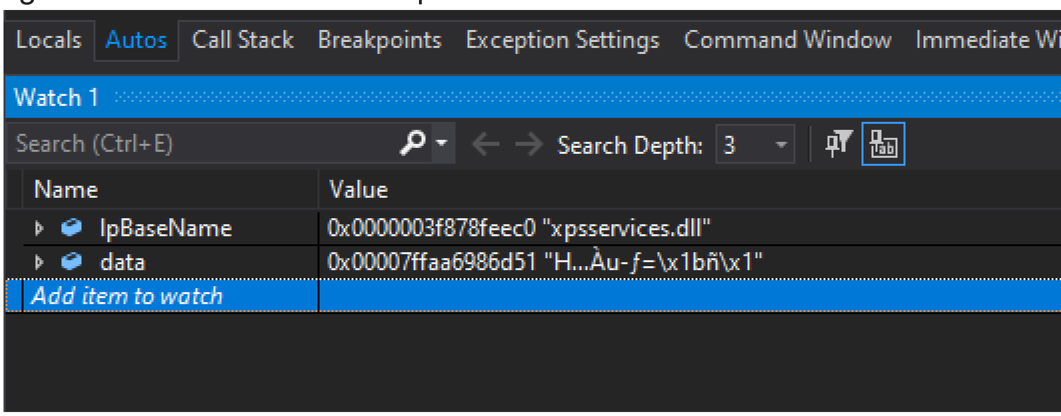


fig 14. Name resolved properly

Here we can see the new stomped DLL calling our hooked malloc, and that our code can successfully resolve calls to this module. If we look at the print statements we would also see all the calls from anything that dont't map to modules have dissapeared.

```
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
Found CS Caller Module!
```

fig 15. Only module callers

And finally we can see in the above screenshot that no callers without module names are observed anymore as all of Cobalt Strike's calls now map to a module on disk, a simple bypass. So now we ask if this technique can be detected as well, and of course, there's a few ways.

The Module Stomp Detection

There are several detections but we will delve into 2 here for module stomping. One is due to a side effect of how Cobalt Strike implements module stomping as well as general IOCs that can be observed when module stomping is performed.

The first is a detection created by Slaeryan (https://github.com/slaeryan/DetectCobaltStomp). In short, this detection works becasue a side effect of Cobalt Strike's implementation is that when loaded in memory, the region appears to be marked as a EXE internally and not a DLL. For those that don't have cobalt strike, he also created a tool to mimic the implementation for people to play with and observe the detection. I won't go into this one too much as he already has a POC and discusses this detection.

The other detection is a much more basic one. Within any executable file, the section where executable code lives is the .TEXT section. If we walk the .TEXT section of a DLL on disk and compare it to the .TEXT section of its equivalent offload in memory the sections in theory should always match, as the code should not change unless the file is polymorphic. The code for this is fairly basic.

```
HMODULE lphModule[1024];
DWORD lpcbNeeded;
// Get a handle to the process.
HANDLE = hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
    PROCESS_VM_READ,
    FALSE, processID);

// Get a list of all the modules in this process.
if (EnumProcessModules(hProcess, lphModule, sizeof(lphModule), &lpcbNeeded))
{
    for (i = 0; i < (lpcbNeeded / sizeof(HMODULE)); i++)
    {
        char szModName[MAX_PATH];

        // Get the full path to the module's file.

        if (K32GetModuleFileNameExA(hProcess, lphModule[i], szModName,
            sizeof(szModName) / sizeof(char)))
        {
            // Do stuff
        }
    }
 }
```

Here we simply start by iterating every module in the process.

```
// Get file Bytes
FILE* pFile;
long lSize;
//SIZE_T lSize;
BYTE* buffer;
size_t result;
pFile = fopen(szModName, "rb");
// obtain file size:
fseek(pFile, 0, SEEK_END);
lSize = ftell(pFile);
rewind(pFile);
// allocate memory to contain the whole file:
buffer = (BYTE*)malloc(sizeof(BYTE) * lSize);
// copy the file into the buffer:
result = fread(buffer, 1, lSize, pFile);
fclose(pFile);

BYTE* buff;
buff = (BYTE*)malloc(sizeof(BYTE) * lSize);
_ReadProcessMemory(hProcess, lphModule[i], buff, lSize, NULL);

PIMAGE_NT_HEADERS64 NtHeader = ImageNtHeader(buff);
PIMAGE_SECTION_HEADER Section = IMAGE_FIRST_SECTION(NtHeader);
WORD NumSections = NtHeader->FileHeader.NumberOfSections;
for (WORD i = 0; i < NumSections; i++)
{
    std::string secName(reinterpret_cast(Section->Name), 5);
    if (secName.find(".text") != std::string::npos) {
        break;
    }
    Section++;
}
```

We then load the relevant module file on disk and store the bytes for comparing memory in the var buffer. We then also read from the base address of the module located in "lphModule[i]" and store all the bytes within the var buff. We then enumerate all the sections in the loaded module until we find the .TEXT section and break the loop. At this point the "Section" variable will contain all our relevant section data.

To be able to match the on disk file to the one in memory we need to use the Section offsets to find the .TEXT section location on disk and in memory. This actually will not match (usually). The offset to the .TEXT section in memory generally gets relocated down a page, 4096 bytes. The offset to the section on disk is usually 1024 bytes in comparison. But we say usually so we of course will simply use "Section->PointerToRawData" to get the offset on disk and "Section->VirtualAddress" to get its offloaded address in memory to be 100% sure.

```
LPBYTE txtSectionFile = buffer + Section->PointerToRawData;
LPBYTE txtSectionMem = buff + Section->VirtualAddress;
```

At this point all you'd have to do is compare each memory region byte for byte and make sure they match.

```
                    int inconsistencies = 0;
                    for (int i = 0; i < Section->SizeOfRawData; i++) {
                        if ((char*)txtSectionFile[i] != (char*)txtSectionMem[i]) {
                            inconsistencies++;
                        }
                    }
```

Now of course we need to account for things like hooks and such, as we know many AV and EDR will perform hooks we know these will provide false positives. As a result we take the amount of the differences and if it's greater than a certain number only do we get concerned.

```
                    if (inconsistencies > 10000) {
                    printf("FOUND DLL HOLLOW.\nNOW MONITORING: %s with %f changes
found. %f%% Overall\n\n", szModName, inconsistencies, icPercent);
                    CHAR* log = (CHAR*)malloc(256);
                    snprintf(log, 255, "FOUND DLL HOLLOW.\nNOW MONITORING: %s with %f
changes found. %f%% Overall\n\n", szModName, inconsistencies, icPercent);
                    LogDetected(&log);
                    free(log);
                    std::string moduleName(szModName, sizeof(szModName) /
sizeof(char));
                    std::transform(moduleName.begin(), moduleName.end(),
moduleName.begin(),
                        [](unsigned char c) { return tolower(c); });
                    dllMonitor = moduleName;
                    break;
                }
```

We arbitrarily pick 10000 as our amount simply because we know it'll certainly be a larger number than any number of hooks any utility would alter for the hooks as well as being small enough we know most raw malware payloads at least are much bigger. This should reduce false positives substantially while finding any altered DLLs in memory. The only caveat to this would be additional false positives from polymorphic DLLs who alter themselves in memory.

Let's run our new detector against our Cobalt Strike payload and the hollowed DLL and observe the results.

```
C:\Users\Arash\source\repos\LockdExeCSScannerStable\x64\Debug\LockdExe.exe (00007FF6B7490000)
C:\Windows\SYSTEM32\ntdll.dll (00007FFAFD7B0000)
C:\Windows\System32\KERNEL32.DLL (00007FFAFC170000)
C:\Windows\System32\KERNELBASE.dll (00007FFAFB210000)
C:\Windows\SYSTEM32\dbghelp.dll (00007FFAED1E0000)
C:\Windows\System32\ucrtbase.dll (00007FFAFB090000)
C:\Users\Arash\source\repos\LockdExeCSScannerStable\x64\Debug\LockdExe.exe (00007FF6B7490000)
C:\Windows\SYSTEM32\ntdll.dll (00007FFAFD7B0000)
Found more than 5 bytes altered, there's potentially hooks here: C:\Windows\SYSTEM32\ntdll.dll Bytes Altered: 10.000000
C:\Windows\System32\KERNEL32.DLL (00007FFAFC170000)
C:\Windows\System32\KERNELBASE.dll (00007FFAFB210000)
C:\Windows\SYSTEM32\dbghelp.dll (00007FFAED1E0000)
C:\Windows\System32\ucrtbase.dll (00007FFAFB090000)
C:\Windows\SYSTEM32\wininet.dll (00007FFAE6A20000)
Found more than 5 bytes altered, there's potentially hooks here: C:\Windows\SYSTEM32\wininet.dll Bytes Altered: 10.000000
C:\Windows\System32\msvcrt.dll (00007FFAFCAD0000)
C:\Windows\SYSTEM32\iertutil.dll (00007FFAF1860000)
C:\Windows\System32\combase.dll (00007FFAFB800000)
C:\Windows\System32\RPCRT4.dll (00007FFAFCB70000)
C:\Windows\System32\sechost.dll (00007FFAFBC80000)
C:\Windows\System32\advapi32.dll (00007FFAFBBD0000)
C:\Windows\System32\shcore.dll (00007FFAFCD20000)
C:\Windows\SYSTEM32\SspiCli.dll (00007FFAFAD90000)
C:\Windows\System32\user32.dll (00007FFAFCEE0000)
C:\Windows\System32\win32u.dll (00007FFAFB190000)
C:\Windows\System32\GDI32.dll (00007FFAFD170000)
C:\Windows\System32\gdi32full.dll (00007FFAFB6F0000)
C:\Windows\System32\msvcp_win.dll (00007FFAFB4E0000)
C:\Windows\System32\IMM32.DLL (00007FFAFBE40000)
C:\Windows\SYSTEM32\windows.storage.dll (00007FFAF8FF0000)
C:\Windows\SYSTEM32\Wldp.dll (00007FFAFA850000)
C:\Windows\System32\shlwapi.dll (00007FFAFD1A0000)
C:\Windows\SYSTEM32\profapi.dll (00007FFAFAE10000)
C:\Windows\System32\WS2_32.dll (00007FFAFBB60000)
C:\Windows\SYSTEM32\ondemandconnroutehelper.dll (00007FFAE2000000)
C:\Windows\SYSTEM32\winhttp.dll (00007FFAF0C50000)
C:\Windows\SYSTEM32\kernel.appcore.dll (00007FFAF8DF0000)
C:\Windows\system32\mswsock.dll (00007FFAFA5B0000)
C:\Windows\SYSTEM32\IPHLPAPI.DLL (00007FFAFA2A0000)
C:\Windows\SYSTEM32\WINNSI.DLL (00007FFAF2BE0000)
C:\Windows\System32\NSI.dll (00007FFAFC160000)
C:\Windows\SYSTEM32\urlmon.dll (00007FFAF1410000)
C:\Windows\SYSTEM32\srvcli.dll (00007FFAF13E0000)
C:\Windows\SYSTEM32\netutils.dll (00007FFAFA3B0000)
C:\Windows\System32\OLEAUT32.dll (00007FFAFD0A0000)
C:\Windows\SYSTEM32\xpsservices.dll (00007FFAA6960000)
Found more than 5 bytes altered, there's potentially hooks here: C:\Windows\SYSTEM32\xpsservices.dll Bytes Altered: 303562.000000
FOUND DLL HOLLOW.
NOW MONITORING: C:\Windows\SYSTEM32\xpsservices.dll with 303562.000000 changes found. 15.265049% Overall
```

fig 16. DLL Hollow Detection

Here we can see a few false positives from our own hooks actually, where we alter 5 bytes to the prologue of each function, 2 functions being altered in each DLL. Finally at the end we can see our hollowed xpsservices.dll and the detection is observed with over 300k bytes altered.

Let's go ahead and turn our tool into a DLL and inject it into everything to observe false positives:

Injecting into everything and logging all data to files we can observe our detection:

```
Found more than 5 bytes altered, there's potentially hooks here: C:\Windows\SYSTEM32\xpsservices.dll Bytes Altered: 303562.000000
FOUND DLL HOLLOW.
NOW MONITORING: C:\Windows\SYSTEM32\xpsservices.dll with 303562.000000 changes found. 15.265049% Overall

Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842960 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C3300 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000786760 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842960 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C2F20 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007866E0 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842660 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C3420 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000786820 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842020 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C31A0 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007866E0 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000008420A0 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C33A0 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007866E0 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842020 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C3140 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000786810 Heap Handle:0000000000760000  Size: 8
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000000842660 Heap Handle:0000000000760000  Size: 41
Suspicious InternetConnectA() from module with name: c:\windows\system32\xpsservices.dll, Name: 192.168.1.182 Creds: (null)[(null)]
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:00000000007C35C0 Heap Handle:0000000000760000  Size: 24
Suspicious Malloc() from module with name:c:\windows\system32\xpsservices.dll LPVOID:0000000002FE0080 Heap Handle:0000000000760000  Size: 27648
```

fig 17. Detection

BUT! Interestingly enough we do observe 1 false positive on what appears to be a polymorphic DLL after all...

```
Found more than 5 bytes altered, there's potentially hooks here: C:\Program Files\VMware\VMware Tools\intl.dll Bytes Altered: 7064.000000
```

fig 18. False positive

Unfortunately not enough bytes are altered to be useful for a hollow target though!

How do you bypass this detection? Now the simple obvious solution is to restore the DLL bytes (per https://twitter.com/solomonsklash's idea) on sleep to prevent this sort of detection and next steps would be hooking those calls and detecting the restores if possible or the constant file reads etc. As we all know Cyber Security is a never ending cat and mouse.

Final Thoughts

As red teamers work on malware often we make discoveries that can lead to new detections too. These observations can be tremendously useful to the community while also pushing researchers to the cutting edge and forcing them to think outside of the box if they'd like this game to continue longer.

As we see above we find detections, make bypasses, find more detections, and the game will never end. Hopefully some interesting new insights could be made to make our defensive industry far more robust overall as we work together towards a goal of secure internet usage.