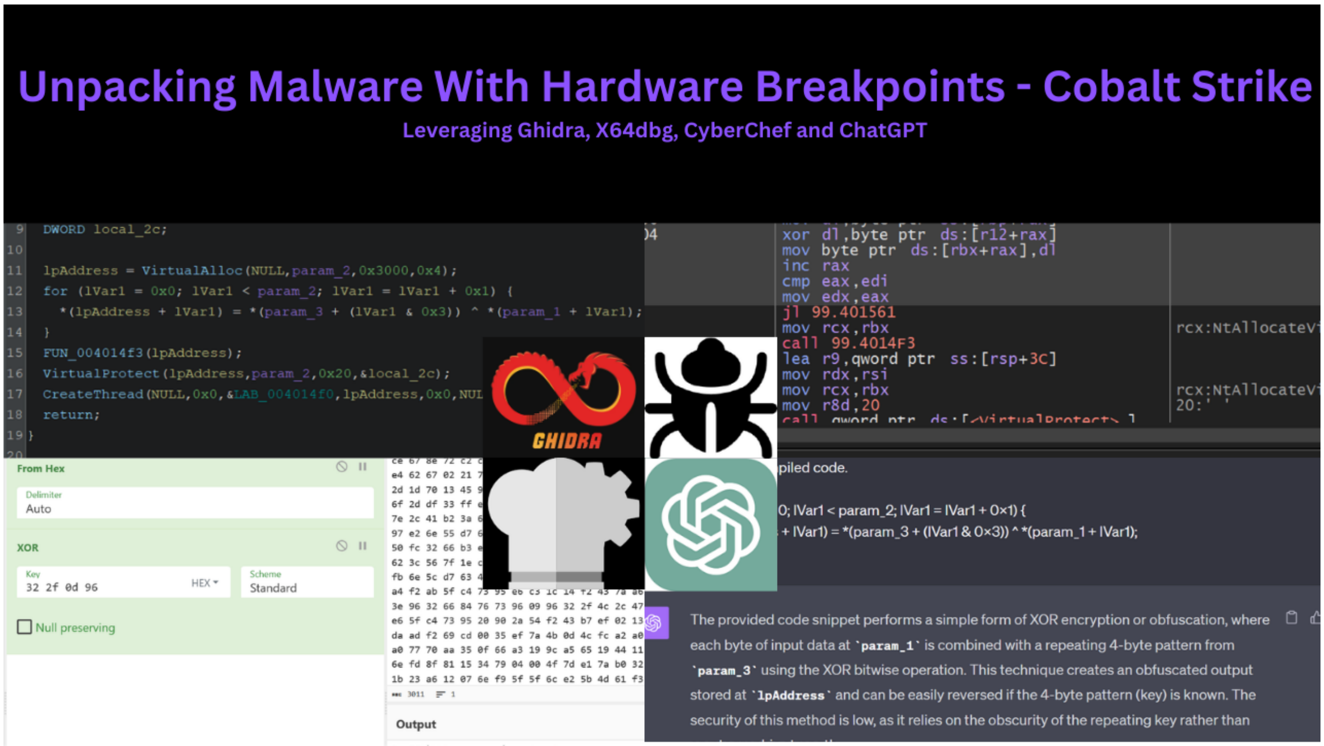
Malware Unpacking With Hardware Breakpoints - Cobalt Strike Shellcode Loader

embee-research.ghost.io/unpacking-malware-with-hardware-breakpoints-cobalt-strike/

Matthew November 6, 2023

Intermediate

Unpacking a simple Cobalt Strike loader using Debuggers and Hardware breakpoints.



In previous posts here and here, we explored methods for extracting cobalt strike shellcode from script-based malware.

In this post, we'll explore a more complex situation where Cobalt Strike shellcode is loaded by a compiled executable `.exe` file. This will require the use of a debugger (x64dbg) in conjunction with Static Analysis (Ghidra) in order to perform a complete analysis.

Overview

The executable is a compiled exe containing hidden and obfuscated Shellcode. The shellcode is decoded using a simple XOR routine and a 4-byte key, is then written to a simple buffer created with `VirtualAlloc`.

We will explore methods for obtaining the decoded shellcode using a debugger, and we will then explore methods for manually locating the Shellcode and associated decryption keys using Ghidra.

We'll also look at a way to pivot between X64dbg and Ghidra, as well as a method for identifying and analysing Ghidra output using ChatGPT.

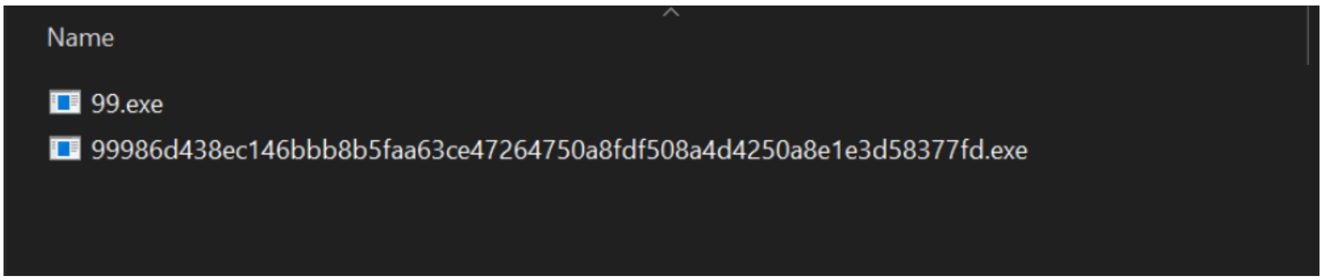
Obtaining the Sample

You can follow along by downloading the sample [here on Malware Bazaar]() (pw:`infected`)

SHA256: **`99986d438ec146bbb8b5faa63ce47264750a8fdf508a4d4250a8e1e3d58377fd`**

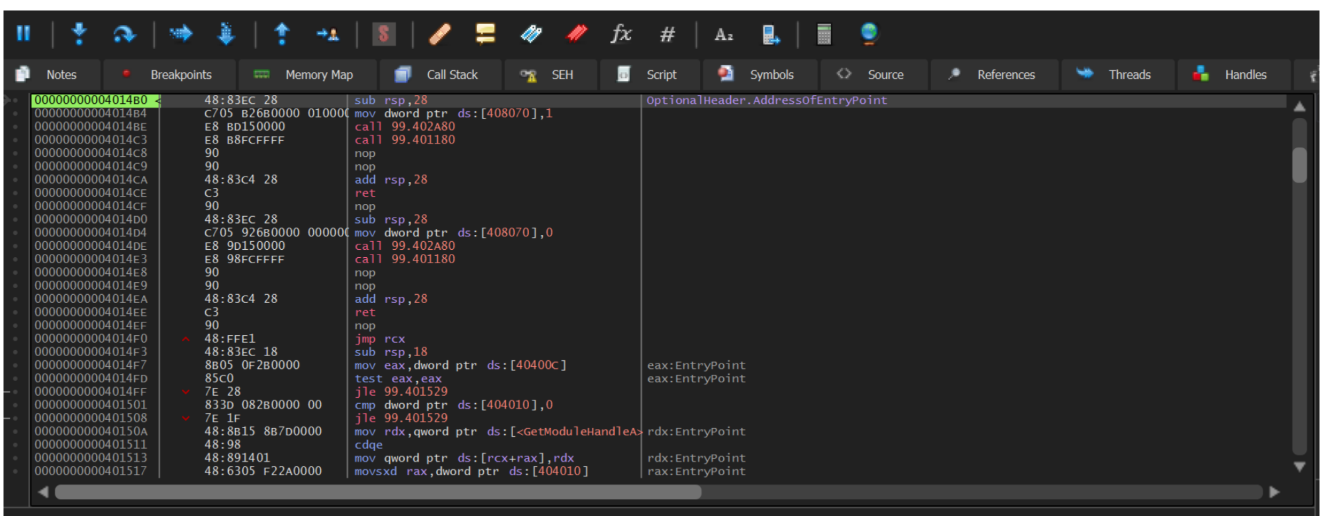
Analysis

We can begin by saving the file to an analysis machine and unzipping it with the password `infected`. From here we can also create a copy with a shorter file name.



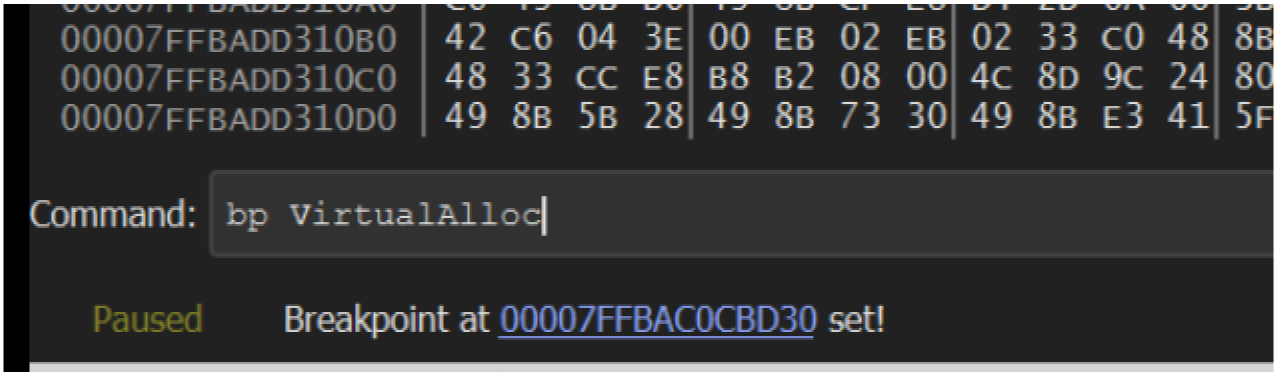
Since the file is a compiled executable, we can attempt to analyse it using a debugger. In this case x64dbg.

We can go ahead and open the file with x64dbg, clicking through until we reach the entry point.



We can now go ahead and create some breakpoints on API's that are commonly (but not always) used when malware is unpacking.

We can go ahead and create 2 breakpoints by running `bp VirtualAlloc` and `bp VirtualProtect`



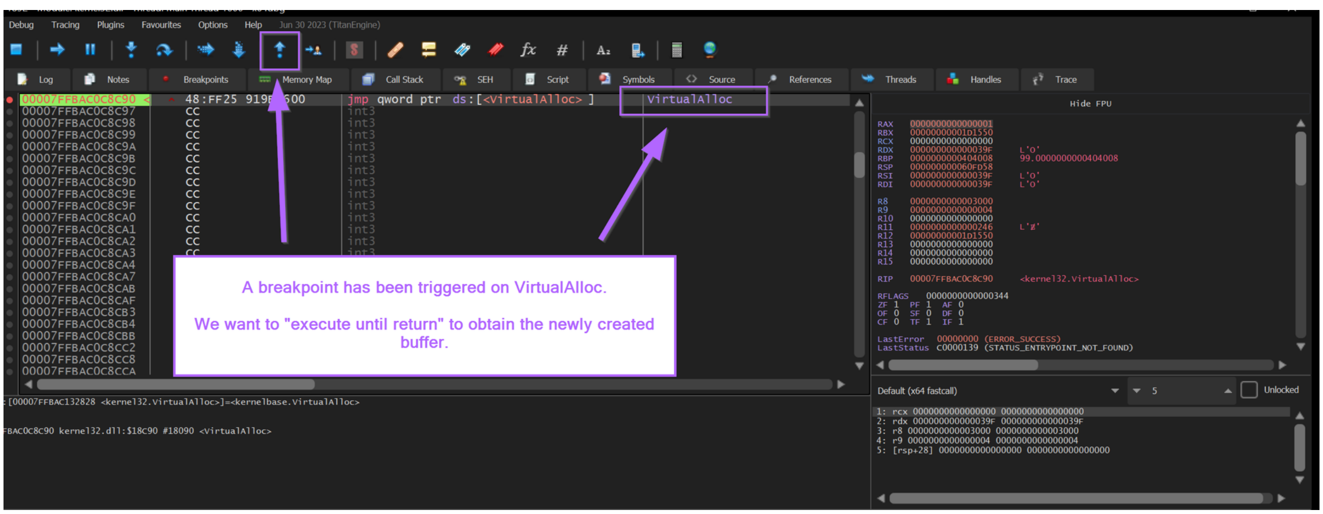
After creating the breakpoints, we can go ahead and allow the malware to continue (F9)

The malware will continue to run and trigger a breakpoint on `VirtualAlloc`.

Our primary purpose here is to obtain the buffer being created by VirtualAlloc, we can do this by using `Execute Until Return`.

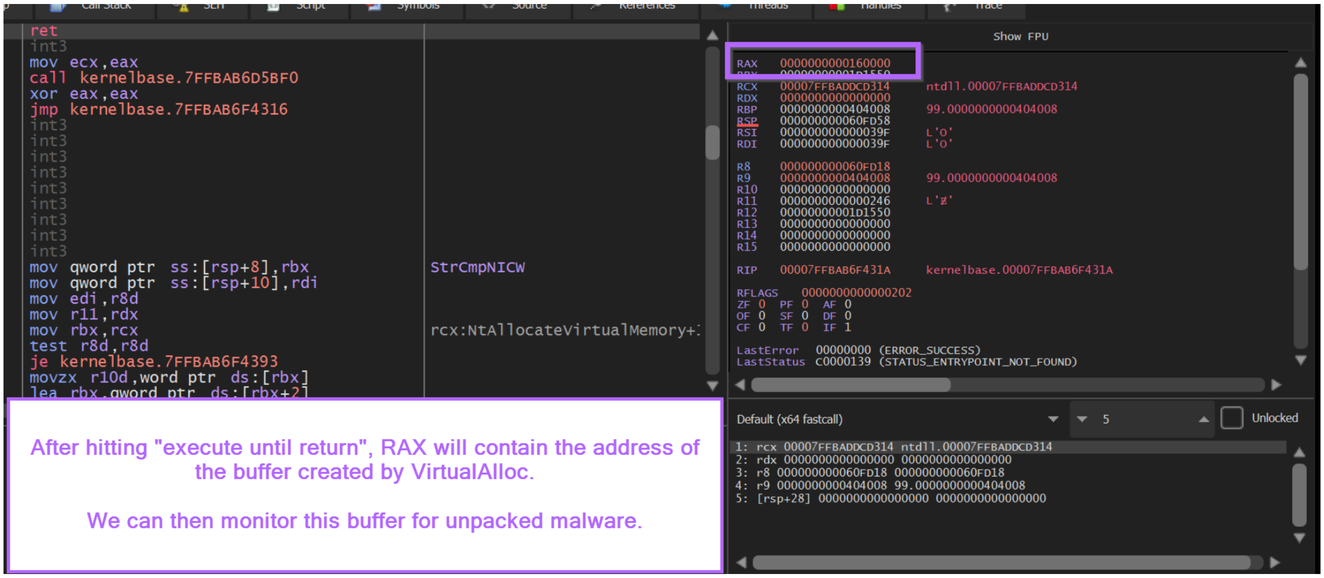
"Execute Until Return" will allow the VirtualAlloc function to complete, but won't allow any further actions to occur. This means we can easily obtain the address of the buffer that was created.

Viewing Memory Created by VirtualAlloc



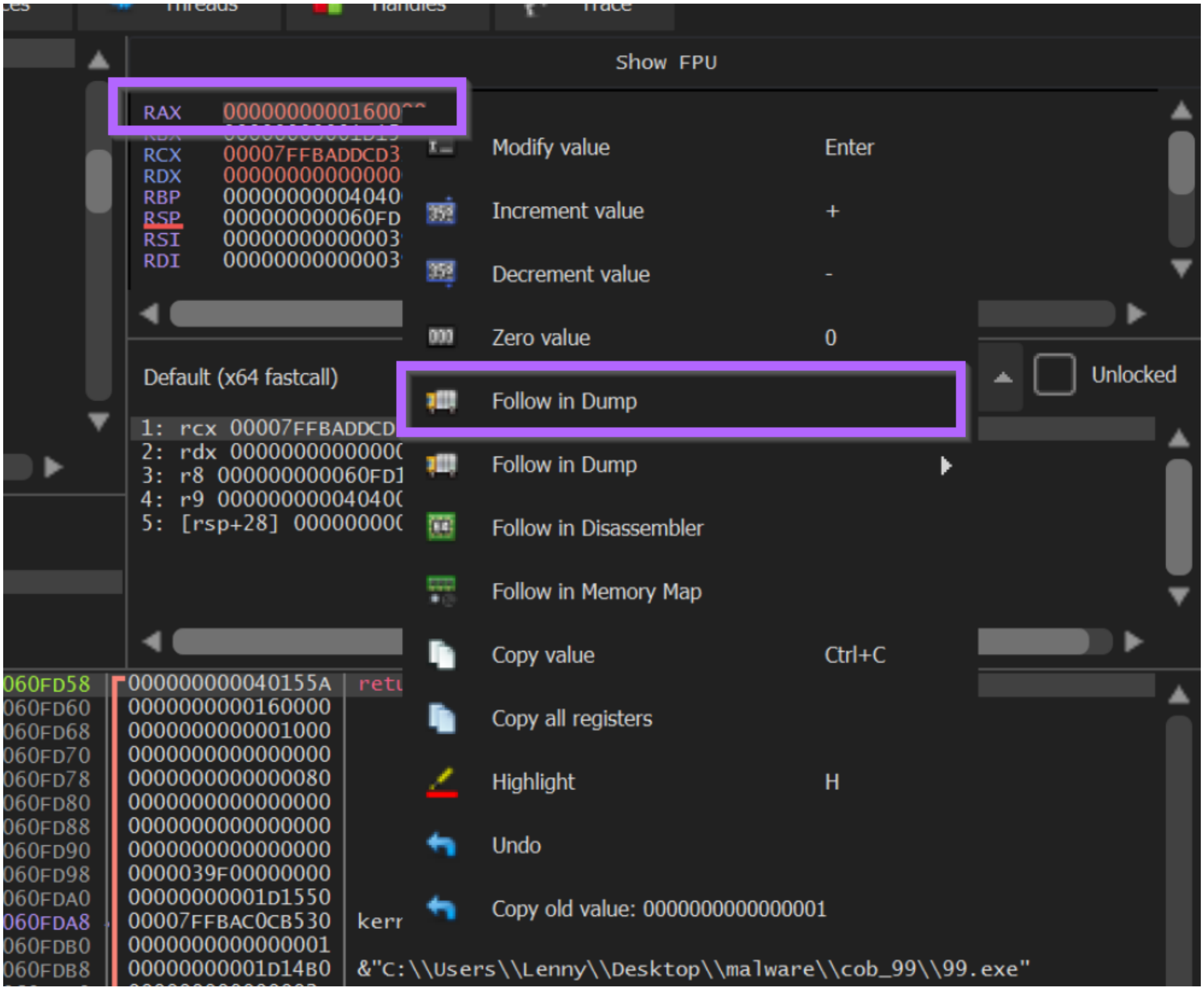
After hitting `execute until return`. We can observe the address of the newly created buffer inside of `RAX`.

We want to go ahead and monitor this buffer for suspicious content and unpacked malware.



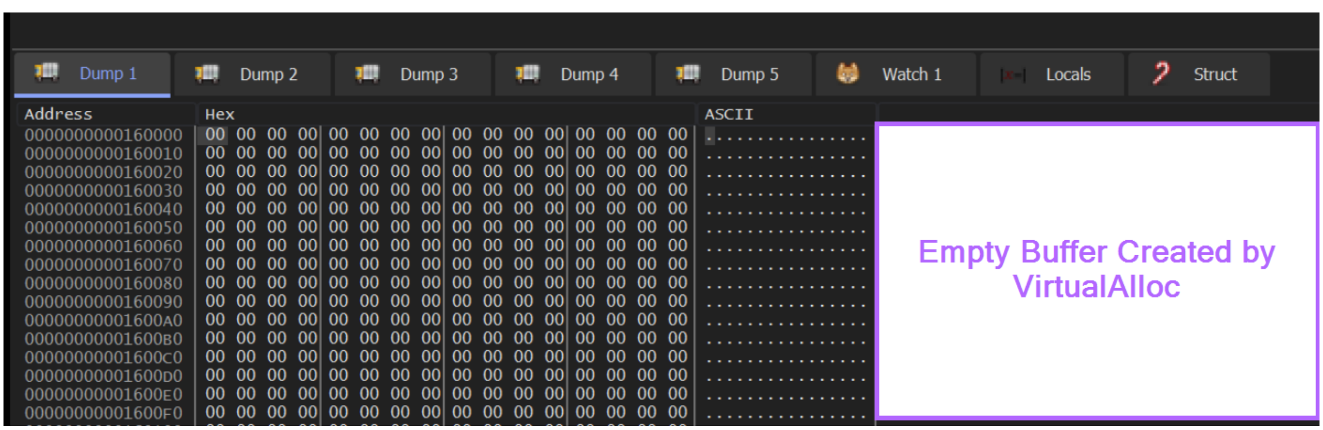
We can begin the monitoring process by right-clicking on the address contained inside of `RAX`.

From here we can select `Follow in Dump`. This will open the content of the buffer in the bottom-left window.



By clicking "Follow In Dump", we can observe the contents of the dump in the bottom-left window.

We can note here that the buffer is empty and contains only `00`.

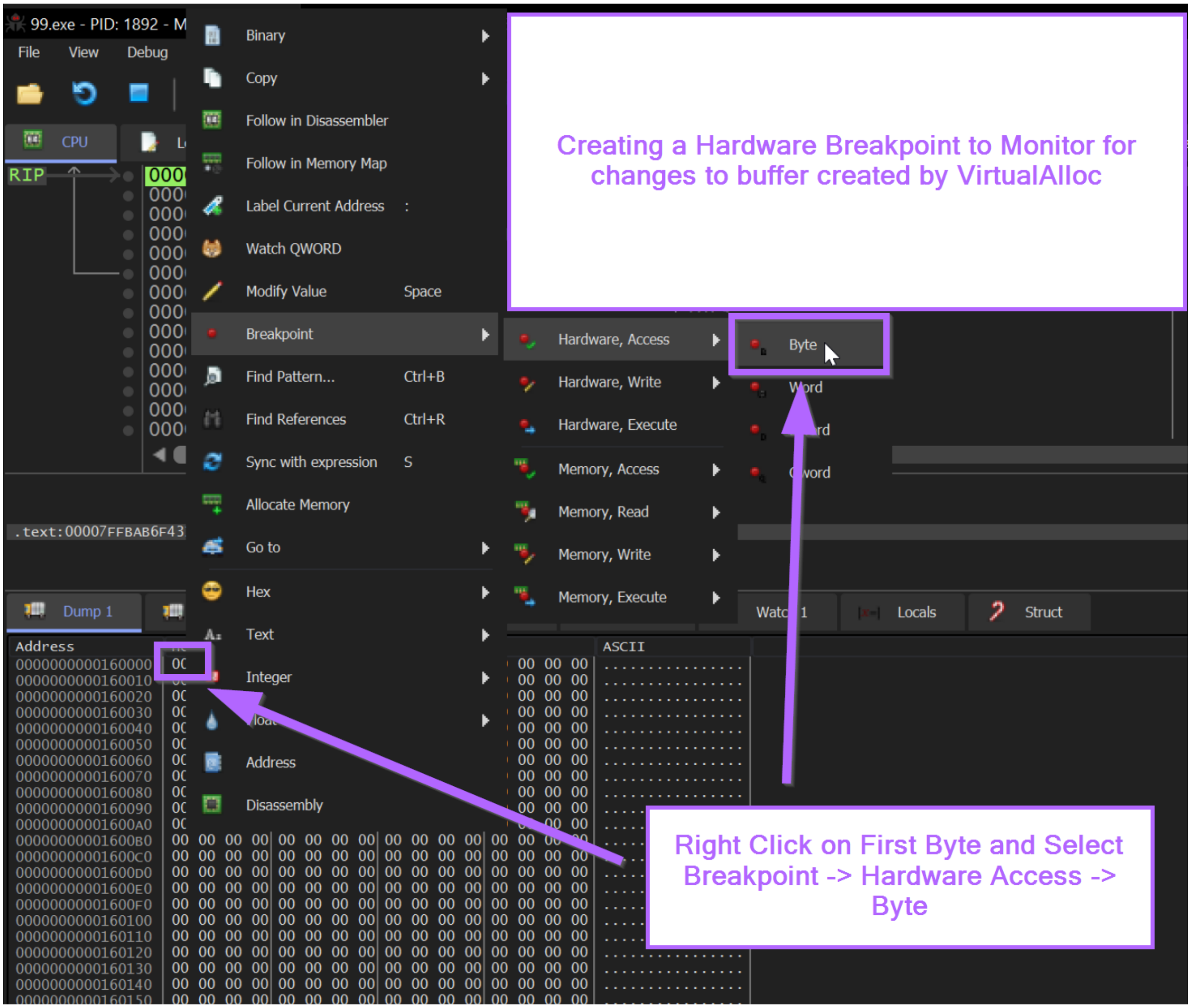


Monitoring Memory With Hardware Breakpoints

VirtualAlloc has finished creating an empty buffer and we have successfully found it.

We can now go ahead and monitor for changes to this buffer by creating a `Hardware Breakpoint`.

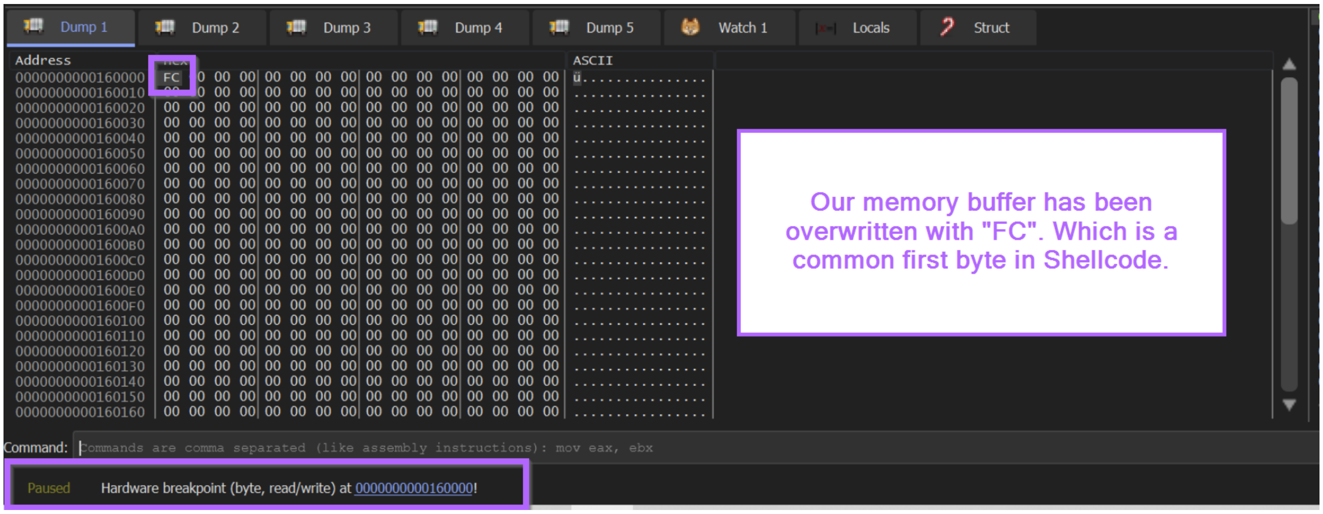
A hardware breakpoint can be created by selecting the first byte in the memory dump and `Right Click -> Breakpoint -> Hardware, Access -> Byte`



From here we can allow the malware to continue to execute.

We should soon see our hardware breakpoint triggered. With an `FC` byte contained in the first part of the buffer.

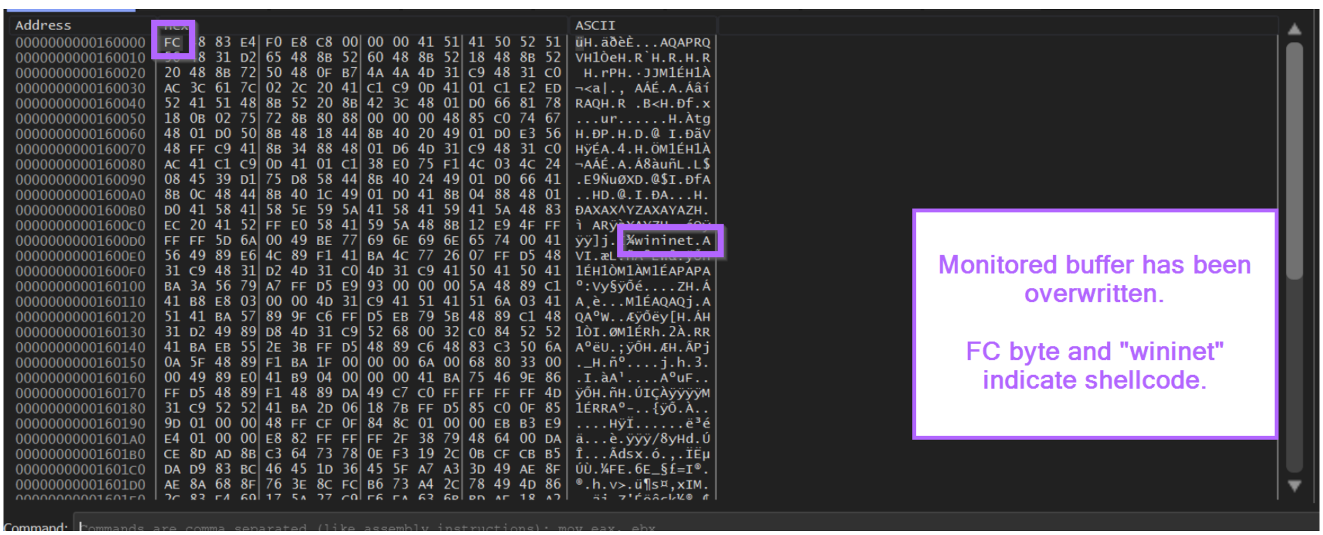
We can recall from previous blogs that `FC` is a very common first byte in shellcode.



At this point we want the malware to continue to fill up the buffer, but we don't want it to do anything after that.

We can go ahead and use another `Execute Until Return` . Which will allow the buffer to fill up. At which point we can monitor it's contents.

Below we can see the buffer after it has filled. We can see the first byte is `0xFC` and there is a `wininet` string present in the initial bytes. From previous blogs (1, 2)we know that this could indicate shellcode.

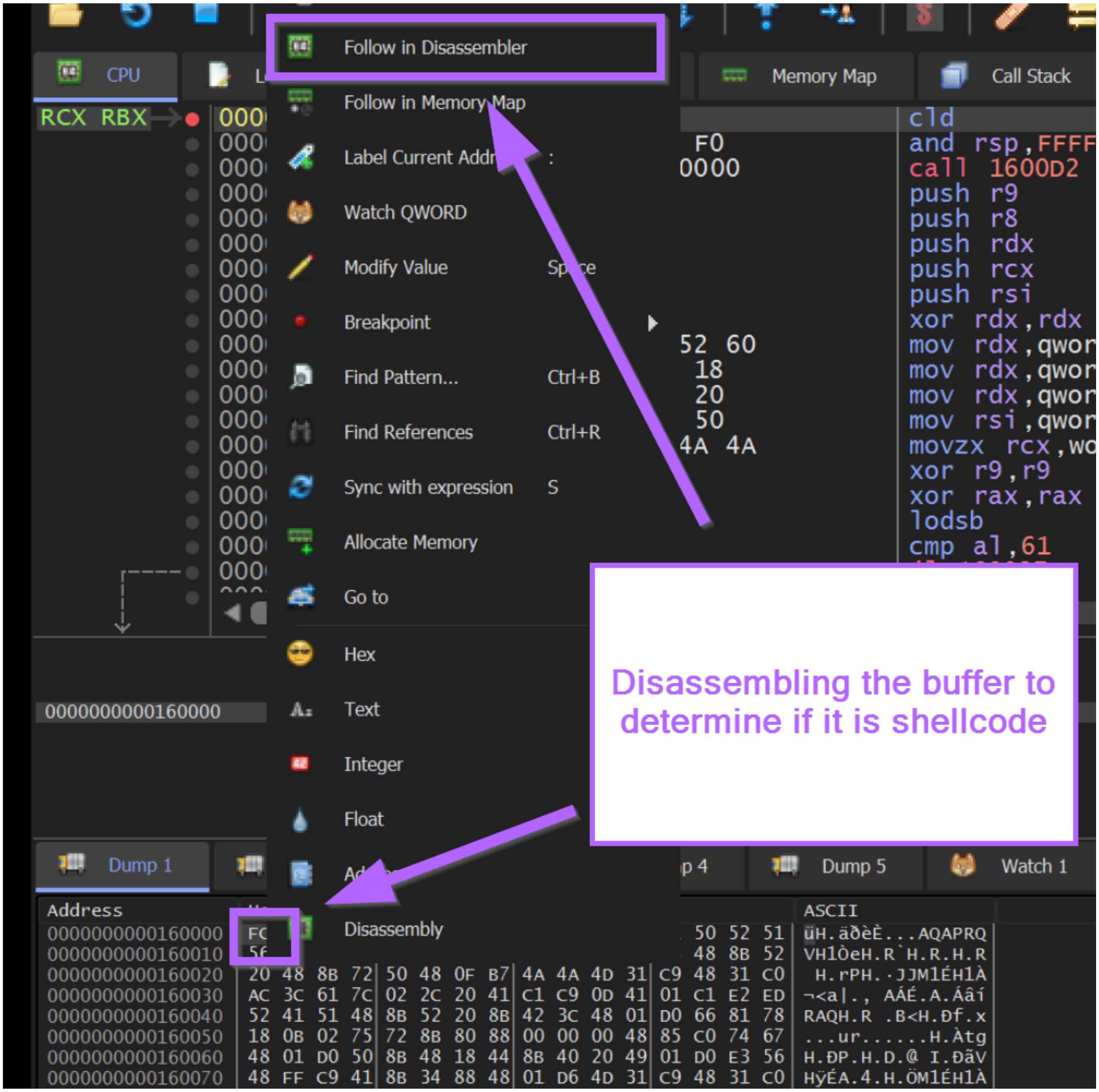


Validating Shellcode Using a Disassembler

Now that we have a reasonable assumption that the buffer contains shellcode, we can go ahead and try to disassemble it using X64dbg.

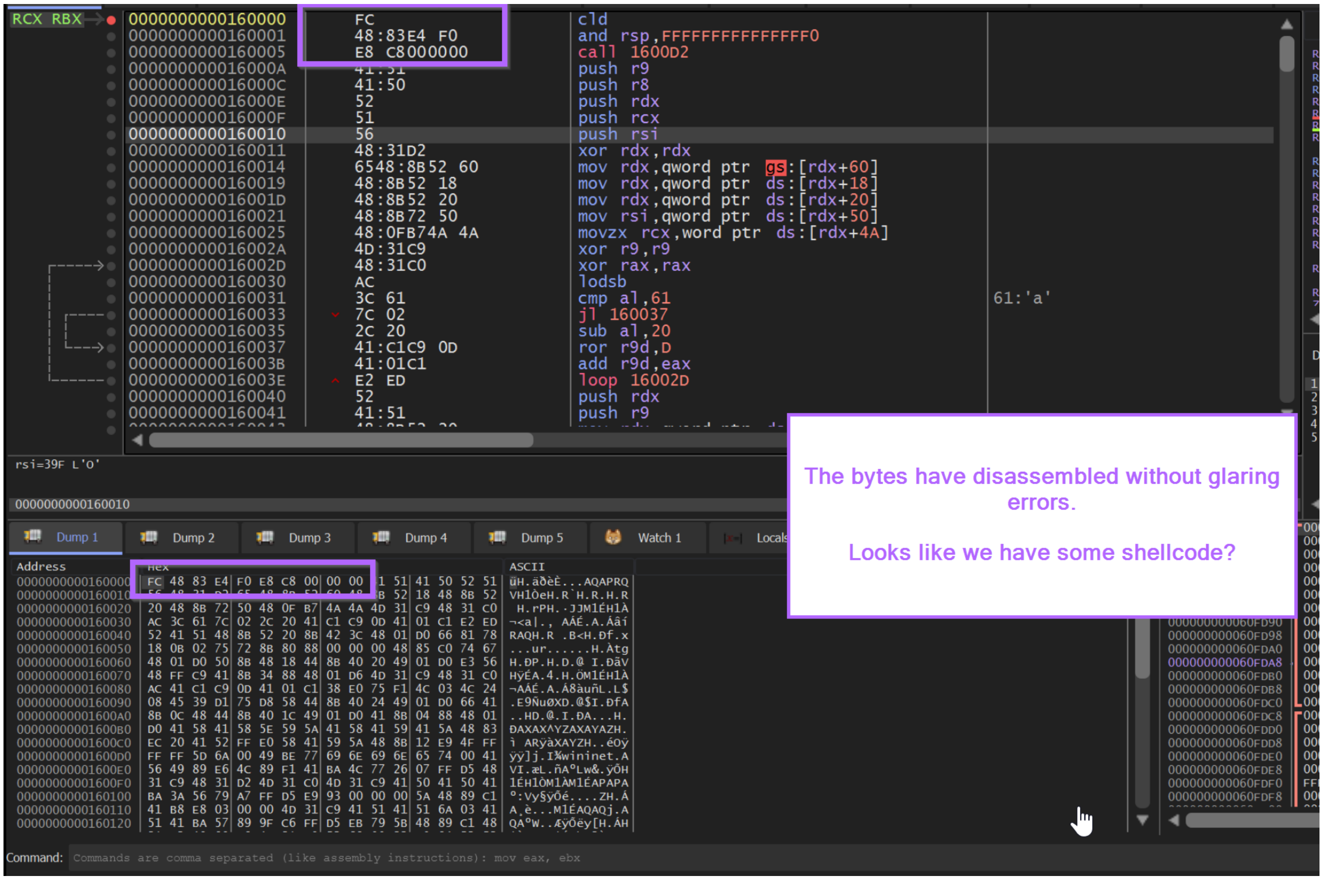
If we disassemble the code and there are no glaring errors, then there is a very high chance that we are looking at shellcode.

We can achieve this by selecting the first FC byte and `Follow in Disassembler`.



X64dbg will now attempt to disassemble the bytes from our buffer.

Below, we can observe the buffer disassembled in the top disassembly window. There appear to be no glaring errors, and there are valid function calls, loops and overall "normal" looking instructions.



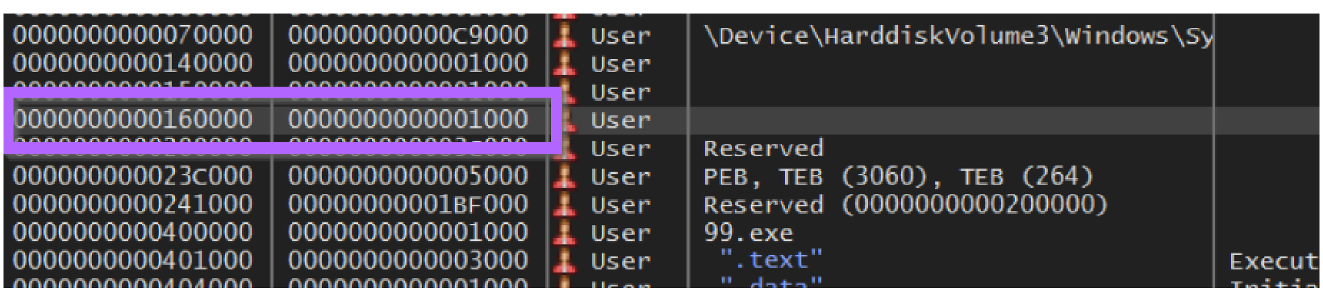
Final Validation Using SpeakEasy Emulator

We now have a very high suspicion that the buffer contains shellcode. So we can go ahead and emulate it using Speakeasy.

> We could also achieve the same thing with X64dbg, but for shellcode, this is a much more involved process that will be covered in a later blog.

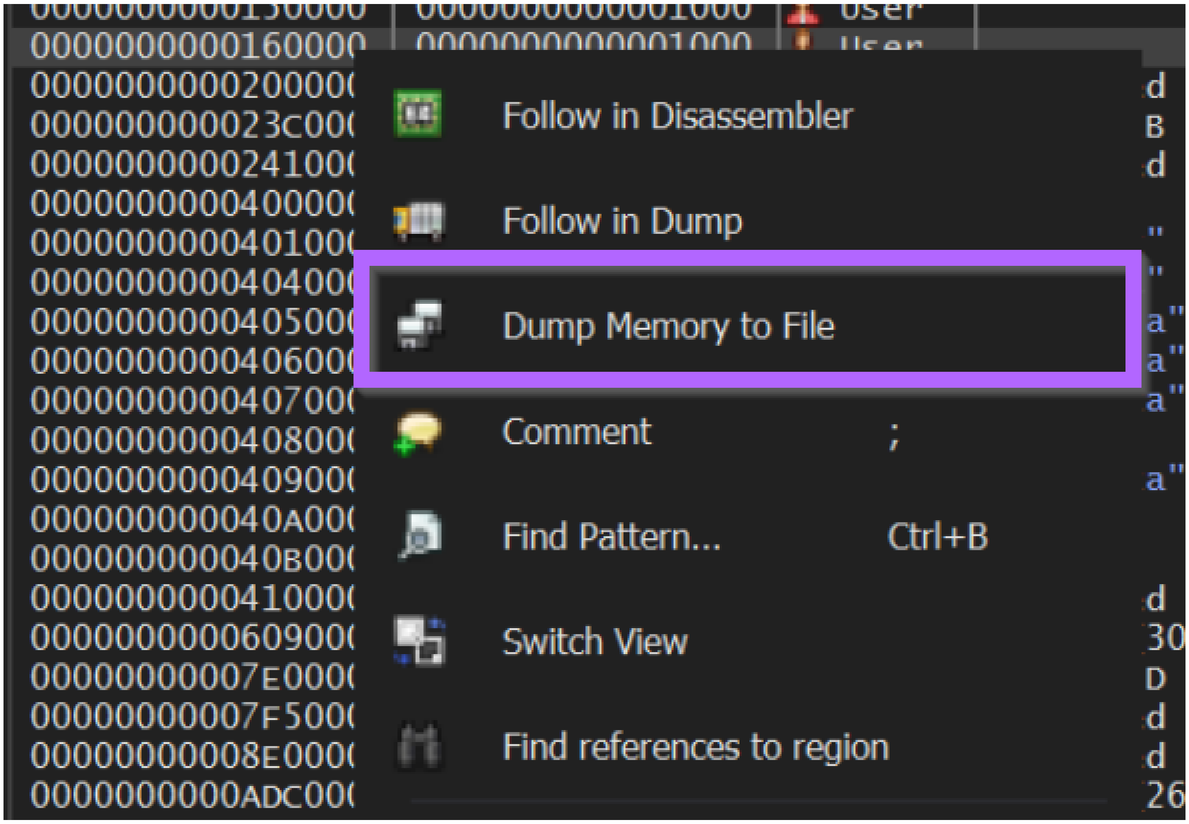
To emulate the shellcode using speakeasy, we first need to save it.

We can select our first FC byte, right-click and go to `Follow in Memory Map`



From here we can save the memory buffer to a file.

I will go ahead and save my file as `memdump.bin`.



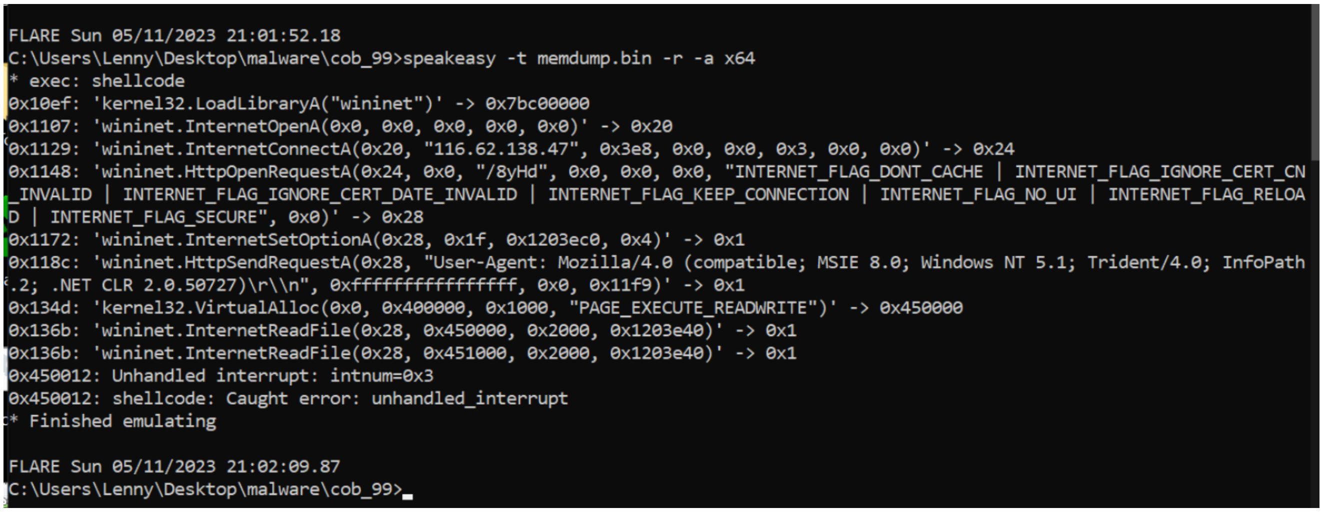
Emulating the Unpacked Shellcode with Speakeasy

With the shellcode buffer now saved to a file `memdump.bin`. We can go ahead and emulate the shellcode using Speakeasy.

We can do this with the command `speakeasy -t memdump.bin -r -a x64`

- `speakeasy` - Runs the speakeasy tool
- `-t` - Which file we want to use
- `-r` - (Raw) - Indicates that we are using shellcode
- `-a x64` - Indicates that our file contains 64-bit instructions. (we know this as we're using x64dbg and not x32dbg)

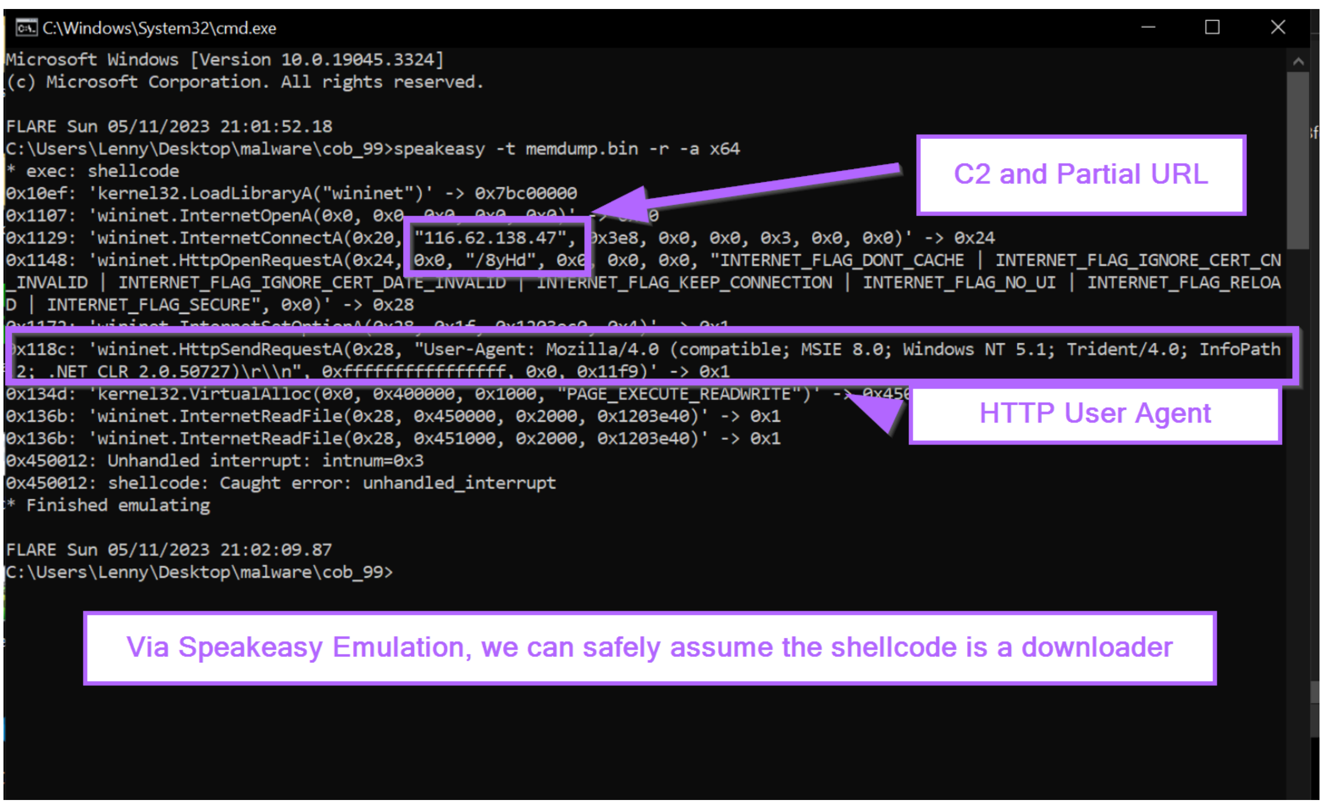
Upon running this command, the shellcode is emulated successfully and we are given a lot of information about it's functionality.



The Speakeasy output shows a C2 address of `116.62[.]138.47`, as well as a partial url of `/8yHd`.

We can also see references to a user agent of `User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727)\r\n`

(This user agent would be a great place to go hunting in proxy logs if you had them available)



Locating the Shellcode Decryption Function In Ghidra

At the point where the hardware breakpoint was first triggered, the primary executable was likely in the middle of the decryption function. We can use this information to locate the same decryption function within Ghidra.

From here, we can do some interesting things which are covered in the next 7 sections.

- Locating the Shellcode Decryption Function In Ghidra
- Identifying Decryption Routine Logic With ChatGPT
- Identifying the Decryption Key Using Ghidra
- Locating the Encrypted Shellcode Using Entropy
- Performing Manual Decoding Using Cyberchef
- Hunting For Additional Samples Using Decryption Bytes
- Creating a Yara Rule Using Decryption Code

These remaining sections are available for paid members of the site.

You can subscribe using the button below.

> Paid members will receive priority access to posts about Ghidra, Static Analysis and Advanced Debugging techniques.