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 <u>Cobalt Strike</u> 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 <u>one of the uploaded TEARDROP samples</u> 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:

help									
c:\users\localadmin\desktop\b820e8a20	57112d0 property	value							
-Jul indicators (wait)	md5	BD842C41B4C1B3C2DEB475D7A3876599							
virustotal (warning)	sha1	F7E61EB028B399B74C73883A2FCCEDBE56ECEA2E							
dos-header (64 bytes)	sha256	B820E8A2057112D0ED73BD7995201DBED79A79E13C79D4BDAD81A22F12387E07							
dos-stub (wait)	md5-without-overlay	wait							
 file-header (Mar.2018) 	sha1-without-overlay	wait							
 b optional-neader (GUI) disastasion (7) 	sha256-without-overlay	wait							
airectories (7)	first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00							
 b libraries (wait) 	first-bytes-text	M Z							
imports (wait)	file-size	530432 (bytes)							
exports (579)	size-without-overlay	wait							
	entropy	7.533							
resources (version)	imphash	3417123AF2F473F771D46841BFCE6D48							
abc strings (wait)	signature	n/a							
∰ debug (n/a)	entry-point	48 83 EC 48 48 8B 05 C5 9E 07 00 83 FA 01 C7 00 00 00 00 00 74 0A 48 83 C4 48 E9 A1 FE FF F9 0 4C							
🗐 manifest (n/a)	file-version	0.14.4.1952							
version (libintl3.dll)	description	GetText: library and tools for native language support							
certificate (n/a)	file-type	dynamic-link-library							
🗋 overlay (wait)	cpu	64-bit							
	subsystem	GUI							
	compiler-stamp	0x5AA35DCE (Sat Mar 10 04:23:43 2018)							
	debugger-stamp	n/a							
	resources-stamp	0x5AA35DCE (Sat Mar 10 04:23:43 2018)							
	exports-stamp	0x5AA35DCF (Sat Mar 10 04:23:43 2018)							
	version-stamp	empty							

sha256.B820EBA2007112D0ED738D795201DBED79A79E13C7904BDAD81A22F12387E07 cpus 64-bit file-type: dynamic-link-library subsystem: GUI entry-point: 0x000013D0 signature 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):

Paused INT3 breakpoint "TLS Callback 1 (teardrop.dll)" at teardrop.000000522C032BE0 (000000522C032BE0)!

In the screenshot above we can see that we've successfully hit <u>teardrop.dll</u> 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 nonlegitimate 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 :

PausedINT3 breakpoint "DIIMain (teardrop.dll)" at <teardrop.EntryPoint> (00000522C0013D0)!Now that we're in the target DLL, all we need to do is breakpoint on theTk_CreateImageTypeexport. To do this we can go to the Symbols tab, selectteardrop.dllin the left pane and then right click on the correct export in the right paneand selectToggleToggleBreakpoint



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:

Paused INT3 breakpoint at <teardrop.Tk CreateImageType>(000000522C033E90)!

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:

Command: bp VirtualAlloc

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:

Paused INT3 breakpoint at <kernel32.VirtualAlloc>(00007FFC0D7EB510)!

Inspecting the <u>documentation</u> for <u>VirtualAlloc</u> shows us that the return value of a successful call is the "base address of the allocated region of pages". By clicking <u>Debug</u> -> <u>Return to User Code</u> after the breakpoint on <u>VirtualAlloc</u> 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":

Hide	FPU						
RAX	00000211DC4E0	00^	Modificualua	Entor	1		
RBX	00000069FD329		Moully value	Enter			
RDX	000000000000000000000000000000000000000	00	Follow in Dump				
RBP	00007FFC0D7EA	06 🚛	Follow in Dump	•	Dump 1		
RSP	00000069FD329	6D	Follow in Disassembler		Dump 2		
RSI	00007FFC007EB				Dump 2		
KDI	0000003220082		Follow in Memory Map		Dump 3		
R8	00000069FD329	68 🖺	Copy value	Ctrl+C	Dump 4		
R9	00007FFC0D7EA	06	Copy all registers		Dump 5		
R10 R11	000000000000000000000000000000000000000	24	Highlight	н			
R12	00000069FD329	BD 💻					
R13	000000000000000000000000000000000000000	00, 🔽	Undo				
R14	000000522c000	00 000	Zero	0			
R15	00000069FD329	BD .	Increment	+			
RIP	000000522c031	48. 355	Decrement	-			
			Justice of the second s				
RFLAG	s000000000000000000000000000000000	00 358	Increase 8				
$\frac{2F}{0F}$ 0	SE 0 DE 0	358	Decrease 8				
CF 0	TF 0 IF 1	₽.	Push				
LastE	rror 0000000		Рор				
LastS	tatus 0000000) Č	Display x87rX				
GS 002B FS 0053 ES 002B DS 002B CS 0033 SS 002B							

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

🚛 Dump 1	🚛 Dum	p 2	🚛 Dum	р 3		Dump ·	4	🛄 D	ump 5	;	۵ 🥹	Vatch	1	[x =]	Locals	Struct	
Address	He	х										AS	CII				^
00000211DC4E0	00 00	00	00 00 00	00	00 00	00 0	00 00	0 00	00 (0 00	0 00)					
00000211DC4E0	010 00	00	00 00 00	00	00 00	00 0	00 0	0 00	00 (0 00	0 00	1					
00000211DC4E0	020 00	00	00 00 00	00	00 00	00 0	00 0	0 00	00 (0 00	0 00						
00000211DC4E0	030 00	00	00 00 00	00	00 00	00 0	00 0	0 00	00	0 0	0 00						
00000211DC4E0	040 00	00	00 00 00	00	00 00	00 0	00 0	0 00	00 (0 00	0 00						
00000211DC4E0	050 00	00	00 00 00	00	00 00	00 00	00 0	0 00	00 (0 0	0 00						
00000211DC4E0	060 00	00	00 00 00	00	00 00	00 00	00 0	0 00	00 0	00 0	0 00	• •					
00000211DC4E0	0/0 00	00	00 00 00	00	00 00	00	00 0	0 00	00 0	000	0 00	2			* * * * * *		
00000211DC4E0	00 080	00		00	00 00		00 0	0 00	00 0	0 00	0 00	1			*****		
00000211DC4E0	90 00	00		00	00 00		00 0	0 00	00 0	0 00	0 00	2					
000002110C4E0		00							00 0			(*****		
000002110C4E0		00										(• •					
000002110C4E0		00										(• •					
000002110C4E0		00										(• •					
000002110C4E0		00										(· ·					
000002110C4E0		00										(· ·					
000002110C4E0		00		00					00 0			(··					
00000211DC4E0		00		00	00 00				00	no o							
00000211DC4E0	30 00	00		00	00 00		õõ õ	õ õõ	00	no o	õ õ						\mathbf{v}
SSSSSSI DCTLO			00 00 00		00 00	100		00			000						_

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:

Jump is taken qword ptr [00007FFC0D845E68 <kernel32.&virtualprotect>]=<kernelbase.virtualprotect></kernelbase.virtualprotect></kernel32.&virtualprotect>								
.text:00007FFC0D7EBF10 kernel32.dll:\$1BF10 #1B310								
💷 Dump 1 💷 Dump	p 2 💷 Dump 3	🚛 Dump 4 🛛 💷 Dump 5	🛞 Watch 1 [x=] Locals	s 🐉 Struct				
Address Hex	x		ASCII					
0000021880F3000 0F 0000021880F30010 0C 0000021880F30010 0F 0000021880F30020 F4 0000021880F30040 F8 0000021880F30050 A 0000021880F30060 AE 0000021880F30080 AE 0000021880F30080 BR 0000021880F30080 F3 0000021880F30100 64 0000021880F30110 F0 0000021880F30120 00 0000021880F30120 00 0000021880F30120 01 0000021880F30120 01 0000021880F30120 01	$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	EC 20 @ M.A.RAUH.AH. 81 C3 A	1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2				

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:

1 START_PATTERNS = { 2 3: b'\x69\x68\x69\x68\x69\x6b..\x69\x6b\x69\x68\x69\x68\x69\x6a', 3 4: b'\x2e\x2f\x2e\x2f\x2e\x2c..\x2e\x2c\x2e\x2f\x2e\x2c..\x2e' 4 } 5 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:

1

-> % python parse_beacon_config.py teardrop_pefile.bin

2 BeaconType - HTTPS 3 Port - 443 4 SleepTime - 14400000 5 MaxGetSize - 1049217 6 Jitter - 23 7 MaxDNS - 255 8 C2Server - infinitysoftwares[.]com,/files/information_055.pdf 9 UserAgent - Not Found 10 HttpPostUri - /wp-admin/new file.php 11 Malleable_C2_Instructions - Remove 313 bytes from the end 12 Remove 324 bytes from the beginning 13 XOR mask w/ random key

```
14
HttpGet_Metadata - Not Found
15
HttpPost_Metadata - Not Found
16
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
```

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