Spoofed Saudi Purchase Order Drops GuLoader – Part 2 | FortiGuard Labs

By James Slaughter

Published: 2022-07-12 · Archived: 2026-04-05 22:00:53 UTC

In part one of this blog, FortiGuard Labs examined a recently discovered e-mail delivered to a coffee company in Ukraine that was seemingly sent by an oil provider in Saudi Arabia. Purporting to contain an attached purchase order, the image of a PDF file was actually a link to an ISO file hosted in the cloud that contained an executable for GuLoader. What makes this case interesting is that this executable uses NSIS (Nullsoft Scriptable Install System) to deploy itself.

GuLoader (also known as CloudEye and vbdropper) dates to at least 2019 and is generally used to deploy other malware variants such as Agent Tesla, Formbook, and Lokibot.

In this second part of the series, I will showcase a dynamic analysis of the main file, PO#23754-1.exe, as well as investigate the shellcode file “rudesbies.Par”. It will also highlight some of the defences it puts in place to hinder analysis.

Affected Platforms: Windows
Impacted Users: Windows users
Impact: Potential to deploy additional malware for additional purposes
Severity Level: Medium

PO#23754-1.exe dynamic analysis

This sample has a basic level of awareness of its surroundings. If it is executed in a virtual environment that has an obvious artefact (e.g., VirtualBox Guest Additions Tray), it will halt immediately.

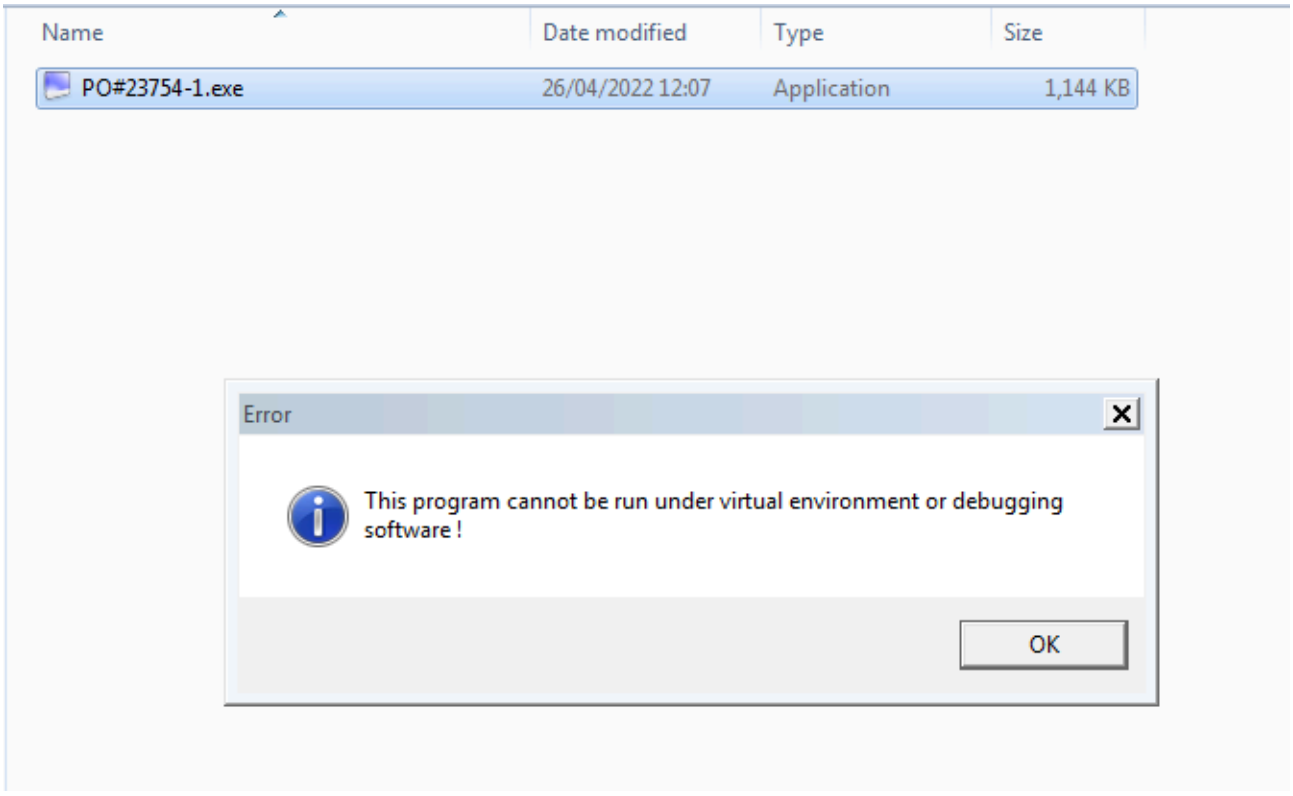

Figure 1. Halting upon detection of a virtual environment.

As the file is executed from inside a mounted ISO file, the execution path will flow from Explorer.

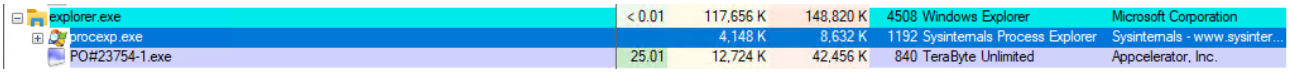

Figure 2. Execution tree.

By deploying from within a container such as an ISO, it is often possible to bypass MOTW (Mark-of-the-Web) controls that would stop files downloaded from the internet from executing.

Since this is a NSIS file, anyone viewing the screen at the time of execution will see also see a progress window as files are installed.

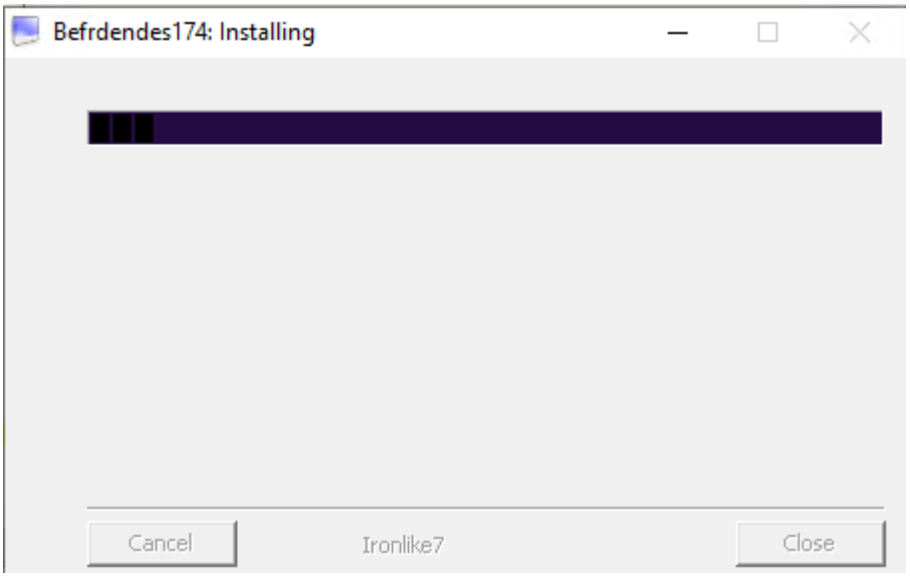

Figure 3. Installation window.

The label use, as shown in the NSIS script from part one of the blog, now becomes apparent when looking at the title bar for the window as well as the lower centre.

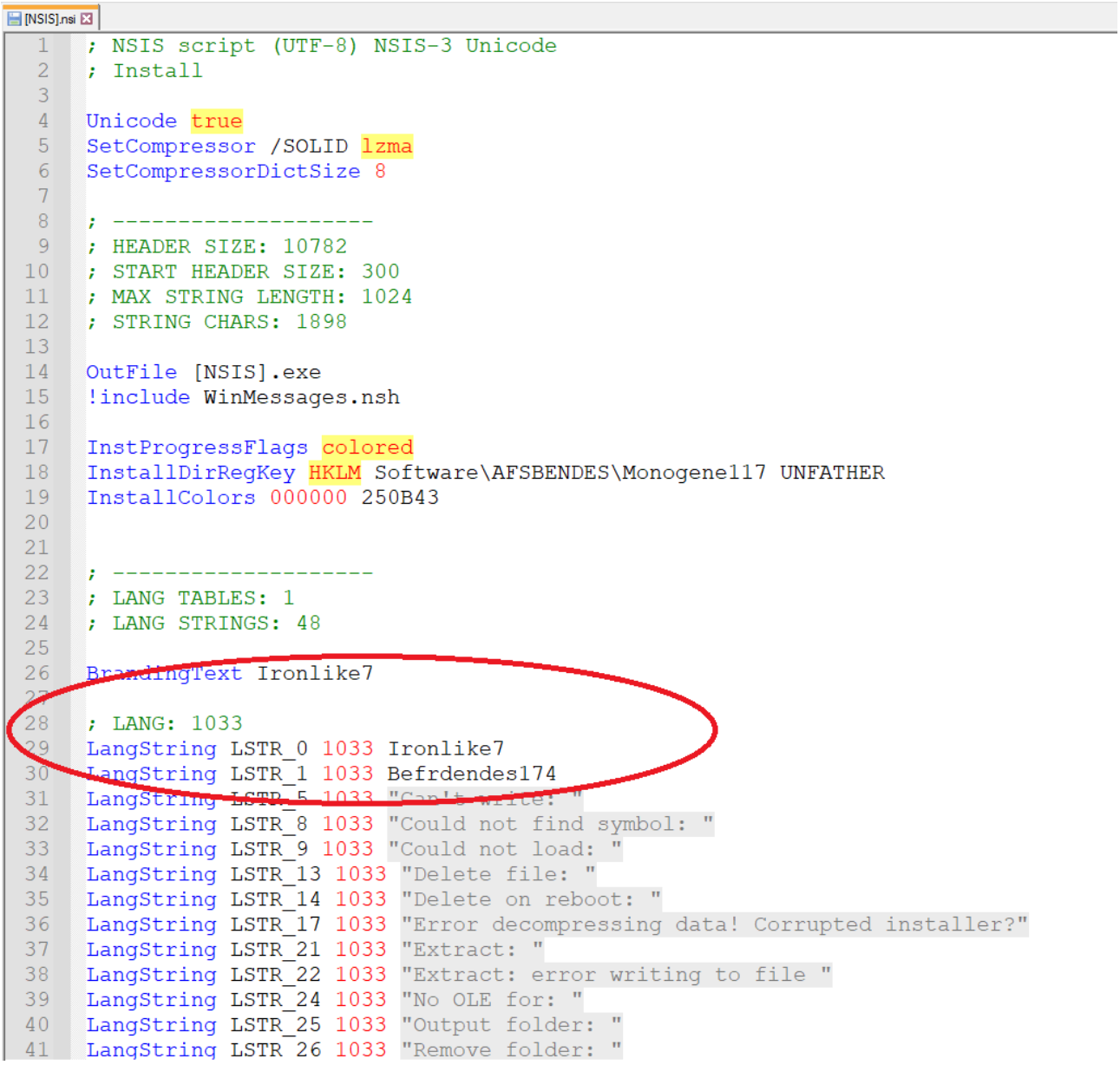

Figure 4. Script image from part one showing the title bar and label use.

As was suspected during static analysis, all identified files have been placed into the “Temp” directory of the active user account.

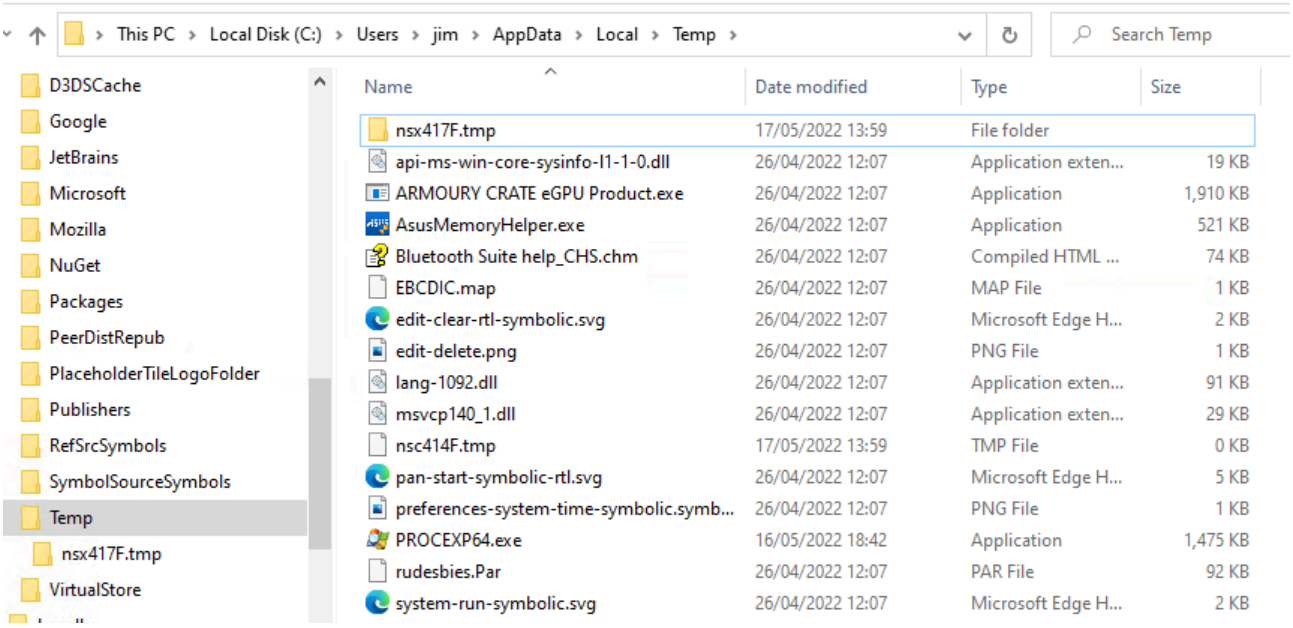

Figure 5. Files installed in $TEMP (note, PROCEXP.exe is not deployed by this malware).

A unique directory name is created (always beginning with “ns”) to store “System.dll”. A temp file that also is uniquely named (and again always prefixed with “ns”) is used in the NSIS deployment process.

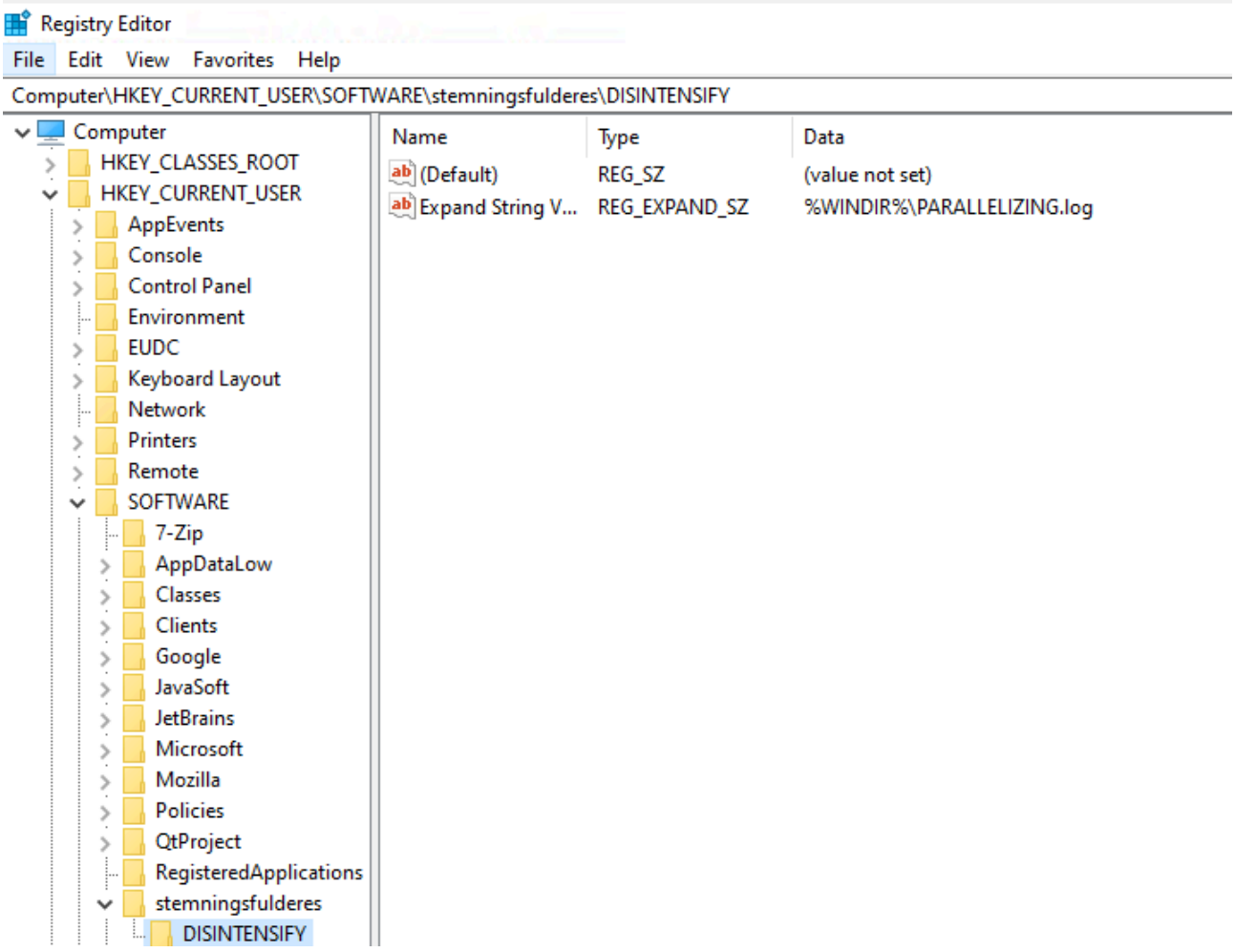

Figure 6. Registry entry created in HKCU\SOFTWARE.

In part one of the blog, a registry key was highlighted in the NSIS script. Figure 6 shows this when implemented on a victim system. It does not appear, however, that this is referenced later by the malware, nor does it appear that a file named “PARALLELIZING.log” is ever committed to disk. It is possible that this is a remnant of the testing phase of the NSIS file that was no longer needed for the active campaign.

System.dll

As mentioned earlier, “System.dll” is dropped into a unique directory. On its own, “System.dll” is a non-malicious file. However, the NSIS script will effectively use this DLL to proxy Windows API calls to “kernel32.dll”. This is done to read the file “rudesbies.Par” and inject its contents into the already running NSIS process. Doing things in this fashion makes it more difficult to trace where the origin of the call is coming from, thereby making it appear to be legitimate.

```
.text:10001817
.text:10001817 ; =============== S U B R O U T I N E =======================================
.text:10001817
.text:10001817 ; Attributes: bp-based frame
.text:10001817
.text:10001817                 public Call
.text:10001817 Call            proc near               ; DATA XREF: .rdata:off_100042D8↓o
.text:10001817
.text:10001817 String1         = word ptr -84h
.text:10001817 var_20          = byte ptr -20h
.text:10001817 arg_4           = dword ptr  0Ch
.text:10001817 arg_8           = dword ptr  10h
.text:10001817 arg_C           = dword ptr  14h
.text:10001817 arg_10          = dword ptr  18h
.text:10001817
.text:10001817                 push    ebp
```

Figure 7. Call function as seen in IDA.

As can be seen in Figure 7, the “Call” function is used to actually make the API call to “kernel32.dll”

```
EIP EAX EDI → 73CC1817   55       push ebp        Call
              73CC1818   8BEC     mov ebp,esp
```

Figure 8. Call function as seen in the debugger.

Despite a different offset, i.e., the last four bytes of the address in Figure 7 (1817), it can be seen in Figure 8 when “Call” is called by the code executing in the debugger.

In all, five calls are made through “System.dll” to “kernel32.dll”. These read “rudesbies.Par” and then allocate appropriate space in memory within the running executable. The series of five figures shown below demonstrate the entire journey to make this happen. Note the spelling of kernel as “KERNel”. Referring again back to part one, this is identical to the representation in the NSIS script.

```
EIP → 73CD1E68   FF75 CC     push dword ptr ss:[ebp-34]   [ebp-34]:L"KERNel32::CreateFilew(t \"C:\\Users\\SHEFFI~1\\AppData\\Local\\Temp\\rudesbies.Par\",
      73CD1E6E   8B3D 0440CD73  mov edi,dword ptr ds:[<&GlobalFree>]
      73CD1E74   FFD7        call edi
      73CD1E76   FF75 EC     push dword ptr ss:[ebp-14]   [ebp-14]:L"C:\\Users\\SHEFFI~1\\AppData\\Local\\Temp\\rudesbies.Par"
      73CD1E79   FFD7        call edi
      73CD1E7B   FF75 D4     push dword ptr ss:[ebp-2C]   [ebp-2C]:L"KERNel32"
      73CD1E7E   FFD7        call edi
```

Figure 9. CreateFileW is called to create a handle (a reference in memory within the OS) to the file “rudesbies.Par”.

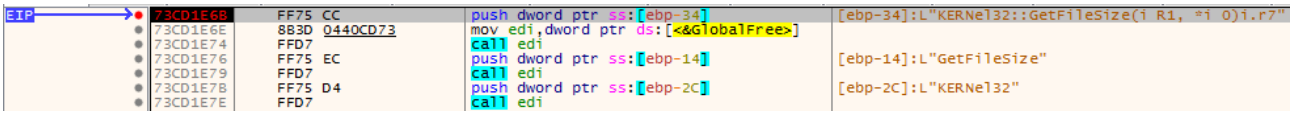

Figure 10. GetFileSize is called to determine the amount of memory required to store “rudesbies.Par”when opened.

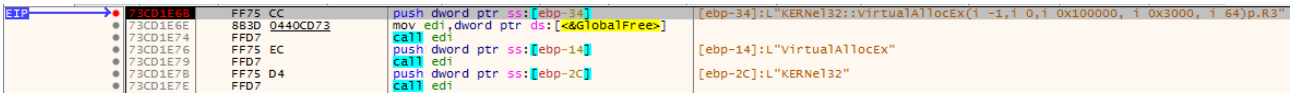

Figure 11. VirtualAllocEx is called to prepare a designated area of memory to store the contents of “rudesbies.Par”.

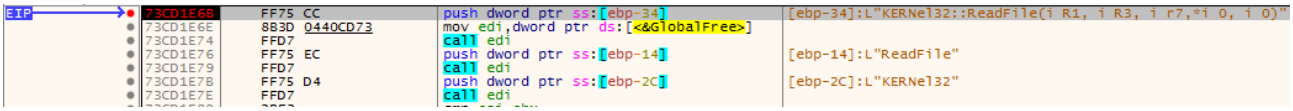

Figure 12. ReadFile is called to take the contents of “rudesbies.Par” and store them in memory.

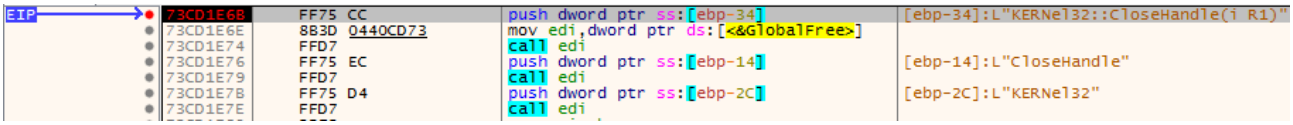

Figure 13. CloseHandle asks the OS to release the handle to “rudesbies.Par”.

Injection

Once rudesbies.Par is successfully read, it must then be injected back into the running NSIS process. This done into a random region of memory within the NSIS process.

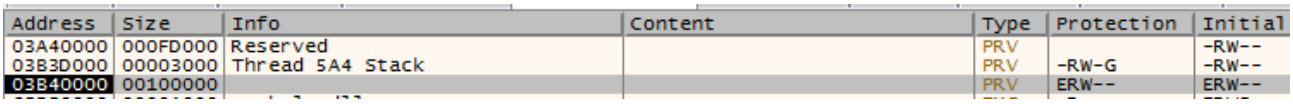

Figure 14. By observing the memory map in the debugger, the addition of a region with the above characteristics can be seen.

Viewing that region of memory initially shows what looks to be a very large number of “cmp” operations.

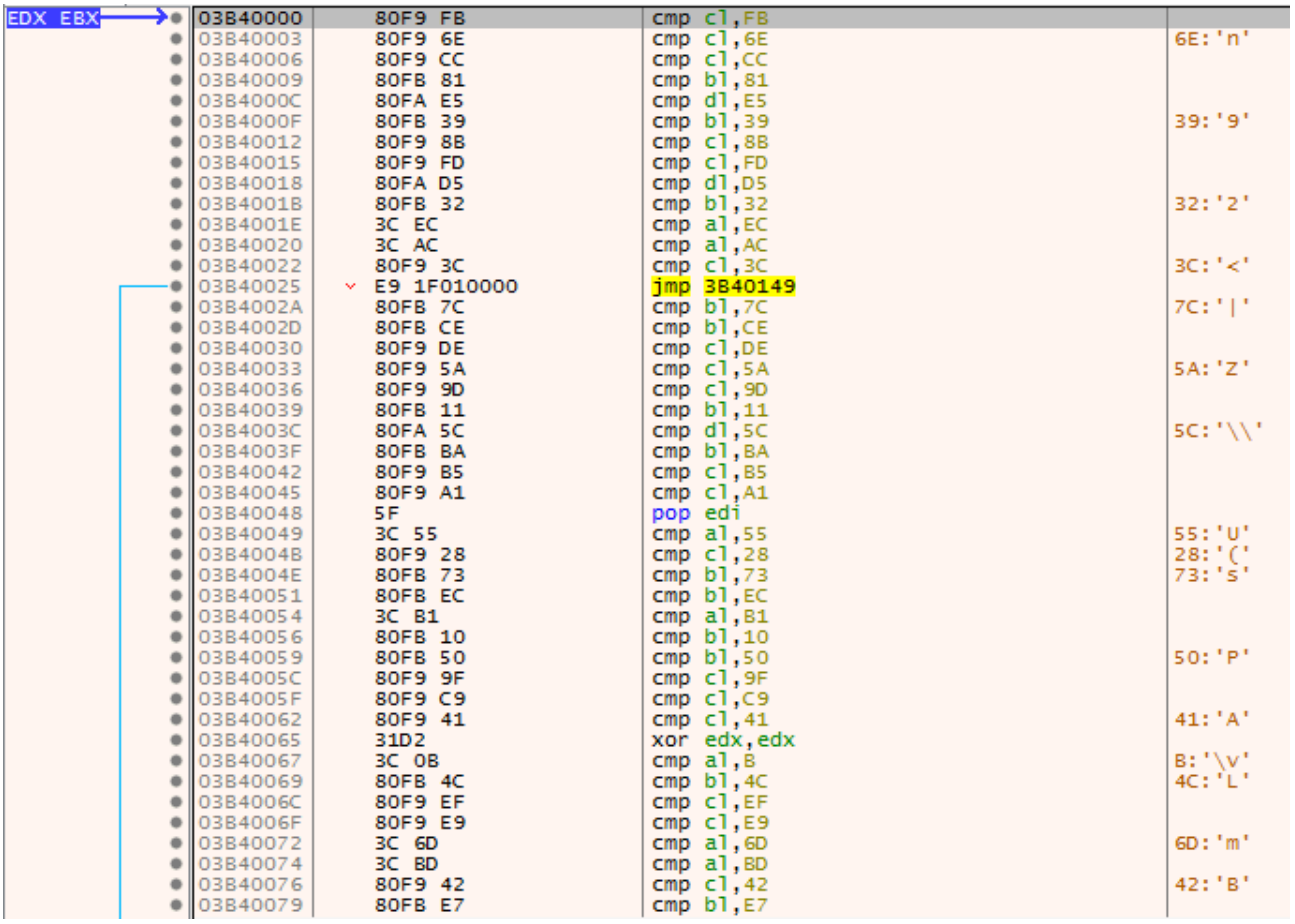

Figure 15. Initial view of the injected shellcode.

Shellcode

The instructions shown in Figure 15 are junk code and don't actually do anything. Regardless of the memory offset used to load the shellcode, the first set of meaningful instructions start at 014E.

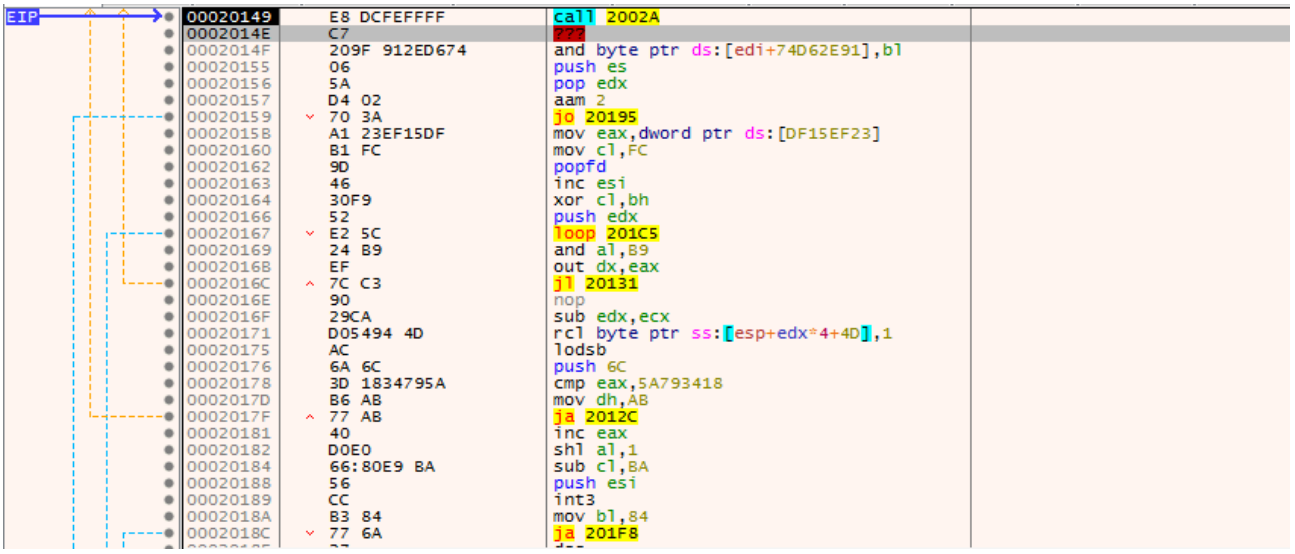

Figure 16. Start of actual instruction set (0002014E in this example, still in its encoded format).

From this point, the code will attempt to decode the next 17208 bytes of “rudesbies.Par”. It does this by setting a counter and then incrementing by four bytes while transiting a loop. Each time it goes through the loop, four bytes are added to the counter until it reaches 17208.

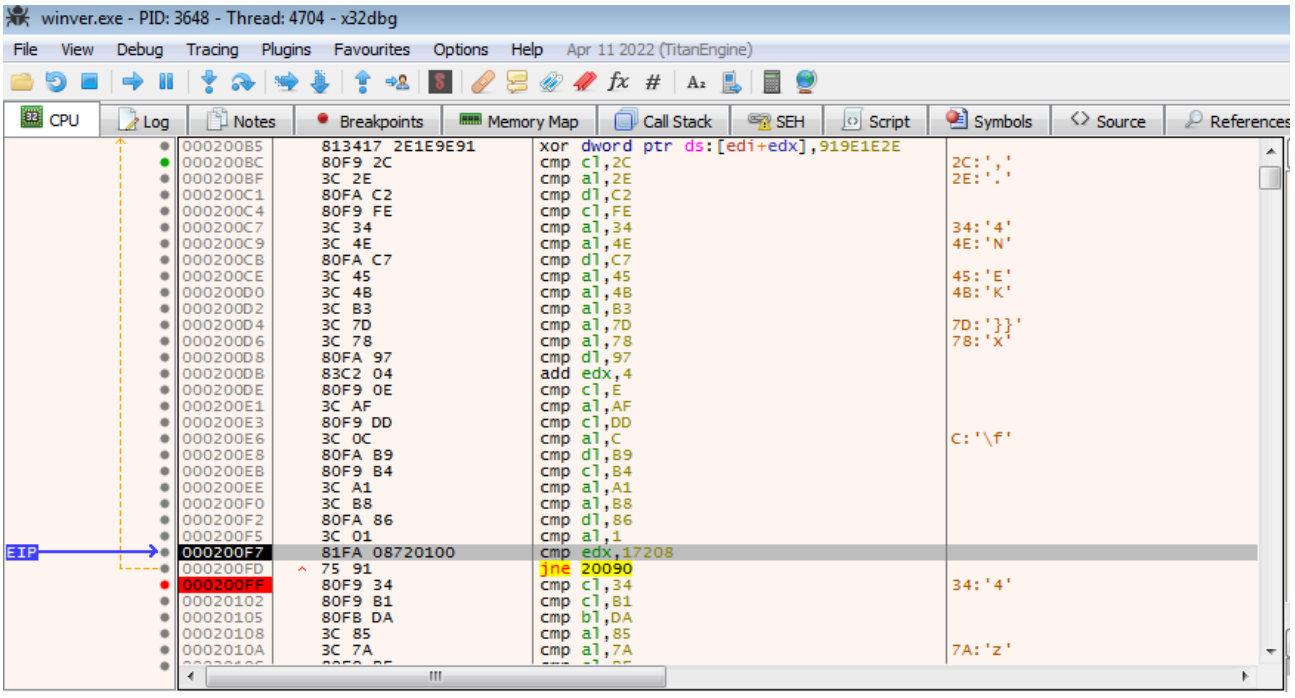

Figure 17. Compare instruction checking if the counter (edx) has reached 17208.

While that is occurring every four bytes, an instruction is XOR’d using the key 919E1E2E. Due to endian-ness, this is actually reversed and represented as 2E1E9E91.

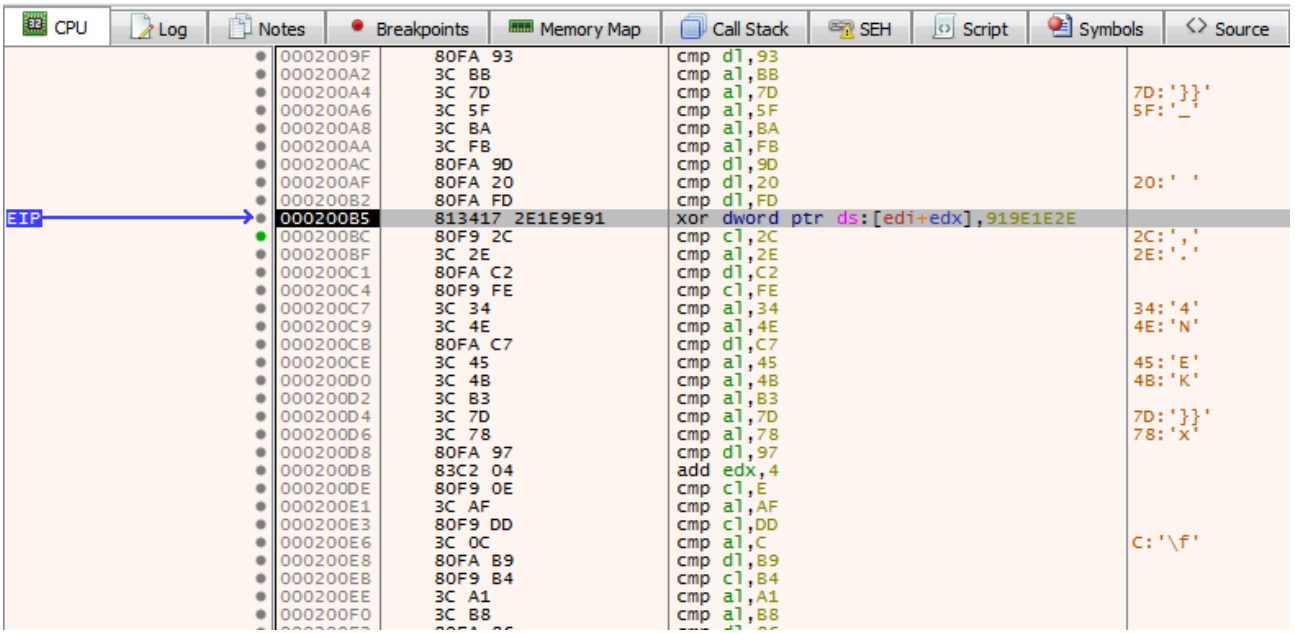

Figure 18. XOR key.

Once this has all been completed, further impediments to analysis have been added to hinder review by analysts such as periodic RDTSC (Read Time-Stamp Counter) instructions. Due to the time difference it takes to step

through instructions manually versus when running normally, this will create a break in the normal flow of the program.

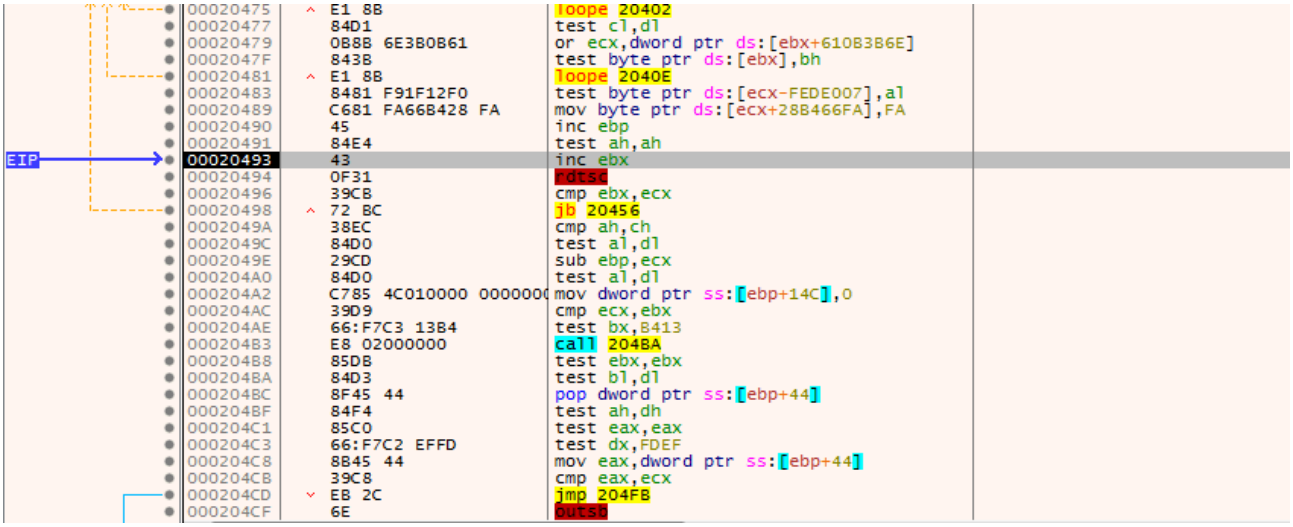

Figure 19. RDTSC instruction.

FortiGuard Labs was unfortunately unable to retrieve the final payload that was to have been dropped by this GuLoader sample. As mentioned earlier, GuLoader has been known to drop different types of malware, such as Agent Tesla, Formbook, and Lokibot.

Conclusion

Despite being a dropper, this sample shows the advantages of when malware developers increase the number of defences against analysis in their code. It slows down analysts attempting to share the details of a campaign, thereby maximizing the amount of time a malicious actor can keep their infrastructure up before becoming known and then compromised or taken down.

GuLoader can be adapted for delivery in different campaigns with different malware types. It is an interesting and active threat that will likely continue for some time to come.

Fortinet Protections

The GuLoader sample mentioned in this blog is detected by the following (AV) signature:

NSIS/Injector.AOW!tr

Fortinet customers are protected from this malware through FortiGuard's Antivirus, and CDR (content disarm and reconstruction) services and FortiMail, FortiClient, and FortiEDR solutions.

Due to the ease of disruption, damage to daily operations, potential impact to the reputation of an organization, and the unwanted destruction or release of personally identifiable information (PII), etc., it is important to keep all AV and IPS signatures up to date.

Fortinet also has multiple solutions designed to help train users to understand and detect phishing threats:

The FortiPhish Phishing Simulation Service uses real-world simulations to help organizations test user awareness and vigilance to phishing threats and to train and reinforce proper practices when users encounter targeted phishing attacks.

In addition, we suggest that organizations also have their end users go through our FREE NSE training: NSE 1 – Information Security Awareness. It includes a module on Internet threats that is designed to help end users learn how to identify and protect themselves from various types of phishing attacks.

IOCs

Filename	SHA256
PO#23754-1.ISO	c4debff9c0ec8a56aea5cd97215c6c906bd475ea8bd521fb9a346a4c992a0448
PO#23754-1.exe	14d52119459ef12be3a2f9a3a6578ee3255580f679b1b54de0990b6ba403b0fe
rudesbies.Par	4a1b6b30209c35ab180fa675a769e3285f54597963dd0bb29f7adb686ba88b79
GuLoader	344362b48b8aa9a89623e0bfd139d62f07e2523e600a79bb5af940f35d0740e5
GuLoader	3e79ce8ac441c8c8e777fe0804b67da0bd908a045d553a31893d95f15ae4ea01
GuLoader	9c5f99c37d042b0d6f2b5614fade06d373b2b954bf021bbf955df03693f2380d
GuLoader	53a0111fa7fca816618b65709ebf5d04ae9a64f9ebcfe08c60117a6a6f9d8030
GuLoader	5805e51dc4825c86b2d38c2a011429259954395e2d7b1fd06d83a2a3ec16fc14
GuLoader	1051d3690e70e4227a2b0a0aa87367fb09c49c55360c7a1880b2acfba0b77490
GuLoader	cc1ad7582d16db389c1b15a1cccdc188a85398165623876f4c7887743e54a9f9

Network IOCs

bounceclick[.]live/VVB/COrg_RYGGqN229.binb

Thanks to Fred Gutierrez who helped contribute to this blog.

Learn more about Fortinet's FortiGuard Labs threat research and intelligence organization and the FortiGuard Security Subscriptions and Services portfolio.
