Nitrogen Dropping Cobalt Strike – A Combination of “Chemical Elements”

By Maurice Fielenbach

Published: 2026-03-30 · Archived: 2026-04-05 17:47:44 UTC

First detected in September 2024 and initially targeting the United States and Canada, the Nitrogen ransomware group has since expanded its reach into parts of Africa and Europe. Many of their victims remain absent from Nitrogen’s public ransomware blog and likely never will be listed. At the time of writing, ransomware.live reports 21 known victims of Nitrogen. Notably, indicators of this malware family surfaced as early as 2023, suggesting links to other ransomware infections.

In this post, we’ll share details from a recent, non-published, Nitrogen ransomware case, including how the attackers gained initial access, their lateral movement across systems (confirmed through user access logs), and how they attempted to cover their tracks by clearing logs. By examining Windows Error Reporting (WER) and crash dump files, we uncovered a Cobalt Strike configuration, along with a Cobalt Strike C2 team server and the attacker’s use of a pivot system.

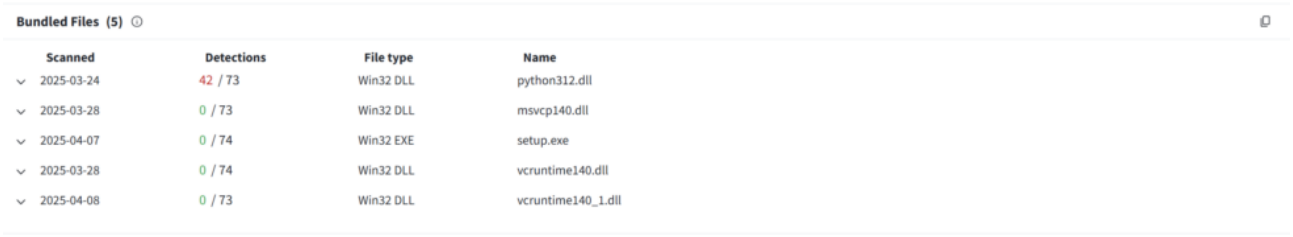
Malvertising to Gain Initial Access

In recent months, threat actors have leveraged targeted Nitrogen-themed malvertising, bundling malicious code within tools that appear legitimate. For instance, thedfirreport documented a Nitrogen campaign that distributed a fake “Advanced IP Scanner,” ultimately leading to a BlackCat ransomware infection. Similar malvertising tactics have been observed with disguised versions of FileZilla and WinRAR.

During one of our recent investigations, a user searching for “WinSCP download” via Microsoft Edge clicked on a suspicious ad served through Bing. The ad redirected them from `ftp-winscp[.]org` to a compromised WordPress site hosting a malicious WinSCP ZIP file — establishing the initial foothold (“beachhead”) in a broader attack chain.

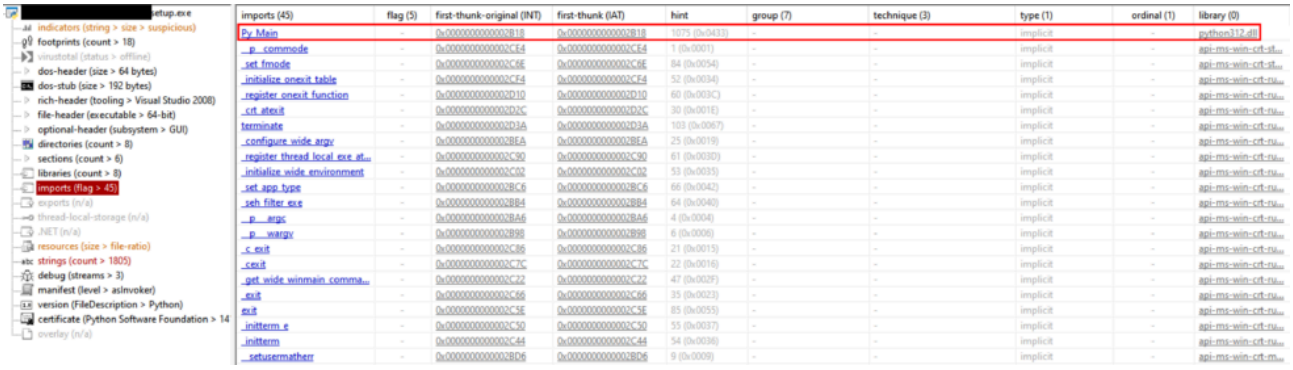
WinSCP ZIP download detected in Microsoft Edge browser history on patient zero

Within the ZIP archive, `WinSCP-6.3.6-Setup.zip` (SHA-256: `fa3eca4d53a1b7c4cfcd14f642ed5f8a8a864f56a8a47acbf5cf11a6c5d2afa2`), several files were bundled: a malicious `python312.dll`, three legitimate DLLs, and a renamed `python.exe` labeled `setup.exe`. Once the user ran `setup.exe`, DLL sideloading occurred — WinSCP was installed in the foreground while the malicious DLL was loaded into the running process.



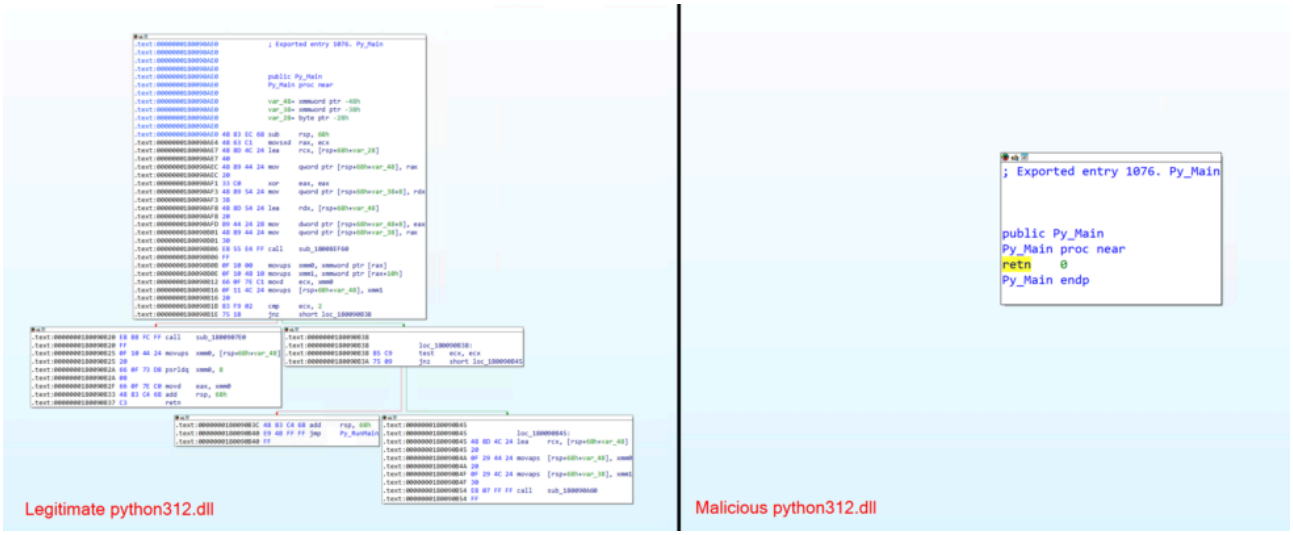
Malicious WinSCP ZIP bundled files

As indicated by the imports in setup.exe, python312.dll is invoked as a dependency at runtime, triggering the execution of the malicious DLL. Because the file path for the DLL is not defined with an absolute file path in setup.exe, Windows relies on its default DLL search order: it first checks the application's directory, then the system directory, the Windows directory, and finally the PATH environment variable if the DLL is still not found.



setup.exe imports

Closer inspection of the malicious DLL, also referenced as the "NitrogenLoader," shows that it mirrors the same exports and ordinals found in a genuine Python DLL. For example, it includes the `Py_Main` export mentioned in the `setup.exe` import table. However, whereas a legitimate `python312.dll` (for instance, `278f22e258688a2afc1b6ac9f3aba61be0131b0de743c74db1607a7b6b934043`) features authentic logic, the malicious file uses a minimalist approach, returning null instructions instead.



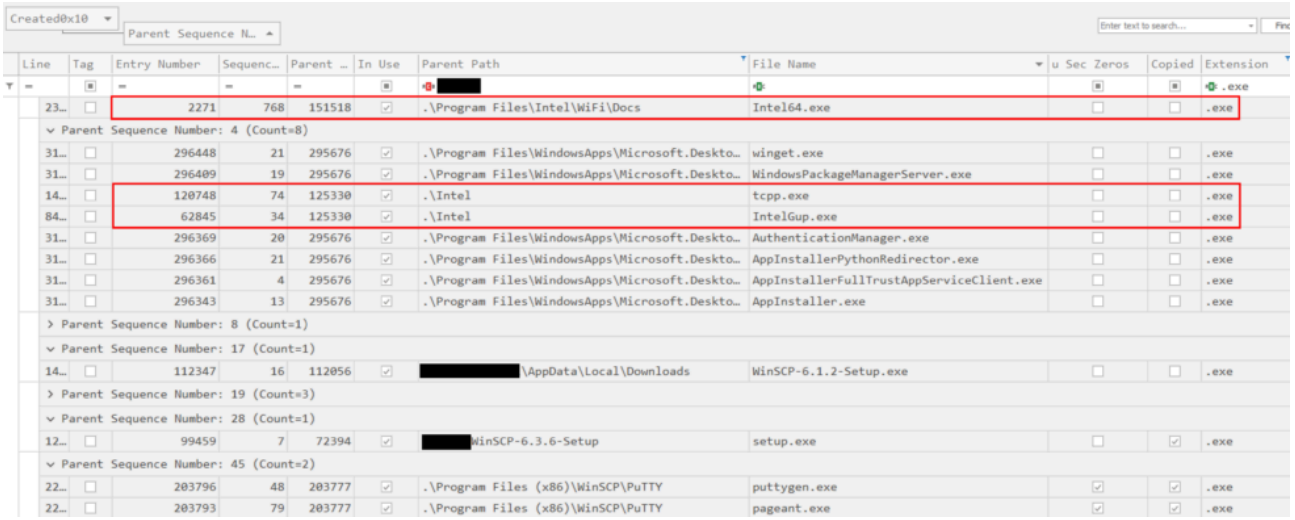
Comparison of a legitimate and malicious python312.dll

Its primary malicious backdoor functionality resides in the DllMain export, in which the packed connect-back logic establishes a C2 connection. Various forensic artifacts — including Prefetch files on the compromised Windows client — confirmed that `setup.exe` and, consequently, `python312.dll` executed successfully, ultimately compromising Patient Zero.

Windows Host Triaging

Typically, when analyzing a system — unless you're performing a scheduled compromise assessment — you have some lead pointing you toward the right direction for your forensic investigation. Doing forensics without a clear lead or well-defined questions is like setting off on vacation without deciding where you want to go. With that in mind, we rely on a battle-tested workflow to analyze systems and determine which tools to run, a process we refer to as "preparational forensics". It's partially automated, so we don't have to deploy the same tools every time manually. As usual, we started off by analyzing "patient zero" with Velociraptor's triage output.

After confirming infection, we took a full disk image. We won't go into every detail of our standard deep-dive workflow here, but one key step we always take is to run THOR and look for recently created executables in the Master File Table. We focused on executables created that same day because we knew the exact timestamp of the WinSCP infection and suspected the threat actor might have used a C2 framework like Cobalt Strike. This approach led us to files named `Intel64.exe`, `tcpp.exe`, and `IntelGup.exe`.

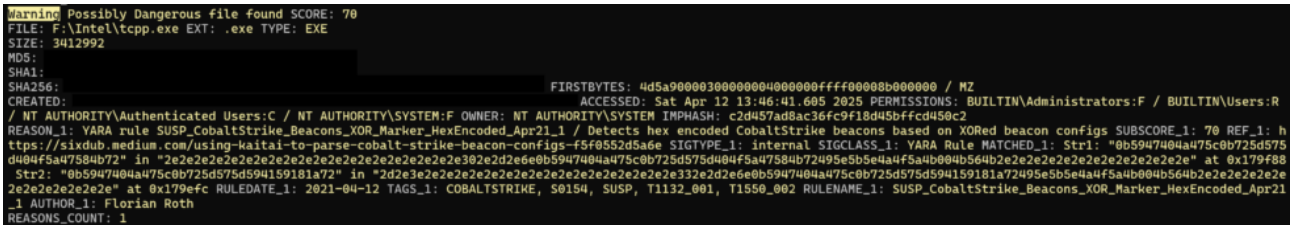


Line	Tag	Entry Number	Sequenc…	Parent …	In Use	Parent Path	File Name	u Sec Zeros	Copied	Extension
23…	☐	2271	768	151518	☑	.\Program Files\Intel\WiFi\Docs	Intel64.exe	☐	☐	.exe
Parent Sequence Number: 4 (Count=8)										
31…	☐	296448	21	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	winget.exe	☐	☐	.exe
31…	☐	296409	19	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	WindowsPackageManagerServer.exe	☐	☐	.exe
14…	☐	120748	74	125330	☑	.\Intel	tcpp.exe	☐	☐	.exe
84…	☐	62845	34	125330	☑	.\Intel	IntelGup.exe	☐	☐	.exe
31…	☐	296369	20	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	AuthenticationManager.exe	☐	☐	.exe
31…	☐	296366	21	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	AppInstallerPythonRedirector.exe	☐	☐	.exe
31…	☐	296361	4	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	AppInstallerFullTrustAppServiceClient.exe	☐	☐	.exe
31…	☐	296343	13	295676	☑	.\Program Files\WindowsApps\Microsoft.Deskto…	AppInstaller.exe	☐	☐	.exe
Parent Sequence Number: 8 (Count=1)										
Parent Sequence Number: 17 (Count=1)										
14…	☐	112347	16	112056	☑	[redacted]\AppData\Local\Downloads	WinSCP-6.1.2-Setup.exe	☐	☐	.exe
Parent Sequence Number: 19 (Count=3)										
Parent Sequence Number: 28 (Count=1)										
12…	☐	99459	7	72394	☑	[redacted]WinSCP-6.3.6-Setup	setup.exe	☐	☑	.exe
Parent Sequence Number: 45 (Count=2)										
22…	☐	203796	48	203777	☑	.\Program Files (x86)\WinSCP\PuTTY	puttygen.exe	☑	☑	.exe
22…	☐	203793	79	203777	☑	.\Program Files (x86)\WinSCP\PuTTY	pageant.exe	☑	☑	.exe

Newly created executables on patient zero

As mentioned before, it's also possible to run THOR against a system image, or, as we did, a mounted disk image by running `thor64.exe --lab -p F:\ --htmlfile A:\Artifacts\case\output.html`, where the `F:\` drive served as the mount point.

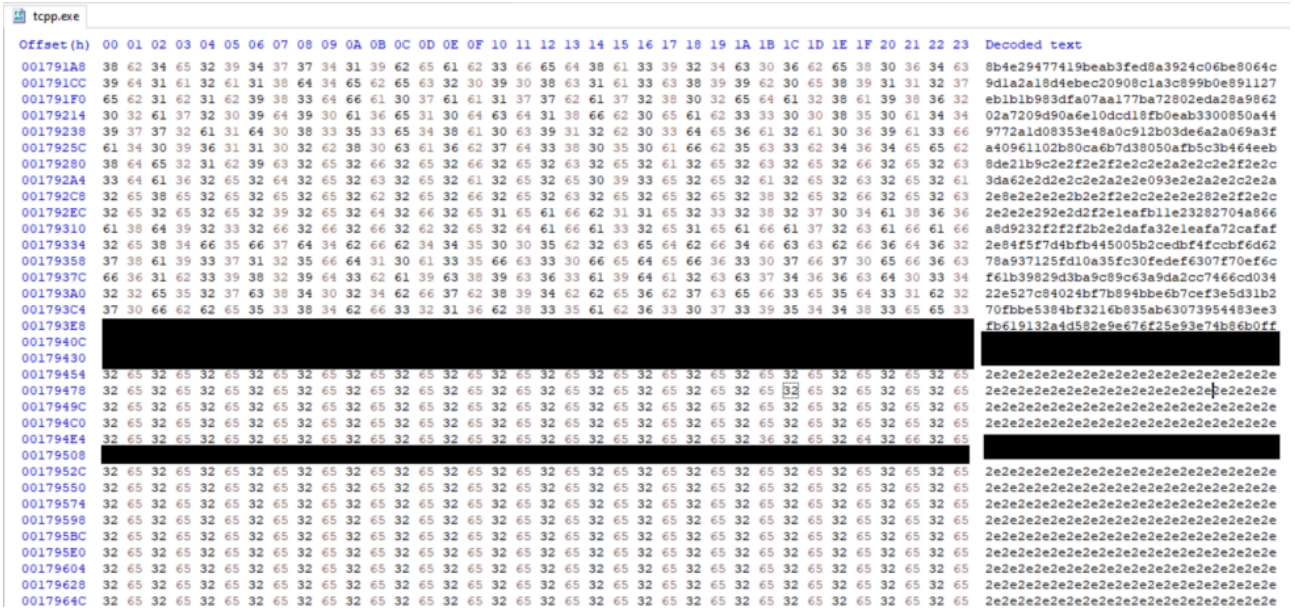
Another option is to aim THOR directly at specific files of interest created on the day of initial infection, which, in this case, flagged `tcpp.exe` as containing a potential Cobalt Strike configuration. A byte pattern in this specific file that stood out was the recurring value `0x2e`, a default XOR key for encrypting configurations in many versions of Cobalt Strike. Whenever we see stretches of `0x2e` or `0x69` in a file, it usually indicates XOR-encrypted null bytes.



```
Warning Possibly Dangerous file found SCORE: 70
FILE: F:\Intel\tcpp.exe EXT: .exe TYPE: EXE
SIZE: 3412992
MD5:
SHA1:
SHA256:                                                    FIRSTBYTES: 4d5a900003000000040000000ffff00008b000000 / MZ
CREATED:                                                   ACCESSED: Sat Apr 12 13:46:41.605 2025 PERMISSIONS: BUILTIN\Administrators:F / BUILTIN\Users:R
/ NT AUTHORITY\Authenticated Users:C / NT AUTHORITY\SYSTEM:F OWNER: NT AUTHORITY\SYSTEM IMPHASH: c2d457ad8ac36fc9f18d45bffcd450c2
REASON_1: YARA rule SUSP_CobaltStrike_Beacons_XOR_Marker_HexEncoded_Apr21_1 / Detects hex encoded CobaltStrike beacons based on XORed beacon configs SUBSCORE_1: 70 REF_1: h
ttps://sixdub.medium.com/using-kaitai-to-parse-cobalt-strike-beacon-configs-f5f0552d5a6e SIGTYPE_1: internal SIGCLASS_1: YARA Rule MATCHED_1: Str1: "0b5947404a475c0b725d575
d404f5a47584b72" in "2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e302e2d2e6e0b5947404a475c0b725d575d404f5a47584b72495e5b5e4a4f5a4b004b564b2e2e2e2e2e2e2e2e2e2e2e2e2e" at 0x179f88
 Str2: "0b5947404a475c0b725d575d594159181a72" in "2d2e3e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e332e2d2e6e0b5947404a475c0b725d575d594159181a72495e5b5e4a4f5a4b004b564b2e2e2e2e2e2e
2e2e2e2e2e2e2e" at 0x179efc RULEDATE_1: 2021-04-12 TAGS_1: COBALTSTRIKE, S0154, SUSP, T1132_001, T1550_002 RULENAME_1: SUSP_CobaltStrike_Beacons_XOR_Marker_HexEncoded_Apr21
_1 AUTHOR_1: Florian Roth
REASONS_COUNT: 1
```

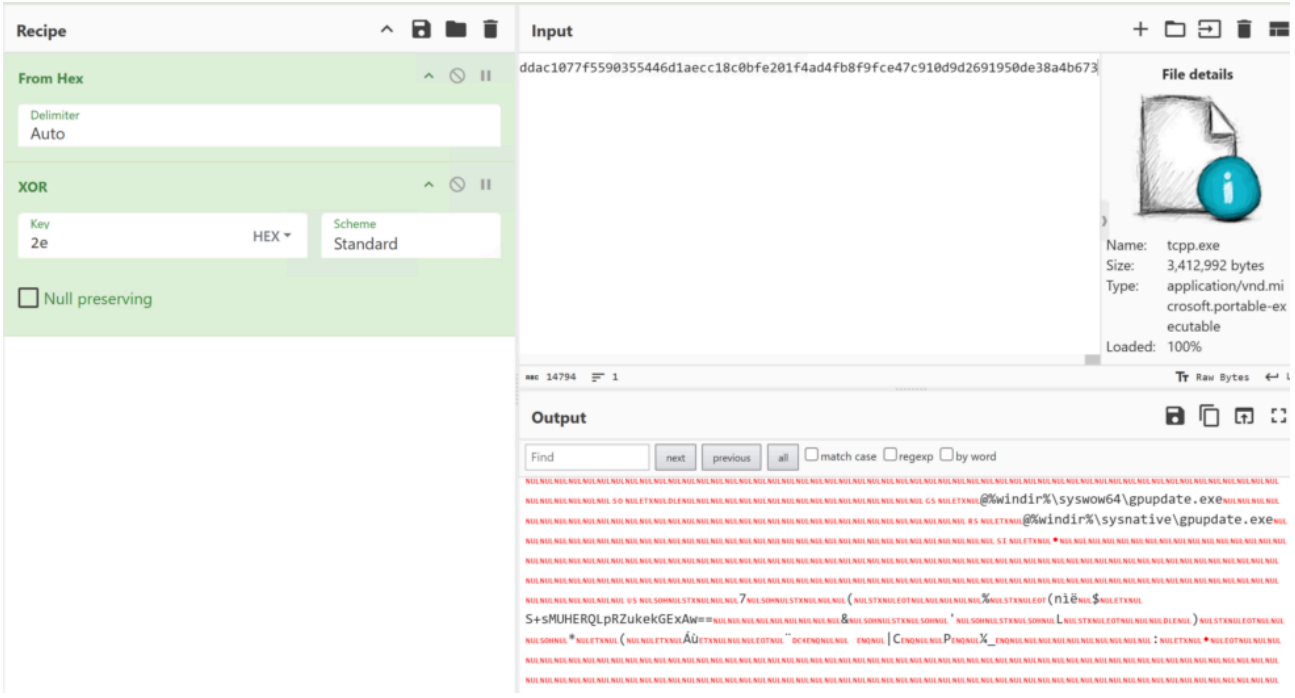
THOR detection highlighting a potential Cobalt Strike byte pattern

Several methods can help reveal more details about this potential Cobalt Strike configuration. The one we typically use is to copy the suspicious byte section and decrypt it using CyberChef or Python. From there, we can export the decrypted data and feed it into a Cobalt Strike parser.



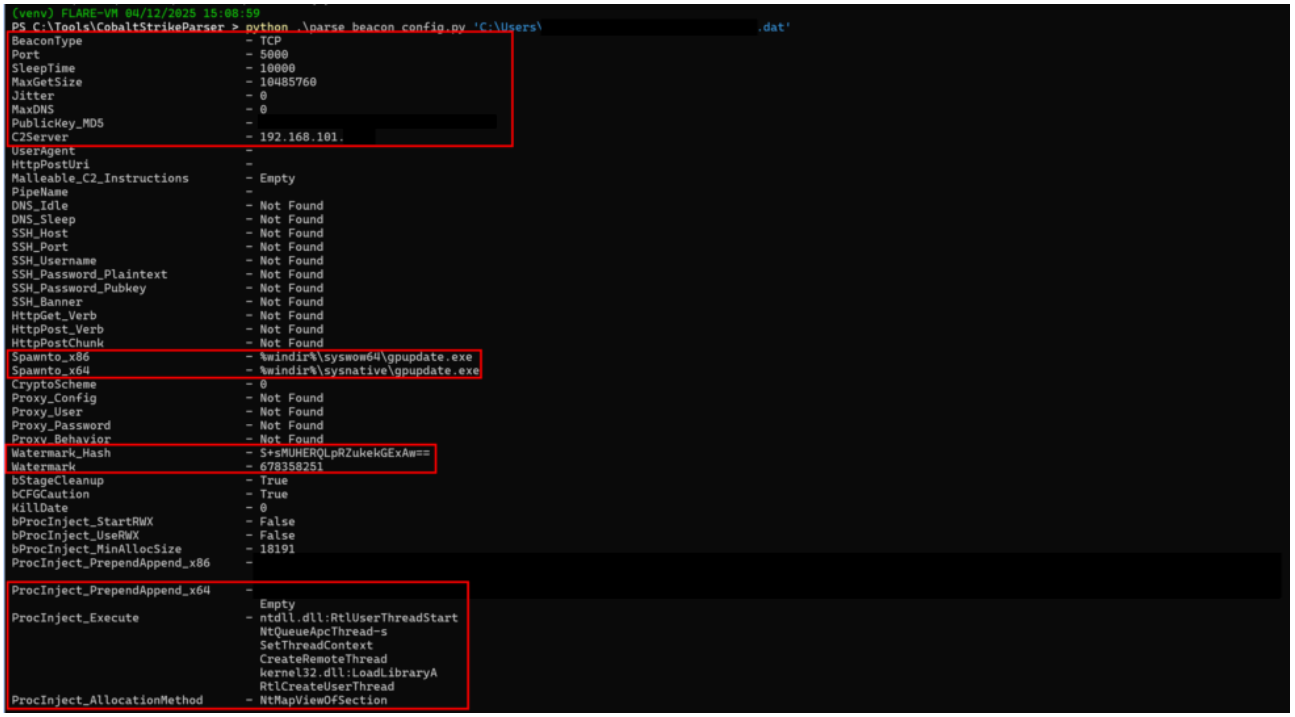
Potential encrypted Cobalt Strike configuration pattern

The first step is to copy the `0x2e` pattern, paste it into CyberChef, and decrypt it using `0x2e`. Straight away, you can see interesting strings appearing.



Cobalt Strike decryption using CyberChef

Next, we can download the decrypted blob and leverage Sentinel One's CobaltStrikeParser, extracting and parsing even more information.



Cobalt Strike configuration detection using CobaltStrikeParser

A particularly noteworthy aspect of the detected Cobalt Strike configuration was its reference to the internal IP address `192.168.101.XXX` on port `5000`, which happened to match patient zero's own IP. This detail strongly suggests that patient zero was being used as a pivot for a Cobalt Strike beacon — a conclusion that became even clearer later in our investigation. We also observed that `gpupdate.exe` was employed as a sacrificial process for Cobalt Strike, as post-compromise payloads are typically injected into dedicated processes.

> **Note:** The manual process described above for extracting Cobalt Strike configurations using the 0x2e pattern will soon be obsolete. THOR v11 includes a built-in feature that automatically detects, decrypts, and parses Cobalt Strike Beacon configurations — directly during the scan, no manual steps required. This feature will be covered in more detail in an upcoming blog post.

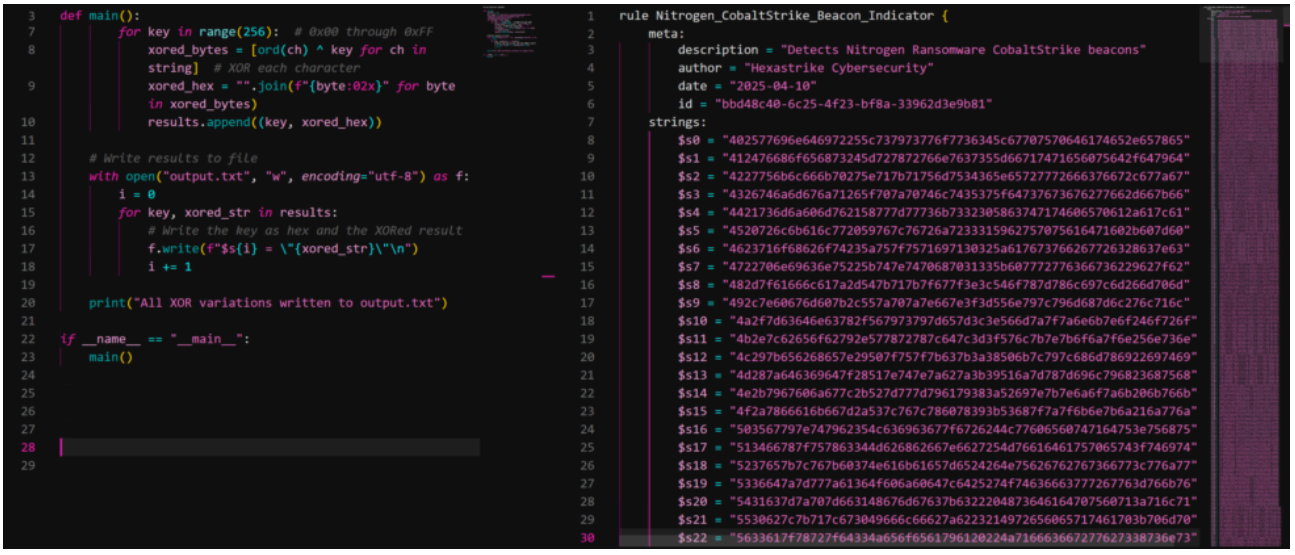
Interjection – Cobalt Strike Detection and Threat Intel

From these strings — for example, `@%windir%\syswow64\gpupdate.exe, @%windir%\sysnative\gpupdate.exe`, and the watermark hash `S+sMUHERQLpRZukekGExAw==` — we can build a custom YARA rule. Encrypting each of these strings with all possible single-byte values makes it possible to detect additional XOR-encrypted Cobalt Strike configurations, not only on patient zero but also on other potentially compromised hosts.

```
#!/usr/bin/env python3

def main():
    results = []
    str_input = ["@%windir%\syswow64\gpupdate.exe", "@%windir%\sysnative\gpupdate.exe", "S+sMUHERQLpl
    for string in str_input:
        for key in range(256):  # 0x00 through 0xFF
```

```
            xored_bytes = [ord(ch) ^ key for ch in string]  # XOR each character
            xored_hex = "".join(f"{byte:02x}" for byte in xored_bytes)
            results.append((key, xored_hex))

    # Write results to file
    with open("output.txt", "w", encoding="utf-8") as f:
        i = 0
        for key, xored_str in results:
            f.write(f"$s{i} = \"{xored_str}\"\n")
            i += 1

    print("All XOR variations written to output.txt")

if __name__ == "__main__":
    main()
```

Using the script's output, we can create a very simple YARA rule to be used during the engagement, potentially highlighting even more suspicious files like the one we already discovered.

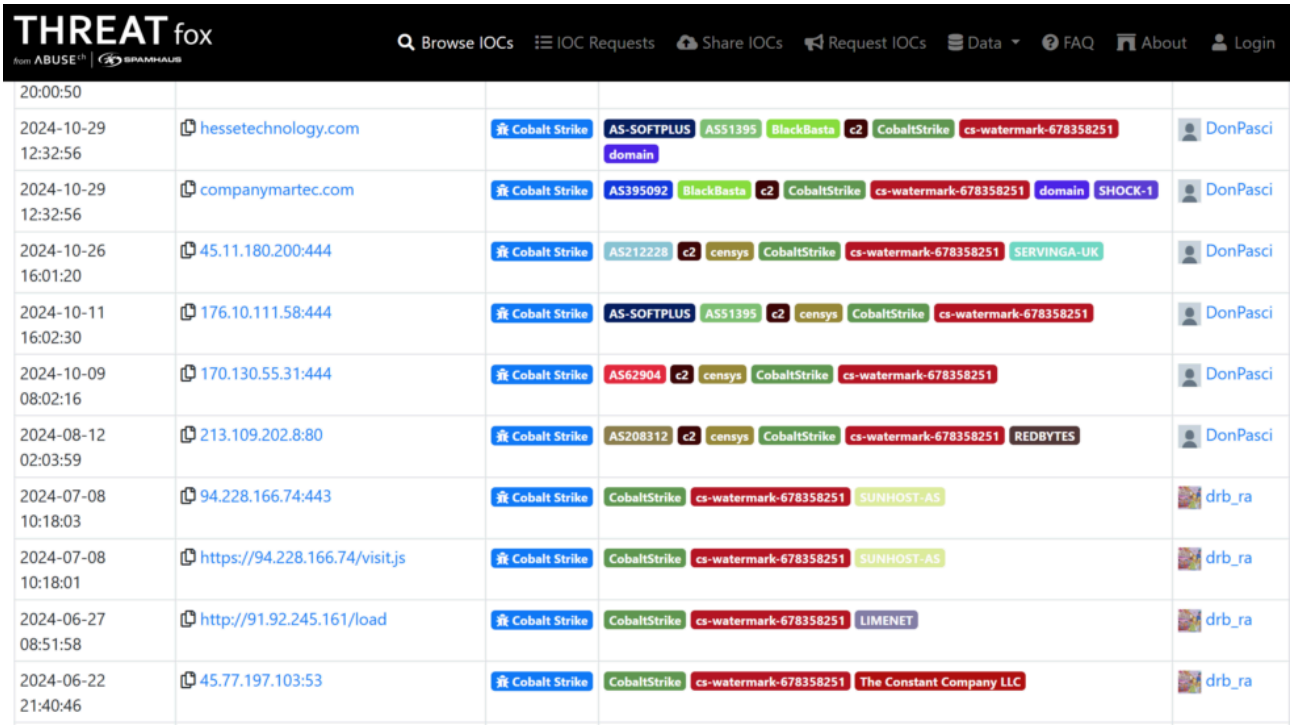


```
def main():
    for key in range(256):  # 0x00 through 0xFF
        xored_bytes = [ord(ch) ^ key for ch in string]  # XOR each character
        xored_hex = "".join(f"{byte:02x}" for byte in xored_bytes)
        results.append((key, xored_hex))

    # Write results to file
    with open("output.txt", "w", encoding="utf-8") as f:
        i = 0
        for key, xored_str in results:
            # Write the key as hex and the XORed result
            f.write(f"$s{i} = \"{xored_str}\"\n")
            i += 1

    print("All XOR variations written to output.txt")

if __name__ == "__main__":
    main()
```

```
rule Nitrogen_CobaltStrike_Beacon_Indicator {
    meta:
        description = "Detects Nitrogen Ransomware CobaltStrike beacons"
        author = "Hexastrike Cybersecurity"
        date = "2025-04-10"
        id = "bbd48c40-6c25-4f23-bf8a-33962d3e9b81"
    strings:
        $s0 = "402577696e646972255c737973776f7736345c677075706461746 52e657865"
        $s1 = "412476686f656873245d727872766e7637355d667174716560756 42f647964"
        $s2 = "4227756b6c666b70275e717b71756d7534365e65727772666376672c677a67"
        $s3 = "4326746a6d676a71265f707a70746c7435375f647376736762776 62d667b66"
        $s4 = "4421736d6a606d762158777d77736b7332305863747174606570612a617c61"
        $s5 = "4520726c6b616c772059767c76726a72333159627570756164716 02b607d60"
        $s6 = "4623716f68626f74235a757f7571697130325a617673766267726328637e63"
        $s7 = "4722706e69636e75225b747e7470687031335b607727763667362296 27f62"
        $s8 = "482d7f61666c617a2d547b717b7f677f3e3c546f787d786c697c6d266d706d"
        $s9 = "492c7e60676d607b2c557a707a7e667e3f3d556e797c796d687d6c276c716c"
        $s10 = "4a2f7d63646e63782f567973797d657d3c3e566d7a7f7a6e6b7e6f246f726f"
        $s11 = "4b2e7c62656f62792e577872787c647c3d3f576c7b7e7b6f6a7f6e256e736e"
        $s12 = "4c297b656268657e29507f757f7b637b3a38506b7c797c686d786922697469"
        $s13 = "4d287a646369647f28517e747e7a627a3b39516a7d787d696c796823687568"
        $s14 = "4e2b7967606a677c2b527d777d796179383a52697e7b7e6a6f7a6b206b766b"
        $s15 = "4f2a7866616b667d2a537c767c786078393b53687f7a7f6b6e7b6a216a776a"
        $s16 = "503567797e747962354c636963677f6726244c7760656074716475 3e756875"
        $s17 = "513466787f757863344d626862667e6627254d766164617570657 43f746974"
        $s18 = "5237657b7c767b60374e616b61657d6524264e7562676276736677 3c776a77"
        $s19 = "5336647a7d777a61364f606a60647c6425274f74636663772767763d766b76"
        $s20 = "5431637d7a707d663148676d67637b6322204873646164707560713a716c71"
        $s21 = "5530627c7b717c673049666c66627a62232149726560657174617 03b706d70"
        $s22 = "5633617f78727f64334a656f6561796120224a71666366727762733873 6e73"
```

YARA Cobalt Strike signature and rule creation

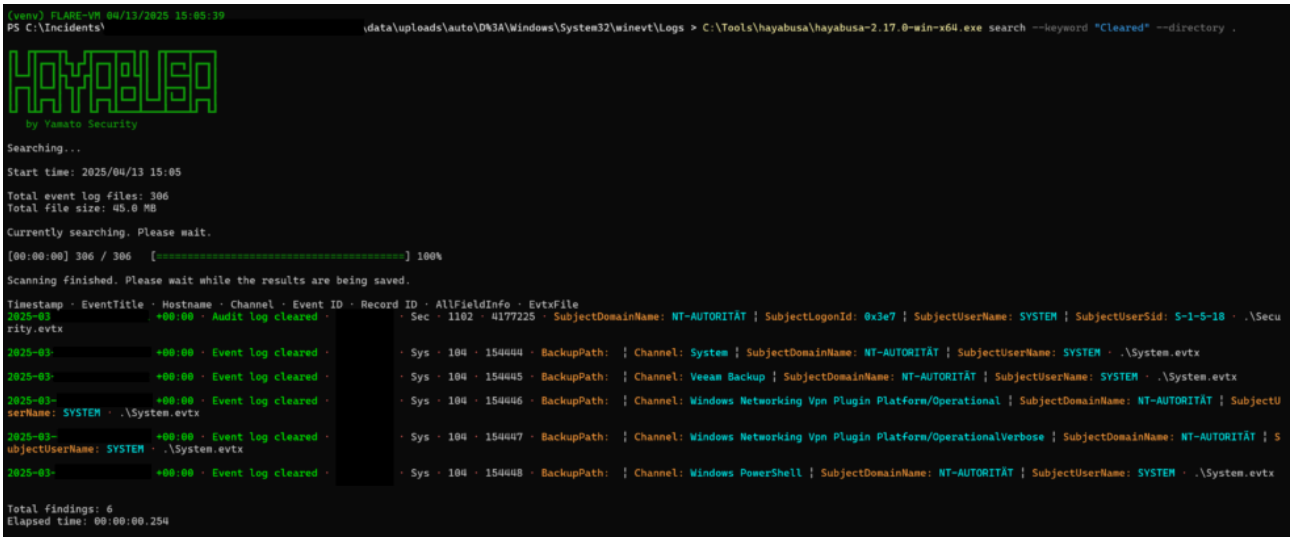
Notably, the identified Cobalt Strike watermark 678358251 has previously been listed on abuse.ch. This watermark has been associated with multiple threat actors, including the ransomware group Black Basta, further highlighting its reuse across malicious campaigns and threat actors. Cobalt Strike watermarks serve as unique identifiers, allowing to track and correlate activity across disparate Cobalt Strike C2 servers observed in the wild.



Cobalt Strike C2 team servers with watermark 678358251

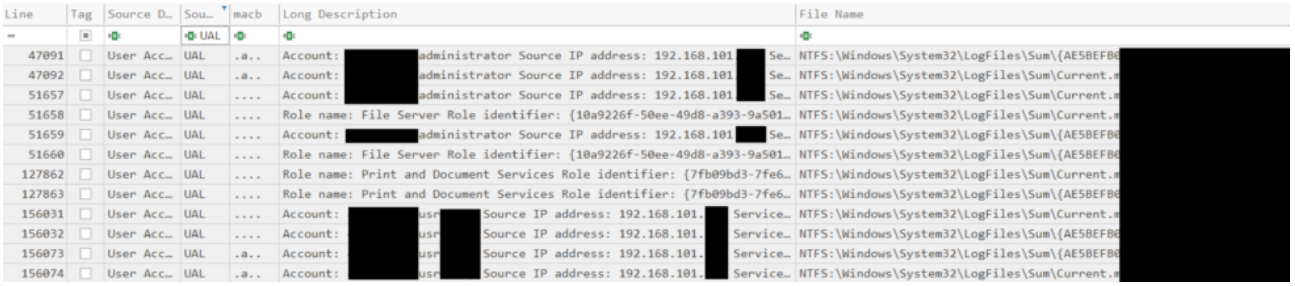
Detecting Lateral Movement with User Access Logging

After identifying patient zero, we set out to locate further compromised hosts. Tracking lateral movement from patient zero proved challenging because artifacts on the source system are typically less thorough than those on the destination. Complicating matters even more, the threat actor had cleared critical Windows event logs — among them the Security, System, and PowerShell logs — on several machines, as shown in the following screenshot.



Log clearing detected using Hayabusa

Nevertheless, not all forensic data was lost. Even with extensive log clearing, we built a supertimeline on one of the client's physical Windows servers, revealing User Access Logging (UAL) entries. These entries provided clear evidence of lateral movement to another Windows Server originating from the beachhead on the exact date of the initial compromise.

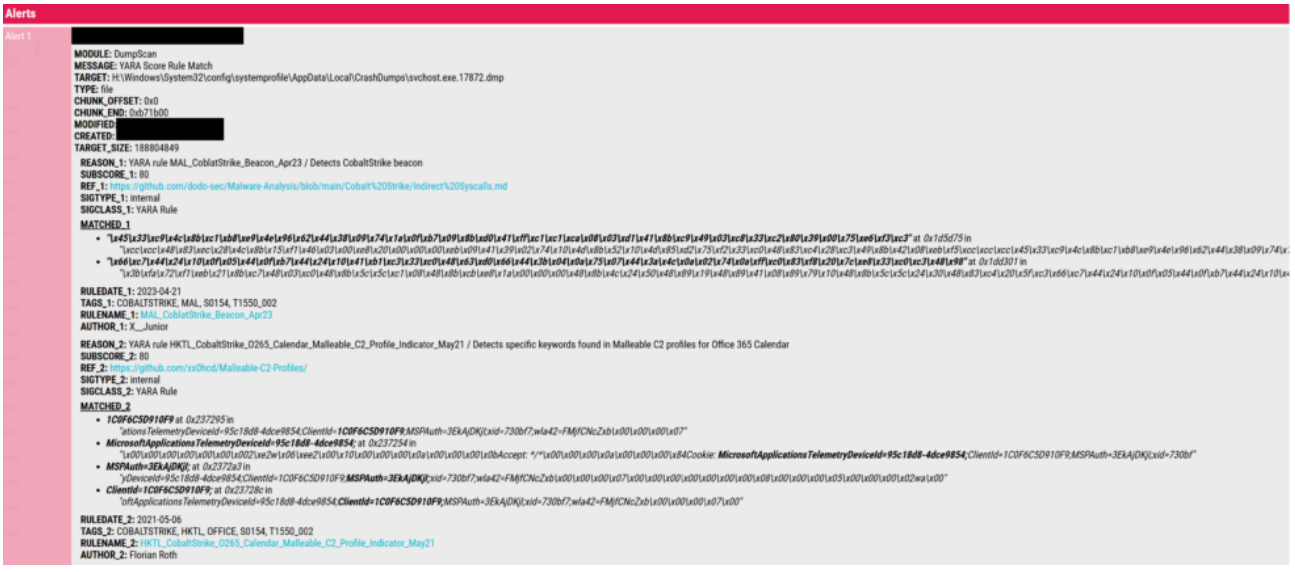


Line	Tag	Source D…	Sou…	macb	Long Description	File Name
47091		User Acc…	UAL	.a..	Account: [redacted]administrator Source IP address: 192.168.101.[redacted] Se…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
47092		User Acc…	UAL	.a..	Account: [redacted]administrator Source IP address: 192.168.101.[redacted] Se…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]
51657		User Acc…	UAL	....	Account: [redacted]administrator Source IP address: 192.168.101.[redacted] Se…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]
51658		User Acc…	UAL	....	Role name: File Server Role identifier: {10a9226f-50ee-49d8-a393-9a501…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]
51659		User Acc…	UAL	....	Account: [redacted]administrator Source IP address: 192.168.101.[redacted] Se…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
51660		User Acc…	UAL	....	Role name: File Server Role identifier: {10a9226f-50ee-49d8-a393-9a501…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
127862		User Acc…	UAL	....	Role name: Print and Document Services Role identifier: {7fb09bd3-7fe6…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]
127863		User Acc…	UAL	....	Role name: Print and Document Services Role identifier: {7fb09bd3-7fe6…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
156031		User Acc…	UAL	....	Account: [redacted]usr[redacted] Source IP address: 192.168.101.[redacted] Service…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]
156032		User Acc…	UAL	....	Account: [redacted]usr[redacted] Source IP address: 192.168.101.[redacted] Service…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
156073		User Acc…	UAL	.a..	Account: [redacted]usr[redacted] Source IP address: 192.168.101.[redacted] Service…	NTFS:\Windows\System32\LogFiles\Sum\{AE5BEFB6[redacted]
156074		User Acc…	UAL	.a..	Account: [redacted]usr[redacted] Source IP address: 192.168.101.[redacted] Service…	NTFS:\Windows\System32\LogFiles\Sum\Current.m[redacted]

User access logging artifacts in a system supertimeline

Basic Crash Dump Triaging

When we reran THOR against the newly uncovered compromised server system, it yielded some additional leads. In this instance, we discovered a suspicious operating system `svchost.exe` file that presented telltale signs of Cobalt Strike activity.



THOR detection of Cobalt Strike artifacts in a user mode crash dump

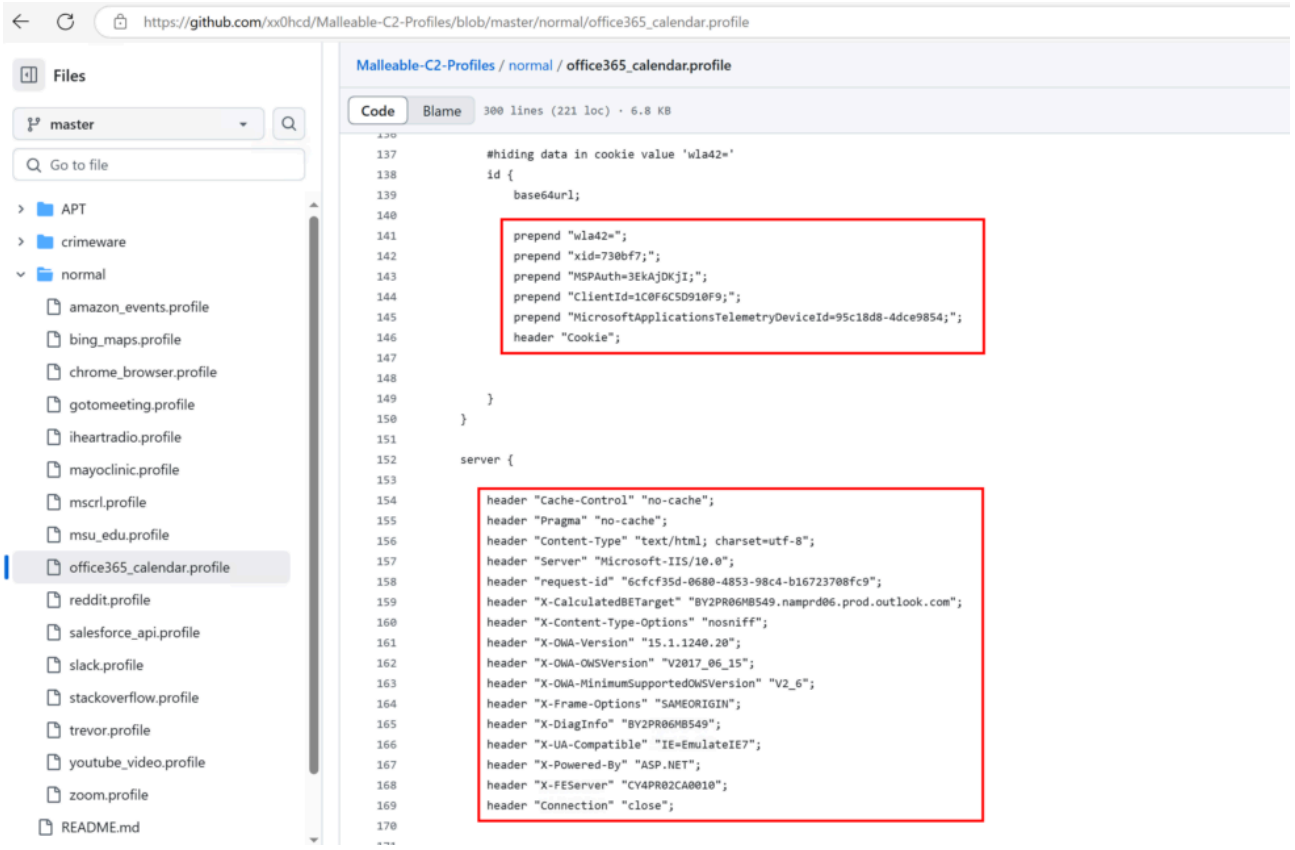
We also found that a Windows Error Reporting (WER) log for this particular `svchost.exe` was saved in `C:\ProgramData\Microsoft\Windows\WER`. WER is a highly underrated artifact capable of capturing detailed debug information, such as the application name, loaded modules, and a heap dump that preserves the memory data active at the time of a crash. If configured to do so, WER also collects user-mode crash dumps and stores them locally whenever an application crashes — exactly the situation THOR detected here. Although crash dumps are disabled by default, administrators can enable them by configuring the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps`.

```
Key path: Microsoft\Windows\Windows Error Reporting\LocalDumps
Last write time: 2022-01-11 13:42:27.1159213

Subkey count: 0
Values count: 3


------------ Value #0 ------------
Name: DumpType (RegDword)
Data: 2

------------ Value #1 ------------
Name: DumpCount (RegDword)
Data: 10

------------ Value #2 ------------
Name: DumpFolder (RegExpandSz)
Data: %LOCALAPPDATA%\CrashDumps
```

Crash dump SOFTWARE registry hive configuration

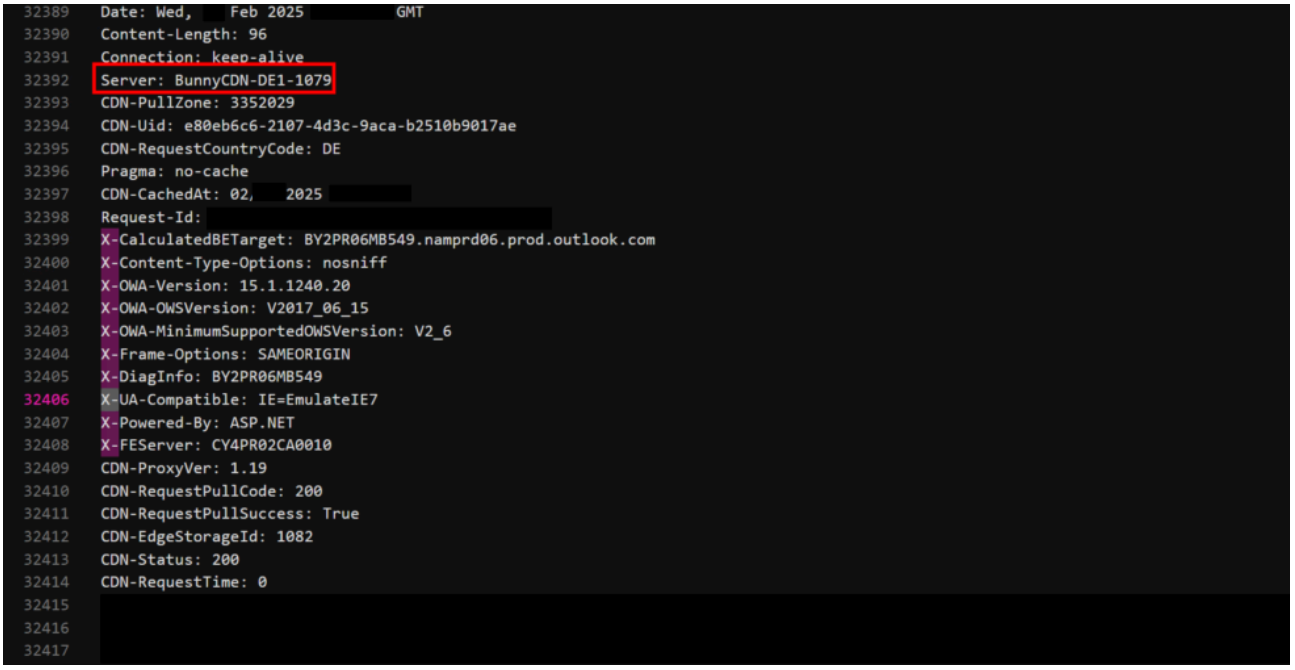
In recent years, these crash dumps have improved considerably and can be analyzed in more depth using tools like WinDBG — a process we'll explore in the next chapter. In this specific scenario, we verified the crash dump settings by reviewing the registry keys and confirmed that a full dump (dump type 2), which includes all virtual memory, was being saved to the `%LOCALAPPDATA%\CrashDumps` directory, with a maximum of ten dump files retained.

From the `svchost.exe.17872.dmp` crash dump we identified through THOR, several suspicious string artifacts pointed to a possible Cobalt Strike beacon configuration. THOR referenced a GitHub repository — "Detects specific keywords found in Malleable C2 profiles for Office 365 Calendar" — indicating that both client and server configuration details, including cookie header values from the client and custom headers from the server, had been embedded within the crash dump.



M365 Calendar Profile on GitHub

To confirm these findings, we used `bstrings.exe` to extract strings from the crash dump running `bstrings.exe -f .\svchost.exe.17872.dmp > .\svchost.exe.17872.dmp.strings.txt`. This process uncovered the precise configuration strings highlighted earlier, revealing what appeared to be an entire HTTP response. We even found a `Server` header that matched the system responding to the request.



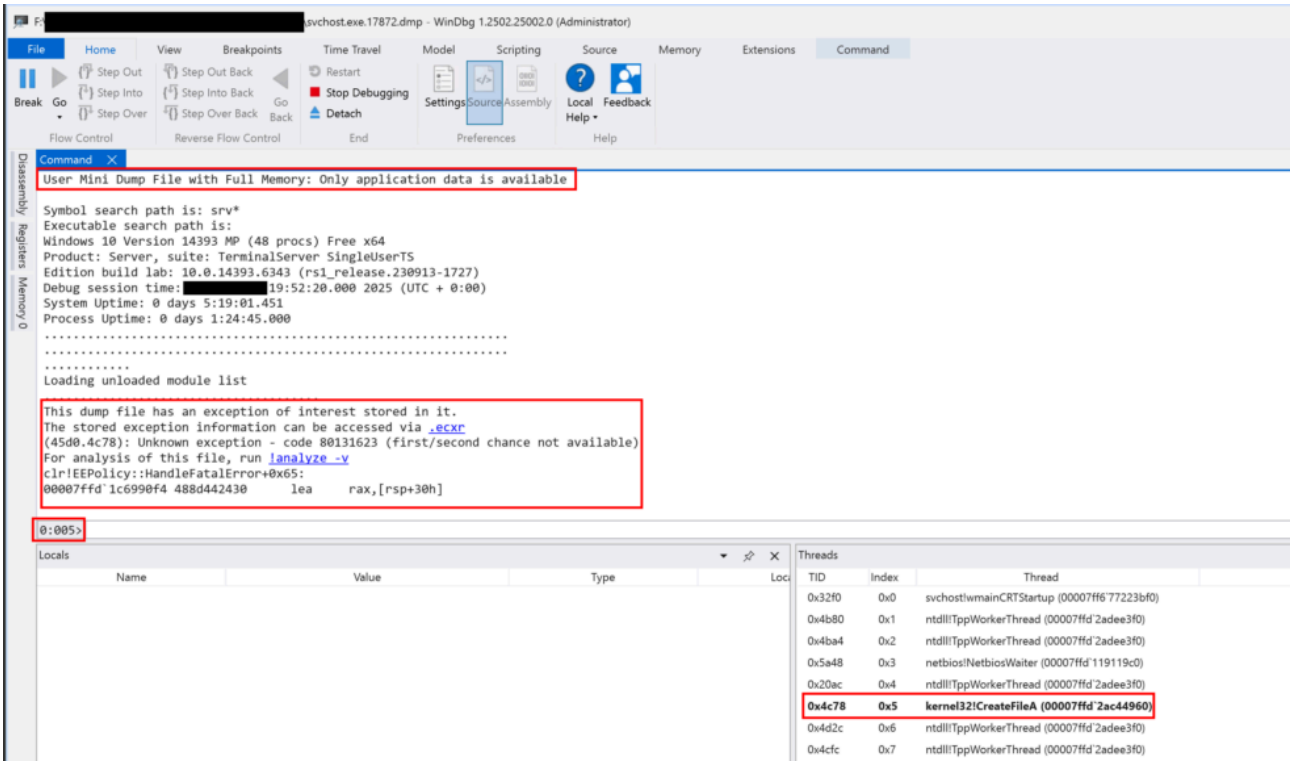
Cobalt Strike HTTP Response in the svchost.exe crash dump

We repeated this methodology until no additional pertinent strings emerged, then ran `bstrings.exe` to focus specifically on URLs: `bstrings.exe --lr url3986 -f .\svchost.exe.17872.dmp -q -sa`. That step exposed the Cobalt Strike team server, confirming our suspicions regarding an active beacon configuration within the crash dump.

C2 Team Server detected in URL strings extracted from the detected crash dump

Crash Dump Analysis with WinDBG

In this scenario, the process crash dump was configured to capture a full user-mode dump that included all virtual memory. Having access to a full dump file allowed for a thorough examination of the process at the time it failed. By loading the crash dump directly into WinDBG, the debugger halted at the specific exception that caused the crash and displayed the associated thread — thread `0x5` with an ID of `0x4c78` — along with a reference to the full memory dump type. The debugger also showed the debug session time, which matched the timestamp of the crash dump's creation.



Crash dump loaded into WinDBG

The available information showed that a failure occurred while the process executed the `kernel32` function `CreateFileA` (`0x4c78 0x5 kernel32!CreateFileA (00007ffd'2ac44960)`). Running `!analyze -v` initiated the exception analysis, revealing details about the operating system version, build, CPU registers, and a stack trace, alongside an error code. Unfortunately, the error code did not yield any additional clues, only indicating that the exception must have happened before the error handling routine at `00007ffd'12c5ac52 mscorlib_ni!System.Environment.ResourceHelper.GetResourceStringCode+0x252` .

```
CONTEXT:  (.ecxr)
rax=000000b978af9e00 rbx=000000b978af9d60 rcx=000000b978af9e00
rdx=0000000000000000 rsi=0000000000000000 rdi=000000b978af9e00
rip=00007ffd1c6990f4 rsp=000000b978af9d30 rbp=000000b978af9e30
 r8=00000000000004d0  r9=0000000000000000 r10=00000000000000f5
r11=000000b978af9e00 r12=0000000000000001 r13=000000b978afaa10
r14=00007ffd1cd3a530 r15=0000000000000000
iopl=0         nv up ei pl nz na po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000205
clr!EEPolicy::HandleFatalError+0x65:
00007ffd`1c6990f4 488d442430      lea     rax,[rsp+30h]
Resetting default scope

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 00007ffd12c5ac52 (mscorlib_ni!System.Environment.ResourceHelper.GetResourceStringCode+0x0000000000000252)
   ExceptionCode: 80131623
  ExceptionFlags: 00000001
NumberParameters: 0

PROCESS_NAME:  svchost.exe

ERROR_CODE: (NTSTATUS) 0x80131623 - <Unable to get error code text>

EXCEPTION_CODE_STR:  80131506

FAULTING_THREAD:  ffffffff

STACK_TEXT:
00000000`00000000 00000000`00000000 svchost.exe!unknown_function+0x0
00000000`00000000 00000000`00000000 unknown![.ecxr]+0x0
000000b9`78af9d30 00007ffd`1c6990f4 clr!EEPolicy::HandleFatalError+0x65
000000b9`78afa310 00007ffd`1ca7be67 clr!SystemNative::GenericFailFast+0x19f
000000b9`78afa3b0 00007ffd`1ca7ba96 clr!SystemNative::FailFast+0xa6
000000b9`78afa500 00007ffd`12c5ac52 mscorlib_ni!System.Environment.ResourceHelper.GetResourceStringCode+0x252
000000b9`78afa570 00007ffd`1c385963 clr!CallDescrWorkerInternal+0x83
000000b9`78afa5b0 00007ffd`1c3856f6 clr!CallDescrWorkerWithHandler+0x4e
```

WinDBG analyze extension output

To gather more insights, the MEX extension provided the command `!mex.di` (or simply `!di` when using built-in aliases). This command revealed information about the user under whose account the process was running, as well as the operating system version, system uptime, and the process ID.

```
0:005> .load mex
Mex External 3.0.0.7172 Loaded!
0:005> !mex.di
Computer Name: [redacted]
User Name: [redacted]
PID: 0x45D0 = 0n17872
Windows 10 Version 14393 MP (48 procs) Free x64
Product: Server, suite: TerminalServer SingleUserTS
Edition build lab: 10.0.14393.6343 (rs1_release.230913-1727)
Debug session time: [redacted] 19:52:20.000 2025 (UTC + 0:00)
System Uptime: 0 days 5:19:01.451
Process Uptime: 0 days 1:24:45.000
  Kernel time: 0 days 0:00:01.000
  User time: 0 days 0:00:05.000
```

Basic MEX triaging in WinDBG

Further investigation involved the `!peb` command, which examined the Process Environment Block (PEB) — a structure containing details on loaded modules, command-line arguments, the image file in use, and the window title for the process. In this instance, the PEB indicated that the process path was `C:\StorageReport\tcpp.exe`, a file previously identified as a Cobalt Strike pivot beacon that facilitated tunneling through the patient zero system. With a Cobalt Strike configuration discovered in memory (as supported by the string analysis), it was apparent that malicious activity had been running within this process.

Pivot beacon identified as windows title metadata in the analyzed crash dump

These same details could have been extracted manually by inspecting the PEB structure without relying on the `!peb` extension. Typically, one would locate the PEB address first by referencing the pseudoregister `$peb` (`dt @$peb`). In a kernel-mode dump, the command `!process -0 0` would also yield the PEB location. With that address in hand — in this case, `0x000000b977fe1000` — the relevant data can be read by issuing a command such as `dt _PEB 000000b977fe1000` .

```
0:005> dt @$peb
Symbol not found at address 000000b977fe1000.
0:005> dt _PEB 000000b977fe1000
ole32!_PEB
   +0x000 Reserved1        : [2]  ""
   +0x002 BeingDebugged    : 0 ''
   +0x003 Reserved2        : [1]  "???"
   +0x008 Reserved3        : [2] 0xffffffff`ffffffff Void
   +0x018 Ldr              : 0x00007ffd`2af203c0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x000001e4`ef132160 _RTL_USER_PROCESS_PARAMETERS
   +0x028 Reserved4        : [3] (null)
   +0x040 AtlThunkSListPtr : (null)
   +0x048 Reserved5        : (null)
   +0x050 Reserved6        : 4
   +0x058 Reserved7        : 0x00007ffd`292bf000 Void
   +0x060 Reserved8        : 0
   +0x064 AtlThunkSListPtr32 : 0
   +0x068 Reserved9        : [45] 0x000001e4`eef20000 Void
   +0x1d0 Reserved10       : [96]  ""
   +0x230 PostProcessInitRoutine : (null)
   +0x238 Reserved11       : [128]  "???"
   +0x2b8 Reserved12       : [1] (null)
   +0x2c0 SessionId        : 0
```

It is in the `_PEB_LDR_DATA` member that key information regarding loaded modules resides, as documented by Microsoft. The `InMemoryOrderModuleList` field within the `_PEB_LDR_DATA` structure is a doubly linked list of loaded modules, so walking this list can provide details on every module.

```
0:005> dt _LIST_ENTRY
ole32!_LIST_ENTRY
   +0x000 Flink            : Ptr64 _LIST_ENTRY
   +0x008 Blink            : Ptr64 _LIST_ENTRY
```

This includes the primary image executable (in this instance, `svchost.exe`) and subsequent items referenced in its `InMemoryOrderLinks` or `InLoadOrderLinks` fields.

Manual PEB iteration

The first loaded module points to the next one via its `InMemoryOrderLinks` or `InLoadOrderLinks` member, which, in this instance, leads to the address `0x000001e4ef132950`. Because that address is also of type `_LIST_ENTRY`, the command `dt _LDR_DATA_TABLE_ENTRY 0x000001e4ef132950` can reveal details about the next link. This manual approach — iterating through the linked list entry by entry — proves especially useful when you need to investigate a specific module or structure in greater depth.

```
0:005> dt _LDR_DATA_TABLE_ENTRY 0x000001e4`ef132950
ole32!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x000001e4`ef132f70 - 0x000001e4`ef132ae0 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x000001e4`ef133450 - 0x00007ffd`2af203f0 ]
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00007ffd`2add0000 - 0x00000000`00000000 ]
   +0x030 DllBase          : 0x00000000`001cf000 Void
   +0x038 EntryPoint       : 0x00000000`003c003a Void
   +0x040 SizeOfImage      : 0xef132840
   +0x048 FullDllName      : _UNICODE_STRING "ntdll.dll"
   +0x058 BaseDllName      : _UNICODE_STRING "Š???"
   +0x068 FlagGroup        : [4]  "`???"
   +0x068 Flags            : 0x2af20160
   +0x068 PackagedBinary   : 0y0
   +0x068 MarkedForRemoval : 0y0
   +0x068 ImageDll         : 0y0
   +0x068 LoadNotificationsSent : 0y0
   +0x068 TelemetryEntryProcessed : 0y0
   +0x068 ProcessStaticImport : 0y1
   +0x068 InLegacyLists    : 0y1
   +0x068 InIndexes        : 0y0
   +0x068 ShimDll          : 0y1
   +0x068 InExceptionTable : 0y0
   +0x068 ReservedFlags1   : 0y00
   +0x068 LoadInProgress   : 0y0
   +0x068 LoadConfigProcessed : 0y0
   +0x068 EntryProcessed   : 0y0
   +0x068 ProtectDelayLoad : 0y0
   +0x068 ReservedFlags3   : 0y10
   +0x068 DontCallForThreads : 0y0
   +0x068 ProcessAttachCalled : 0y0
   +0x068 ProcessAttachFailed : 0y1
   +0x068 CorDeferredValidate : 0y1
   +0x068 CorImage         : 0y1
   +0x068 DontRelocate     : 0y1
   +0x068 CorILOnly        : 0y0
   +0x068 ReservedFlags5   : 0y101
```

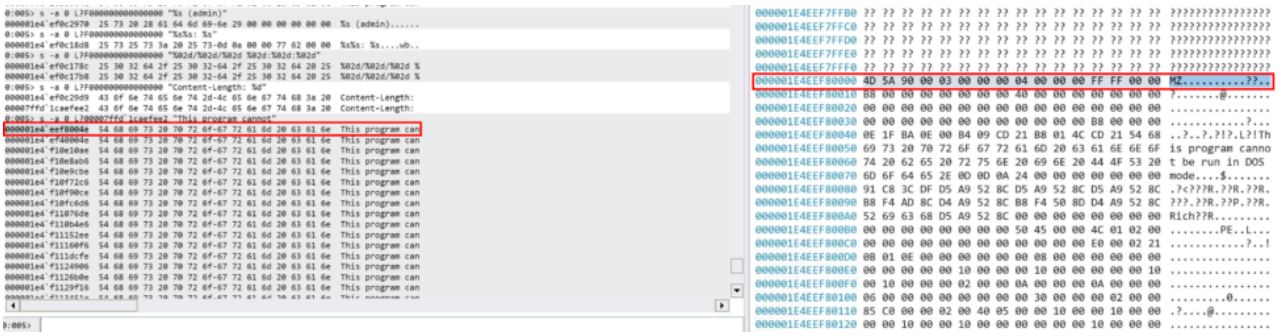
Second module in the PEB

Returning to the original purpose — gathering conclusive evidence of a Cobalt Strike beacon residing in memory — analysis continued by examining suspicious strings and testing them against a Cobalt Strike YARA rule by Elastic.

```
PS F:\                                    E:\yara-v4.5.2-2326-win64\yara64.exe .\cs.yar .\svchost.exe.17872.dmp --print-strings
Windows_Trojan_CobaltStrike_ee756db7 .\svchost.exe.17872.dmp
0x1f4861:$a46: %s (admin)
0x1f37c9:$a48: %s%s: %s
0x1f367d:$a50: %02d/%02d/%02d %02d:%02d:%02d
0x1f36a9:$a50: %02d/%02d/%02d %02d:%02d:%02d
0x1f48ca:$a51: Content-Length: %d
0x8893dd3:$a51: Content-Length: %d
Windows_Trojan_CobaltStrike_663fc95d .\svchost.exe.17872.dmp
0x1de82d:$a: 48 89 5C 24 08 57 48 83 EC 20 48 8B 59 10 48 8B F9 48 8B 49 08 FF 17 33 D2 41 B8 00 80 00 00
```

Cobalt Strike YARA rule by Elastic

Observed strings were traced to the corresponding memory address within the dump, revealing that all originated from a similar region. Searching for the MZ header indicated the presence of what looked like a loaded binary at that location.



WinDBG potential Cobalt Strike string search

By investigating the DOS header (`ntdll!_IMAGE_DOS_HEADER` at `000001e4'eef80000`), one can identify the PE header offset (`e_lfanew`), determine the approximate size of the binary (`SizeOfImage`), and theoretically dump that data. However, it is important to note that paging can cause portions of memory to be absent from the dump file, so the extracted DLL may be incomplete or partially overwritten.

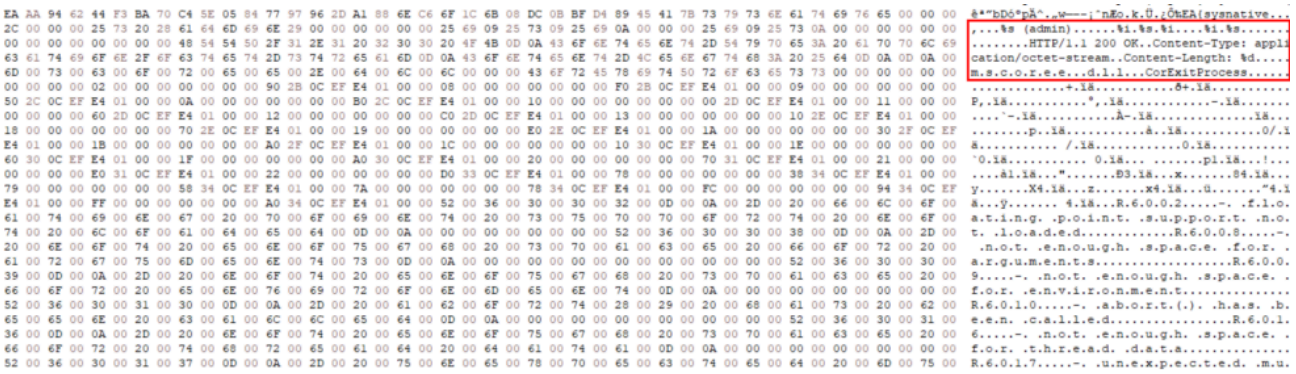
```
0:005> dt -r ntdll!_IMAGE_DOS_HEADER 000001e4`eef80000
   +0x000 e_magic          : 0x5a4d
   +0x002 e_cblp           : 0x90
   +0x004 e_cp             : 3
  [⋯]
   +0x028 e_res2           : [10] 0
   +0x03c e_lfanew         : 0n184
0:005> ? 000001e4`eef80000 + 0n184
Evaluate expression: 2082773401784 = 000001e4`eef800b8
0:005> dt -r _IMAGE_NT_HEADERS 000001e4`eef800b8
ole32!_IMAGE_NT_HEADERS
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
      +0x000 Machine          : 0x14c
      +0x002 NumberOfSections : 2
      +0x004 TimeDateStamp    : 0
      +0x008 PointerToSymbolTable : 0
      +0x00c NumberOfSymbols  : 0
      +0x010 SizeOfOptionalHeader : 0xe0
      +0x012 Characteristics  : 0x2102
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
      +0x000 Magic            : 0x10b
  [⋯]
      +0x038 SizeOfImage      : 0x3000
      +0x03c SizeOfHeaders    : 0x200
      +0x040 CheckSum         : 0xc085
      +0x044 Subsystem        : 2
      +0x046 DllCharacteristics : 0x540
      +0x048 SizeOfStackReserve : 0x100000
      +0x04c SizeOfStackCommit : 0x1000
      +0x050 SizeOfHeapReserve : 0x100000
      +0x054 SizeOfHeapCommit : 0x1000
      +0x058 LoaderFlags      : 0
      +0x05c NumberOfRvaAndSizes : 0x10
      +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
         +0x000 VirtualAddress   : 0
         +0x004 Size             : 0
```

Using `.writemem` in WinDBG with an appropriate address range (`000001e4'eef80000 L3000`) attempts to write this region to disk. In this case, portions of memory at `000001e4'eef81000` were unreadable, likely due to paging, and the range did not encompass the exact strings indicative of the beacon configuration.

```
0:005> da 000001e4`eef800b8
000001e4`eef800b8  "PE"
0:005> .writemem C:\temp\svchost_cs.dat 000001e4`eef80000 L3000
Writing 3000 bytes..
Unable to read memory at 000001e4`eef81000, file is incomplete
0:005> .writemem C:\temp\svchost_cs.dat 000001e4`ef0c0000 L4000
Writing 4000 bytes........
```

Memory sections written to disk using .writemem WinDBG extension

Consequently, additional blocks of memory were dumped around the suspicious strings — for instance, those containing `%02d/%02d/%02d %02d:%02d:%02d`, `%s (admin)`, or `Content-Length: %d` — in an effort to capture more complete data. Although this did not yield a fully parsable beacon configuration in this specific instance, the discovered indicators, combined with previous string analysis, further reinforced that a Cobalt Strike payload had indeed been running within the process at the time of the crash.



Dumped memory region containing potential Cobalt Strike strings

Summing Up

The Nitrogen ransomware group exemplifies a modern, multi-stage intrusion operation that blends social engineering, evasive malware, and post-exploitation frameworks. By abusing malvertising — often disguising payloads as legitimate tools like WinSCP, Advanced IP Scanner, or FileZilla — Nitrogen establishes initial access via DLL sideloading, with malicious loaders delivering backdoor functionality through NitrogenLoader.

Once inside the network, Cobalt Strike becomes their tool of choice for lateral movement, command and control, and post-compromise activity. In our case study, Nitrogen used a compromised host as a pivot system while simultaneously wiping critical Windows logs to hinder detection and response efforts.

Throughout this post, we highlighted various ways to detect and extract Cobalt Strike configurations, including pattern analysis, byte-level XOR decryption, and custom YARA rules. In particular, we emphasized the power of crash dump analysis — specifically using Windows Error Reporting (WER) artifacts and WinDBG — to uncover in-memory indicators of Cobalt Strike beacons, configuration strings, and HTTP response structures embedded in dump files.

With that being said—stay safe, make use of lesser-known artifacts like WER, crash dumps, and UAL — and always read the labels before you install something from an ad.

About the author:

Maurice Fielenbach

Maurice Fielenbach trains cybersecurity professionals in reverse engineering and malware analysis — his main area of focus — and digital forensics through his company, Hexastrike Cybersecurity. The company also develops tools for red and blue teams and publishes technical blog posts covering both offensive and defensive topics. He also serves as Head of CERT at r-tec, leading a team of forensic specialists, managing and investigating a wide range of security incidents.
