Extracting the Cobalt Strike Config

blog.securehat.co.uk/malware-analysis/extracting-the-cobalt-strike-config-from-a-teardrop-loader

Extracting the Cobalt Strike Config from a TEARDROP Loader

This blog post will cover how to use dynamic analysis to extract the underlying Cobalt Strike config from a recent TEARDROP sample

---

Introduction

During the analysis of the SolarWinds supply chain compromise in 2020, a second-stage payload was identified and dubbed TEARDROP. Analysis of the discovered samples showed that TEARDROP ultimately loaded a Cobalt Strike beacon into memory. A good overview of the SolarWinds supply chain attack and follow on compromise activity can be found here:

Sunburst: Supply Chain Attack Targets SolarWinds Users

symantec

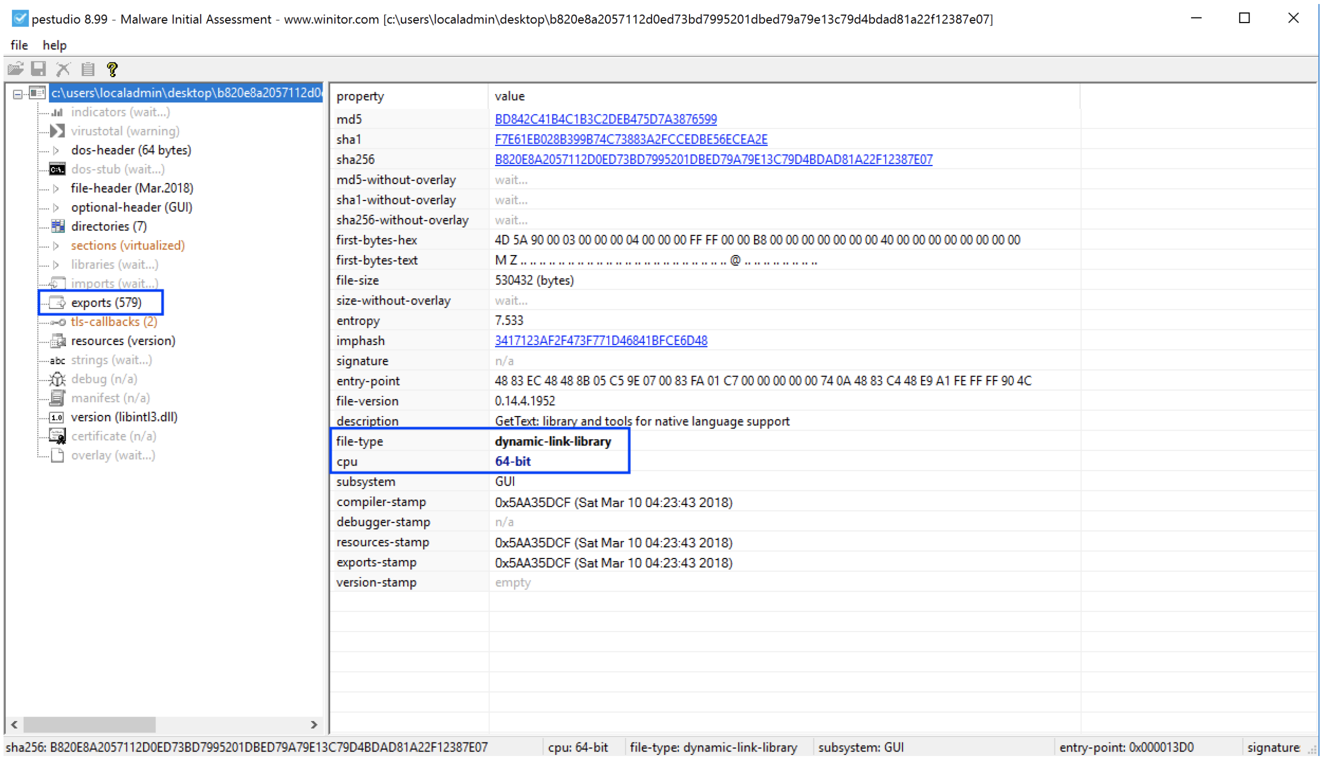
Despite wide discussion and coverage in security industry, actual samples of the TEARDROP malware were not initially made publicly accessible. However on 05-02-2021, the two TEARDROP samples referenced in the Symantec blog above were uploaded to VirusTotal.

For the remainder of this blog post we will analyse one of the uploaded TEARDROP samples with the goal of extracting the underlying Cobalt Strike config.
This blog post is not an exhaustive analysis of the TEARDROP loader and its behaviour, it focuses purely on extracting the Cobalt Strike beacon and the associated config information.

Analysis of the TEARDROP Sample

Initial Analysis

Loading the sample into PEStudio we can see that we're dealing with a 64bit DLL with a lot of DLL exports:



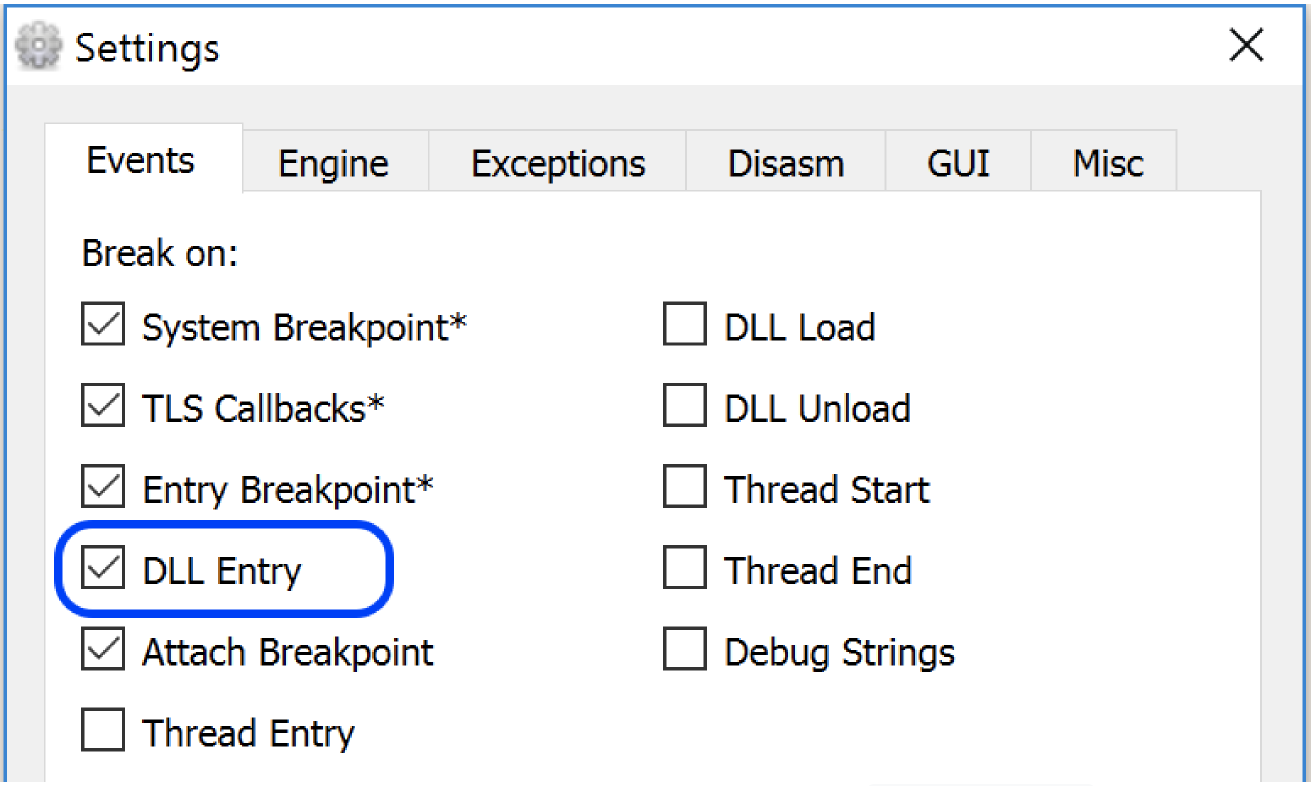
The high number of DLL exports makes it slightly more challenging to move onto dynamic analysis as we first need to identify which export we want to analyse. To keep this blog post digestible, we will skip this step for now and instead we can use the information provided in the Symantec report (as linked above) to give us the correct DLL export for the starting point of our analysis: `Tk_CreateImageType`

How to Analyse a Specific DLL Export in x64dbg

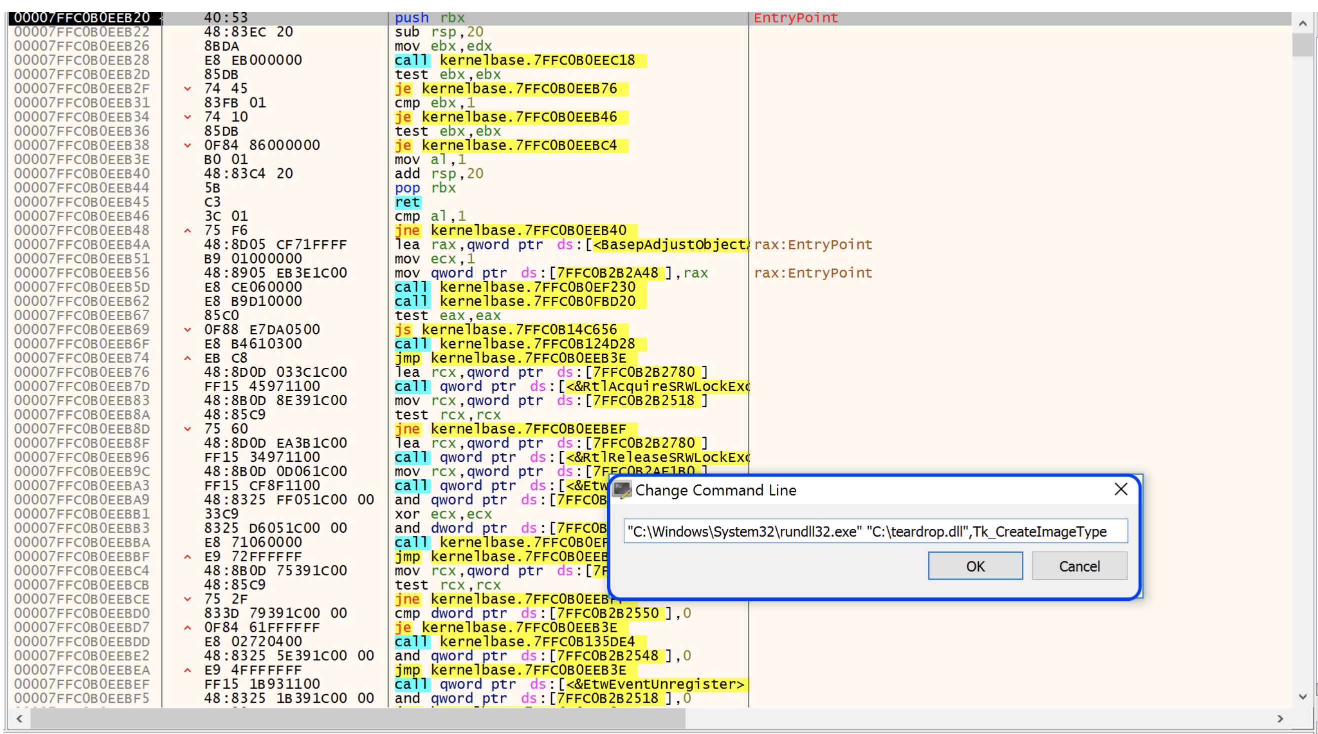
Now that we know we want to analyse the `Tk_CreateImageType` export, we need to get to that location in our favourite debugging tool. This is a little more challenging to do with a DLL as we can't directly call the export using x64dbg, but fortunately it's still easy enough to achieve with the following steps:

1. 1.
   Open `rundll32.exe` in x64dbg
2. 2.
   Configure x64dbg to automatically break on DLL entry
3. 3.
   Configure the rundll32.exe command line arguments to call our DLL and desired export
4. 4.
   Breakpoint on the entrypoint of our desired DLL export and run until we get to that location

First of all we're going to open `rundll32.exe` in x64dbg (File -> Open -> `C:\Windows\System32\rundll32.exe`) and then also configure x64dbg to automatically pause on every DLL entry point (Options -> Preferences -> DLL Entry):



Now we can change the command line arguments passed to `rundll32.exe` so that when it's launched it will execute our DLL at the `Tk_CreateImageType` export. To do this go to File -> Change Command Line:



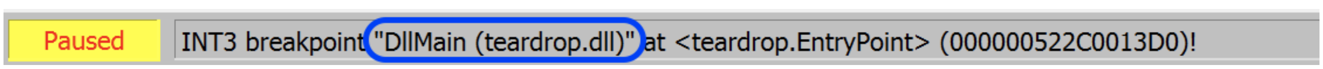
After hitting "OK", and restarting the debugging process (Debug -> Restart), we can now allow execution to proceed knowing that x64dbg will automatically breakpoint at the entrypoint of every DLL, including our target DLL. Keep pressing F9 (or Debug -> Run) while keeping an eye on current DLL location listed in the bottom bar of x64ddb until we see that we've hit our target DLL (teardrop.dll in this case):

In the screenshot above we can see that we've successfully hit `teardrop.dll` in x64dbg and specifically we are at the first TLS Callback of the DLL. Thread Local Storage (TLS) callbacks execute before the main entry point of PE files and have both legitimate and non-legitimate use-cases. Some malware samples have been known to leverage TLS Callbacks as a way to check if the process is being analysed before the main execution of the sample is reached:

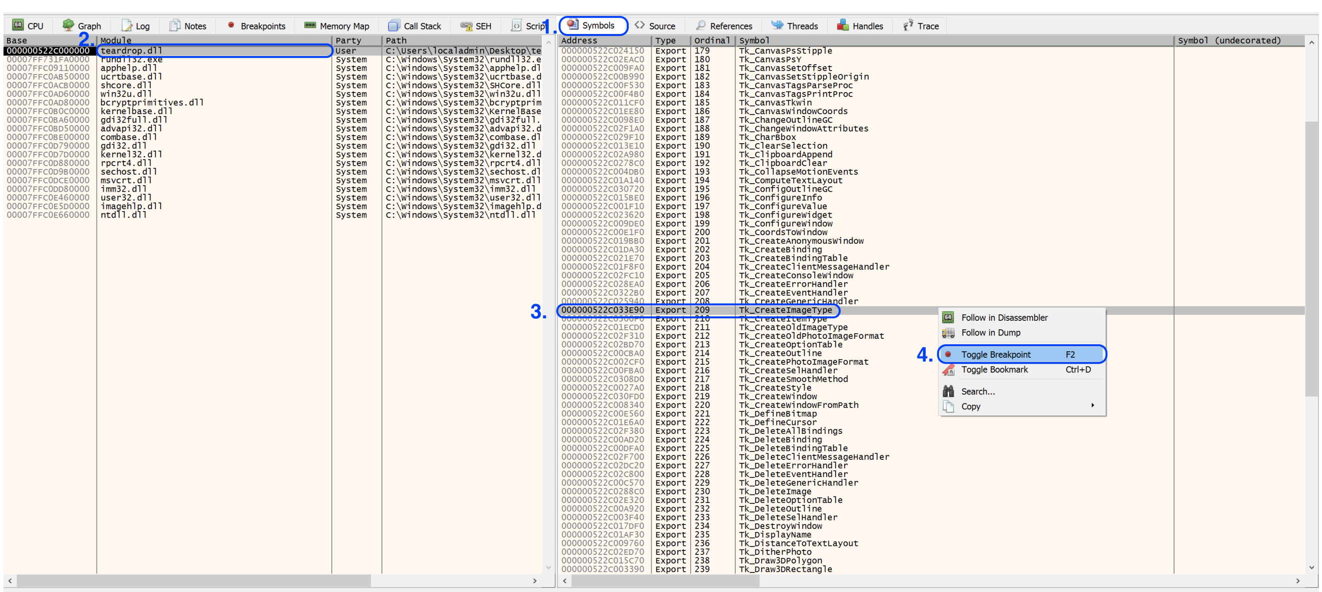
https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html

www.fireeye.com

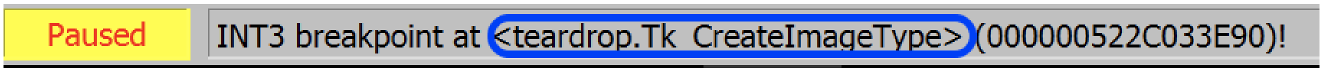
For now we will move past the TLS callbacks and circle back later on if we need to dig deeper into the sample. Keep pressing F9 until we reach `DLLMain` of `teardrop.dll` :



Now that we're in the target DLL, all we need to do is breakpoint on the `Tk_CreateImageType` export. To do this we can go to the `Symbols` tab, select `teardrop.dll` in the left pane and then right click on the correct export in the right pane and select `Toggle Breakpoint` :



Finally we can once again allow debugging to continue (Debug -> Run) until the status bar in the bottom of the x64dbg window shows that we've reached the start of the `Tk_CreateImageType` export:



At this point you may want to change the preferences of x64dbg so that it no longer breaks on every DLL entry, this will make debugging easier and we can easily jump back to the CreateImageType export now that we have a breakpoint set.

Tracking Memory Activity

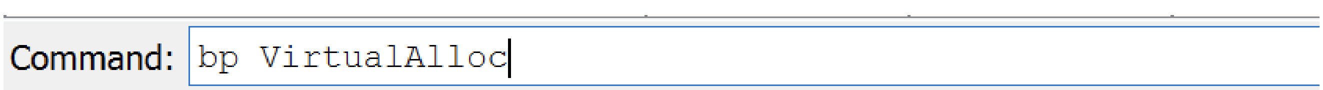
As mentioned at the start of this blog post, the main aim of the TEARDROP loader is to load a Cobalt Strike beacon into memory on the victim machine. Using this knowledge we can make an assumption that by breakpointing on memory related API calls we should hopefully be able to find the Cobalt Strike beacon being loaded into memory.

To start off this analysis route, we will set breakpoints on the following APIs:

> **VirtualAlloc** - allocate memory regions in the current process
>
> **VirtualProtect** - used to change the protection on a memory region
>
> **VirtualQuery** - gather information about a memory region in the current process
>
> **VirtualFree** - releases a memory region in the current process

If suspect that a sample may do some form of process injection, we may also want to set breakpoints on APIs such as **VirtualAllocEx, OpenProcess, CreateRemoteThread, CreateProcessInternalW** etc. However for this sample these breakpoints won't be necessary.

The easiest way to do this is to type `bp <API>` in the command window towards the bottom of the x64dbg window:



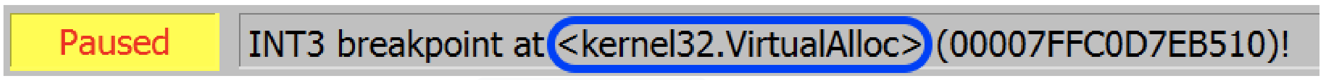
Now that we have our breakpoints set, the initial plan is to follow the steps below:

1. 1.
   Break on calls to VirtualAlloc
2. 2.
   Track the allocated memory regions returned by the API call
3. 3.
   Inspect these regions as execution continues to identify interesting content being loaded into memory

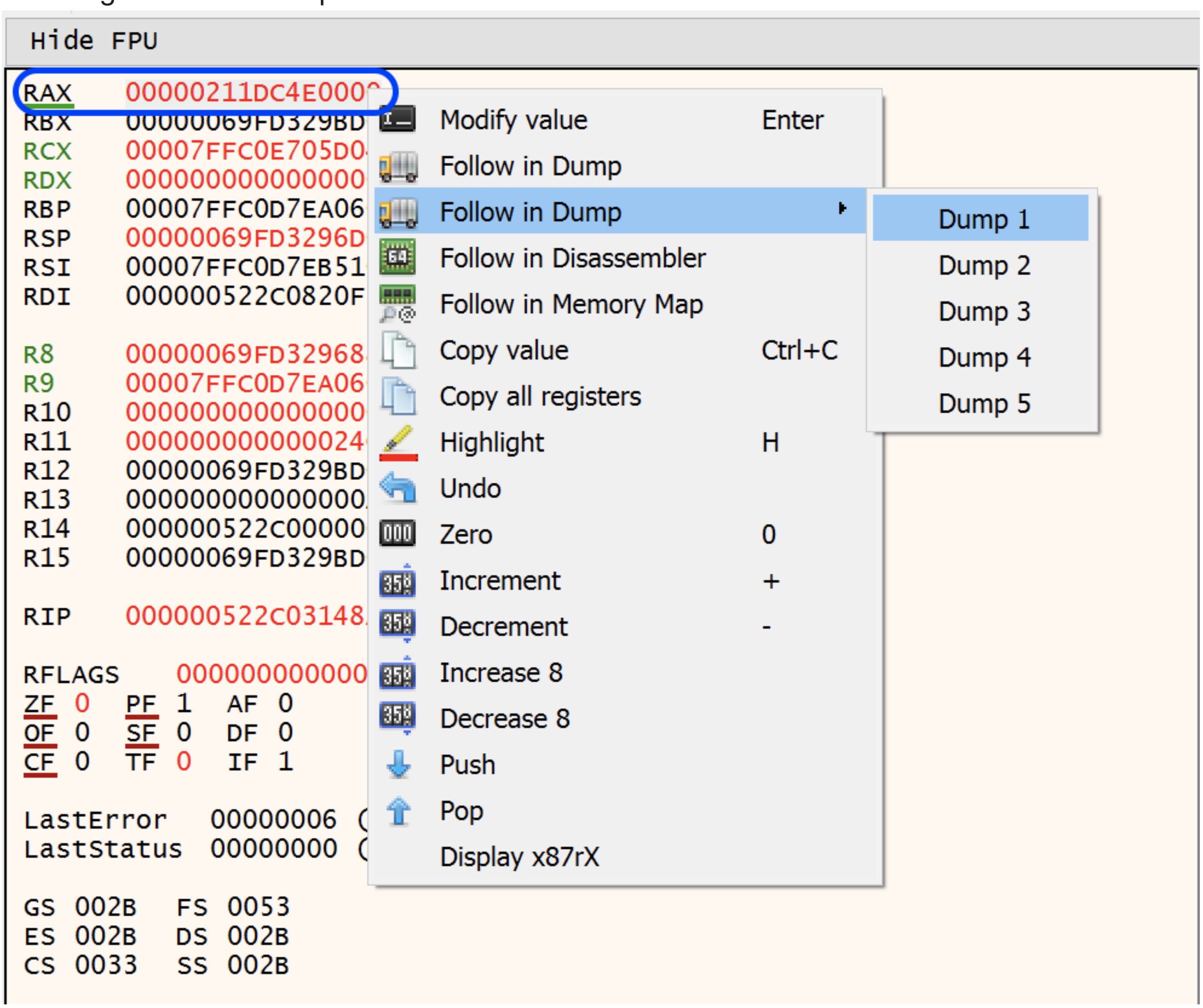
In its default configuration Cobalt Strike beacons are loaded into memory in the form of a PE file, so by tracking the contents of allocated memory regions we should hopefully be able to spot the Cobalt Strike PE file being loaded into memory prior to execution.

Tracking the Memory Regions Allocated by VirtualAlloc

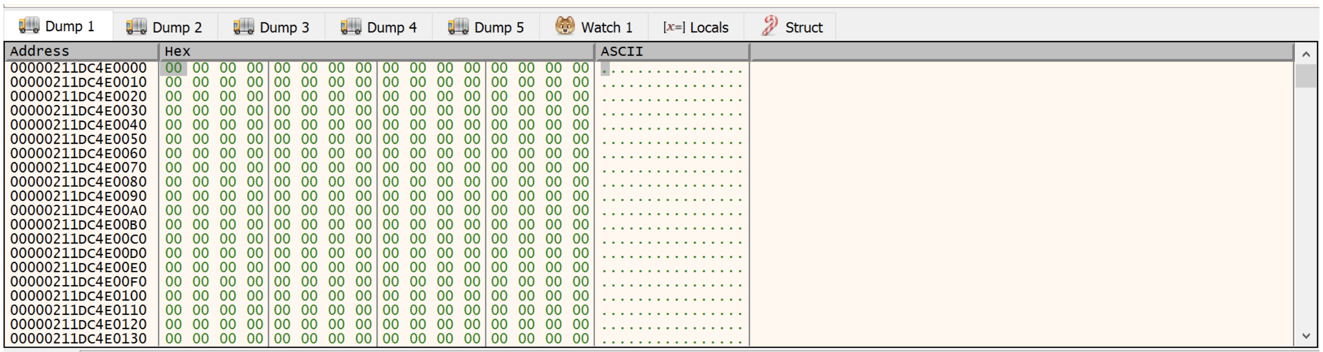
Allow the execution of the program to continue until we hit the first instance of `VirtualAlloc` being called:



Inspecting the documentation for `VirtualAlloc` shows us that the return value of a successful call is the "base address of the allocated region of pages". By clicking `Debug -> Return to User Code` after the breakpoint on `VirtualAlloc` we can allow the API call to complete and the base address of the new memory region should be now be stored in the RAX register. We can follow this memory region by right clicking on the RAX register and selecting "Follow in Dump":

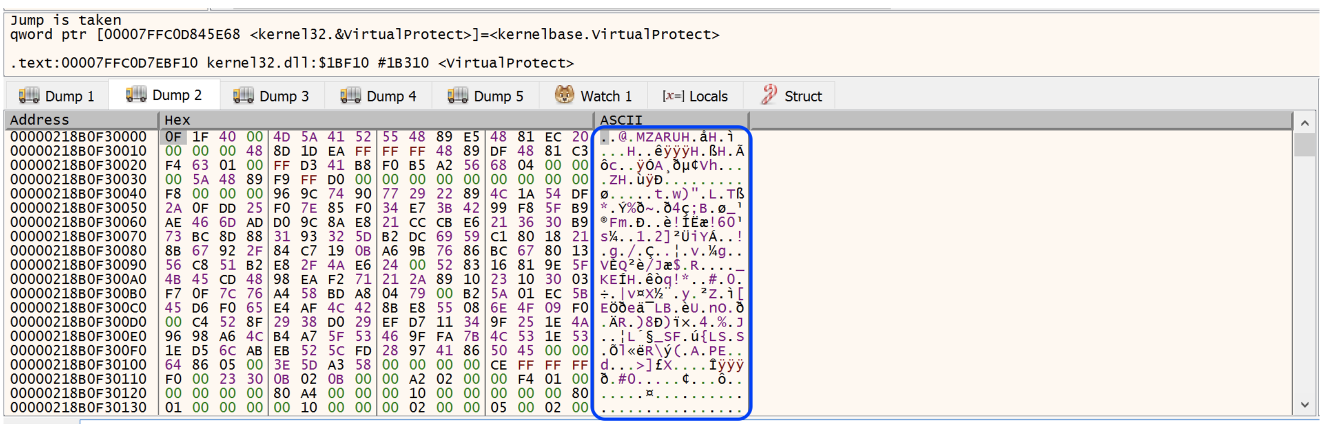


After clicking on **Follow in dump** we can see the allocated memory region in the x64dbg dump window towards the bottom on the screen:



As we allow the execution of the program to continue we should see the contents of this memory region change, and hopefully we will see a PE file appear in this window relating to the Cobalt Strike beacon. If we hit subsequent `VirtualAlloc` calls, we can follow the same process as above and track the regions in the different dump tabs.

After allowing the execution of the program to continue, tracking a number of allocated memory regions, and allowing execution to continue over a number of `VirtualProtect` breakpoints, we can spot the start of a PE file in one of the memory regions:



Dumping the PE File to Disk

Almost any time we see a PE file being loaded into memory during malware analysis, it's worth dumping it to disk and analysing it further. In this case we're hoping that this is our Cobalt Strike beacon, so to progress the analysis of this sample we can dump this region of memory to disk.

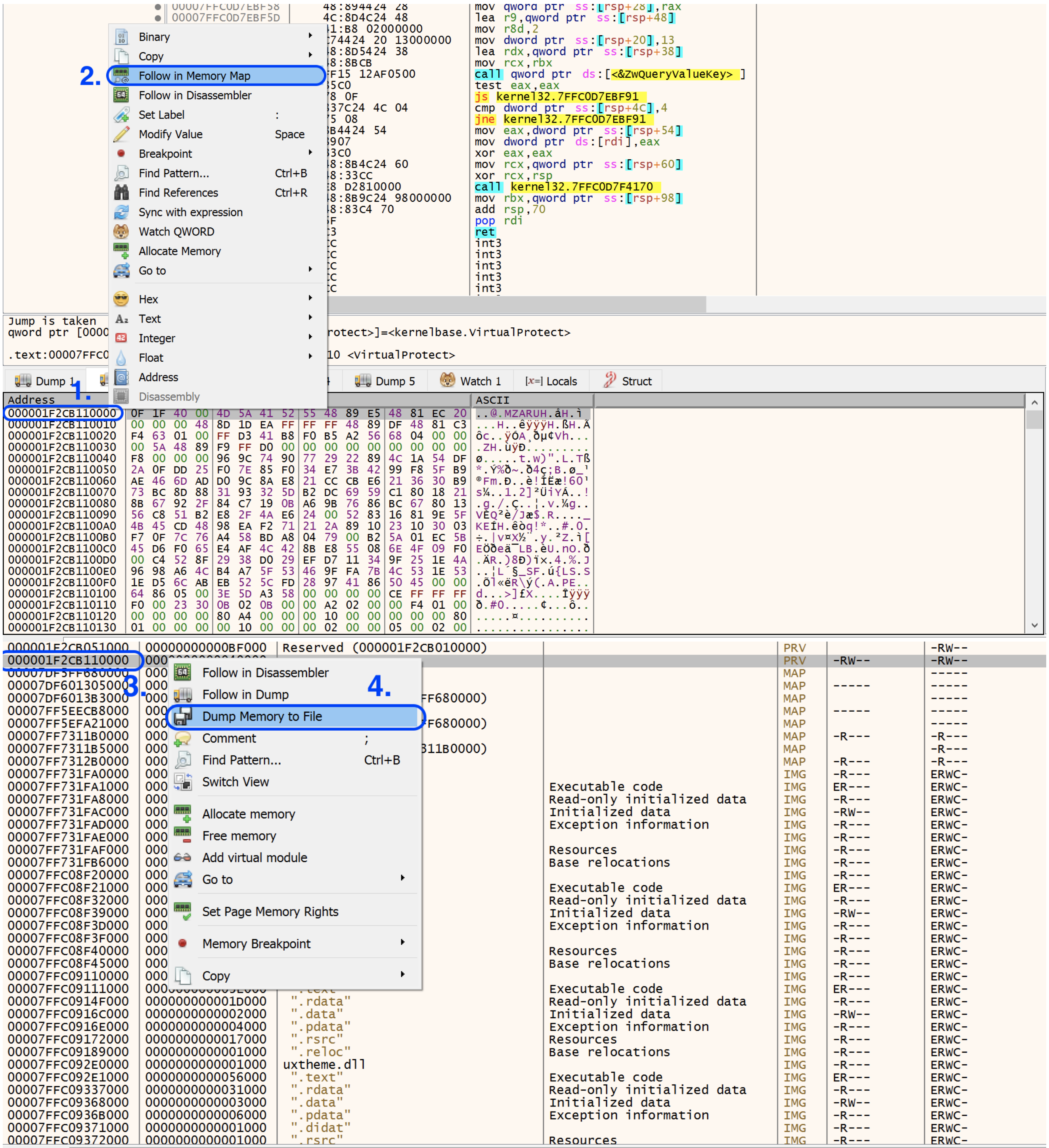
To do this we can do the following:

1. 1.
   Right click on the desired memory region in the dump tab

2. 2.
   Click "Follow In Memory Map"

3. 3.
   In the new window that appears, right click on the highlighted memory region

4. 4.
   Click "Dump Memory to File"



Now that we have the PE file on disk, the final step is to attempt to extract the Cobalt Strike config information so that we can identify IOCs and configuration information. Although we haven't confirmed that this PE file is definitely a Cobalt Strike beacon at this point, it's a safe bet when we take into account that the Symantec blog post reported that the sample will ultimately load a beacon into memory.

Fortunately it's very easy to check by using Sentinal One's Cobalt Strike config extractor:

GitHub - Sentinel-One/CobaltStrikeParser

GitHub

Inspecting the parser code we can see that it looks for one of three byte patterns in order to identify the presence of a Cobalt Strike config. If any of the byte patterns are found, then the parser will attempt to decode and print the configuration information of the Cobalt Strike beacon. The byte patterns that the parser looks for are:

```
START_PATTERNS = {
3: b'\x69\x68\x69\x68\x69\x6b..\x69\x6b\x69\x68\x69\x6b..\x69\x6a',
4: b'\x2e\x2f\x2e\x2f\x2e\x2c..\x2e\x2c\x2e\x2f\x2e\x2c..\x2e'
}
START_PATTERN_DECODED = b'\x00\x01\x00\x01\x00\x02..\x00\x02\x00\x01\x00\x02..\x00'
```

Copied!

The first two patterns reflect the two different XOR keys used in version 3 (0x69) and version 4 (0x2e).

Running the parser over the PE file that we extracted from the TEARDROP sample confirms that the file is a Cobalt Strike beacon and that we can successfully extract the config:

```
-> % python parse_beacon_config.py teardrop_pefile.bin
BeaconType - HTTPS
Port - 443
SleepTime - 14400000
MaxGetSize - 1049217
Jitter - 23
MaxDNS - 255
C2Server - infinitysoftwares[.]com,/files/information_055.pdf
UserAgent - Not Found
HttpPostUri - /wp-admin/new_file.php
Malleable_C2_Instructions - Remove 313 bytes from the end
Remove 324 bytes from the beginning
XOR mask w/ random key
```

14

HttpGet_Metadata - Not Found

15

HttpPost_Metadata - Not Found

16

SpawnTo - b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

17

PipeName -

18

DNS_Idle - 208.67.220.220

19

DNS_Sleep - 0

20

SSH_Host - Not Found

21

SSH_Port - Not Found

22

SSH_Username - Not Found

23

SSH_Password_Plaintext - Not Found

24

SSH_Password_Pubkey - Not Found

25

HttpGet_Verb - GET

26

HttpPost_Verb - POST

27

HttpPostChunk - 0

28

Spawnto_x86 - %windir%\syswow64\print.exe

29

Spawnto_x64 - %windir%\sysnative\msiexec.exe

30

CryptoScheme - 0

31

Proxy_Config - Not Found

32

Proxy_User - Not Found

33

Proxy_Password - Not Found

34

Proxy_Behavior - Use IE settings

35

Watermark - 943010104

36

bStageCleanup - True

37

bCFGCaution - False

38

KillDate - 0

39

bProcInject_StartRWX - False

40

bProcInject_UseRWX - False

41

bProcInject_MinAllocSize - 8493

42

ProcInject_PrependAppend_x86 - b'\x90\x90'

43

Empty

44

ProcInject_PrependAppend_x64 - b'\x0f\x1f\x00'

45

Empty

46

ProcInject_Execute - ntdll:RtlUserThreadStart

47

CreateThread

48

NtQueueApcThread

49

SetThreadContext

50

ProcInject_AllocationMethod - NtMapViewOfSection

51

bUsesCookies - True

52

HostHeader -

Copied!

References

Sunburst: Supply Chain Attack Targets SolarWinds Users

symantec

GitHub - Sentinel-One/CobaltStrikeParser

GitHub