Hive ransomware gets upgrades in Rust

microsoft.com/security/blog/2022/07/05/hive-ransomware-gets-upgrades-in-rust/

July 5, 2022

Hive ransomware is only about one year old, having been first observed in June 2021, but it has grown into one of the most prevalent ransomware payloads in the ransomware-as-a-service (RaaS) ecosystem. With its latest variant carrying several major upgrades, Hive also proves it's one of the fastest evolving ransomware families, exemplifying the continuously changing ransomware ecosystem.

The upgrades in the latest variant are effectively an overhaul: the most notable changes include a full code migration to another programming language and the use of a more complex encryption method. The impact of these updates is far-reaching, considering that Hive is a RaaS payload that Microsoft has observed in attacks against organizations in the healthcare and software industries by large ransomware affiliates like DEV-0237.

Microsoft Threat Intelligence Center (MSTIC) discovered the new variant while analyzing detected Hive ransomware techniques for dropping *.key* files. We know that Hive drops its encryption keys file, which contains encrypted keys used to decrypt encrypted files, and uses a consistent naming pattern:

```
[KEY_NAME].key.[VICTIM_IDENTIFIER]
(e.g., BiKtPupMjgyESaene0Ge5d0231uiKq1PFMFUEBNhAYv_.key.ab123)
```

The said *.key* files were missing the [VICTIM_IDENTIFIER] part of the file name, prompting deeper analysis of the Hive ransomware that dropped them. This analysis led to the discovery of the new Hive variant and its multiple versions, which exhibit slightly different available parameters in the command line and the executed processes.

Analyzing these patterns in samples of the new variants, we discovered even more samples, all with a low detection rate and none being correctly identified as Hive. In this blog we will share our in-depth analysis of the new Hive variant, including its main features and upgrades, with the aim of equipping analysts and defenders with information to better identify and protect organizations against malware attacks relying on Hive.

Analysis and key findings

The switch from GoLang to Rust

The main difference between the new Hive variant and old ones is the programming language used. The old variants were written in Go (also referred to as GoLang), while the new Hive variant is written in Rust.

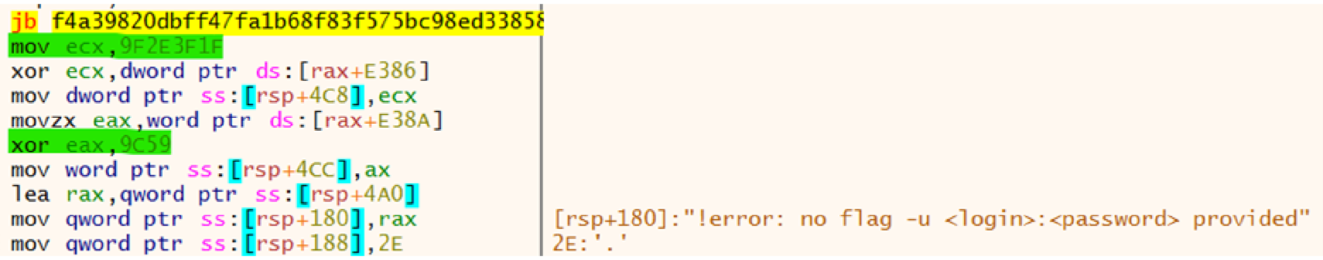
Hive isn’t the first ransomware written in Rust—BlackCat, another prevalent ransomware, was the first. By switching the underlying code to Rust, Hive benefits from the following advantages that Rust has over other programming languages:

- It offers memory, data type, and thread safety
- It has deep control over low-level resources
- It has a user-friendly syntax
- It has several mechanisms for concurrency and parallelism, thus enabling fast and safe file encryption
- It has a good variety of cryptographic libraries
- It’s relatively more difficult to reverse-engineer

String encryption

The new Hive variant uses string encryption that can make it more evasive. Strings reside in the *.rdata* section and are decrypted during runtime by XORing with constants. The constants that are used to decrypt the same string sometimes differ across samples, making them an unreliable basis for detection.

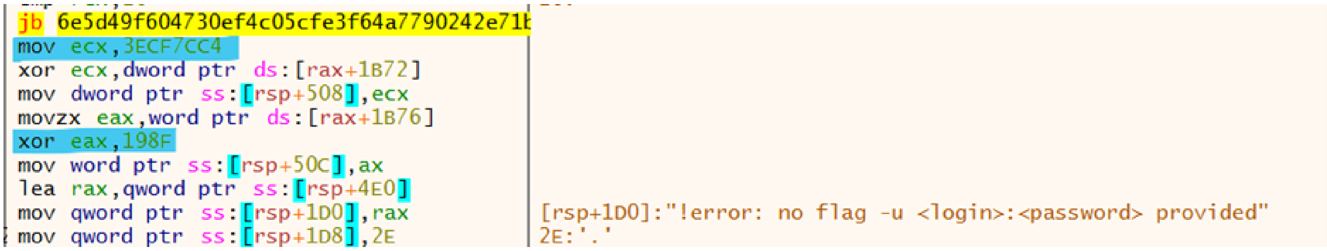
For example, let’s look at the section where part of the string *“!error no flag -u <login>:<password> provided”* is decrypted. In one sample (SHA-256: f4a39820dbff47fa1b68f83f575bc98ed33858b02341c5c0464a49be4e6c76d3), the constants are 0x9F2E3F1F and 0x95C9:



```
jb f4a39820dbff47fa1b68f83f575bc98ed33858
mov ecx,9F2E3F1F
xor ecx,dword ptr ds:[rax+E386]
mov dword ptr ss:[rsp+4C8],ecx
movzx eax,word ptr ds:[rax+E38A]
xor eax,9C59
mov word ptr ss:[rsp+4CC],ax
lea rax,qword ptr ss:[rsp+4A0]
mov qword ptr ss:[rsp+180],rax          [rsp+180]:"!error: no flag -u <login>:<password> provided"
mov qword ptr ss:[rsp+188],2E           2E:'.'
```

Figure 1 – String decryption using constants 0x9F2E3F1F and 0x95C9

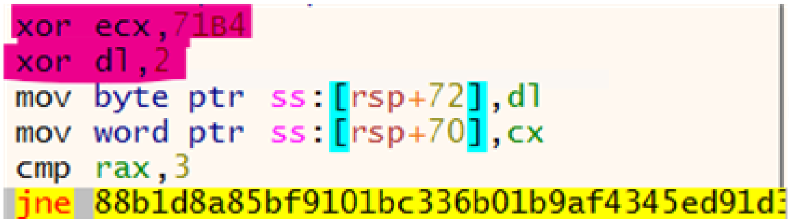
In another sample (SHA-256: 6e5d49f604730ef4c05cfe3f64a7790242e71b4ecf1dc5109d32e811acf0b053), the constants are 0x3ECF7CC4 and 0x198F:



```
jb 6e5d49f604730ef4c05cfe3f64a7790242e71b
mov ecx,3ECF7CC4
xor ecx,dword ptr ds:[rax+1B72]
mov dword ptr ss:[rsp+508],ecx
movzx eax,word ptr ds:[rax+1B76]
xor eax,198F
mov word ptr ss:[rsp+50C],ax
lea rax,qword ptr ss:[rsp+4E0]
mov qword ptr ss:[rsp+1D0],rax          [rsp+1D0]:"!error: no flag -u <login>:<password> provided"
mov qword ptr ss:[rsp+1D8],2E           2E:'.'
```

Figure 2 – String decryption using constants 0x3ECF7CC4 and 0x198F

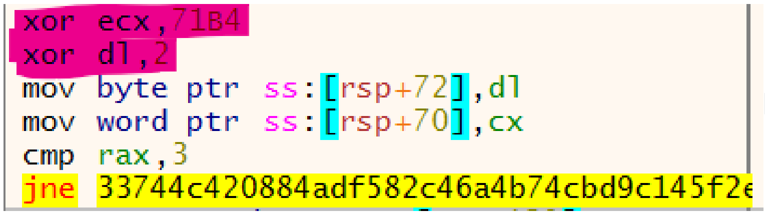
Some samples do share constants when decrypting the same string. For example, let’s look where the parameter string *“-da”* is decrypted. In one sample (SHA-256: 88b1d8a85bf9101bc336b01b9af4345ed91d3ec761554d167fe59f73af73f037), the constants are 0x71B4 and 2:



```
xor ecx,71B4
xor dl,2
mov byte ptr ss:[rsp+72],dl
mov word ptr ss:[rsp+70],cx
cmp rax,3
jne 88b1d8a85bf9101bc336b01b9af4345ed91d
```

Figure 3 – String decryption using constants 0x71B4 and 2

In another sample (SHA-256: 33744c420884adf582c46a4b74cbd9c145f2e15a036bb1e557e89d6fd428e724), the constants are the same:

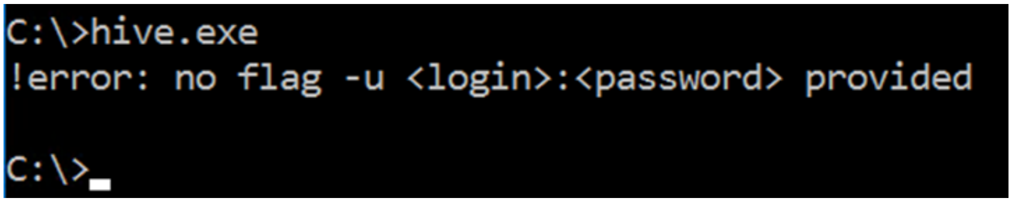


```
xor ecx,71B4
xor dl,2
mov byte ptr ss:[rsp+72],dl
mov word ptr ss:[rsp+70],cx
cmp rax,3
jne 33744c420884adf582c46a4b74cbd9c145f2e
```

Figure 4 – String decryption in a different sample also using constants 0x71B4 and 2

Command-line parameters

In old Hive variants, the username and the password used to access the Hive ransom payment website are embedded in the samples. In the new variant, these credentials must be supplied in the command line under the *"-u"* parameter, which means that they can't be obtained by analysts from the sample itself.



```
C:\>hive.exe
!error: no flag -u <login>:<password> provided

C:\>_
```

Figure 5 – Without a username and a password, the sample won't continue its execution

Like most modern ransomware, Hive introduces command-line parameters, which allow attackers flexibility when running the payload by adding or removing functionality. For example, an attacker can choose to encrypt files on remote shares or local files only or select the minimum file size for encryption. In the new Hive variant, we found the following parameters across different samples:

Parameter	Functionality
-no-local	Don't encrypt local files
-no-mounted	Don't encrypt files on mounted network shares
-no-discovery	Don't discover network shares
-local-only	Encrypt only local files
-network-only	Encrypt only files on network shares

-explicit-only	Encrypt specific folder(s). For example, *‘-explicit-only c:\mydocs c:\myphotos’*
-min-size	Minimum file size, in bytes, to encrypt. For example, *‘-min-size 102400’* will encrypt files with size equal or greater than 100kb
-da	[Usage is being analyzed.]
-f	[Usage is being analyzed.]
-force	[Usage is being analyzed.]
-wmi	[Usage is being analyzed.]

Overall, it appears different versions have different parameters that are constantly updated. Unlike in previous variants where there was a *‘help’* menu, in the new variant, the attacker must know the parameters beforehand. Since all strings are encrypted, it makes finding the parameters challenging for security researchers.

Stopped services and processes

Like most sophisticated malware, Hive stops services and processes associated with security solutions and other tools that might get in the way of its attack chain. Hive tries to impersonate the process tokens of *trustedinstaller.exe* and *winlogon.exe* so it can stop Microsoft Defender Antivirus, among other services.

Hive stops the following services:

```
windefend, msmpsvc, kavsvc, antivirservice, zhudongfungyu, vmm, vmwp, sql, sap, oracle,
mepocs, veeam, backup, vss, msexchange, mysql, sophos, pdfservice, backupexec, gxblr, gxvss,
gxclmgrs, gxvcd, gxcimgr, gxmmm, gxvsshwprov, gxfwd, sap, qbcfmonitorservice, qbidpservice,
acronisagent, veeam, mvarmor, acrsch2svc
```

It also stops the following processes:

```
dbsnmp, dbeng50, bedbh, excel, encsvc, visios, firefox, isqlplussvc, mspub, mydesktopqos,
notepad, ocautoupds, ocomm, ocssd, onenote, outlook, sqbcoreservice, sql, steam,
tbirdconfig, thunderbird, winword, wordpad, xfssvccon, vxmon, benetns, bengien, pvlsvr,
raw_agent_svc, cagservice, sap, qbidpservice, qbcfmonitorservice, teamviewer_service,
teamviewer, tv_w32, tv_x64, cvd, saphostexec, sapstartsrv, avscc, dellsystemdetect,
enterpriseclient, veeam, thebat, cvfwd, cvods, vsnapvss, msaccess, vaultsvc, beserver,
appinfo, qbdmgrn, avagent, spooler, powerpnt, cvmountd, synctime, oracle, wscsvc, winmgmt,
*sql*
```

Launched processes

As part of its ransomware activity, Hive typically runs processes that delete backups and prevent recovery. There are differences between versions, and some samples may not execute all these processes, but one sample that starts the most processes is SHA-256: 481dc99903aa270d286f559b17194b1a25deca8a64a5ec4f13a066637900221e:

- *“vssadmin.exe delete shadows /all /quiet”*

- *“wmic.exe shadowcopy delete”*
- *“wbadmin.exe delete systemstatebackup”*
- *“wbadmin.exe delete catalog -quiet”*
- *“bcdedit.exe /set {default} recoveryenabled No”*
- *“bcdedit.exe /set {default} bootstatuspolicy ignoreallfailures”*
- *“wbadmin.exe delete systemstatebackup -keepVersions:3”*

Ransom note

Hive’s ransom note has also changed, with the new version referencing the *.key* files with their new file name convention and adding a sentence about virtual machines (VMs).

The older variants had an embedded username and password (marked as *hidden*). In the new variant, the username and password are taken from the command line parameter *-u* and are labeled *test_hive_username and test_hive_password*.

Old ransom note text:

```
 Your network has been breached and all data were encrypted.
Personal data, financial reports and important documents are ready to disclose.

To decrypt all the data and to prevent exfiltrated files to be disclosed at
http://hive[REDACTED].onion/
you will need to purchase our decryption software.

Please contact our sales department at:

   http://hive[REDACTED].onion/

      Login:    [REDACTED]
      Password: [REDACTED]

To get an access to .onion websites download and install Tor Browser at:
   https://www.torproject.org/ (Tor Browser is not related to us)


Follow the guidelines below to avoid losing your data:

- Do not modify, rename or delete *.key.abc12 files. Your data will be
   undecryptable.
- Do not modify or rename encrypted files. You will lose them.
- Do not report to the Police, FBI, etc. They don't care about your business.
   They simply won't allow you to pay. As a result you will lose everything.
- Do not hire a recovery company. They can't decrypt without the key.
   They also don't care about your business. They believe that they are
   good negotiators, but it is not. They usually fail. So speak for yourself.
- Do not reject to purchase. Exfiltrated files will be publicly disclosed.
```

New ransom note text:

```
Your network has been breached and all data were encrypted.
Personal data, financial reports and important documents are ready to disclose.

To decrypt all the data and to prevent exfiltrated files to be disclosed at
http://hive[REDACTED].onion/
you will need to purchase our decryption software.

Please contact our sales department at:

   http://hive[REDACTED].onion/

      Login:    test_hive_username
      Password: test_hive_password

To get an access to .onion websites download and install Tor Browser at:
   https://www.torproject.org/ (Tor Browser is not related to us)


Follow the guidelines below to avoid losing your data:

- Do not delete or reinstall VMs. There will be nothing to decrypt.
- Do not modify, rename or delete *.key files. Your data will be
   undecryptable.
- Do not modify or rename encrypted files. You will lose them.
- Do not report to the Police, FBI, etc. They don't care about your business.
   They simply won't allow you to pay. As a result you will lose everything.
- Do not hire a recovery company. They can't decrypt without the key.
   They also don't care about your business. They believe that they are
   good negotiators, but it is not. They usually fail. So speak for yourself.
- Do not reject to purchase. Exfiltrated files will be publicly disclosed.
```

Encryption

The most interesting change in the Hive variant is its cryptography mechanism. The new variant was first uploaded to VirusTotal on February 21, 2022, just a few days after a group of researchers from Kookmin University in South Korea published the paper “A Method for Decrypting Data Infected with Hive Ransomware” on February 17, 2022. After a certain period of development, the new variant first appeared in Microsoft threat data on February 22.

The new variant uses a different set of algorithms: Elliptic Curve Diffie-Hellmann (ECDH) with Curve25519 and XChaCha20-Poly1305 (authenticated encryption with ChaCha20 symmetric cipher).

A unique encryption approach

The new Hive variant uses a unique approach to file encryption. Instead of embedding an encrypted key in each file that it encrypts, it generates two sets of keys in memory, uses them to encrypt files, and then encrypts and writes the sets to the root of the drive it encrypts, both with .key extension.

To indicate which keys set was used to encrypt a file, the name of the .key file containing the corresponding encryption keys is added to the name of the encrypted file on disk, followed by an underscore and then a Base64 string (also adding underscore and hyphen to the character set).

Once it’s Base64-decoded, the string contains two offsets, with each offset pointing to a different location in the corresponding *.key* file. This way, the attacker can decrypt the file using these offsets.

For example, after running Hive, we got the following files dropped to the *C:\* drive:

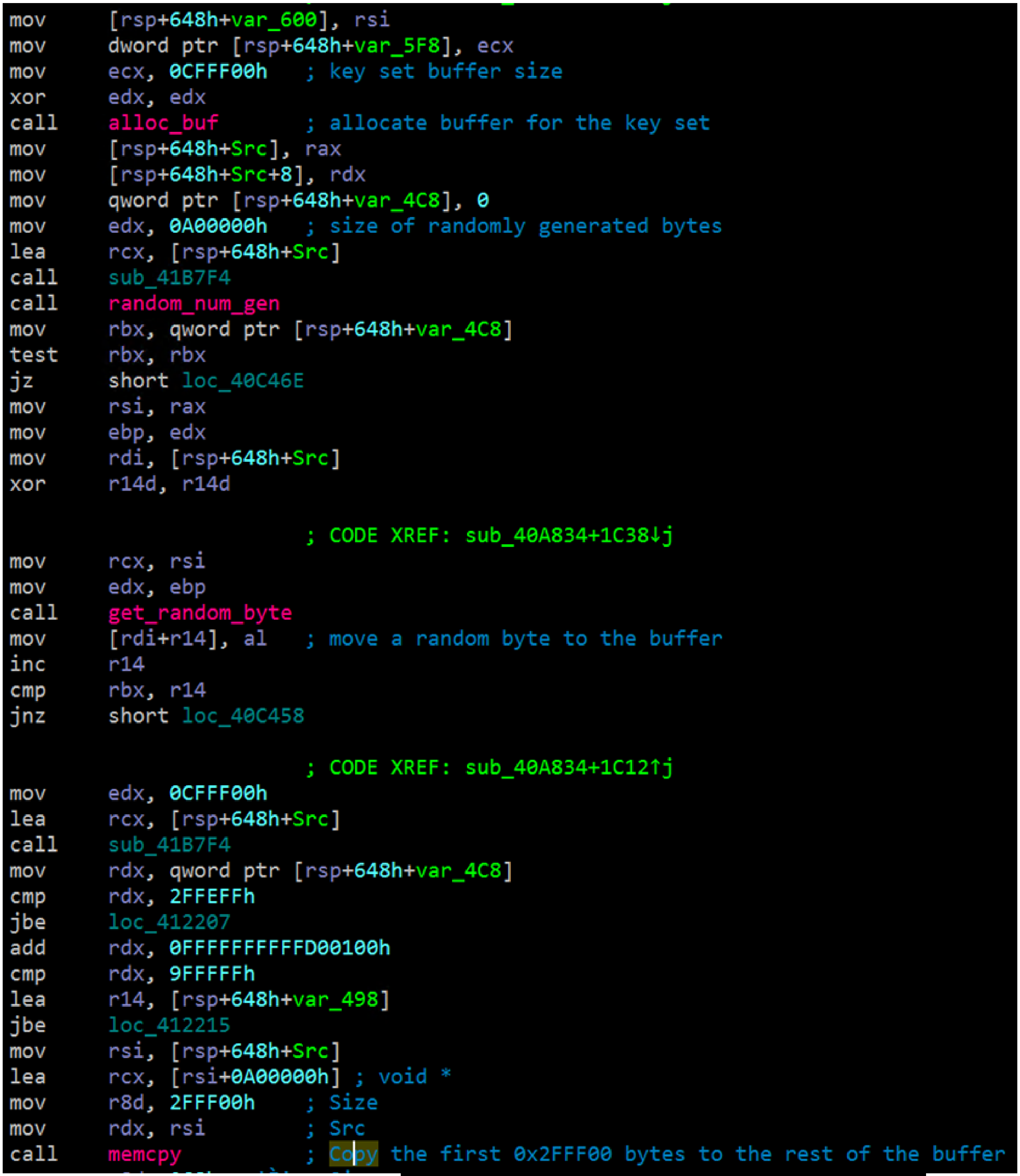
- *C:\3bcVwj6j.key*
- *C:\l0Zn68cb.key*

In this example, a file named *myphoto.jpg* would be renamed to *C:\myphoto.jpg.l0Zn68cb _ -B82BhIaGhI8*. As we discuss in the following sections, the new variant’s keys set generation is entirely different from old variants. However, its actual file encryption is very similar.

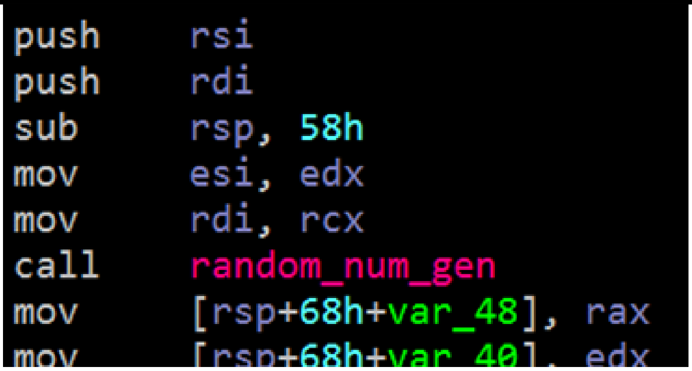
Keys set generation

A buffer of size 0xCFFF00 bytes is allocated. Using two custom functions to generate random bytes (labeled *“random_num_gen”* and *“random_num_gen_2”* for demonstration purposes) the buffer is filled. The first 0xA00000 bytes of this buffer are filled with random bytes and the remaining 0x2FFF00 bytes are simply copied from the first 0x2FFF00 random bytes that were copied earlier to the buffer.

The content of each buffer is a keys set (a collection of symmetric keys). Since two buffers are allocated, there are two keys sets. In the encryption process, the malware randomly selects different keys (byte sequences) for each file from one of the keys set and uses them to encrypt the file by XORing the byte sequence of the keys with the file’s content.

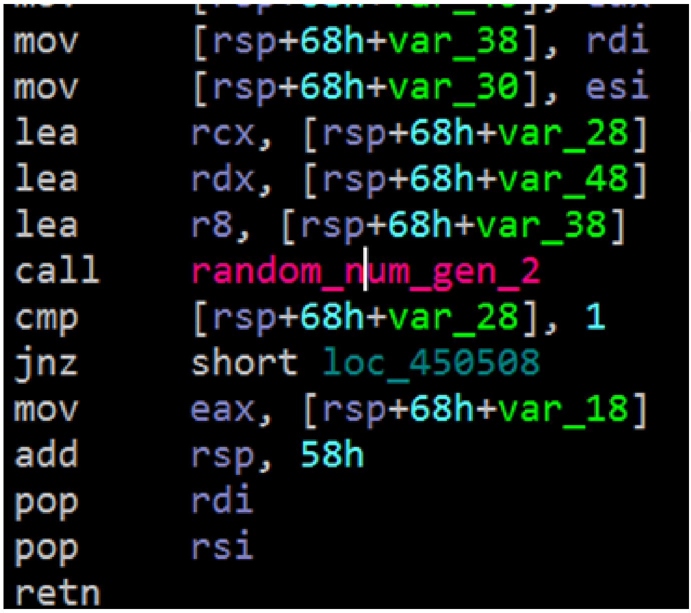


```
mov     [rsp+648h+var_600], rsi
mov     dword ptr [rsp+648h+var_5F8], ecx
mov     ecx, 0CFFF00h   ; key set buffer size
xor     edx, edx
call    alloc_buf       ; allocate buffer for the key set
mov     [rsp+648h+Src], rax
mov     [rsp+648h+Src+8], rdx
mov     qword ptr [rsp+648h+var_4C8], 0
mov     edx, 0A00000h   ; size of randomly generated bytes
lea     rcx, [rsp+648h+Src]
call    sub_41B7F4
call    random_num_gen
mov     rbx, qword ptr [rsp+648h+var_4C8]
test    rbx, rbx
jz      short loc_40C46E
mov     rsi, rax
mov     ebp, edx
mov     rdi, [rsp+648h+Src]
xor     r14d, r14d

                ; CODE XREF: sub_40A834+1C38↓j
mov     rcx, rsi
mov     edx, ebp
call    get_random_byte
mov     [rdi+r14], al   ; move a random byte to the buffer
inc     r14
cmp     rbx, r14
jnz     short loc_40C458

                ; CODE XREF: sub_40A834+1C12↑j
mov     edx, 0CFFF00h
lea     rcx, [rsp+648h+Src]
call    sub_41B7F4
mov     rdx, qword ptr [rsp+648h+var_4C8]
cmp     rdx, 2FFEFFh
jbe     loc_412207
add     rdx, 0FFFFFFFFFFD00100h
cmp     rdx, 9FFFFFh
lea     r14, [rsp+648h+var_498]
jbe     loc_412215
mov     rsi, [rsp+648h+Src]
lea     rcx, [rsi+0A00000h] ; void *
mov     r8d, 2FFF00h    ; Size
mov     rdx, rsi        ; Src
call    memcpy          ; Copy the first 0x2FFF00 bytes to the rest of the buffer
```



```
push    rsi
push    rdi
sub     rsp, 58h
mov     esi, edx
mov     rdi, rcx
call    random_num_gen
mov     [rsp+68h+var_48], rax
mov     [rsp+68h+var_40], edx
```

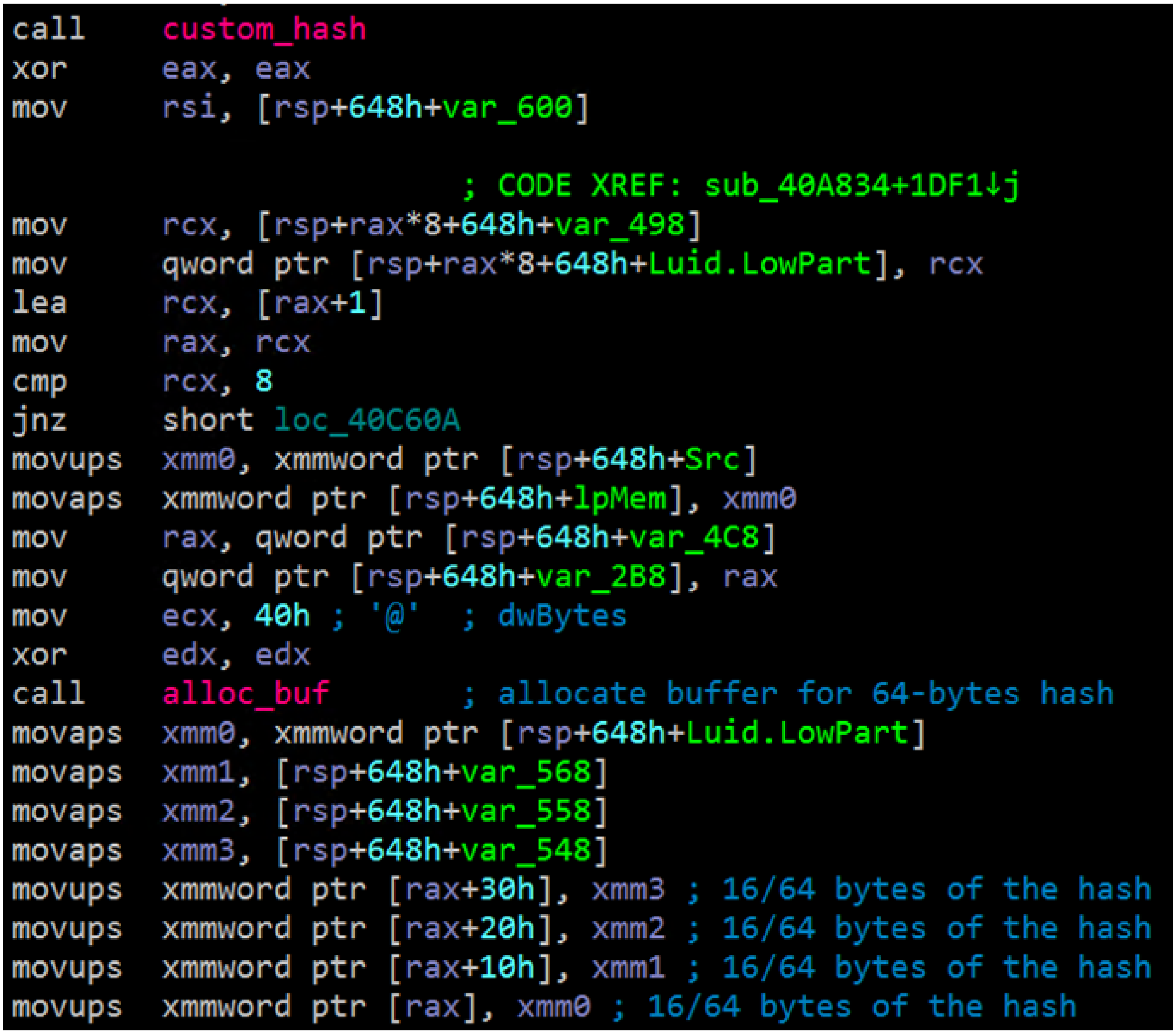
Figure 6 – Original keys set generation



```
mov     [rsp+68h+var_38], rdi
mov     [rsp+68h+var_30], esi
lea     rcx, [rsp+68h+var_28]
lea     rdx, [rsp+68h+var_48]
lea     r8, [rsp+68h+var_38]
call    random_num_gen_2
cmp     [rsp+68h+var_28], 1
jnz     short loc_450508
mov     eax, [rsp+68h+var_18]
add     rsp, 58h
pop     rdi
pop     rsi
retn
```

Figure 7 – Inside get_random_byte

A custom 64-byte hash is prepared for each keys set. This hash will be used later.

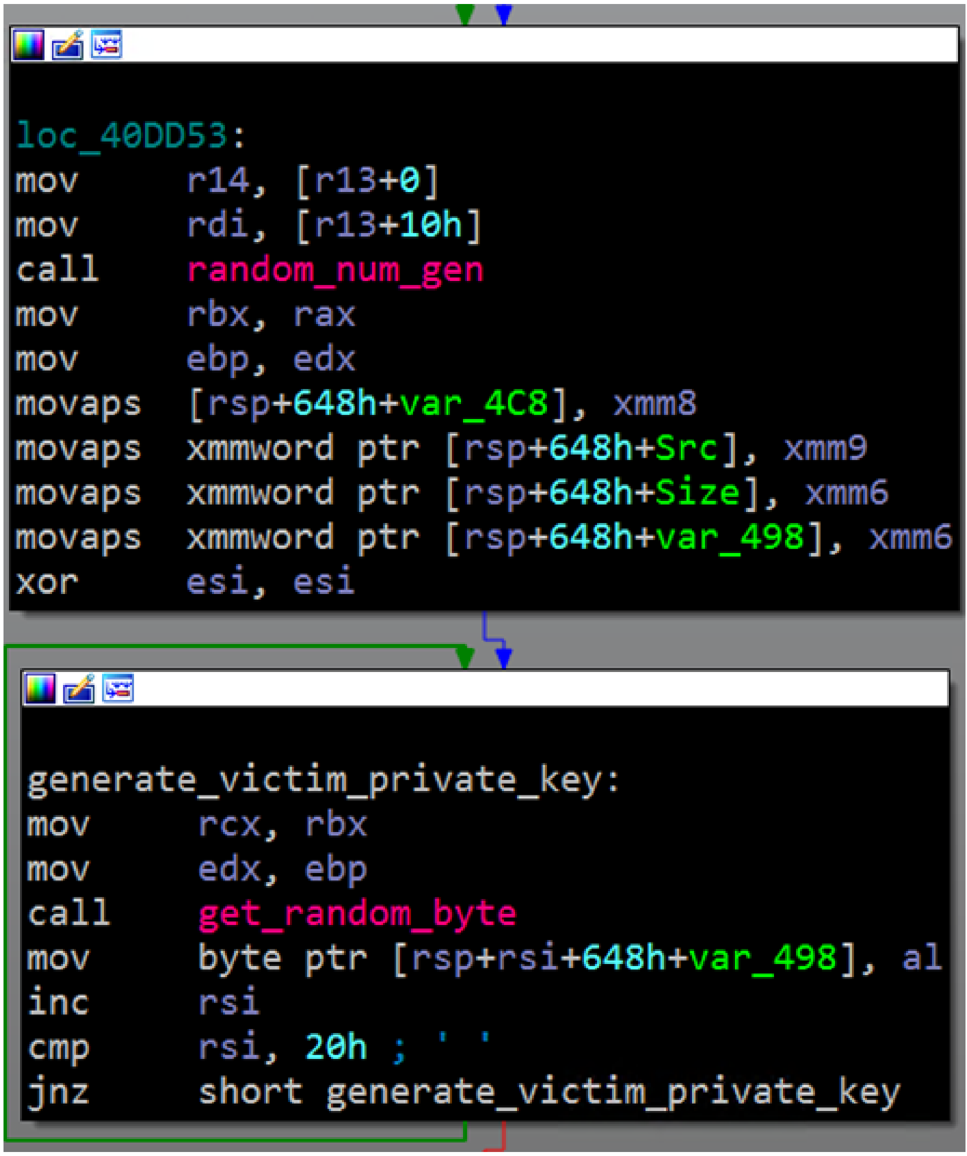


```
call    custom_hash
xor     eax, eax
mov     rsi, [rsp+648h+var_600]

                        ; CODE XREF: sub_40A834+1DF1↓j
mov     rcx, [rsp+rax*8+648h+var_498]
mov     qword ptr [rsp+rax*8+648h+Luid.LowPart], rcx
lea     rcx, [rax+1]
mov     rax, rcx
cmp     rcx, 8
jnz     short loc_40C60A
movups  xmm0, xmmword ptr [rsp+648h+Src]
movaps  xmmword ptr [rsp+648h+lpMem], xmm0
mov     rax, qword ptr [rsp+648h+var_4C8]
mov     qword ptr [rsp+648h+var_2B8], rax
mov     ecx, 40h ; '@'  ; dwBytes
xor     edx, edx
call    alloc_buf       ; allocate buffer for 64-bytes hash
movaps  xmm0, xmmword ptr [rsp+648h+Luid.LowPart]
movaps  xmm1, [rsp+648h+var_568]
movaps  xmm2, [rsp+648h+var_558]
movaps  xmm3, [rsp+648h+var_548]
movups  xmmword ptr [rax+30h], xmm3 ; 16/64 bytes of the hash
movups  xmmword ptr [rax+20h], xmm2 ; 16/64 bytes of the hash
movups  xmmword ptr [rax+10h], xmm1 ; 16/64 bytes of the hash
movups  xmmword ptr [rax], xmm0 ; 16/64 bytes of the hash
```

Figure 8 – Preparing the custom hash of the keys set

After the hash is computed and several other strings are decrypted, the encryption process takes the following steps:

1. Generate *victim_private_key* using the same functions introduced above.

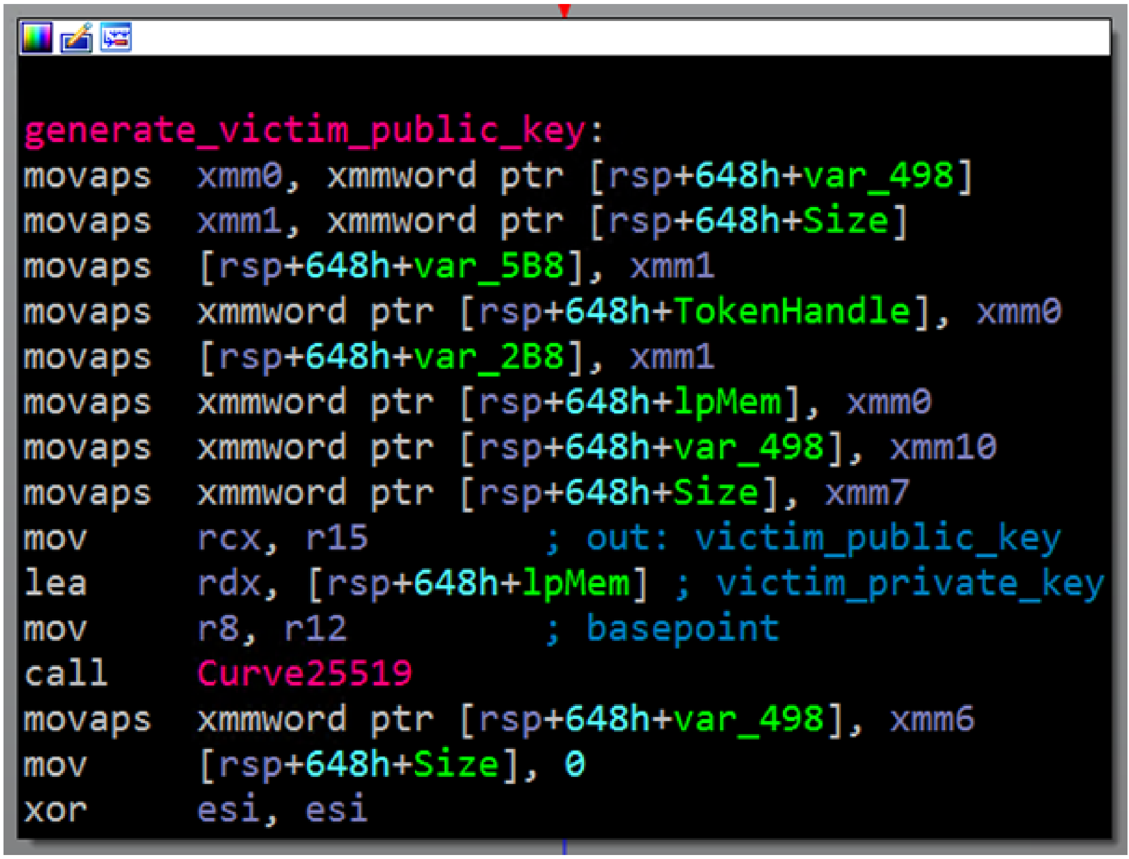


```
loc_40DD53:
mov     r14, [r13+0]
mov     rdi, [r13+10h]
call    random_num_gen
mov     rbx, rax
mov     ebp, edx
movaps  [rsp+648h+var_4C8], xmm8
movaps  xmmword ptr [rsp+648h+Src], xmm9
movaps  xmmword ptr [rsp+648h+Size], xmm6
movaps  xmmword ptr [rsp+648h+var_498], xmm6
xor     esi, esi
```

```
generate_victim_private_key:
mov     rcx, rbx
mov     edx, ebp
call    get_random_byte
mov     byte ptr [rsp+rsi+648h+var_498], al
inc     rsi
cmp     rsi, 20h ; ' '
jnz     short generate_victim_private_key
```

Figure 9 – Generating victim_private_key

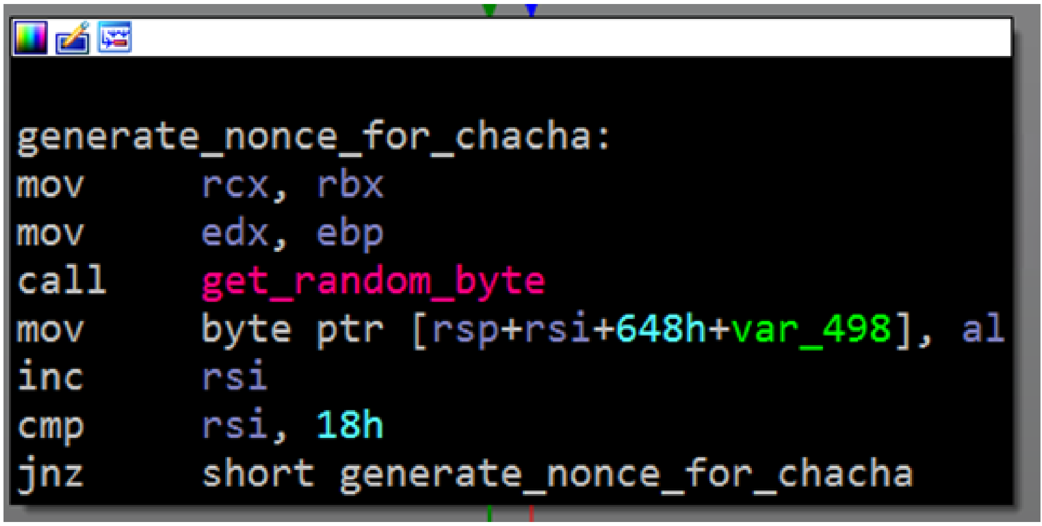
1. Generate *victim_public_key* using ECDH with Curve25519. The input is *victim_private_key* and the basepoint is 9 followed by 31 zeros (embedded in the sample).



```
generate_victim_public_key:
movaps  xmm0, xmmword ptr [rsp+648h+var_498]
movaps  xmm1, xmmword ptr [rsp+648h+Size]
movaps  [rsp+648h+var_5B8], xmm1
movaps  xmmword ptr [rsp+648h+TokenHandle], xmm0
movaps  [rsp+648h+var_2B8], xmm1
movaps  xmmword ptr [rsp+648h+lpMem], xmm0
movaps  xmmword ptr [rsp+648h+var_498], xmm10
movaps  xmmword ptr [rsp+648h+Size], xmm7
mov     rcx, r15         ; out: victim_public_key
lea     rdx, [rsp+648h+lpMem] ; victim_private_key
mov     r8, r12          ; basepoint
call    Curve25519
movaps  xmmword ptr [rsp+648h+var_498], xmm6
mov     [rsp+648h+Size], 0
xor     esi, esi
```

Figure 10 – Generating victim_public_key

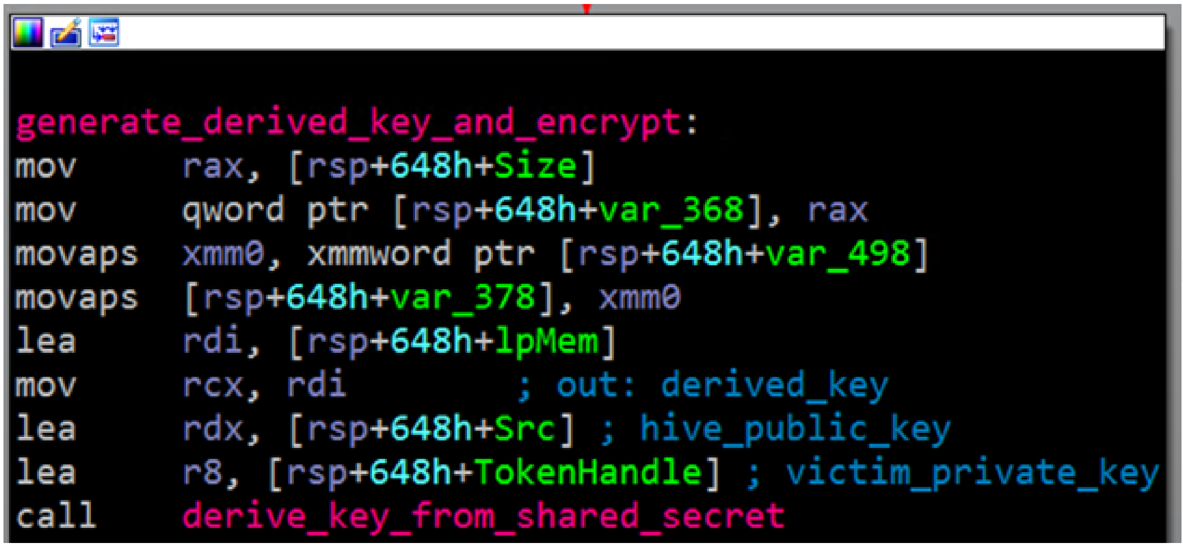
1. Generate a 24-byte nonce for the XChaCha algorithm, later in Poly1305-XChaCha20.



```
generate_nonce_for_chacha:
mov     rcx, rbx
mov     edx, ebp
call    get_random_byte
mov     byte ptr [rsp+rsi+648h+var_498], al
inc     rsi
cmp     rsi, 18h
jnz     short generate_nonce_for_chacha
```

Figure 11 – Generating a 24-byte nonce

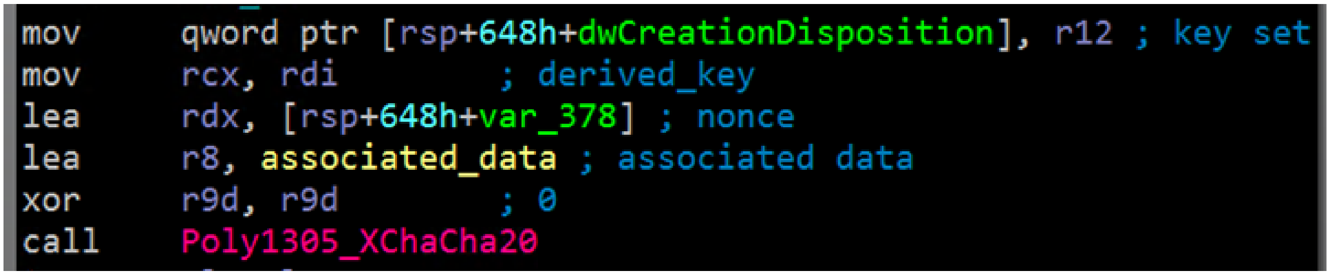
1. Generate *shared_secret* using ECDH with Curve25519. The input is *victim_private_key* and *hive_public_key*. Then, the *shared_secret* (as a key) with *hive_public_key* (as a nonce) is used to derive the *derived_key* using ChaCha20.



```
generate_derived_key_and_encrypt:
mov     rax, [rsp+648h+Size]
mov     qword ptr [rsp+648h+var_368], rax
movaps  xmm0, xmmword ptr [rsp+648h+var_498]
movaps  [rsp+648h+var_378], xmm0
lea     rdi, [rsp+648h+lpMem]
mov     rcx, rdi          ; out: derived_key
lea     rdx, [rsp+648h+Src] ; hive_public_key
lea     r8, [rsp+648h+TokenHandle] ; victim_private_key
call    derive_key_from_shared_secret
```

Figure 12 – Generating shared_secret

1. Encrypt the keys set using Poly1305-XChaCha20. The values used for the encryption are the keys set, *derived_key*, nonce, and the embedded associated data (AD). This function encrypts the keys set and adds a 16-byte authentication tag at the end of the buffer of the encrypted keys. It's unclear if the authentication tag is ever checked.

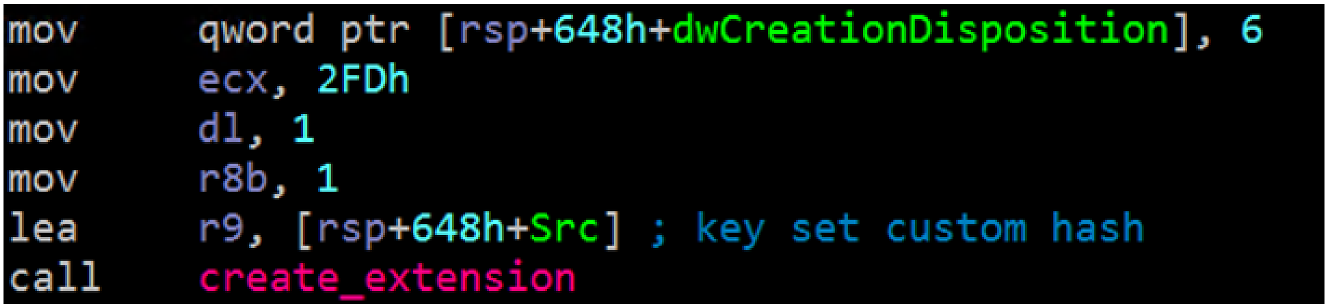


```
mov     qword ptr [rsp+648h+dwCreationDisposition], r12 ; key set
mov     rcx, rdi          ; derived_key
lea     rdx, [rsp+648h+var_378] ; nonce
lea     r8, associated_data ; associated data
xor     r9d, r9d          ; 0
call    Poly1305_XChaCha20
```

Figure 13 – Encrypting the keys set

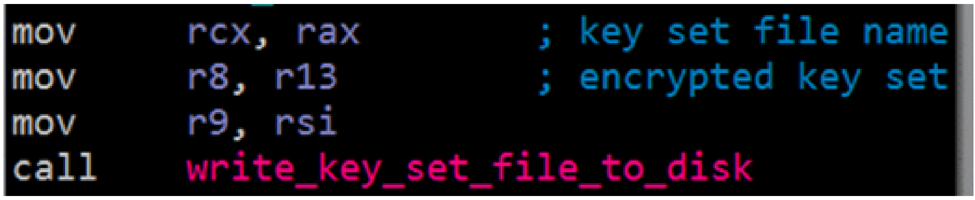
Now that the keys set is finally encrypted, the nonce, *victim_public_key*, the now-encrypted keys set, and the authentication tag are copied to a new buffer, one after another. This buffer (which we label *encrypted_structure_1*) is treated as a new keys set, which is again encrypted using the same method described above but with a second *hive_public_key*. This time, the function outputs new nonce, *victim_private_key*, and others. Only the associated data is the same.

Finally, the new buffer, which contains the *second_nonce*, *second_victim_public_key*, and the encrypted*encrypted_structure_1*, is written to the root of the drive it's encrypting (for example, *C:\*). The *create_extension* function generates a Base64 string based on the first six bytes of the custom hash that was created earlier. This Base64 string serves as the file name, and the extension of the file is simply *".key"*.



```
mov     qword ptr [rsp+648h+dwCreationDisposition], 6
mov     ecx, 2FDh
mov     dl, 1
mov     r8b, 1
lea     r9, [rsp+648h+Src] ; key set custom hash
call    create_extension
```

Figure 14 – Generating a Base64 string based on the first six bytes of the custom hash



```
mov     rcx, rax        ; key set file name
mov     r8, r13         ; encrypted key set
mov     r9, rsi
call    write_key_set_file_to_disk
```

Figure 15 – Using the Base64 string as the file name

The diagram below illustrates the encryption scheme described above:

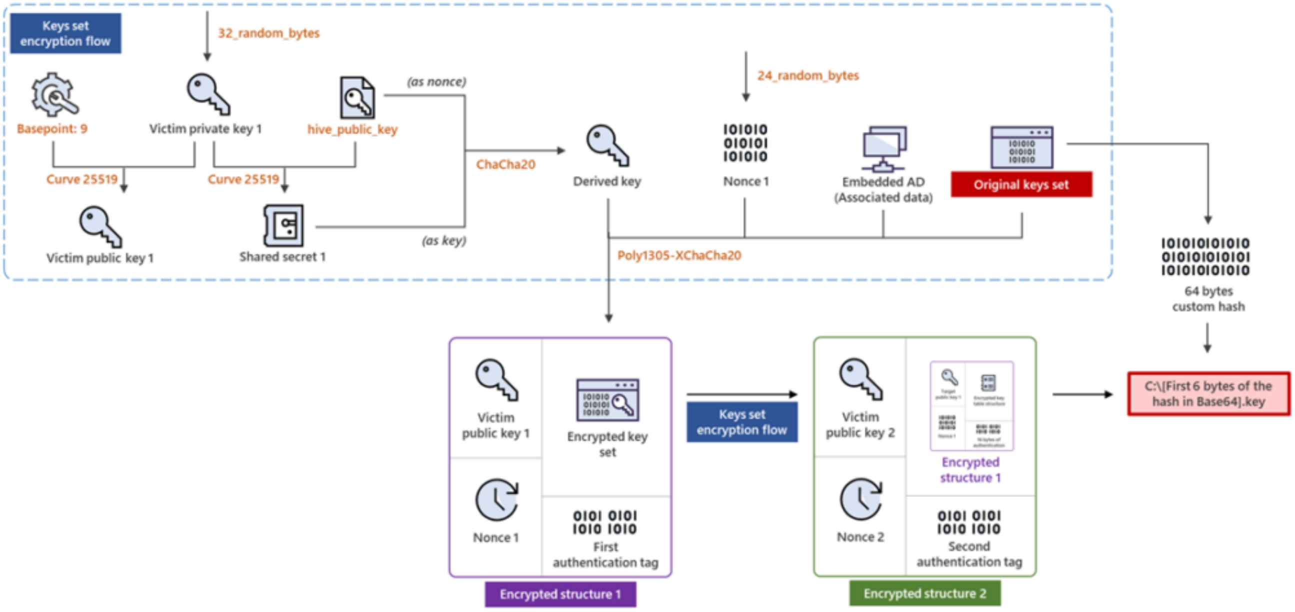


Figure 16 – The keys set encryption scheme of the new Hive variant

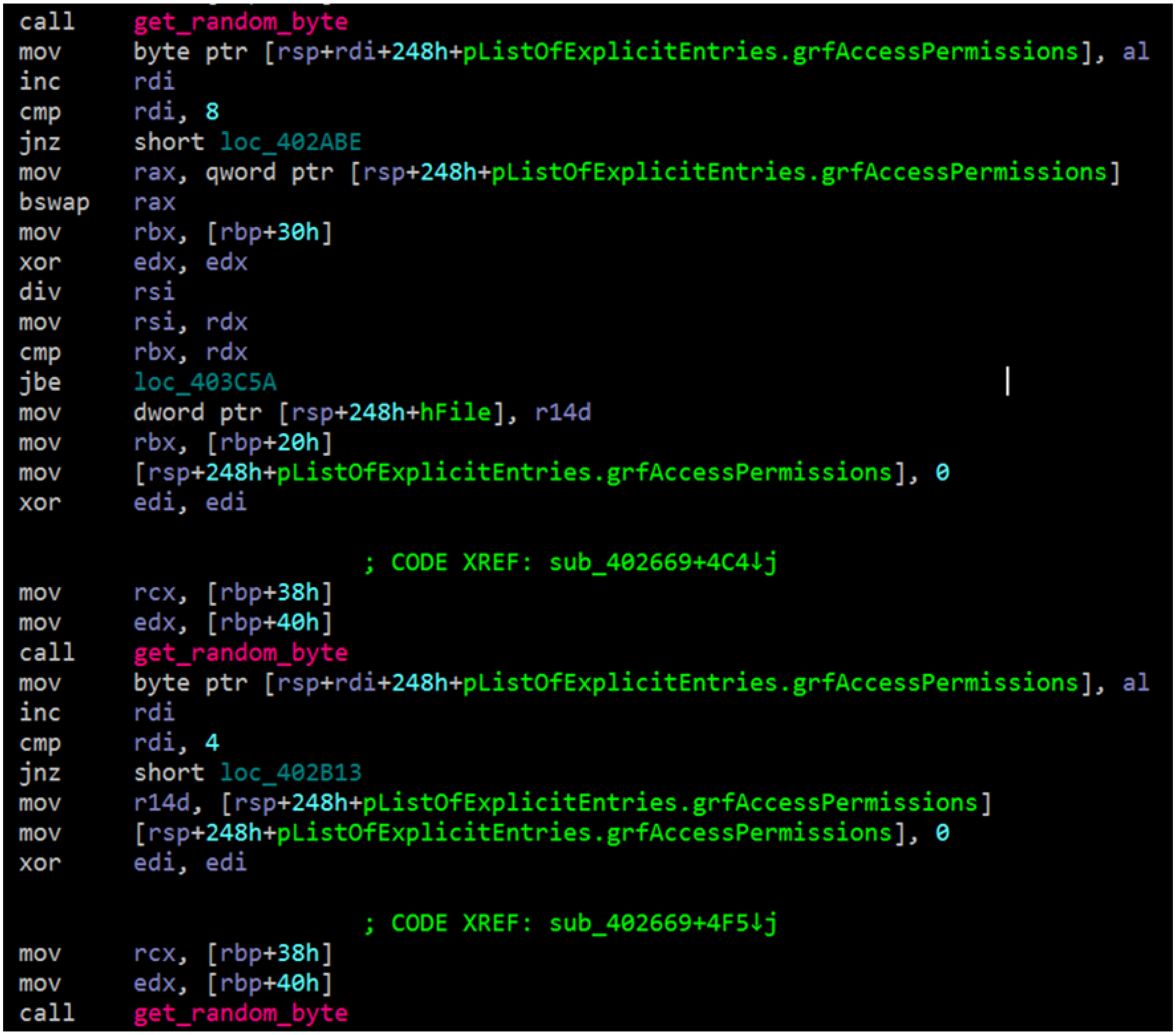
As seen in the diagram above, “Keys sets encryption flow” is executed twice. In the first round it is executed with the original keys set as an input. In the second round it is executed with the “encrypted structure 1” as an input. In its second execution, all other input values are different except the AD (associated data) and the Basepoint 9.

Hence, the following values are new in the second execution: *victim_private_key*, *victim_public_key*, *hive_public_key*, *nonce*, *shared_secret* and *derived_key*.

File encryption

After both keys files are written to the disk, the multi-threaded file encryption starts. Before encrypting each file, the malware checks its name and extension against a list of strings. If there is a match, then the file will not be encrypted. For example, a file with .exe extension will not be encrypted if .exe is in the list of strings. It should be noted that this list is encrypted and decrypted during runtime.

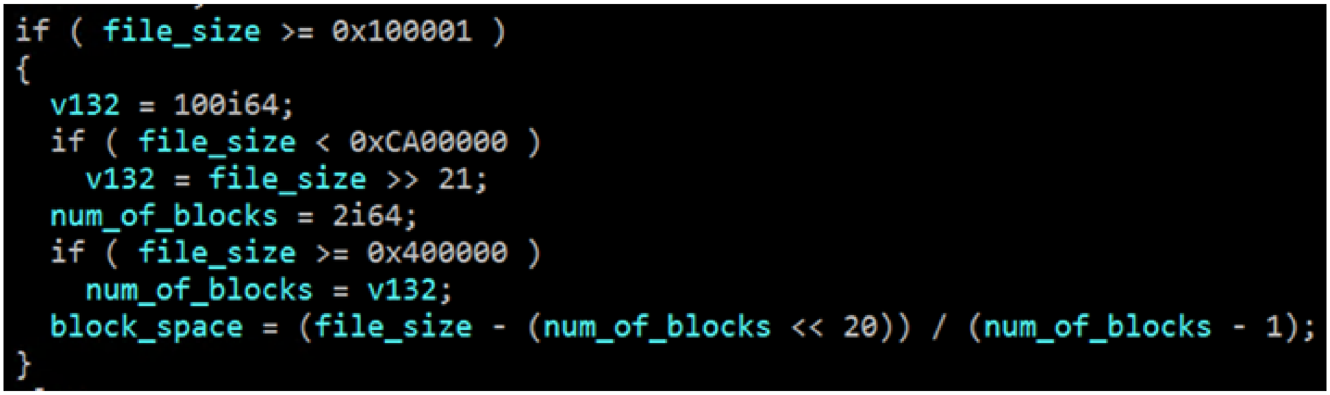
The same file encryption method seen in old variants is used in the new one: two random numbers are generated and used as offsets to the keys set. Each offset is four bytes:



```
call    get_random_byte
mov     byte ptr [rsp+rdi+248h+pListOfExplicitEntries.grfAccessPermissions], al
inc     rdi
cmp     rdi, 8
jnz     short loc_402ABE
mov     rax, qword ptr [rsp+248h+pListOfExplicitEntries.grfAccessPermissions]
bswap   rax
mov     rbx, [rbp+30h]
xor     edx, edx
div     rsi
mov     rsi, rdx
cmp     rbx, rdx
jbe     loc_403C5A
mov     dword ptr [rsp+248h+hFile], r14d
mov     rbx, [rbp+20h]
mov     [rsp+248h+pListOfExplicitEntries.grfAccessPermissions], 0
xor     edi, edi

                        ; CODE XREF: sub_402669+4C4↓j
mov     rcx, [rbp+38h]
mov     edx, [rbp+40h]
call    get_random_byte
mov     byte ptr [rsp+rdi+248h+pListOfExplicitEntries.grfAccessPermissions], al
inc     rdi
cmp     rdi, 4
jnz     short loc_402B13
mov     r14d, [rsp+248h+pListOfExplicitEntries.grfAccessPermissions]
mov     [rsp+248h+pListOfExplicitEntries.grfAccessPermissions], 0
xor     edi, edi

                        ; CODE XREF: sub_402669+4F5↓j
mov     rcx, [rbp+38h]
mov     edx, [rbp+40h]
call    get_random_byte
```

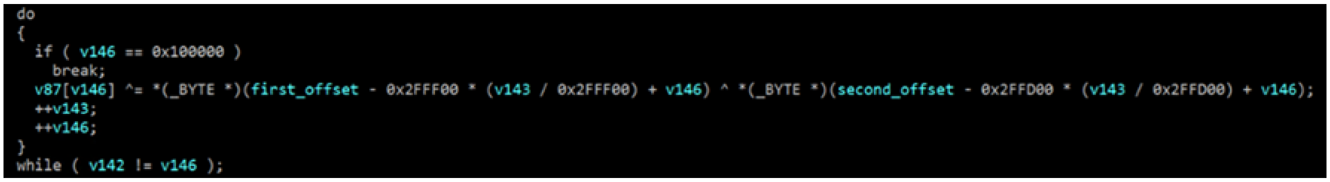
Figure 17 – Generating the offsets

For the encryption, the file’s content is XORed with bytes from the keys set, according to the offsets. The file bytes are XORed twice—once according to the first offset and a second time according to the second offset. Files are encrypted in blocks of 0x100000 bytes, with the maximum number of blocks at 100. There is an interval between the encrypted blocks as defined by *block_space*. After the encryption is finished in memory, the encrypted data is written to the disk, overwriting the original file.



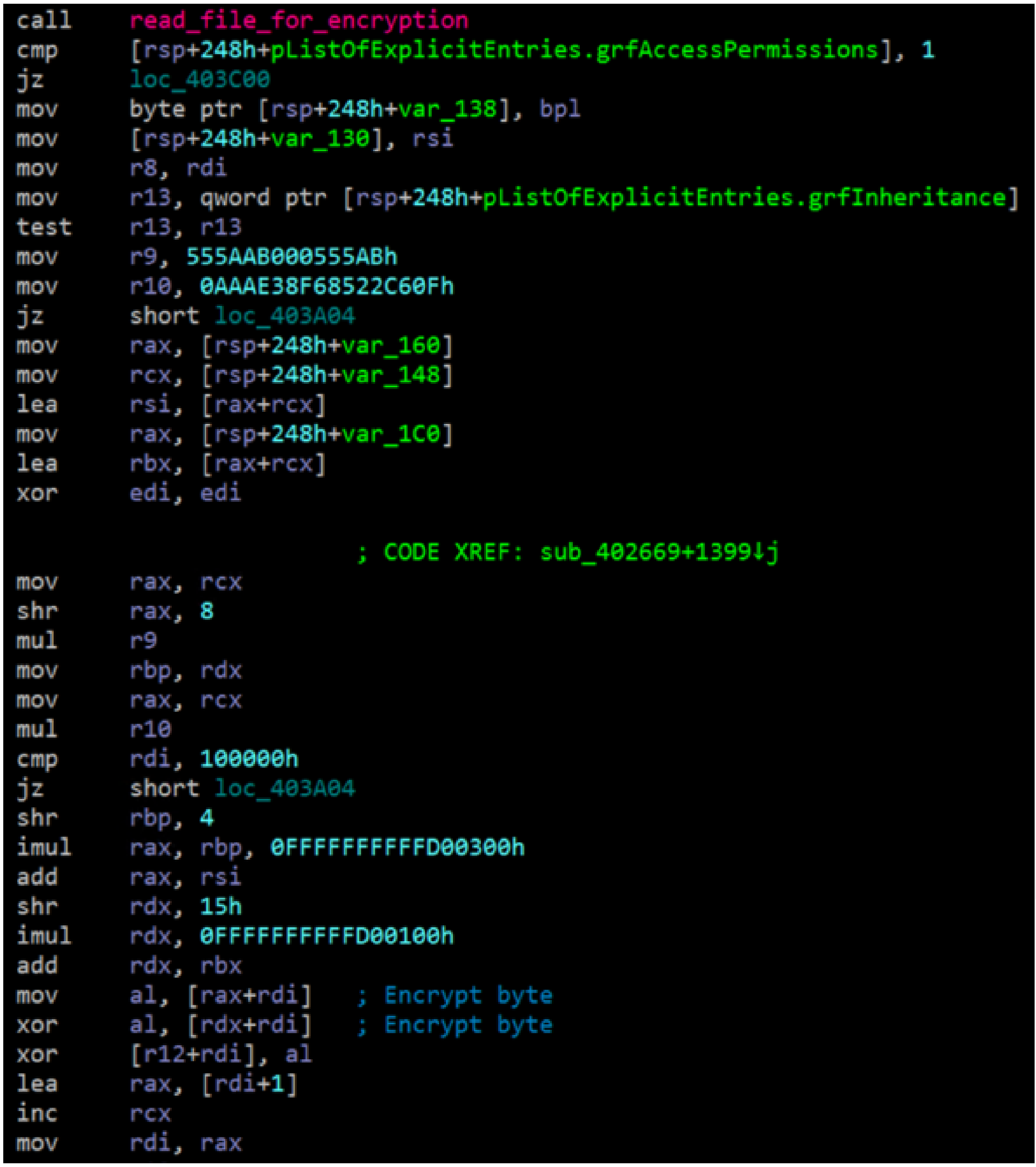
```
if ( file_size >= 0x100001 )
{
  v132 = 100i64;
  if ( file_size < 0xCA00000 )
    v132 = file_size >> 21;
  num_of_blocks = 2i64;
  if ( file_size >= 0x400000 )
    num_of_blocks = v132;
  block_space = (file_size - (num_of_blocks << 20)) / (num_of_blocks - 1);
}
```

Figure 18 – Calculation of number of blocks



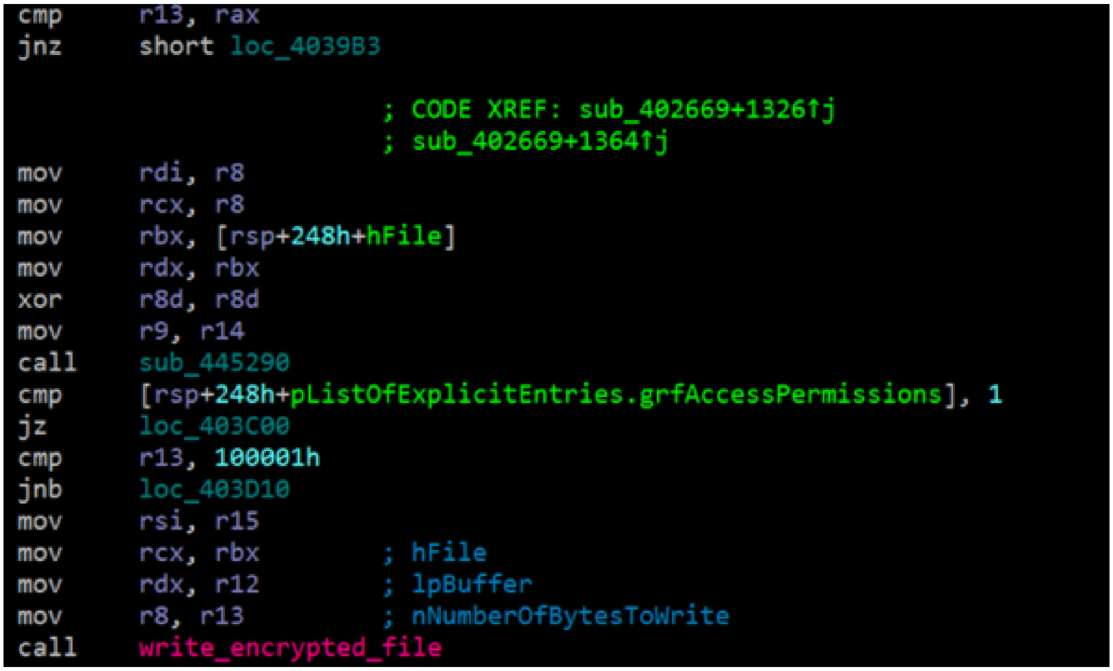
```
do
{
  if ( v146 == 0x100000 )
    break;
  v87[v146] ^= *(_BYTE *)(first_offset - 0x2FFF00 * (v143 / 0x2FFF00) + v146) ^ *(_BYTE *)(second_offset - 0x2FFD00 * (v143 / 0x2FFD00) + v146);
  ++v143;
  ++v146;
}
while ( v142 != v146 );
```

Figure 19 – Actual encryption of the file bytes



```
call    read_file_for_encryption
cmp     [rsp+248h+pListOfExplicitEntries.grfAccessPermissions], 1
jz      loc_403C00
mov     byte ptr [rsp+248h+var_138], bpl
mov     [rsp+248h+var_130], rsi
mov     r8, rdi
mov     r13, qword ptr [rsp+248h+pListOfExplicitEntries.grfInheritance]
test    r13, r13
mov     r9, 555AAB000555ABh
mov     r10, 0AAAE38F68522C60Fh
jz      short loc_403A04
mov     rax, [rsp+248h+var_160]
mov     rcx, [rsp+248h+var_148]
lea     rsi, [rax+rcx]
mov     rax, [rsp+248h+var_1C0]
lea     rbx, [rax+rcx]
xor     edi, edi

                        ; CODE XREF: sub_402669+1399↓j
mov     rax, rcx
shr     rax, 8
mul     r9
mov     rbp, rdx
mov     rax, rcx
mul     r10
cmp     rdi, 100000h
jz      short loc_403A04
shr     rbp, 4
imul    rax, rbp, 0FFFFFFFFFFD00300h
add     rax, rsi
shr     rdx, 15h
imul    rdx, 0FFFFFFFFFFD00100h
add     rdx, rbx
mov     al, [rax+rdi]   ; Encrypt byte
xor     al, [rdx+rdi]   ; Encrypt byte
xor     [r12+rdi], al
lea     rax, [rdi+1]
inc     rcx
mov     rdi, rax
```

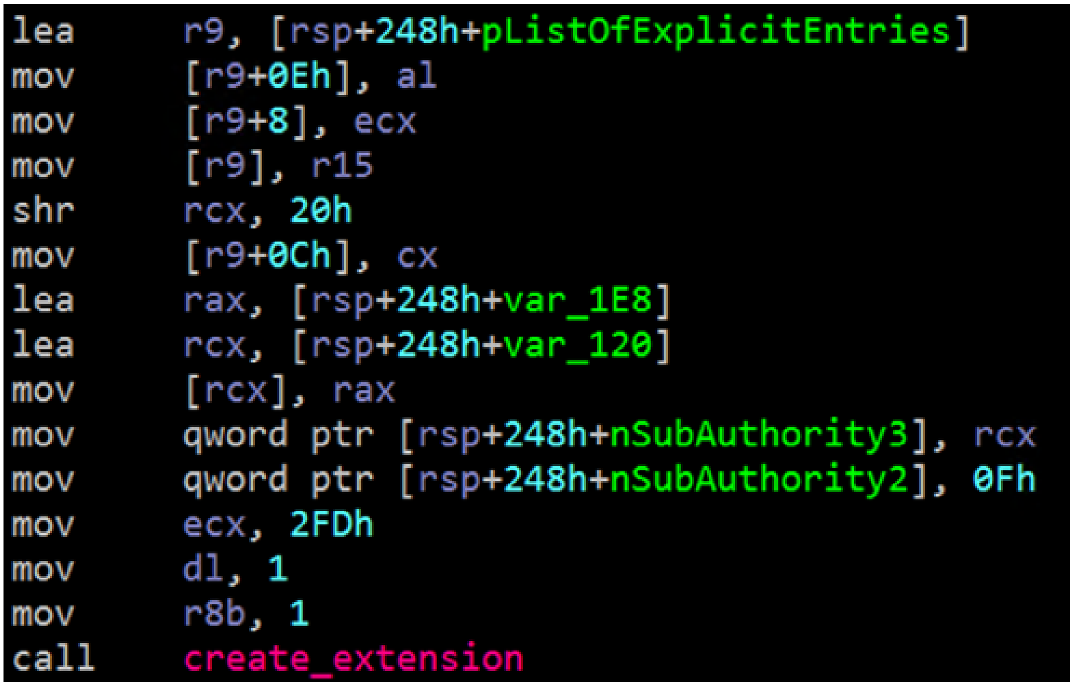
Figure 20 –



```
cmp     r13, rax
jnz     short loc_4039B3

                        ; CODE XREF: sub_402669+1326↑j
                        ; sub_402669+1364↑j
mov     rdi, r8
mov     rcx, r8
mov     rbx, [rsp+248h+hFile]
mov     rdx, rbx
xor     r8d, r8d
mov     r9, r14
call    sub_445290
cmp     [rsp+248h+pListOfExplicitEntries.grfAccessPermissions], 1
jz      loc_403C00
cmp     r13, 100001h
jnb     loc_403D10
mov     rsi, r15
mov     rcx, rbx        ; hFile
mov     rdx, r12        ; lpBuffer
mov     r8, r13         ; nNumberOfBytesToWrite
call    write_encrypted_file
```

Reading a file, encrypting it, and writing it back to the disk

Looking at when *create_extension* is called once file encryption has started, we recognized a similar structure in the previous variant:

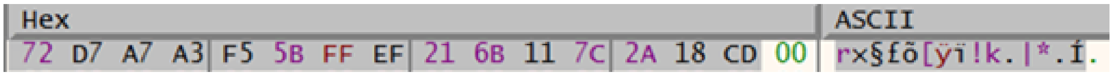


```
lea     r9, [rsp+248h+pListOfExplicitEntries]
mov     [r9+0Eh], al
mov     [r9+8], ecx
mov     [r9], r15
shr     rcx, 20h
mov     [r9+0Ch], cx
lea     rax, [rsp+248h+var_1E8]
lea     rcx, [rsp+248h+var_120]
mov     [rcx], rax
mov     qword ptr [rsp+248h+nSubAuthority3], rcx
mov     qword ptr [rsp+248h+nSubAuthority2], 0Fh
mov     ecx, 2FDh
mov     dl, 1
mov     r8b, 1
call    create_extension
```

Figure 21 –

Creating the extension for the file

Let us look at the value (72 D7 A7 A3 F5 5B FF EF 21 6B 11 7C 2A 18 CD 00) in the address of r9 register just before *create_extension* is called on a file called *EDBtmp.log*

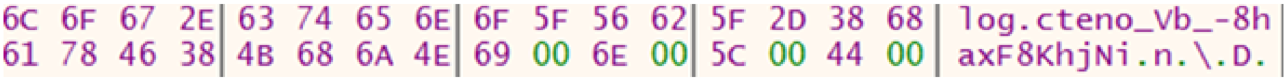


```
Hex                                              | ASCII
72 D7 A7 A3|F5 5B FF EF|21 6B 11 7C|2A 18 CD 00| rx§£õ[ÿï!k.|*.Í.
```

Recall that in the older variants, 0xFF was used as a delimiter to separate the key file name from the offset values. We can also see it here. Converting the first six bytes (72 D7 A7 A3 F5 5B) to Base64 yields the following:

```
cteno/Vb
```

And if we step over *create_extension*, the result is similar—we get *cteno_Vb* as the *.key* file name (note: Since Hive uses a different Base64 character set, “/” was replaced with “_”):



```
6C 6F 67 2E| 63 74 65 6E| 6F 5F 56 62| 5F 2D 38 68| log.cteno_Vb_-8h|
61 78 46 38| 4B 68 6A 4E| 69 00 6E 00| 5C 00 44 00| axF8KhjNi.n.\.D.|
```

Microsoft will continue to monitor the Hive operators’ activity and implement protections for our customers. The current detections, advanced detections, and indicators of compromise (IOCs) in place across our security products are detailed below.

Recommended customer actions

The techniques used by the new Hive variant can be mitigated by adopting the security considerations provided below:

> Use the included IOCs to investigate whether they exist in your environment and assess for potential intrusion.

Our recent blog on the ransomware-as-a-service economy has an exhaustive guide on how to protect yourself from ransomware threats that dive deep into each of the following areas. We encourage readers to refer to that blog for a comprehensive guide on:

- Building credential hygiene
- Auditing credential exposure
- Prioritizing deployment of Active Directory updates
- Cloud hardening
  - Implement the Azure Security Benchmark and general best practices for securing identity infrastructure.
  - Ensure cloud admins/tenant admins are treated with the same level of security and credential hygiene as Domain Admins.
  - Address gaps in authentication coverage.
- Enforce MFA on all accounts, remove users excluded from MFA, and strictly require MFA from all devices, in all locations, at all times.
- Enable passwordless authentication methods (for example, Windows Hello, FIDO keys, or Microsoft Authenticator) for accounts that support passwordless. For accounts that still require passwords, use authenticator apps like Microsoft Authenticator for MFA.
- Disable legacy authentication.

For Microsoft 365 Defender customers, the following checklist eliminates security blind spots:

- Turn on cloud-delivered protection in Microsoft Defender Antivirus to cover rapidly evolving attacker tools and techniques, block new and unknown malware variants, and enhance attack surface reduction rules and tamper protection.
- Turn on tamper protection features to prevent attackers from stopping security services.

- Run EDR in block mode so that Microsoft Defender for Endpoint can block malicious artifacts, even when a non-Microsoft antivirus doesn't detect the threat or when Microsoft Defender Antivirus is running in passive mode. EDR in block mode also blocks indicators identified proactively by Microsoft Threat Intelligence teams.
- Enable network protection to prevent applications or users from accessing malicious domains and other malicious content on the internet.
- Enable investigation and remediation in full automated mode to allow Microsoft Defender for Endpoint to take immediate action on alerts to resolve breaches.
- Use device discovery to increase visibility into the network by finding unmanaged devices and onboarding them to Microsoft Defender for Endpoint.
- Protect user identities and credentials using Microsoft Defender for Identity, a cloud-based security solution that leverages on-premises Active Directory signals to monitor and analyze user behavior to identify suspicious user activities, configuration issues, and active attacks.

Indicators of compromise (IOCs)

The below list provides a partial list of the IOCs observed during our investigation and included in this blog. We encourage our customers to investigate these indicators in their environments and implement detections and protections to identify past related activity and prevent future attacks against their systems.

Indicator	Type	Description
f4a39820dbff47fa1b68f83f575bc98ed33858b02341c5c0464a49be4e6c76d3	SHA-256	Hive Rust variant payload
88b1d8a85bf9101bc336b01b9af4345ed91d3ec761554d167fe59f73af73f037	SHA-256	Hive Rust variant payload
065208b037a2691eb75a14f97bdbd9914122655d42f6249d2cca419a1e4ba6f1	SHA-256	Hive Rust variant payload
33744c420884adf582c46a4b74cbd9c145f2e15a036bb1e557e89d6fd428e724	SHA-256	Hive Rust variant payload
afab34235b7f170150f180c7afb9e3b4e504a84559bbd03ab71e64e3b6541149	SHA-256	Hive Rust variant payload
36759cab7043cd7561ac6c3968832b30c9a442eff4d536e901d4ff70aef4d32d	SHA-256	Hive Rust variant payload
481dc99903aa270d286f559b17194b1a25deca8a64a5ec4f13a066637900221e	SHA-256	Hive Rust variant payload

6e5d49f604730ef4c05cfe3f64a7790242e71b4ecf1dc5109d32e811acf0b053	SHA-256	Hive Rust variant payload
32ff0e5d87ec16544b6ff936d6fd58023925c3bdabaf962c492f6b078cb01914	SHA-256	Hive Rust variant payload

NOTE: These indicators shouldn’t be considered exhaustive for this observed activity.

Detections

Microsoft 365 Defender

Microsoft Defender Antivirus

Microsoft Defender Antivirus provides detection for this threat under the following family names with build version 1.367.405.0 or later.

- Ransom:Win64/Hive
- Ransom:Win32/Hive

Microsoft Defender for Endpoint detection

Microsoft Defender for Endpoint customers may see any or a combination of the following alerts as an indication of possible attack. These alerts are not necessarily an indication of a Hive compromise, but should be investigated:

- Ransomware behavior detected in the file system
- File backups were deleted
- Possible ransomware infection modifying multiple files
- Possible ransomware activity
- Ransomware-linked emerging threat activity group detected

Advanced hunting queries

Microsoft Sentinel

To locate possible Hive ransomware activity mentioned in this blog post, Microsoft Sentinel customers can use the queries detailed below:

Identify Hive ransomware IOCs

This query identifies a match across various data feeds for IOCs related to Hive ransomware.

https://github.com/Azure/Azure-Sentinel/blob/master/Detections/MultipleDataSources/HiveRansomwareJuly2022.yaml

Identify backup deletion

This hunting query helps detect a ransomware’s attempt to delete backup files.

https://github.com/Azure/Azure-Sentinel/blob/master/Hunting%20Queries/MultipleDataSources/BackupDeletion.yaml

Identify Microsoft Defender Antivirus detection of Hive ransomware

This query looks for Microsoft Defender Antivirus detections related to the Hive ransomware and joins the alert with other data sources to surface additional information such as device, IP, signed-in users, etc.

https://github.com/Azure/Azure-Sentinel/blob/master/Detections/SecurityAlert/HiveRansomwareAVHits.yaml