HelloKitty Linux version malware analysis

Jul 17, 2021

Please read the disclaimer

Introduction

This report contains technical details of the new linux version of **HelloKitty** that targets VMware ESXi servers.

The encryption used by this variant is **AES_CBC** and **Elliptic-curve Diffie–Hellman (ECDH)** to protect the keys.

Encryption Overview

The malware generates an **ECDH** keypair, then using the hardcoded public key of the threat actor, it generates an **ECDH secret** key, then an AES **KEY/IV** are randomly generated at run time, this key will be used to encrypt a file.

Note: the AES **KEY/IV** are different for each file.

The AES **KEY/IV** is encrypted using a randomly generated **IV** and the previous **ECDH secret** key with AES algorithm.

Finally a structure is populated with the ECDH public key of the malware, the encrypted AES **KEY/IV** used for file encryption and other stuff.

The structure is appended with the ransom note and the encrypted file `.crypt`

The Threat actor can recover the **ECDH secret** to decrypt the encrypted **AES KEY/IV** used for encrypting the file with his Private ECDH key and the malware Public ECDH key.

Ransom Note

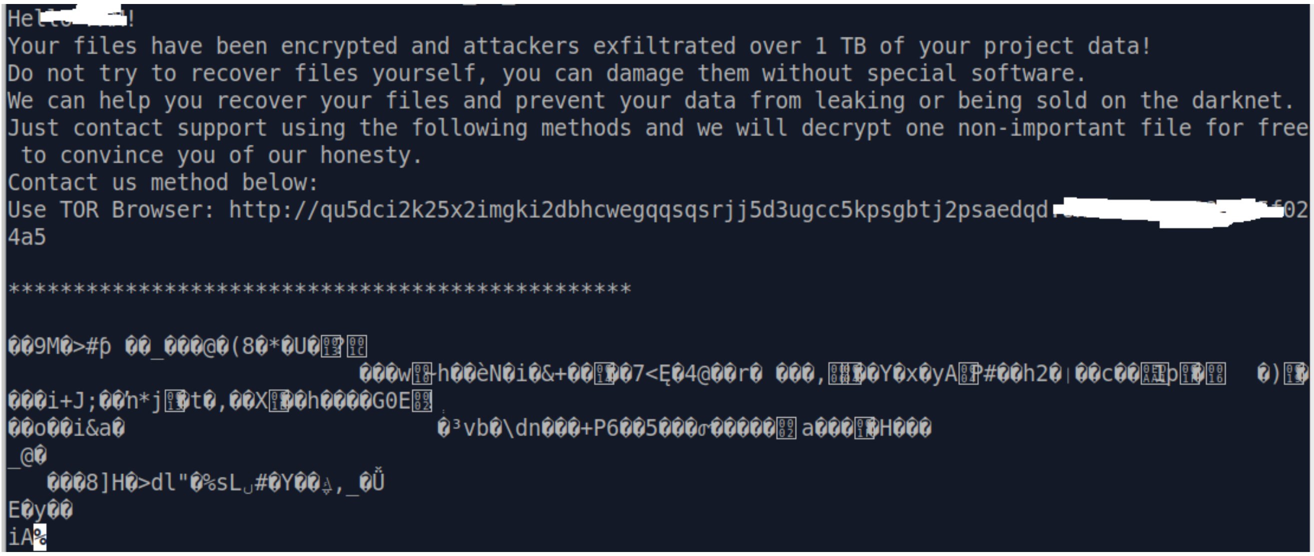
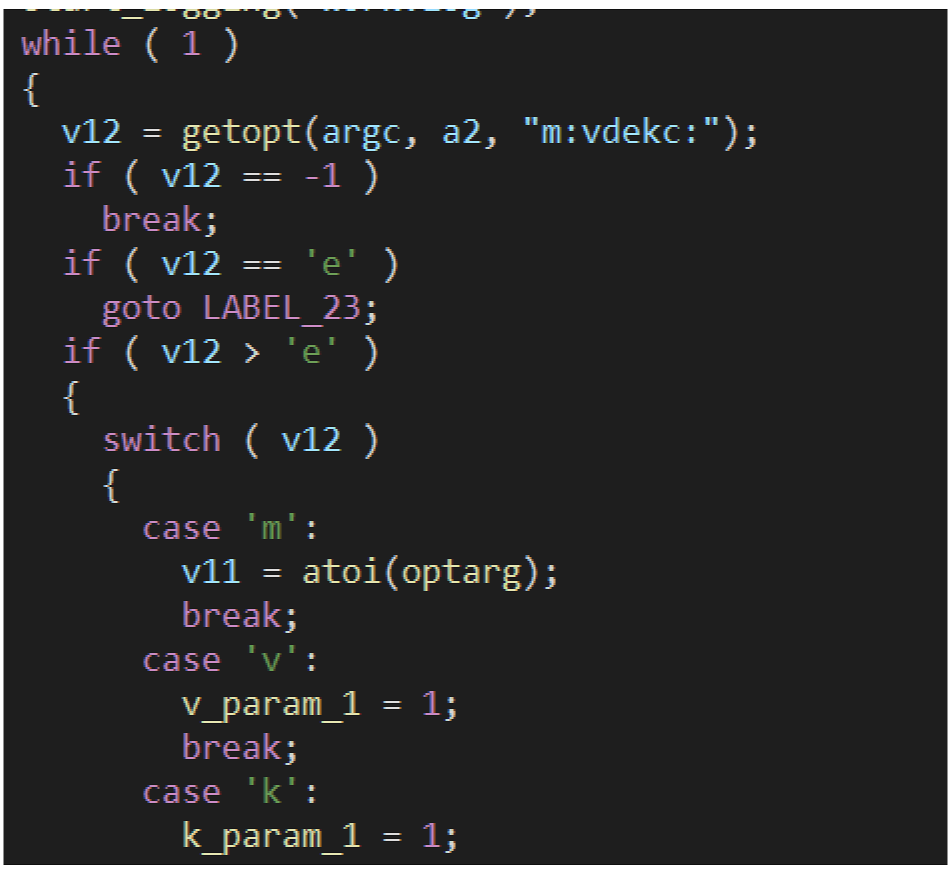


Figure: Ransom note

Technical Analysis

CommandLine arguments



```
while ( 1 )
{
  v12 = getopt(argc, a2, "m:vdekc:");
  if ( v12 == -1 )
    break;
  if ( v12 == 'e' )
    goto LABEL_23;
  if ( v12 > 'e' )
  {
    switch ( v12 )
    {
      case 'm':
        v11 = atoi(optarg);
        break;
      case 'v':
        v_param_1 = 1;
        break;
      case 'k':
        k_param_1 = 1;
```

Figure: Parsing commandLine arguments

option	Functionality

option	Functionality
-k	Kill VM processes
-d	Run as daemon
-e	Encrypt VM files
-v	Enable verbose

Dynamically loading libcrypto API

The malware loads some OpenSSL API from **libcrypto.so** using **dlopen/dlsym**.

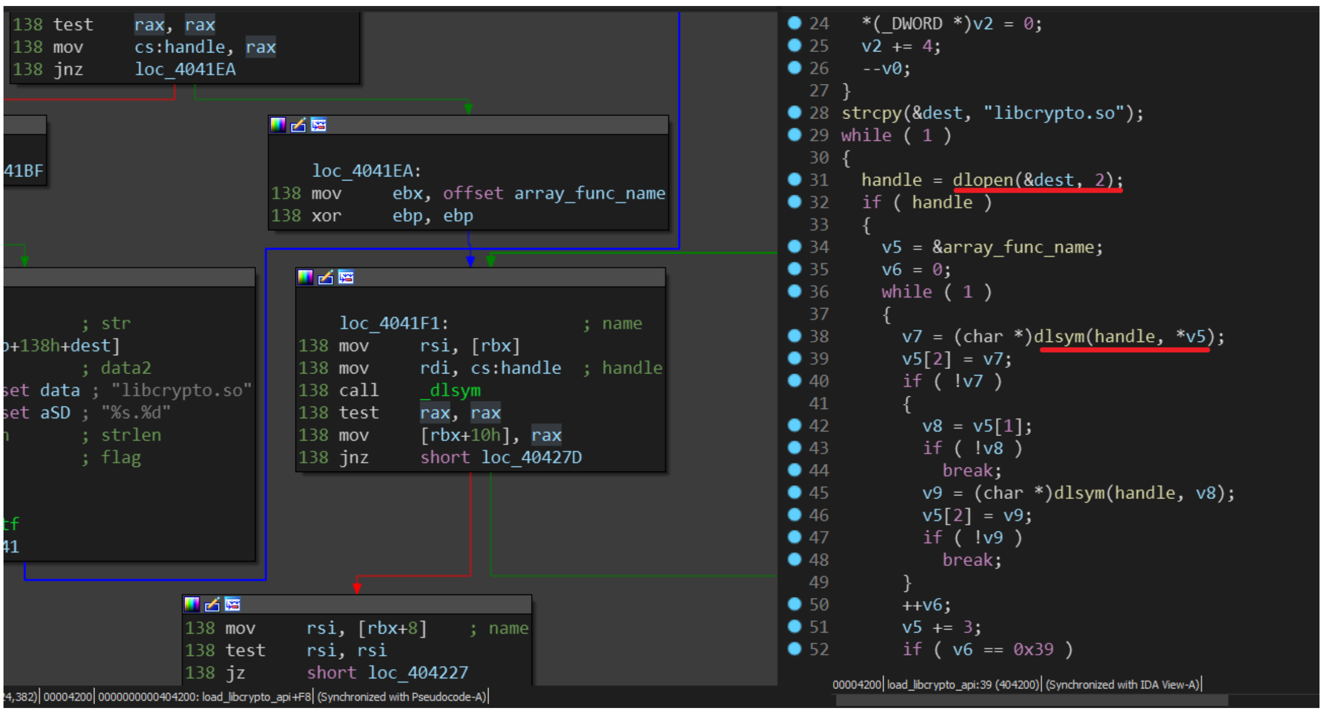


Figure: Dynamically loading libcrypto API

Looking at the array, we can see that each entry is a structure of 3 pointers:

```
        {
                unsigned char*
new_function_name;
                unsigned char*
old_function_name;
                void* pointer_to_api;
        }
```

First the ransomware tries to get the address of the function name stored in **new_function_name**, if not found (which means the library is old) it uses **old_function_name**, if the API was found , it's pointer will be stored in **pointer_to_api**

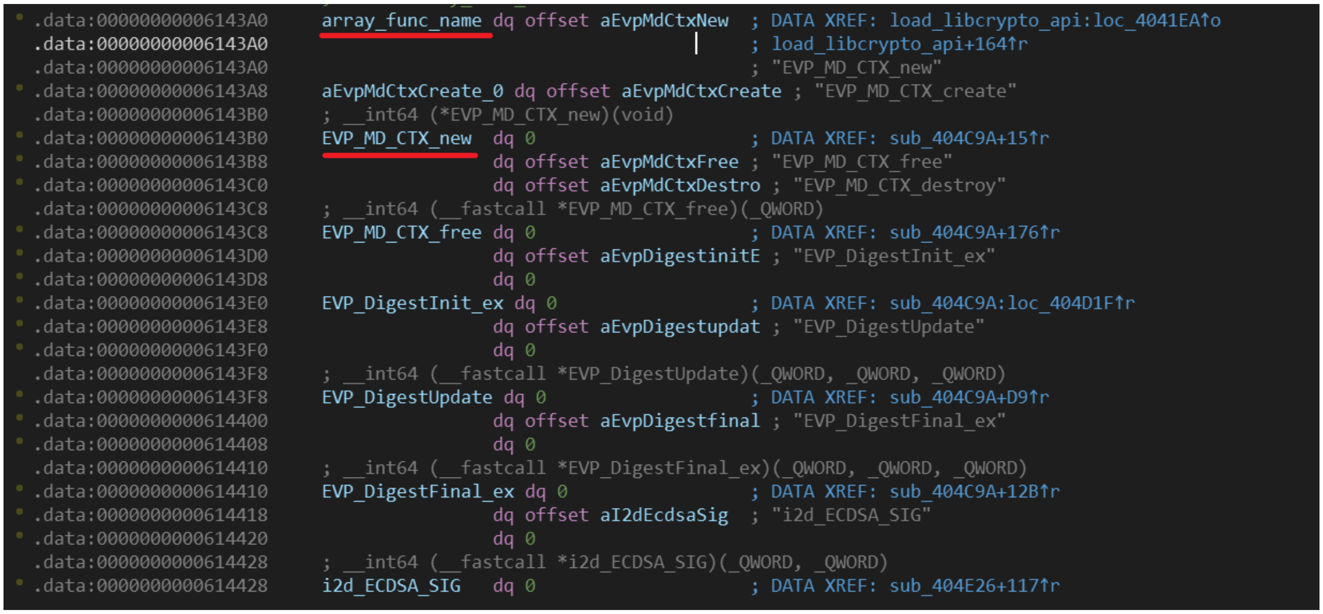


Figure: Array of libcrypto API names and addresses

HISTORY

The EVP_MD_CTX_create() and EVP_MD_CTX_destroy() functions were renamed to EVP_MD_CTX_new() and EVP_MD_CTX_free() in OpenSSL 1.1.0, respectively.

We can rename the **pointer_to_api** with this pythonIDA script, it gets **new_function_name** string and it then set the name for **pointer_to_api**

```
start = get_name_ea(0, "array_func_name")
for i in xrange(0x39):
        func_name =
get_strlit_contents(Qword(start))
        set_name(start+0x10, func_name)
        start += 0x18
```

Ignores signals

The malware will ignore the following signals, so that it won't be interrupted during encryption, this will prevent half encrypting files which leads to file corruption.

- SIGCHLD
- SIGTSTP
- SIGTTOU
- SIGTTIN
- SIGHUP
- SIGTERM

```
signal(SIGCHLD, (__sighandler_t)SIG_IGN);     // ignoring signals
signal(SIGTSTP, (__sighandler_t)SIG_IGN);
signal(SIGTTOU, (__sighandler_t)SIG_IGN);
signal(SIGTTIN, (__sighandler_t)SIG_IGN);
```

Figure: Malware ignores some signals

List VM processes

It executes the command `esxcli vm process list` to list every VirtualMachine processes currently running on the infected machine. It then parses through the output to extract **Process ID** and **Config File** which is basically the path to the **VMX file of the VM** This data is saved in a array of stucture of type

```
{
        uint64_t Process_ID;
        unsigned char *Vmx_Path;
}
```

kill VM porcesses

Using the previous array, the malware first tries to kill the processes with a soft kill `esxcli vm process kill -t=soft -w=<Process_PID>` if it fails it uses a hard kill option `esxcli vm process kill -t=hard -w=<Process_PID>` .

Recursive file search

It uses the paths given as command line arguments and explore recursively the directories using **opendir readdir**.

For each file read, it first checks if the file is not **.** or **..** and does not contain the strings **.crypt** or **.README_TO_RESTORE**

```
fd_path = opendir(path);
if ( fd_path )
{
LABEL_2:
  while ( 1 )
  {
    v3 = readdir64(fd_path);
    if ( !v3 )
      break;
    name_of_a_file = v3->d_name;
    v5 = 0x24LL;
    v6 = &stat_buf;
    while ( v5 )
    {
      LODWORD(v6->st_dev) = 0;
      v6 = (struct stat64 *)((char *)v6 + 4);
      --v5;
    }
    if ( strcmp(name_of_a_file, ".")
      && strcmp(name_of_a_file, "..")
      && !strstr(name_of_a_file, ".crypt")
      && !strstr(name_of_a_file, ".README_TO_RESTORE") )
    {
      v7 = 1024LL;
      v8 = &fill_pathfile;
      while ( v7 )
      {
        *(_DWORD *)v8 = 0;
        v8 += 4;
        --v7;
```

Switch (file type)

case directory:

It checks if it is not one of the following directories:

- /bin
- /boot
- /dev
- /etc
- /lib
- /lib32
- /lib64
- /lost+found
- /proc

- /run
- /sbin
- /usr/bin
- /usr/include
- /usr/lib
- /usr/lib32
- /usr/lib64
- /usr/sbin
- /sys
- /usr/libexec
- /usr/share
- /var/lib

In case the check pass, it calls recursively the same function with the new directory path as argument.

```
    v14 = stat_buf.st_mode & 0xF000;
    if ( v14 == __S_IFDIR )
    {
      v19 = 0LL;
      v20 = 1;
      do
      {   signed int v20; // ebp
        if ( !strcmp(&fill_pathfile, dir_list_dont_encrypt[v19]) )
          v20 = 0;
        ++v19;
      }
      while ( v19 != 21 );
      if ( v20 )                              // whitelisted directory
        browse_file_system(&fill_pathfile);
    }
```

Figure: Recursive directory search

case file:

In case it was a file and the **-e** option was specified in command line arguments, it will check if the file does not contain the following strings **.crypt, .tmp_, .README_TO_RESTORE**, then checks if it contains one of the following strings

- **.vmdk**
- **.vmx**
- **.vmsd**
- **.vmsn**

```
do
{
  if ( strstr(name_of_a_file, vm_files_extension[v21])
    && !strstr(name_of_a_file, ".crypt")
    && !strstr(name_of_a_file, ".tmp_")    char *[8]
    && !strstr(name_of_a_file, ".README_TO_RESTORE")
    && (unsigned __int64)sub_4048C3(&fill_pathfile) > 256 )// size > 256
  {
    if ( v_param_1 && stream_log_file )
    {
      abstime.tv_nsec = 0LL;
      abstime.tv_sec = 1LL;
      sem_timedwait(&sem, &abstime);
      __fprintf(stream_log_file, 1LL, "Find ESXi:%s\n", &fill_pathfile, v22, v23);
      fflush(stream_log_file);
      sem_post(&sem);
    }
    abstime.tv_sec = (__time_t)strdup(&fill_pathfile);
    sub_403FAE((__int64)&files_to_encrypt, &abstime.tv_sec, v24, v25);
```

if **-e** was not specified, it will check if the filename does not contain one of the following strings

- .crypt
- .READ_ME_TO_RESTORE
- .tmp_
- .a
- .so
- .la

Finally if the size of the file is bigger than 256 bytes, it saves the path to the file for later usage (encrypting it… of course)

switch to daemon process

If the **-d** option was specified in command line arguments the malware calls **daemon** to detach itself from the controlling terminal and run in the background as system daemons.

```c
if ( d_param_1 )
{
  if ( stream_log_file )
  {
    abstime.tv_nsec = 0LL;
    abstime.tv_sec = 1LL;
    sem_timedwait(&sem, &abstime);
    fputs("switch to daemon \n", stream_log_file);
    fflush(stream_log_file);
    sem_post(&sem);
  }
  fputs("switch to daemon \n", stderr);
  daemon(0, 1);                              // detach and run as daemon
}
clock();
```

Figure: Detach and run as daemon

start thread

it starts a thread at address **0x402AA2** then creates 2 strings, `filename + .crypt` and `filename + .tmp_`

```c
__sprintf(file_.crypt, 1, 0xFFFFFFFFFFFFFFFFLL, "%s%s", &filename, ".crypt");// file + .crypt
__sprintf(file_.tmp, 1, 0xFFFFFFFFFFFFFFFFLL, "%s%s", &filename, ".tmp_");// file + .tmp_
```

A function is called that tries to set a lock on the file using `fcntl` , if it fails it will get the PID of the process that is currently locking the file

```c
    *__errno_location() = 22;
    result = -1;
  }
  else
  {
    *(_DWORD *)&v5.l_type = O_WRONLY;
    v5.l_start = 0LL;
    v5.l_len = 0LL;
    result = fcntl(fd, F_SETLK, &v5, *(_QWORD *)&v5.l_type, 0LL, 0LL);// set lock
    if ( result )
    {
      v2 = __errno_location();
      v3 = *v2;
      v4 = v2;
      *(_DWORD *)&v5.l_type = 1;
      v5.l_start = 0LL;
      v5.l_len = 0LL;
      v5.l_pid = 0;
      if ( fcntl(fd, F_GETLK64, &v5, *(_QWORD *)&v5.l_type, 0LL, 0LL) || (result = v5.l_pid, v5.l_pid <= 0) )// get PID of process that has lock on fil
      {
        *v4 = v3;
        result = -1;
      }
    }
  }
  return result;
```

Figure: Malware try to set a lock on the file

If the PID is greater than 10 (not a system process), it will kill it with the command `kill -9 <PID>`

```
}
if ( (signed int)pid_file_open > 10 )
{
  LOBYTE(j) = -128;
  __snprintf_chk(&command, 128LL, 1LL, j, "kill -9 %d", pid_file_open);
  v21 = execute_command(&command);
```

The malware then rename the file to `filename + .tmp_` then it will call a function (**0x405D64**) to encrypt the file. In case of failure it will roll back to the original filename

In case of successful encryption it will rename the `.tmp_` to `.crypt`

Encryption

Generation of keys

It derives an **AES_256_CBC KEY** and **IV** with libcrypto function **EVP_BytesToKey** with a randomly generated **salt** and **data** using **RAND_bytes API** that will be used for file content encryption.

```
if ( !(unsigned int)RAND_bytes(&random_b32, 0x20LL) )
{
  if ( !stream_log_file )
    goto LABEL_35;
  abstime.tvFILE *= 0LL;
  abstime.tv_sec = 1LL;
  sem_timedwait(&sem, &abstime);
  v16 = file_.tmp_;
  v17 = 103LL;
  goto LABEL_33;
}
EVP_BytesToKey_ = (unsigned int (__fastcall *)(__int64, void *, __int64 *, __int64 *, signed __int64, signed __int64, char *, char *))EVP_BytesToKey
v19 = (void *)EVP_sha256();
EVP_MD_v22_aes = EVP_aes_256_cbc();
if ( EVP_BytesToKey_(EVP_MD_v22_aes, v19, &random_b8, &random_b32, 32LL, 12LL, &KEY, IV) != 32 )// derives an AES KEY and IV
```

Figure: Generate AES KEY/IV

EVP_BytesToKey - password based encryption routine

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,
                   const unsigned char *salt,
                   const unsigned char *data, int datal, int count,
                   unsigned char *key,unsigned char *iv);
```

DESCRIPTION

EVP_BytesToKey() derives a key and IV from various parameters. **type** is the cipher to derive the key and IV for. **md** is the message digest to use. The **salt** parameter is used as a salt in the derivation: it should point to an 8 byte buffer or NULL if no salt is used. **data** is a buffer containing **datal** bytes which is used to derive the keying data. **count** is the iteration count to use. The derived key and IV will be written to **key** and **iv** respectively.

Figure: Official documentation of OpenSSL API https://www.openssl.org/docs/man1.0.2/man3/EVP_BytesToKey.html

Afterwards it generates the malware ECDH private/public keys.

```
v6 = v2;
if ( !(unsigned int)EC_KEY_generate_key(v2) )
{
  if ( stream_log_file )
  {
    v9 = 0LL;
    v8 = 1LL;
    sem_timedwait(&sem, (const struct timespec *)&v8);
    v5 = 324LL;
    goto LABEL_7;
```

Figure: Generate client ECDH keypair

Next it will call a function at address **0x4054F1** (I named it **func_compute_secret**) with the newly generate **EC_KEY** and the public key of the author

```
v21 = (__int64 (__fastcall *)(__int64, __int64, signed __int64, __int64, signed __int64, _QWORD))EC_POINT_point2oct;
v22 = EC_KEY_get0_public_key(EC_KEY_key);
v23 = EC_KEY_get0_group(generated_EC_KEY_key);
v24 = v21(v23, v22, 4LL, v14 + 92, 72LL, 0LL);
v25 = author_publickey_hardcoded;
EC_POINT_hex2point_ = (__int64 (__fastcall *)(__int64, char *, _QWORD, _QWORD))EC_POINT_hex2point;
*(_DWORD *)(v14 + 88) = v24;
v27 = EC_KEY_get0_group(generated_EC_KEY_key);
peerkey_pubkey = EC_POINT_hex2point_(v27, v25, 0LL, 0LL);
v29 = peerkey_pubkey;
if ( peerkey_pubkey )
{
  if ( (unsigned int)func_compute_secret(&Psecret_, (__int64)generated_EC_KEY_key, peerkey_pubkey) == 0x20 )// https://man.openbsd.org/ECDH_comput
  {
```

```
db '046250168DEB3A40F1A34C4D0A5EA303306DA15C3A19A39A3F04A07C289953DC3'
                              ; DATA XREF: .data:author_publickey_hardcoded↓o
db 'E6409103C87374603AD312764DB0BA5BE001F706D50236F0FC6C25BBD3BC8AA93'
db 0
```

Figure: Author public key

Then It calls libcrypto api **ECDH_compute_key** to generate an **ECDH shared secret**.

```
EC_GROUP_get_degree_ = (__int64 (__fastcall *)(__int64))EC_GROUP_get_degree;
v5 =__int64 peerkey_pubkey; // r12;
secret_len = (signed int)((unsigned __int64)EC_GROUP_get_degree_(v5) + 7) / 8;
secret = malloc(secret_len);
*Psecret_ = secret;
if ( secret )
{
  v8 = ECDH_compute_key(secret, secret_len, peerkey_pubkey, EC_KEY_key, 0LL);
}
else
```

Figure: Generate ECDH secret

After that it populate a custom structure `(I named it custom_structure00)` of the following type with the **AES KEY, IV** and the size of the file.

```
{
        unsigned char*
save_AES_IV[0x10];
        unsigned char*
save_AES_KEY[0x20];
        unsigned __int64
size_of_file;
        unsigned int
defined_constant;
        unsigned int alignemnt;
} custom_structure00;
```

```
sub_40439D((__int64)&save_aes_data, IV_, 0x10uLL);// save_IV
sub_40439D((__int64)&save_aes_data.save_AES_KEY, (__int64)KEY_, 0x20uLL);
v33 = 0x40LL;
*(_DWORD *)&save_aes_data.bool = v8;
save_aes_data.size_of_file = v6;
```

Figure: Populate the above structure with AES key data

```
db 0F3h                                    ; save_AES_IV
db 23h, 2Eh, 7Fh, 29h, 38h, 96h, 8Eh, 0D5h, 0D2h, 40h, 2, 0D6h, 85h, 0B6h
db 61h
db 0D2h                                    ; save_AES_KEY
db 30h, 6Dh, 43h, 3Eh, 30h, 41h, 67h, 7Ah, 0A1h, 0F1h, 51h, 1Eh, 97h, 0D6h
db 4Fh, 0F4h, 77h, 0C0h, 0FAh, 0F4h, 94h, 37h, 0C4h, 35h, 43h, 0DCh, 0A9h
db 0DCh, 38h, 0CCh, 0ADh
dq 1389h                                   ; size_of_file
dw 0FFh                                    ; bool
dw 0                                       ; alignment
```

Figure: Example of the structure populated with data

Then it encrypts the structure `custom_structure00` using the **ECDH secret** and a randomly generated **IV** of 16 bytes with AES algorithm.

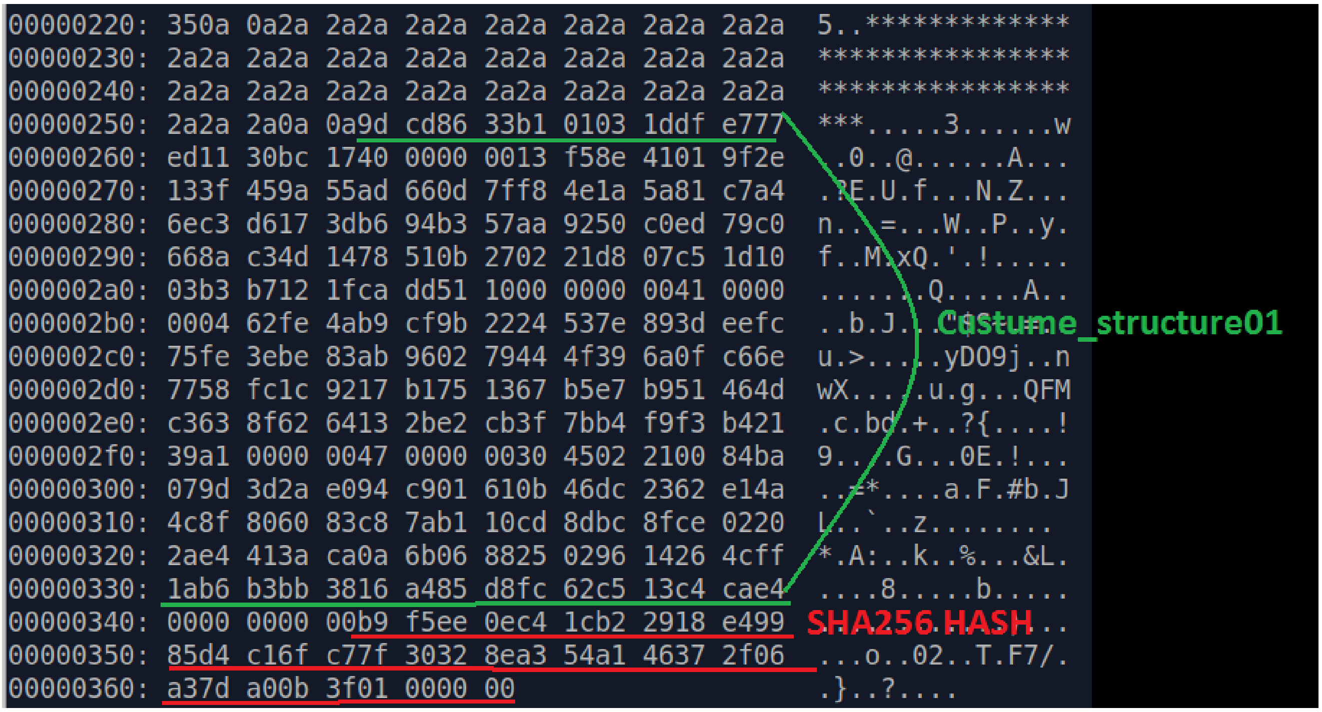
```
ECDH_secret = Psecret_;
EVP_EncryptInit_ex_ = (unsigned int (__fastcall *)(__int64, __int64, _QWORD, void *, __int64 *))EVP_EncryptInit_ex;
aes_engine_cypher = EVP_aes_256_cbc();
if ( EVP_EncryptInit_ex_(v38_cypher_context, aes_engine_cypher, 0LL, ECDH_secret, &v60) )// secret
{
  if ( (unsigned int)EVP_EncryptUpdate(v38_cypher_context, &v63, &v57, &save_aes_data, 0x40LL) )
  {                                  // (EVP_CIPHER_CTX *ctx, unsigned char *out,
                                     //          int *outl, const unsigned char *in, int inl);
    v41 = v57;
    v42 = &v63;
```

Figure: Encrypt the above structure with the **ECDH secret**

After that, it populates yet another important structure `custom_structure01` of following type:

```
{
        // The IV used to encrypt the custom_structure00 structure. | offset 0 -
0x10
        unsigned char secret_IV[0x10];
        // The size of custom_structure00 | offset 0x10 - 0x14
        unsigned int size_of_custom_structure00;
        // Encrypted custom_structure00 | offset 0x14 - 0x34
        custom_structure00 encrypted_custom_structure00;
        // Size of the client public key | offset 0x58 - x5c
        unsigned int size_of_public_key ;
        // Client public key | offset 0x5c - 0xa0
        unsigned char public_key[0x44];
        // Size of the sig | offset 0xa0 - 0xa4
        unsigned int size of sig;
        //  Sig | offset 0xa4 - 0xf0
        unsigned char sig[0x47];
} custom_structure01;
```

Finally it writes to `filename + .README_TO_RESTORE` the ransomware note, then it append the previous structure at the end of the file. It also append the SHA256 of the `<original file content + appended data>` (see next) Example:



`.README_TO_RESTORE` file tail, showcasing the above structure `(custom_structure01)` and the SHA256 hash of the file

It then appended the same structure `(custom_structure01)` to the target file.

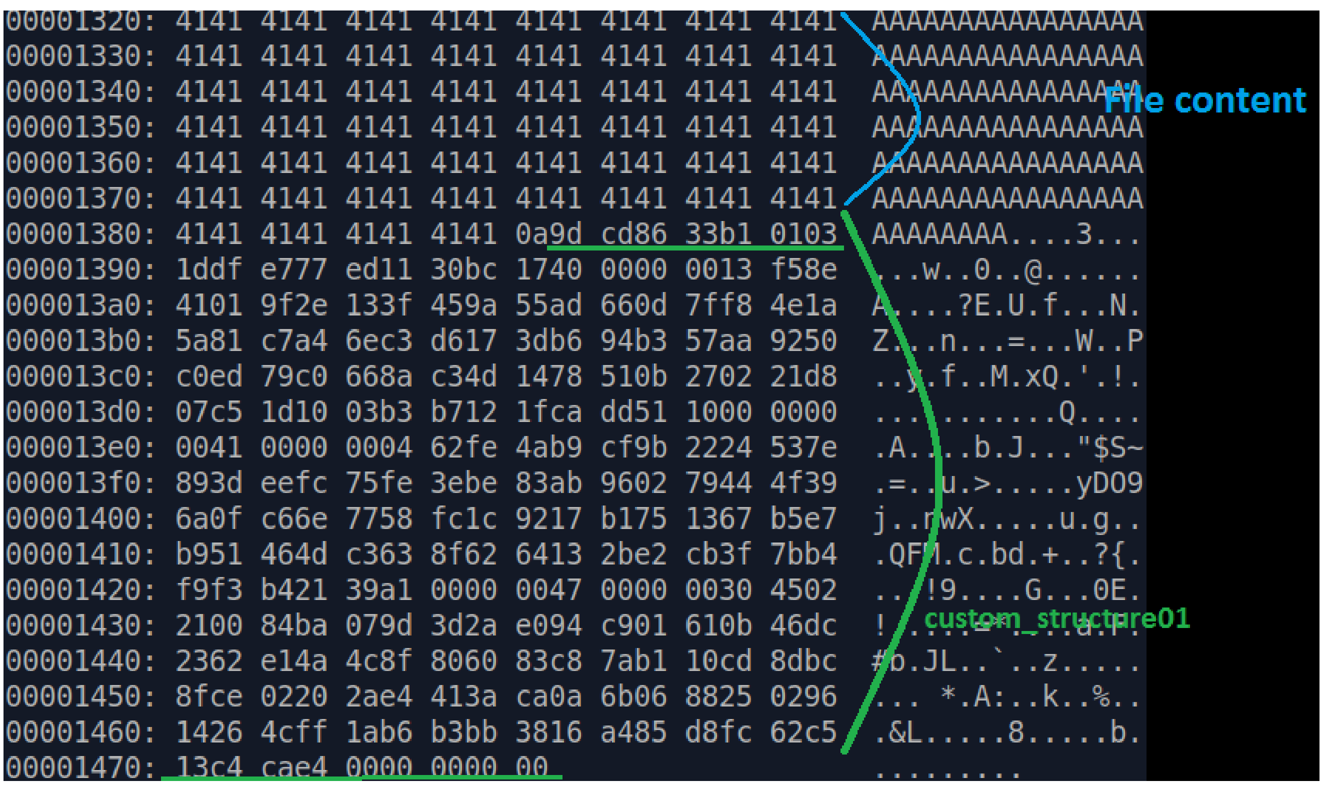


Figure: Original content of the file + `(custom_structure01)`

Finally it reads the data of the target file and uses the file encryption **AES key** to encrypt it.

```c
while ( 1 )
{
  v33 = read(filetmp_fd_, file_content, 0x100000uLL);// READFILE
  v34 = v33;
  if ( v33 <= 0 )
    break;
  if ( !v51 )
  {
    ptra_maybe_sha_of_content = hash_content((__int64)file_content, v33, (__int64)&v52);
    sub_404425(v25, filetmp_fd_, ptra_maybe_sha_of_content, v52);
  }
  lseek64(filetmp_fd_, -v34, 1);
  encrypt(&compute_key_iv, (__int64)v30, (__int64)file_content, v34);
  if ( write(filetmp_fd_, v30, v34) != v34 )
```

Figure: Malware encrypt the content of the target file

```
000012c0: ead0 5a44 c580 7fb6 01f1 f7a6 45a4 8339  ..ZD........E..9
000012d0: 9f85 bbf2 5bfa 3f16 fb5b c424 1b59 940a  ....[.?..[.$.Y..
000012e0: ff4e a834 8872 c365 953d bb50 f2fb 8d64  .N.4.r.e.=.P...d
000012f0: ba18 cbc0 5ebe 17f5 6c77 e558 8b0c 802e  ....^...lw.X....
00001300: 96fd 359a 59a2 3f94 e275 6d67 8ddc 174f  ..5.Y.?..umg...O
00001310: b921 0aa7 9664 eaa2 c4bd a8a0 19e9 1fd8  .!...d..........
00001320: 4d97 efd3 48d4 bdc4 bcae a149 eec6 bbb6  M...H......I....
00001330: aff7 ebb4 3393 261c b42a 6e28 1fb6 3015  ....3.&..*n(..0.
00001340: 9bcf 4564 2e2c 12c1 e8ec 62c1 303e cbb0  ..Ed.,....b.0>..
00001350: 83a0 447f ca54 9b2f ac9e c476 3ad4 d9c1  ..D..T./...v:...
00001360: ea0b 928f 59b5 2d9a 3e83 36c4 7f89 43e6  ....Y.-.>.6...C.
00001370: 4af2 6334 c867 0a8f 2817 aa33 c514 70b1  J.c4.g..(..3..p.
00001380: 1c36 02b2 5ea5 fd2e 5f9f 8c91 0b7a 29e8  .6..^..._....z).
00001390: 0564 f7e3 bb02 6e2c f065 682d eb22 5f6d  .d....n,.eh-."_m
000013a0: 7f25 6731 1430 152b 8fd6 ffed c629 4482  .%g1.0.+.....)D.
000013b0: 5022 48e0 da6e ebdb 4320 0047 f160 6041  P"H..n..C .G.``A
000013c0: 301e 98e8 ebaf 9e23 3076 0031 c166 a9d9  0......#0v.1.f..
000013d0: ad8f 03ce f477 f626 9988 13df d37f 915d  .....w.&.......]
000013e0: 2456 9362 8e53 38b1 a63c ffed ee87 324f  $V.b.S8..<....2O
000013f0: 0ce7 a970 a21d ac01 3eee ad3f ff02 7bc9  ...p....>..?..{.
00001400: eff8 6f99 fd98 82bc 4660 5637 a361 179c  ..o.....F`V7.a..
00001410: 98ad 0c2a 471a dd10 f67c 01e1 a0f3 bc80  ...*G....|......
00001420: c191 e295 a953 171f 7b40 741e 3a29 ac55  .....S..{@t.:).U
00001430: 8d10 8981 f6bc 7439 6acd 9e5a 0915 c097  ......t9j..Z....
00001440: 7c91 92f8 9aaa ab8b df1f 06cb bafb 6ee6  |.............n.
00001450: 8602 3074 7758 49b1 5da5 59a1 c46b ff1b  ..0twXI.].Y..k..
00001460: 74a1 f6db 5985 97bd fdad b7f5 279a d6a6  t...Y.......'...
00001470: c8a5 de80 620d c607 c4                   ....b....
```

Figure: File encrypted

Finally it append again the structure + the sha256 of the file.

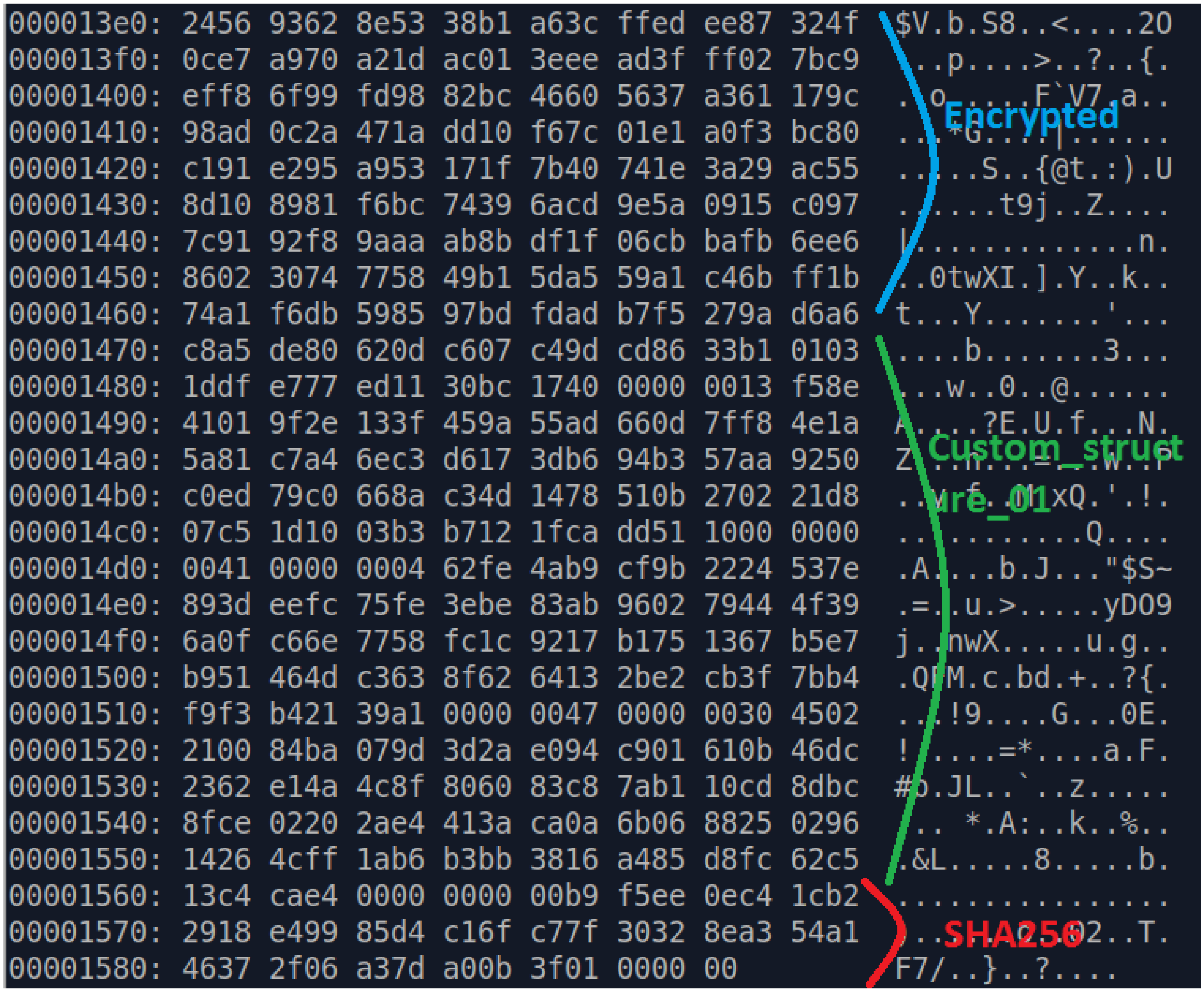


Figure: Encrypted file + custom_structure01 + sha256

Conclusion

This linux variant can target Virtual machines files, which can be crucial to companies without backup and replication.

The encryption scheme used utilize ECDH (Elliptic-curve Diffie–Hellman) algorithm, which means without the private key owned by the threat actor, it will be near impossible to decrypt the encrypted files.

YARA Rule

```
rule hellokitty_linux {
    meta:
        description = "YARA rule HelloKitty linux variant ransomware"
        reference = "https://soolidsnake.github.io/2021/07/17/hellokitty_linux.html"
        author = "@soolidsnakee"
        date = "2021-07-17"
    strings:
        $str1 = ".crypt"
        $str2 = ".README_TO_RESTORE"
        $str5 = "switch to daemon"
        $str6 = "esxcli vm process kill -t=hard -w=%d"
        $str7 = "work.log"
        $str8 = "m:vdekc:"
    condition:
        all of ($str*)
}
```