KrustyLoader - Rust malware linked to Ivanti ConnectSecure compromises

Rédigé par Théo Letailleur - 29/01/2024 - dans CSIRT - Téléchargement

On 10th January 2024, Ivanti disclosed two zero-day critical vulnerabilities affecting Connect Secure VPN product: CVE-2024-21887 and CVE-2023-46805 allowing unauthenticated remote code execution. Volexity and Mandiant published articles reporting how these vulnerabilities were actively exploited by a threat actor. On 18th January, Volexity published new observations including hashes of Rust payloads downloaded on compromised Ivanti Connect Secure instances. This article presents a malware analysis of these unidentified Rust payloads that I labelled as KrustyLoader.

Introduction

On 10th January 2024, Ivanti disclosed two zero-day critical vulnerabilities affecting Connect Secure VPN product: CVE-2024-21887 and CVE-2023-46805[1] allowing unauthenticated remote code execution. Volexity[2] and Mandiant[3] published several articles showing how these vulnerabilities were actively exploited by a threat actor, tracked by Volexity as UTA0178 and by Mandiant as UNC5221.

On 18th January, Volexity published new indicators of compromise[4] including Rust payloads downloaded on compromised Ivanti Connect Secure appliances. Then on 21st and 24th of January, I published two posts on X[5] [6] summarizing the behaviour of those 12 Rust payloads. They share almost 100% code similarity and their main purpose is to download and execute a Sliver backdoor. I personally labelled this piece of malware as *KrustyLoader*.

Therefore, the purpose of this article is to provide more insights on this malware, reversing tips, as well as a script that automatically extracts the encrypted URL from any similar sample.

Basic information

KrustyLoader basic information

SHA256	47ff0ae9220a09bfad2a2fb1e2fa2c8ffe5e9cb0466646e2a940ac2e0cf55d04 816754f6eaf72d2e9c69fe09dcbe50576f7a052a1a450c2a19f01f57a6e13c17 c26da19e17423ce4cb4c8c47ebc61d009e77fc1ac4e87ce548cf25b8e4f4dc28 c7ddd58dcb7d9e752157302d516de5492a70be30099c2f806cb15db49d466026 d14122fa7883b89747f273c44b1f71b81669a088764e97256f97b4b20d945ed0 6f684f3a8841d5665d083dcf62e67b19e141d845f6c13ee8ba0b6ccdec591a01 a4e1b07bb8d6685755feca89899d9ead490efa9a6b6ccc00af6aaea071549960 ef792687b8bcd3c03bed4b09c4722bba921536802afe01f7cdb01cc7c3c60815 76902d101997df43cd6d3ac10470314a82cb73fa91d212b97c8f210d1fa8271f e47b86b8df43c8c1898abef15b8b7feffe533ae4e1a09e7294dd95f752b0fbb2 73657c062a7cc50a3d51853ec4df904bcb291fdc9cdd08eecaecb78826eb49b6 030eb56e155fb01d7b190866aaa8b3128f935afd0b7a7b2178dc8e2eb84228b0
File type	ELF 64-bit LSB pie executable x86_64 stripped, static-pie linked
File size	878824 bytes

Threat	Linux Rust downloader

Screenshots and extracts on this article are based on sample *030eb56e1[...]84228b0* (the highlighted hash above), but – as they are similar – the logic is the same for the other payloads.

Code analysis approach

You will not find a deep analysis into assembly code with tons of IDA screenshots, because it does not bring much value in this context. However, I find more interesting to explain what is my approach to quickly spot the useful parts of the code and get a general idea of its behaviour.

Usually we would start from the entry point and determine the flow of execution, symbols, and API functions. However, there are several difficulties to consider when reversing a Rust-based executable:

- The executable is statically linked, meaning that libraries are embedded into the executable, including Rust crates and the libc: it adds lots of functions that are not important to spend time during malware analysis.
- Since Rust is a high-level programming language, its abstractions tend to bring a “natural” obfuscation to the program code with lots of additional checks, temporary variables and built-in structures.
- Moreover, this sample is stripped, meaning that symbols and debug information are removed from the executable. In practice, it means that the disassembler will not be able to retrieve functions names of the program – and of the libraries – as well as structures, variable and constant names, etc.

As a result, with more than 2000 unnamed functions, it becomes quite tedious to determine what is the actual code of the developer, and what is not.

Therefore, I first executed the sample in a controlled environment (a Linux Debian-based virtual machine), to monitor any system and network activity.

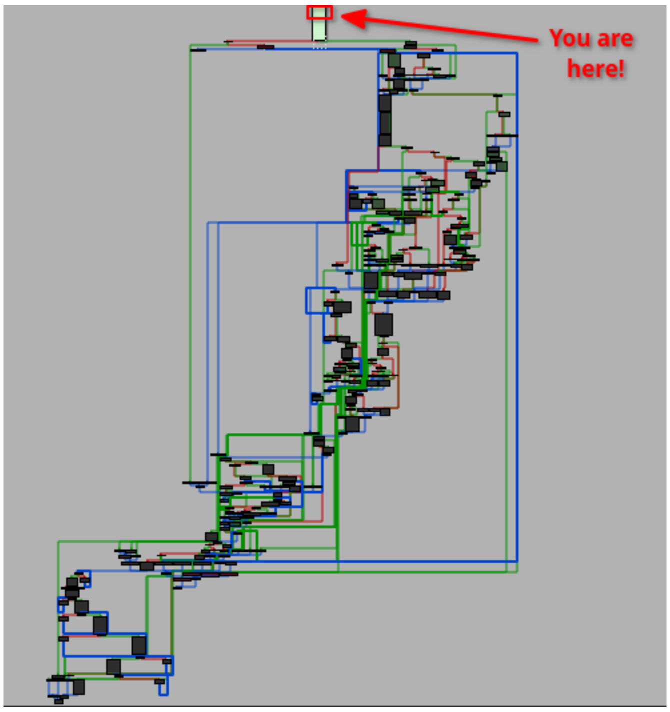
```
$ strace ./030eb56e155fb01._bad_elf
execve("./030eb56e155fb01._bad_elf", ["./030eb56e155fb01._bad_elf"], 0x7ffc20b137a0
/* 50 vars */) = 0
[...]
readlink("/proc/self/exe", "/home/user/iv/030eb56/030eb56e"..., 256) = 48
open("/home/user/iv/030eb56/030eb56e155fb01._bad_elf",
O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_PATH) = 6
readlink("/proc/self/fd/6", "/home/user/iv/030eb56/030eb56e"..., 4095) = 48
fstat(6, {st_mode=S_IFREG|0755, st_size=878824, ...}) = 0
stat("/home/user/iv/030eb56/030eb56e155fb01._bad_elf", {st_mode=S_IFREG|0755,
st_size=878824, ...}) = 0
close(6)                                = 0
unlink("/home/user/iv/030eb56/030eb56e155fb01._bad_elf") = 0
getppid()                               = 3033
readlink("/proc/self/exe", "/home/user/iv/030eb56/030eb56e"..., 256) = 58
readlink("/proc/self/exe", "/home/user/iv/030eb56/030eb56e"..., 256) = 58
stat("/tmp/0", 0x7fffe54a8700)          = -1 ENOENT (No such file or directory)
[...]
exit_group(0)                           = ?
```

I was first disappointed because the process exited instantaneously with no network activity and no impact on the filesystem. But there was a few interesting system calls executed:

- `readlink("/proc/self/exe"...)`: reads the value (the path) pointed by the symbolic link `/proc/self/exe`, meaning its executable (here `/home/user/iv/030eb56/030eb56e155fb01._bad_elf`);
- Then it opens its executable with `open` syscall, checks its file status with `fstats` (not sure why) and closes it;
- `unlink("/home/user/iv/030eb56/030eb56e155fb01._bad_elf")`: deletes itself;
- `stat("/tmp/0", ...)`: tests the existence of `/tmp/0` file, in this running context you can see the error explaining that it does not exist;
- Exits.

We can use this information to find the beginning of the main *useful* function by searching any references to `readlink` and `unlink` system calls, as well as `/proc/self/exe` and `/tmp/0` strings. However, those two strings did not bring interesting results (as I discovered later, they are stack strings so no reference!). But `/tmp/` and the two mentioned system calls were directly referenced from a big function that I determined as the main routine.

The main routine is called by another big function that I identified as a *Tokio* worker thread, responsible for running asynchronous tasks. *Tokio*7 is a famous Rust crate, very handy when building asynchronous network applications. I quickly identified the purpose of this function thanks to a reference to `TOKIO_WORKER_THREADS` string, which allowed me to completely skip its code flow and go straight to the main routine.



KrustyLoader main routine

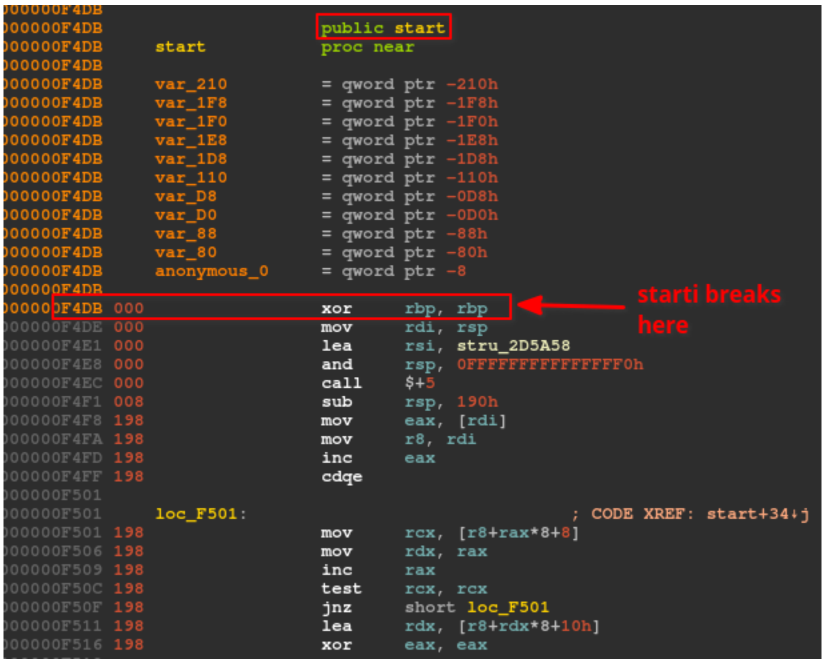
Once we identify the exception/error handling code inside the function, the execution flow becomes more obvious. To help with the reverse engineering, I debugged the program alongside with GDB. Since it is a stripped PIE (Position Independent Executable) binary – simply put, code segment base address is randomized – we can neither break on function names nor predictable addresses. The `start` GDB command is not able to break on the main function in this configuration. Thankfully, GDB has another command called `starti`8 that sets a temporary breakpoint at the very first instruction of a program's execution and then invokes the 'run' command. This command allows us to start the process, break instantaneously, and get the base address of the code segment loaded in memory.

```
$ gdb 030eb56e155fb01._bad_elf
[...]
Reading symbols from 030eb56e155fb01._bad_elf...
(No debugging symbols found in 030eb56e155fb01._bad_elf)
gef➤  starti
Starting program: /home/user/iv/030eb56/030eb56e155fb01._bad_elf
[*] Failed to find objfile or not a valid file format: [Errno 2] No such file or
directory: 'system-supplied DSO at 0x7ffff7fde000'

Program stopped.
0x00007ffff7d364db in ?? ()
[...]
──────────────────────────────────────────────────────────────────────────────────────────

 code:x86:64 ────
   0x7ffff7d364cd                  call   0x7ffff7dc9bc4
   0x7ffff7d364d2                  mov    edi, DWORD PTR [rsp+0xc]
   0x7ffff7d364d6                  call   0x7ffff7dc742e
 → 0x7ffff7d364db                  xor    rbp, rbp
   0x7ffff7d364de                  mov    rdi, rsp
   0x7ffff7d364e1                  lea    rsi, [rip+0x2c6570]        #
0x7ffff7ffca58
   0x7ffff7d364e8                  and    rsp, 0xfffffffffffffff0
   0x7ffff7d364ec                  call   0x7ffff7d364f1
   0x7ffff7d364f1                  sub    rsp, 0x190
──────────────────────────────────────────────────────────────────────────────────────────
 threads ────
[#0] Id 1, Name: "030eb56e155fb01", stopped 0x7ffff7d364db in ?? (), reason: STOPPED
──────────────────────────────────────────────────────────────────────────────────────────
 trace ────
[#0] 0x7ffff7d364db → xor rbp, rbp
──────────────────────────────────────────────────────────────────────────────────────────

gef➤
```



```
000000F4DB
000000F4DB                              public start
000000F4DB      start                   proc near
000000F4DB
000000F4DB      var_210                 = qword ptr -210h
000000F4DB      var_1F8                 = qword ptr -1F8h
000000F4DB      var_1F0                 = qword ptr -1F0h
000000F4DB      var_1E8                 = qword ptr -1E8h
000000F4DB      var_1D8                 = qword ptr -1D8h
000000F4DB      var_110                 = qword ptr -110h
000000F4DB      var_D8                  = qword ptr -0D8h
000000F4DB      var_D0                  = qword ptr -0D0h
000000F4DB      var_88                  = qword ptr -88h
000000F4DB      var_80                  = qword ptr -80h
000000F4DB      anonymous_0             = qword ptr -8
000000F4DB
000000F4DB 000                          xor     rbp, rbp
000000F4DE 000                          mov     rdi, rsp
000000F4E1 000                          lea     rsi, stru_2D5A58
000000F4E8 000                          and     rsp, 0FFFFFFFFFFFFFFF0h
000000F4EC 000                          call    $+5
000000F4F1 008                          sub     rsp, 190h
000000F4F8 198                          mov     eax, [rdi]
000000F4FA 198                          mov     r8, rdi
000000F4FD 198                          inc     eax
000000F4FF 198                          cdqe
000000F501
000000F501      loc_F501:                                         ; CODE XREF: start+34↓j
000000F501 198                          mov     rcx, [r8+rax*8+8]
000000F506 198                          mov     rdx, rax
000000F509 198                          inc     rax
000000F50C 198                          test    rcx, rcx
000000F50F 198                          jnz     short loc_F501
000000F511 198                          lea     rdx, [r8+rdx*8+10h]
000000F516 198                          xor     eax, eax
```

IDA disassembly view - KrustytLoader's first instruction

GDB breaks at the first instruction at address `0x7ffff7d364db` in our example. IDA disassembly view shows that the first instruction of the program (pointed by start symbol) is at `0xF4DB`. Then, using a subtle mathematical operation, we can retrieve the base address and determine the address of the main routine: `0x7ffff7d364db - 0xF4DB + 2E70B (offset of the main routine) = 0x7ffff7d5570b`. We can now break at `0x7ffff7d5570b` and finally start debugging the main routine normally.

The next section describes the results of my analysis based on this approach. I also used Sysdig[9] to monitor the system calls and general activity on the virtual machine. This is a great system monitoring tool that would deserve its own article!

KrustyLoader Behaviour

Based on reverse engineering and dynamic analysis, the behaviour of KrustyLoader can be summarized in the following main points:

- The malware reads `/proc/self/exe` to gets its path (*readlink*) and deletes itself (*unlink*)
- Then the following checks must be validated else the program exits:
    - It gets the process parent ID (PPID) using *getppid* syscall and exits if PPID is 1.
    - Anti-debug checks: it reads `/proc/self/exe` again (now the value suffixed with `" (deleted)"`) and exits if it contains **gdb** or **lldb** (both debuggers) strings.
    - It checks the existence of `/tmp/0` and exits if it does not.
    - It checks if its executable (pointed by `/proc/self/exe`) is located in `/tmp/` directory. If it's not in `/tmp/` directory, it exits.
- Once all the checks successfully passed, the malware starts doing interesting stuff:
    - It creates in `/tmp` directory a new file with a filename made of 10 random alphanumeric characters.
    - It decrypts a hardcoded URL, and sends a GET HTTP request to that URL.
    - In result, it receives an encrypted response from the remote server.
    - The content is decrypted and written to the random file.
    - It makes the random file executable using system command `chmod +x /tmp/randomfile`.
    - Finally, it tries to execute the newly created executable and exits.

As a general point, there is a bit of obfuscation: most symbols are XOR-encrypted stack strings.

The process of decryption used by the malware to retrieve the URL has three steps:

1. It hex-decodes (the equivalent of `bytes.fromhex()` in Python) the encrypted URL;
2. XOR each byte with a 1-byte key;
3. And uses **AES-128 CFB-1 mode**[10] with hardcoded key and initialization vector to decrypt and get the URL.

AES-128 CFB is also used to decrypt the payload sent by the remote HTTP server.

What about the executed payloads? Based on my observations, all the samples download a Sliver (Golang) backdoor, though from different URLs. The Sliver backdoors contact their C2 server using HTTP/HTTPS communication. Sliver[11] is an open-source adversary simulation tool that is gaining popularity amongst threat actors, since it provides a practical command and control framework.

The list of domains and URLs can be found in this GitHub repository: https://github.com/synacktiv/krustyloader-analysis.

Extraction and detection

Extraction of the URL

I developed a simple script to statically retrieve and decrypt the URL used by *KrustyLoader* to get the *Sliver* backdoor. It allows extracting the pieces of information we only need without executing the malware. The script is available here: https://github.com/synacktiv/krustyloader-analysis/blob/main/krusty_extractor.py. It requires pycryptodome Python package and a decent Python version to run. It automatically extracts the XOR key, the AES key, the AES initialization vector and the encrypted URL.

```
$ python krusty_extractor.py 030eb56/030eb56e155fb01._bad_elf
Sample SHA256sum: 030eb56e155fb01d7b190866aaa8b3128f935afd0b7a7b2178dc8e2eb84228b0
XOR KEY: 0x81
AES-128 CFB KEY: b1e228b4b5723d41a575d993b70c906b
AES-128 CFB IV: 27bb7db8021cd9ade3520a6e67f43ac5
Decrypted Stage Hoster URL:
http://bringthenoiseappnew.s3.amazonaws.com/iEgJ4J7Uc9YgC

$ python krusty_extractor.py a4e1b07/a4e1b0._bad_elf
Sample SHA256sum: a4e1b07bb8d6685755feca89899d9ead490efa9a6b6ccc00af6aaea071549960
XOR KEY: 0x81
AES-128 CFB KEY: b1e228b4b5723d41a575d993b70c906b
AES-128 CFB IV: 27bb7db8021cd9ade3520a6e67f43ac5
Decrypted Stage Hoster URL: http://bbr-promo.s3.amazonaws.com/NWEUW983Ve4g1
```

As you can observe in the extract above, it successfully decrypts the URL of both samples (and it works for all 12 samples). When I first ran the script on all samples, I was quite disappointed to notice that they also share the exact same cryptographic parameters. 😄 At least it sped up my analysis, and it could still be handy in case there are new variants with different XOR key or AES parameters.

Detection

You can find a Yara rule here to detect similar KrustyLoader samples: https://github.com/synacktiv/krustyloader-analysis/blob/main/KrustyLoader.yar. It searches specific strings I mentioned and some AES routines.

Conclusion

Rust payloads detected by Volexity team turn out to be pretty interesting Sliver downloaders as they were executed on Ivanti Connect Secure VPN after the exploitation of CVE-2024-21887 and CVE-2023-46805. KrustyLoader – as I dubbed it – performs specific checks in order to run only if conditions are met. The fact that KrustyLoader was developed in Rust brings additional difficulties to obtain a good overview of its behaviour. A script as well as a Yara rule are publicly available to help detection and extraction of indicators.

If any organization needs assistance in doubt removal or responding to a compromise, please feel free to contact Synacktiv.