Offset Training Solutions

Recognizing Common Cryptographic Algorithms - Encryption

Introduction

This will be one of about three theoretical PDFs that showcase different algorithms, and how you can recognize them in malware/software, whether through constants, function structure, or byte patterns!

This is less of a guide on "*here are exact methods of determining exactly what algorithm is in use*", and more of a "*here is how I analysed this specific case and discovered what algorithm is being used*". The main reasoning behind this is threat actors are constantly improving every day. It is very simple to obfuscate an RC4 encryption algorithm, to a point where you are using completely foreign assembly instructions to generate a substitution box. If you checked out the Shellcode chapter, you'll know we saw an RC4 algorithm in GuLoader that used very uncommon assembly instructions to achieve the same outcome.
Algorithm determination will differ between samples, and so it requires a certain level of detective work to find specific values that can be attributed to a specific algorithm. While I could show you every major algorithmic constant in the world, it is only of use if an attacker does not obfuscate/encrypt those constants. The same thing goes for function structure, and byte patterns.
If I can show you *how* to go about analyzing a specific algorithmic function, or at least give pointers on what I personally look for, you can take that knowledge and look for similar features in malware you are looking at. Features such as function length like in Serpent, function structure like in RC4, and function core like in Salsa20.
In certain cases I have limited explanation to constants, and that is due to the fact that the algorithms are very large and complex, and would take you a large amount of time to analyze it by hand. However, these algorithms will always contain dead-giveaways that they are using substitution boxes, like with AES and it's use of multiple loops XORing tables in memory with data.

This is definitely a first for me, as I have not developed something like this before, so any and all feedback would be greatly appreciated, as well as any additions I could make! There are definitely more (a)symmetric algorithms, but from my perspective these are the main ones that we see being used in malware everyday - if there is something you'd like to see, drop me a message and I can definitely look into updating this!

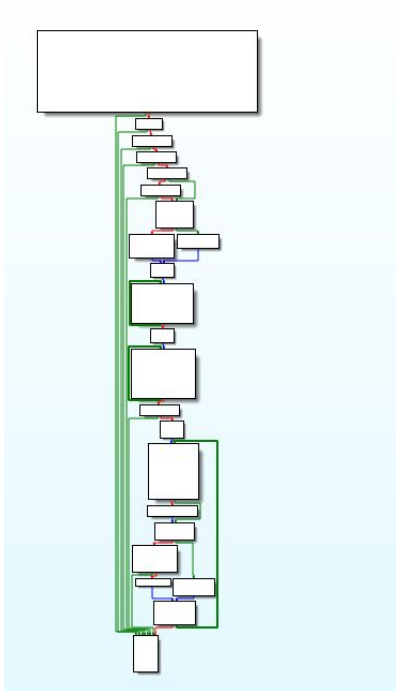
Encryption Algorithms

Symmetric Algorithms

RC4 - Stream Cipher

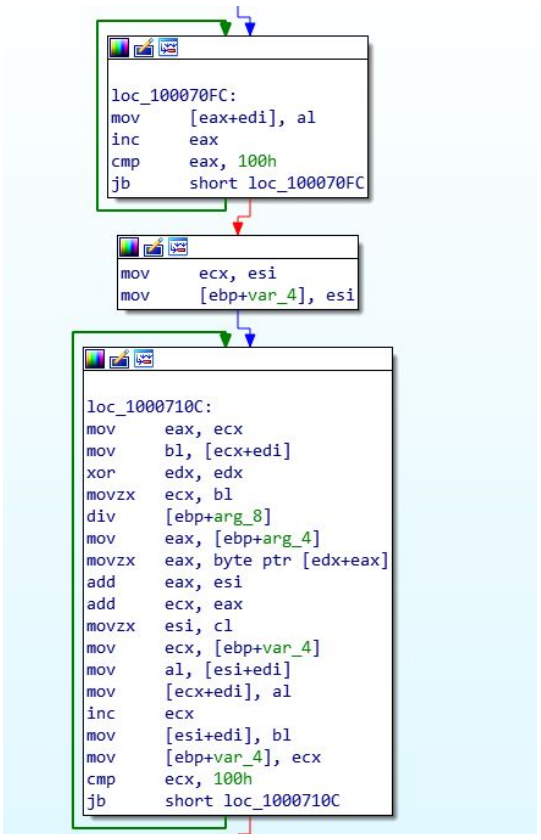
Rivest Cipher 4 is one of the most common algorithms you'll discover in malware, whether to decrypt strings or binaries, or encrypt network communications. It is a small algorithm, composed of 3 main parts. The key-scheduling algorithm (initialization stage), the pseudo-random generation stage (scrambling stage), and the XOR stage.
The initialization stage will simply create what is known as a substitution box, a table of values from **0x00** to **0xFF**, which has a size of 256 bytes (**0x100**). One of the giveaways RC4 is being used is the usage of **0x100** in 2 loops. Firstly in the initialization stage, and then in the scrambling stage. The scrambling stage will use the RC4 key to scramble the substitution box, creating a stream of semi-random bytes. This leads into the final stage, which will XOR the plaintext or ciphertext against the created stream, either encrypting or decrypting the data. In the case of RC4, RC4Encrypt() is the same as RC4Decrypt().
The RC4 keys can be between 1 and 256 bytes in length, however it is usually recommended that it is above 5 bytes. Commonly, RC4 keys are 16 bytes in length.

On the right you will see a disassembly graph from IDA. While the RC4 algorithm is fairly small, threat actors will try to obfuscate their code as much as possible, especially if they are sophisticated. The graph on the right was taken from a 2020 Dridex sample, and contains a lot more code than 3 simple loops - but if you look hard enough, you might notice 2 boxes that are quite similar in size, and seem to loop around itself before continuing execution. While this is not typically enough to determine RC4, it gives you a good hint that it could be.
When we dive deeper into the code, you'll see that regardless of how much obfuscation is added to the code, as long as there are 2 loops responsible for both generating and scrambling a substitution box, it should be fairly simple to determine if something is RC4. In certain cases, the number may be more than 256, in which case, you will have to check the scrambling stage and XOR stage to confirm.

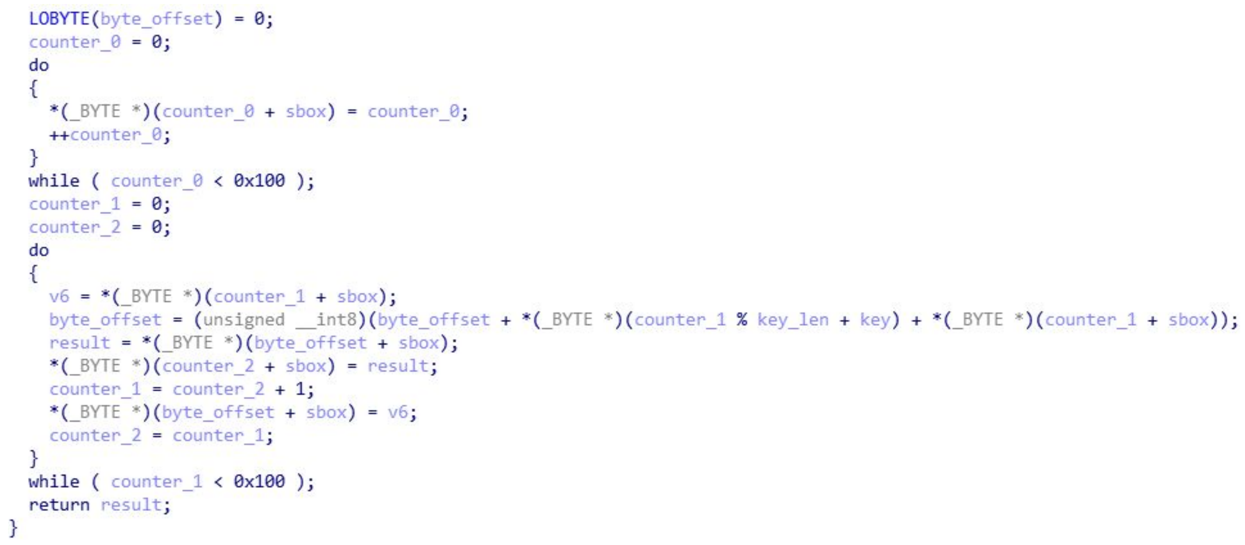


Initialization and Scrambling Stage:
On the right is a close-up image of the RC4 initialization and scrambling routine, this time taken from the REvil ransomware. The first loop will run 256 times, and will increment *eax* on each loop, which will be used to create the substitution box by moving *al* into *[eax+edi]*, where *edi* is a pointer to memory. The second loop will scramble this substitution box, using the key to generate pseudo-random bytes. **arg_8** is the key length, **arg_4** is the key, and **var_4** is used as a counter. A decompiled and commented version of this routine can be seen below as well, for further clarity.

This routine will be followed by the majority of malware families, rather than re-implement the algorithm, so in most cases when you see a similar routine, you will be able to easily identify it as RC4. You can use a tool such as CyberChef to decrypt it quickly before developing any kind of configuration extractor or string decryptor.

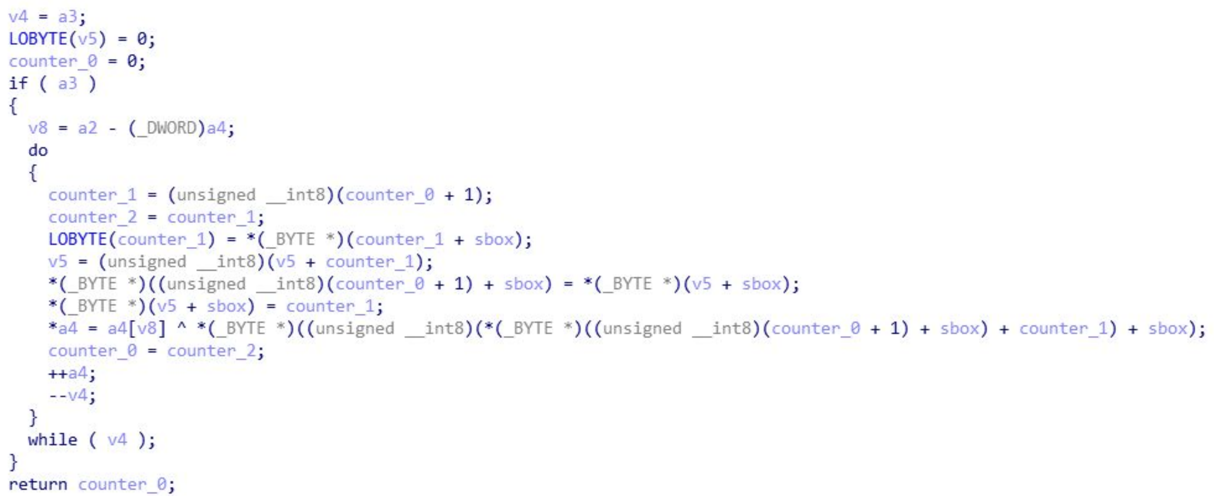


```
loc_100070FC:
mov     [eax+edi], al
inc     eax
cmp     eax, 100h
jb      short loc_100070FC

mov     ecx, esi
mov     [ebp+var_4], esi

loc_1000710C:
mov     eax, ecx
mov     bl, [ecx+edi]
xor     edx, edx
movzx   ecx, bl
div     [ebp+arg_8]
mov     eax, [ebp+arg_4]
movzx   eax, byte ptr [edx+eax]
add     eax, esi
add     ecx, eax
movzx   esi, cl
mov     ecx, [ebp+var_4]
mov     al, [esi+edi]
mov     [ecx+edi], al
inc     ecx
mov     [esi+edi], bl
mov     [ebp+var_4], ecx
cmp     ecx, 100h
jb      short loc_1000710C
```



```
  LOBYTE(byte_offset) = 0;
  counter_0 = 0;
  do
  {
    *(_BYTE *)(counter_0 + sbox) = counter_0;
    ++counter_0;
  }
  while ( counter_0 < 0x100 );
  counter_1 = 0;
  counter_2 = 0;
  do
  {
    v6 = *(_BYTE *)(counter_1 + sbox);
    byte_offset = (unsigned __int8)(byte_offset + *(_BYTE *)(counter_1 % key_len + key) + *(_BYTE *)(counter_1 + sbox));
    result = *(_BYTE *)(byte_offset + sbox);
    *(_BYTE *)(counter_2 + sbox) = result;
    counter_1 = counter_2 + 1;
    *(_BYTE *)(byte_offset + sbox) = v6;
    counter_2 = counter_1;
  }
  while ( counter_1 < 0x100 );
  return result;
}
```

XOR Stage:
Finally, the last stage of RC4 is the XOR stage, where the plaintext or ciphertext is XOR'ed against the generated keystream. Sometimes this loop can be fairly simple and recognizable with a **% 256** (mod 256) in the decompiler view, for data that is larger than 256 bytes. In this case, REvil uses multiple incremental counters, however we can see that the substitution box is used to XOR data in **a2**, and store it in **a4** which is most likely an out-buffer.



```
v4 = a3;
LOBYTE(v5) = 0;
counter_0 = 0;
if ( a3 )
{
  v8 = a2 - (_DWORD)a4;
  do
  {
    counter_1 = (unsigned __int8)(counter_0 + 1);
    counter_2 = counter_1;
    LOBYTE(counter_1) = *(_BYTE *)(counter_1 + sbox);
    v5 = (unsigned __int8)(v5 + counter_1);
    *(_BYTE *)((unsigned __int8)(counter_0 + 1) + sbox) = *(_BYTE *)(v5 + sbox);
    *(_BYTE *)(v5 + sbox) = counter_1;
    *a4 = a4[v8] ^ *(_BYTE *)((unsigned __int8)(*(_BYTE *)((unsigned __int8)(counter_0 + 1) + sbox) + counter_1) + sbox);
    counter_0 = counter_2;
    ++a4;
    --v4;
  }
  while ( v4 );
}
return counter_0;
```

The RC4 algorithm can take many different shapes depending on the obfuscation used, key size, substitution box size, however as long as you can see 2 code blocks looping **n** amount of times, with an XOR code block seemingly XORing data against what was generated in the previous loops, you can make a calculated assumption you are dealing with RC4.

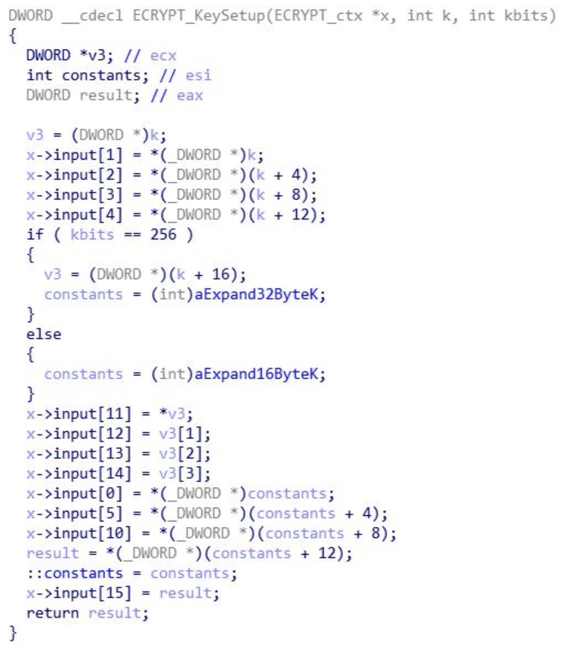
Salsa20 - Stream Cipher

The Salsa20 stream cipher is becoming more and more common in strains of ransomware, due to the speed at which it can encrypt fairly large chunks of data. While Salsa20 is commonly used to encrypt files, network communications are typically encrypted with AES/RC4, where speed isn't a necessity. Salsa20 has a 256 bit mode, as well as a 128 bit mode, however it is extremely easy to recognize this algorithm based on a string constant:

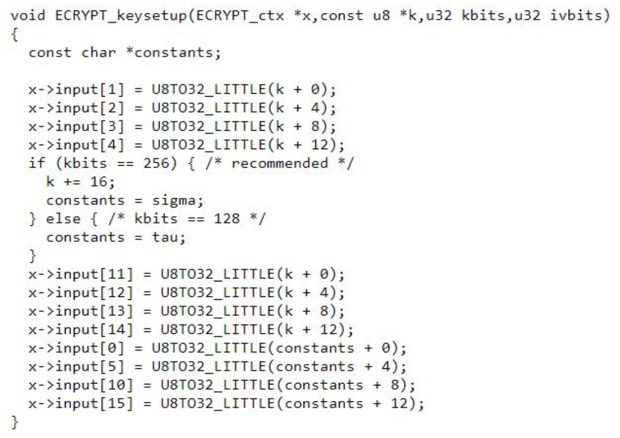
expand 32-byte k
expand 16-byte k

On the right you will see an example of the Salsa20 key setup. Again, this is taken from REvil ransomware, and as you can see there is direct interaction with the two string constants, based on whether the third argument is equal to **256** or not. Not only are we able to determine that the sample uses Salsa20, but we can also find out where the attackers copied the code from. Doing some searching for **salsa20 implementation C** yields a large number of results, however an interesting one can be located here: https://cr.yp.to/snuffle.html. This is an official implementation by the creator himself, and comparing the key setup function to the key setup function in IDA, you'll notice a lot of similarities. These continue on with the IV setup, and finally the actual encryption.

Salsa20 requires an IV and a key to encrypt/decrypt data, compared to RC4 which requires only a key. In most cases, the IV will be nulled, but as this is ransomware, it will be randomly generated.



```
DWORD __cdecl ECRYPT_KeySetup(ECRYPT_ctx *x, int k, int kbits)
{
  DWORD *v3; // ecx
  int constants; // esi
  DWORD result; // eax

  v3 = (DWORD *)k;
  x->input[1] = *(_DWORD *)k;
  x->input[2] = *(_DWORD *)(k + 4);
  x->input[3] = *(_DWORD *)(k + 8);
  x->input[4] = *(_DWORD *)(k + 12);
  if ( kbits == 256 )
  {
    v3 = (DWORD *)(k + 16);
    constants = (int)aExpand32ByteK;
  }
  else
  {
    constants = (int)aExpand16ByteK;
  }
  x->input[11] = *v3;
  x->input[12] = v3[1];
  x->input[13] = v3[2];
  x->input[14] = v3[3];
  x->input[0] = *(_DWORD *)constants;
  x->input[5] = *(_DWORD *)(constants + 4);
  x->input[10] = *(_DWORD *)(constants + 8);
  result = *(_DWORD *)(constants + 12);
  ::constants = constants;
  x->input[15] = result;
  return result;
}
```

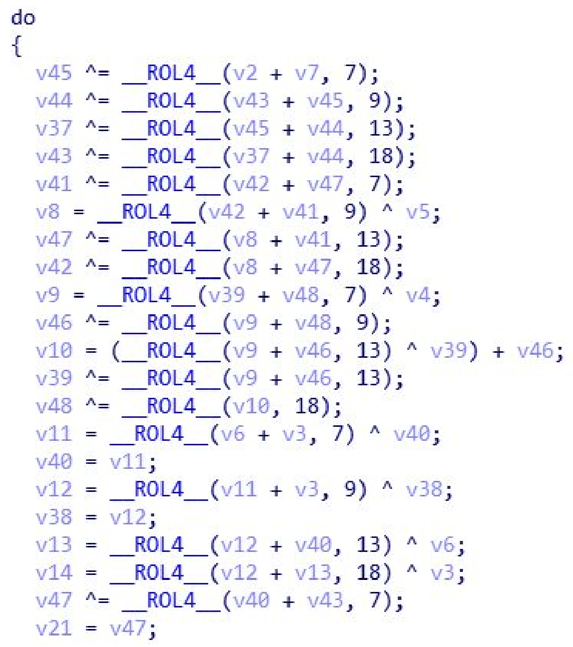


```
void ECRYPT_keysetup(ECRYPT_ctx *x,const u8 *k,u32 kbits,u32 ivbits)
{
  const char *constants;

  x->input[1] = U8TO32_LITTLE(k + 0);
  x->input[2] = U8TO32_LITTLE(k + 4);
  x->input[3] = U8TO32_LITTLE(k + 8);
  x->input[4] = U8TO32_LITTLE(k + 12);
  if (kbits == 256) { /* recommended */
    k += 16;
    constants = sigma;
  } else { /* kbits == 128 */
    constants = tau;
  }
  x->input[11] = U8TO32_LITTLE(k + 0);
  x->input[12] = U8TO32_LITTLE(k + 4);
  x->input[13] = U8TO32_LITTLE(k + 8);
  x->input[14] = U8TO32_LITTLE(k + 12);
  x->input[0] = U8TO32_LITTLE(constants + 0);
  x->input[5] = U8TO32_LITTLE(constants + 4);
  x->input[10] = U8TO32_LITTLE(constants + 8);
  x->input[15] = U8TO32_LITTLE(constants + 12);
}
```

If you are still trying to determine if something is Salsa20, you can take a look at the main encryption loop. The main component of the Salsa20 algorithm is the **Quarter-Round**, which takes a 4-word input and produces a 4-word output. Pseudocode from Wikipedia can be seen below:

```
#define QR(a, b, c, d)(
      b ^= ROTL(a + d, 7),
      c ^= ROTL(b + a, 9),
      d ^= ROTL(c + b,13),
      a ^= ROTL(d + c,18)
)
```

From this, you're mainly looking for a loop that will perform several rotate-left operations, with the values 7, 9, 13 and 18, as well as addition and XOR instructions. Looking at REvil, that's exactly what we have:

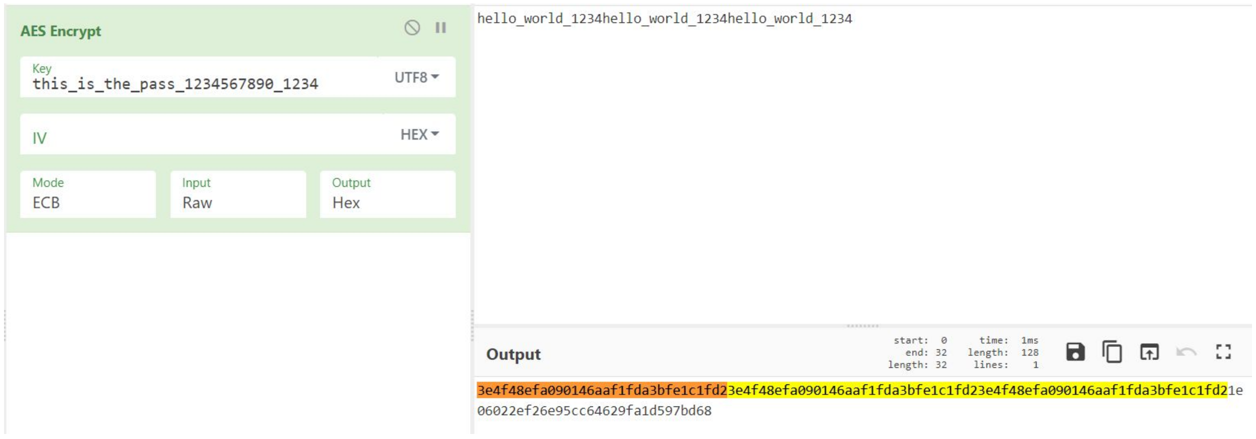


```
do
{
  v45 ^= __ROL4__(v2 + v7, 7);
  v44 ^= __ROL4__(v43 + v45, 9);
  v37 ^= __ROL4__(v45 + v44, 13);
  v43 ^= __ROL4__(v37 + v44, 18);
  v41 ^= __ROL4__(v42 + v47, 7);
  v8 = __ROL4__(v42 + v41, 9) ^ v5;
  v47 ^= __ROL4__(v8 + v41, 13);
  v42 ^= __ROL4__(v8 + v47, 18);
  v9 = __ROL4__(v39 + v48, 7) ^ v4;
  v46 ^= __ROL4__(v9 + v48, 9);
  v10 = (__ROL4__(v9 + v46, 13) ^ v39) + v46;
  v39 ^= __ROL4__(v9 + v46, 13);
  v48 ^= __ROL4__(v10, 18);
  v11 = __ROL4__(v6 + v3, 7) ^ v40;
  v40 = v11;
  v12 = __ROL4__(v11 + v3, 9) ^ v38;
  v38 = v12;
  v13 = __ROL4__(v12 + v40, 13) ^ v6;
  v14 = __ROL4__(v12 + v13, 18) ^ v3;
  v47 ^= __ROL4__(v40 + v43, 7);
  v21 = v47;
```

Additional obfuscation can be done, but it is quite uncommon unless the attackers have a great understanding of the algorithm, so that they do not break it by trying to obfuscate it.

AES - Block Cipher

The **Advanced-Encryption-Standard** algorithm is considered one of the most secure symmetric encryption algorithms, but it can also be used for random byte generation. It supports 128, 192, and 256 bit key lengths, and certain modes utilize IV's. By certain modes I am referring to AES-ECB, AES-CBC, AES-CFB, AES-OFB, and AES-CTR. AES-ECB is the weakest of the 5, as it does not use an IV and will encrypt each block of 128 bits with the same key. AES-CBC is one of the more commonly used modes, which uses an IV and a key. If you encrypted 3 128 bit blocks of the same string with AES-CBC, they would be completely different, whereas with AES-ECB they would be the same. If you want to learn more about the different modes, I'd highly recommend you check out this [here]().



AES Encrypt

Key: this_is_the_pass_1234567890_1234 (UTF8)

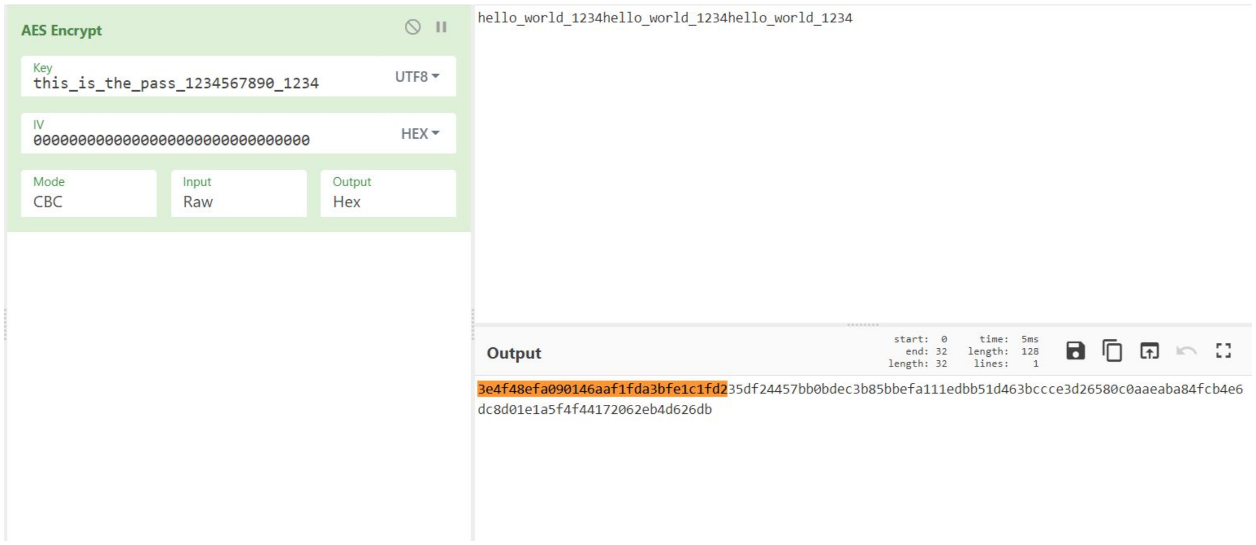
IV: (HEX)

Mode: ECB
Input: Raw
Output: Hex

```
hello_world_1234hello_world_1234hello_world_1234
```

Output

```
start:  0    time:  1ms
  end: 32  length: 128
length: 32   lines:   1

3e4f48efa090146aaf1fda3bfe1c1fd23e4f48efa090146aaf1fda3bfe1c1fd23e4f48efa090146aaf1fda3bfe1c1fd21e
06022ef26e95cc64629fa1d597bd68
```



AES Encrypt

Key: this_is_the_pass_1234567890_1234 (UTF8)

IV: 00000000000000000000000000000000 (HEX)

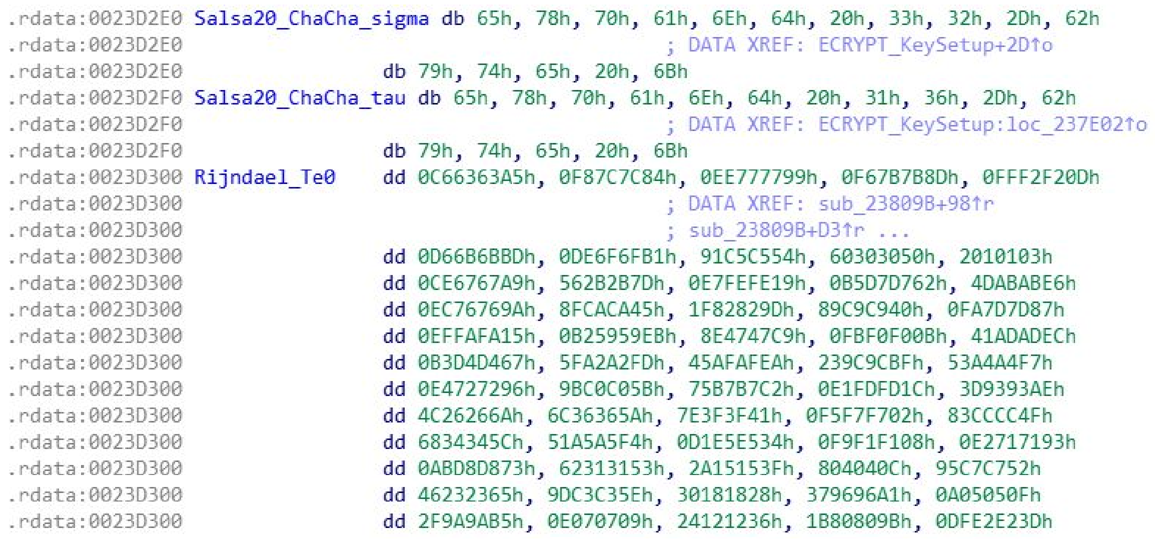
Mode: CBC
Input: Raw
Output: Hex

```
hello_world_1234hello_world_1234hello_world_1234
```

Output

```
start:  0    time:  5ms
  end: 32  length: 128
length: 32   lines:   1

3e4f48efa090146aaf1fda3bfe1c1fd235df24457bb0bdec3b85bbefa111edbb51d463bccce3d26580c0aaeaba84fcb4e6
dc8d01e1a5f4f44172062eb4d626db
```

The easiest way to discover the AES algorithm in a sample of malware is through the lookup tables, either S-Boxes or T-Tables, depending on the implementation. To speed up your searching, you can use a tool such as **Find-Crypt**, which is an IDA Python plugin used to recognize algorithm constants. In this case (REvil), it has not only found the Salsa20 constants, but also AES Tables:

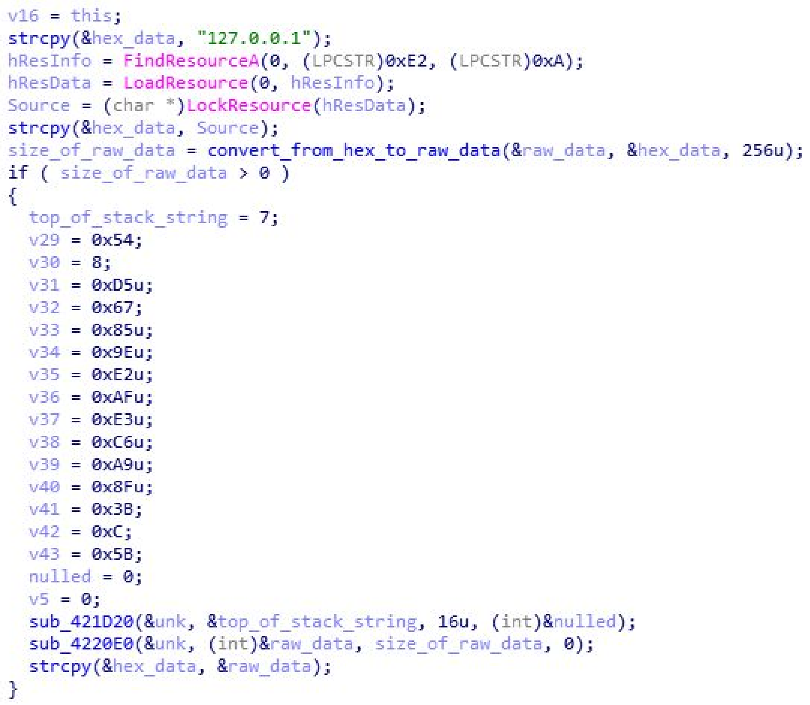


```
.rdata:0023D2E0 Salsa20_ChaCha_sigma db 65h, 78h, 70h, 61h, 6Eh, 64h, 20h, 33h, 32h, 2Dh, 62h
.rdata:0023D2E0                                         ; DATA XREF: ECRYPT_KeySetup+2D↑o
.rdata:0023D2E0                 db 79h, 74h, 65h, 20h, 6Bh
.rdata:0023D2F0 Salsa20_ChaCha_tau db 65h, 78h, 70h, 61h, 6Eh, 64h, 20h, 31h, 36h, 2Dh, 62h
.rdata:0023D2F0                                         ; DATA XREF: ECRYPT_KeySetup:loc_237E02↑o
.rdata:0023D2F0                 db 79h, 74h, 65h, 20h, 6Bh
.rdata:0023D300 Rijndael_Te0    dd 0C66363A5h, 0F87C7C84h, 0EE777799h, 0F67B7B8Dh, 0FFF2F20Dh
.rdata:0023D300                                         ; DATA XREF: sub_23809B+98↑r
.rdata:0023D300                                         ; sub_23809B+D3↑r ...
.rdata:0023D300                 dd 0D66B6BBDh, 0DE6F6FB1h, 91C5C554h, 60303050h, 2010103h
.rdata:0023D300                 dd 0CE6767A9h, 562B2B7Dh, 0E7FEFE19h, 0B5D7D762h, 4DABABE6h
.rdata:0023D300                 dd 0EC76769Ah, 8FCACA45h, 1F82829Dh, 89C9C940h, 0FA7D7D87h
.rdata:0023D300                 dd 0EFFAFA15h, 0B25959EBh, 8E4747C9h, 0FBF0F00Bh, 41ADADECh
.rdata:0023D300                 dd 0B3D4D467h, 5FA2A2FDh, 45AFAFEAh, 239C9CBFh, 53A4A4F7h
.rdata:0023D300                 dd 0E4727296h, 9BC0C05Bh, 75B7B7C2h, 0E1FDFD1Ch, 3D9393AEh
.rdata:0023D300                 dd 4C26266Ah, 6C36365Ah, 7E3F3F41h, 0F5F7F702h, 83CCCC4Fh
.rdata:0023D300                 dd 6834345Ch, 51A5A5F4h, 0D1E5E534h, 0F9F1F108h, 0E2717193h
.rdata:0023D300                 dd 0ABD8D873h, 62313153h, 2A15153Fh, 804040Ch, 95C7C752h
.rdata:0023D300                 dd 46232365h, 9DC3C35Eh, 30181828h, 379696A1h, 0A05050Fh
.rdata:0023D300                 dd 2F9A9AB5h, 0E070709h, 24121236h, 1B80809Bh, 0DFE2E23Dh
```

A quick search for the DWORD **0xC66363A5** or the string **Rijndael_Te0** will pull up multiple results about the AES encryption algorithm, such as **this** from the OpenSSH project. In this case, the DWORD in question is part of a **T-Table** used in AES instead of the usual substitution box. The same method of searching can be used if a substitution box is used, such as the DWORD **0x637C777B**. The AES algorithm also contains a reverse substitution box, where the first DWORD is **0xD56A0952**. Due to the size of the algorithm and complexity of it, it is much easier in this case to base your assumption on boxes and tables you discover, at least in the static analysis phase. When you get to performing dynamic analysis, you simply need to discover the key being used to decrypt/encrypt data, as well as the IV, and then you can run tests on something like CyberChef to confirm your findings. This is the most efficient way to analyse algorithms, especially if you're not a crypto-wizard like myself.

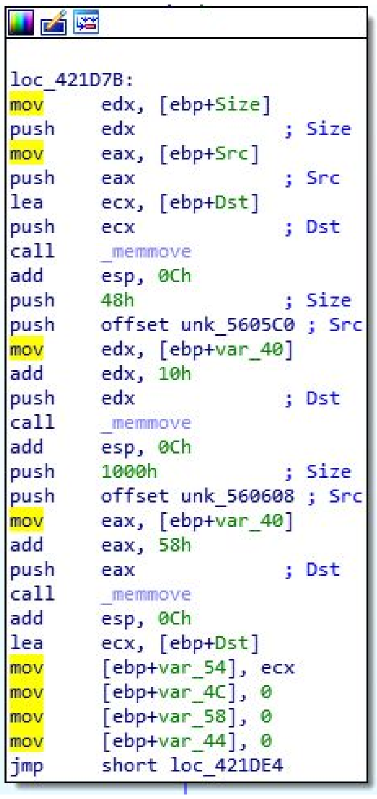
Blowfish - Block Cipher

Blowfish was developed as an alternative to DES, and is a symmetric-key block cipher. This means it will encrypt data in blocks, rather than stream ciphers such as RC4. The block-size for Blowfish is 8 bytes. Key lengths can be between 4 bytes, and 56 bytes. Blowfish is another fairly simple algorithm to determine, based on the required S-boxes and P-arrays. In this example we will be looking at the SendSafe spambot, recently shared with me by @avman1995. In the image below, you'll notice there is a function that will convert resource data from hex to raw bytes, and then some kind of stack string. After decoding the data from hex, we are able to determine it is encrypted, and so we can assume some decryption is going on here. Initially I notice the stack "string" being passed into a function alongside the value 16, a nulled DWORD, and some kind of uninitialized variable. Just underneath this another function is called, which accepts the raw data, size of raw data, and the **unk** variable used in the above function. At this point, I make the assumption there is some kind of key initialization, potentially with an IV (the nulled DWORD), and then a decryption routine, so it isn't a simple substitution function.

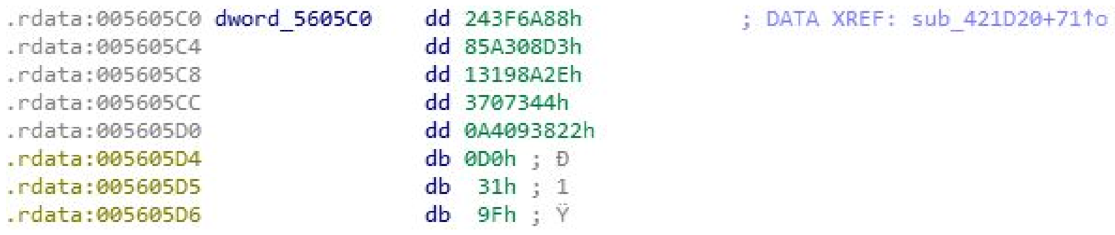


```
v16 = this;
strcpy(&hex_data, "127.0.0.1");
hResInfo = FindResourceA(0, (LPCSTR)0xE2, (LPCSTR)0xA);
hResData = LoadResource(0, hResInfo);
Source = (char *)LockResource(hResData);
strcpy(&hex_data, Source);
size_of_raw_data = convert_from_hex_to_raw_data(&raw_data, &hex_data, 256u);
if ( size_of_raw_data > 0 )
{
  top_of_stack_string = 7;
  v29 = 0x54;
  v30 = 8;
  v31 = 0xD5u;
  v32 = 0x67;
  v33 = 0x85u;
  v34 = 0x9Eu;
  v35 = 0xE2u;
  v36 = 0xAFu;
  v37 = 0xE3u;
  v38 = 0xC6u;
  v39 = 0xA9u;
  v40 = 0x8Fu;
  v41 = 0x3B;
  v42 = 0xC;
  v43 = 0x5B;
  nulled = 0;
  v5 = 0;
  sub_421D20(&unk, &top_of_stack_string, 16u, (int)&nulled);
  sub_4220E0(&unk, (int)&raw_data, size_of_raw_data, 0);
  strcpy(&hex_data, &raw_data);
}
```

Diving into the possible key initialization function, we can locate 2 mentions of addresses in memory, which could indicate the usage of substitution boxes or some other hardcoded value, which is exactly what we're looking for.
Sure enough, both of these offsets contain what seems to be substitution boxes, however only one returns useful information, where the other returns one link to a report in Chinese, which does not seem to contain the value (at least in the preview). The useful substitution box can be seen below, and it points us to several repositories containing code to encrypt and decrypt data using the Blowfish algorithm.

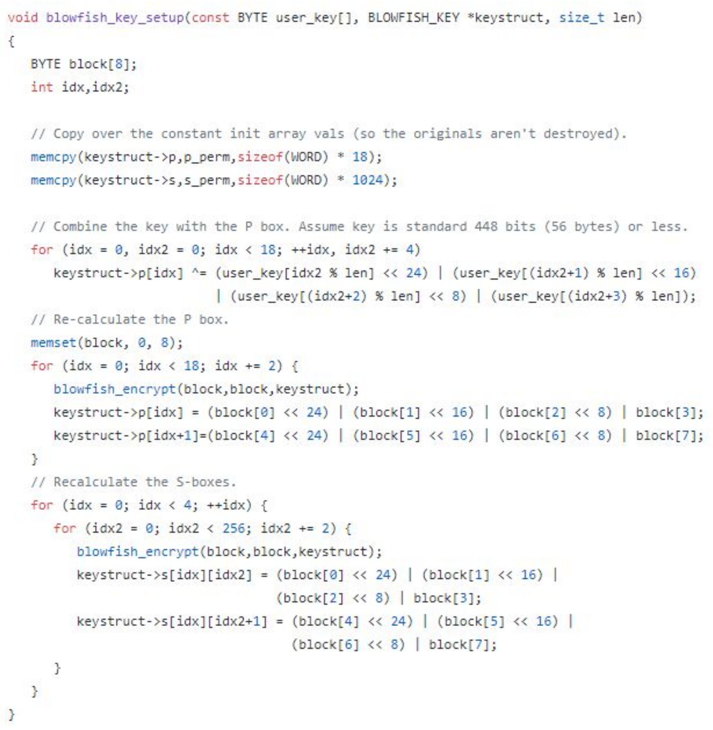


```
loc_421D7B:
mov     edx, [ebp+Size]
push    edx             ; Size
mov     eax, [ebp+Src]
push    eax             ; Src
lea     ecx, [ebp+Dst]
push    ecx             ; Dst
call    _memmove
add     esp, 0Ch
push    48h             ; Size
push    offset unk_5605C0 ; Src
mov     edx, [ebp+var_40]
add     edx, 10h
push    edx             ; Dst
call    _memmove
add     esp, 0Ch
push    1000h           ; Size
push    offset unk_560608 ; Src
mov     eax, [ebp+var_40]
add     eax, 58h
push    eax             ; Dst
call    _memmove
add     esp, 0Ch
lea     ecx, [ebp+Dst]
mov     [ebp+var_54], ecx
mov     [ebp+var_4C], 0
mov     [ebp+var_58], 0
mov     [ebp+var_44], 0
jmp     short loc_421DE4
```

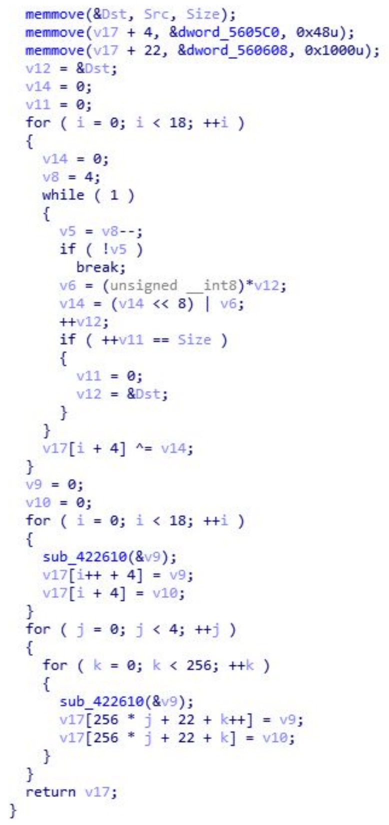


```
.rdata:005605C0 dword_5605C0    dd 243F6A88h            ; DATA XREF: sub_421D20+71↑o
.rdata:005605C4                 dd 85A308D3h
.rdata:005605C8                 dd 13198A2Eh
.rdata:005605CC                 dd 3707344h
.rdata:005605D0                 dd 0A4093822h
.rdata:005605D4                 db 0D0h ; Ð
.rdata:005605D5                 db  31h ; 1
.rdata:005605D6                 db  9Fh ; Ÿ
```

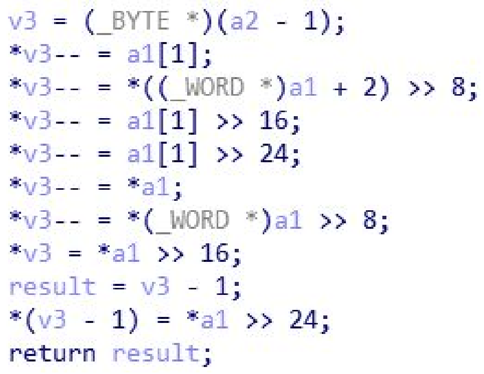
While none of the repositories match our code exactly, there are clear-cut similarities between this **repo** and the decompiled view. Similarities such as the 2 loops of 18, used to initialize the P-Array, as well as a loop that runs 4 times, with an internal loop that will run 256 times - this is used to initialize the S-Boxes.



```
void blowfish_key_setup(const BYTE user_key[], BLOWFISH_KEY *keystruct, size_t len)
{
   BYTE block[8];
   int idx,idx2;

   // Copy over the constant init array vals (so the originals aren't destroyed).
   memcpy(keystruct->p,p_perm,sizeof(WORD) * 18);
   memcpy(keystruct->s,s_perm,sizeof(WORD) * 1024);

   // Combine the key with the P box. Assume key is standard 448 bits (56 bytes) or less.
   for (idx = 0, idx2 = 0; idx < 18; ++idx, idx2 += 4)
      keystruct->p[idx] ^= (user_key[idx2 % len] << 24) | (user_key[(idx2+1) % len] << 16)
                           | (user_key[(idx2+2) % len] << 8) | (user_key[(idx2+3) % len]);
   // Re-calculate the P box.
   memset(block, 0, 8);
   for (idx = 0; idx < 18; idx += 2) {
      blowfish_encrypt(block,block,keystruct);
      keystruct->p[idx] = (block[0] << 24) | (block[1] << 16) | (block[2] << 8) | block[3];
      keystruct->p[idx+1]=(block[4] << 24) | (block[5] << 16) | (block[6] << 8) | block[7];
   }
   // Recalculate the S-boxes.
   for (idx = 0; idx < 4; ++idx) {
      for (idx2 = 0; idx2 < 256; idx2 += 2) {
         blowfish_encrypt(block,block,keystruct);
         keystruct->s[idx][idx2] = (block[0] << 24) | (block[1] << 16) |
                                   (block[2] << 8) | block[3];
         keystruct->s[idx][idx2+1] = (block[4] << 24) | (block[5] << 16) |
                                     (block[6] << 8) | block[7];

      }
   }
}
```

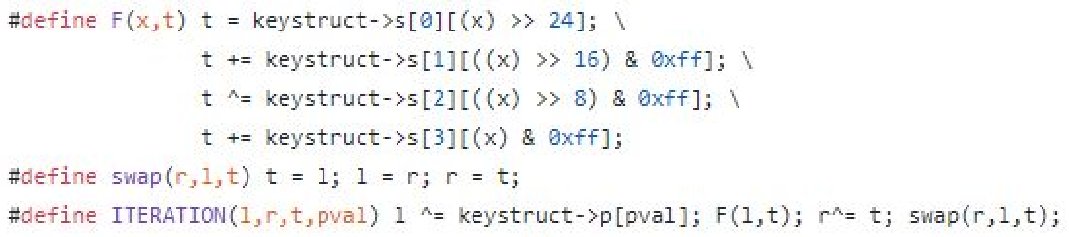


```
  memmove(&Dst, Src, Size);
  memmove(v17 + 4, &dword_5605C0, 0x48u);
  memmove(v17 + 22, &dword_560608, 0x1000u);
  v12 = &Dst;
  v14 = 0;
  v11 = 0;
  for ( i = 0; i < 18; ++i )
  {
    v14 = 0;
    v8 = 4;
    while ( 1 )
    {
      v5 = v8--;
      if ( !v5 )
        break;
      v6 = (unsigned __int8)*v12;
      v14 = (v14 << 8) | v6;
      ++v12;
      if ( ++v11 == Size )
      {
        v11 = 0;
        v12 = &Dst;
      }
    }
    v17[i + 4] ^= v14;
  }
  v9 = 0;
  v10 = 0;
  for ( i = 0; i < 18; ++i )
  {
    sub_422610(&v9);
    v17[i++ + 4] = v9;
    v17[i + 4] = v10;
  }
  for ( j = 0; j < 4; ++j )
  {
    for ( k = 0; k < 256; ++k )
    {
      sub_422610(&v9);
      v17[256 * j + 22 + k++] = v9;
      v17[256 * j + 22 + k] = v10;
    }
  }
  return v17;
}
```



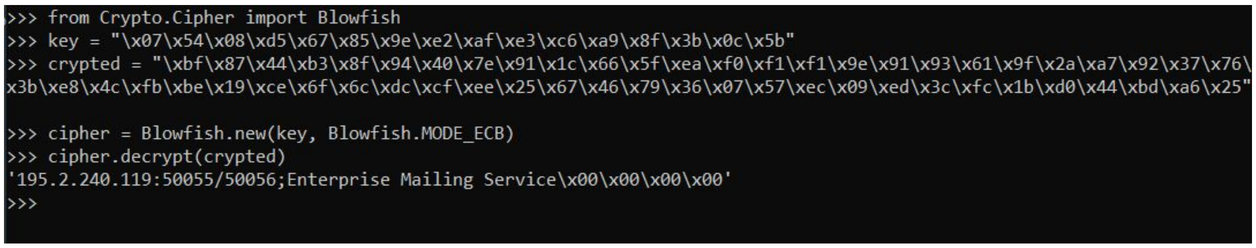
```
v3 = (_BYTE *)(a2 - 1);
*v3-- = a1[1];
*v3-- = *((_WORD *)a1 + 2) >> 8;
*v3-- = a1[1] >> 16;
*v3-- = a1[1] >> 24;
*v3-- = *a1;
*v3-- = *(_WORD *)a1 >> 8;
*v3 = *a1 >> 16;
result = v3 - 1;
*(v3 - 1) = *a1 >> 24;
return result;
```

That is just the key initialization routine however, jumping into the actual encryption algorithm we can also see similarities. For example, on the right is a decompilation of one of the routines called as part of the decryption phase, and below you can see source code that is fairly similar to the decompiled code. Due to the definitions of different variables, and how the code was compiled, it is difficult to get a good overview of the algorithm, however with everything we have discovered so far, we can safely say we are dealing with Blowfish. We have the key (the stack string), as well as a nulled IV, so we can go ahead and decrypt the data using something like CyberChef - though as that only supports 8 byte keys, we can use PyCryptoDome to decrypt the data.



```
#define F(x,t) t = keystruct->s[0][(x) >> 24]; \
               t += keystruct->s[1][((x) >> 16) & 0xff]; \
               t ^= keystruct->s[2][((x) >> 8) & 0xff]; \
               t += keystruct->s[3][(x) & 0xff];
#define swap(r,l,t) t = l; l = r; r = t;
#define ITERATION(l,r,t,pval) l ^= keystruct->p[pval]; F(l,t); r^= t; swap(r,l,t);
```

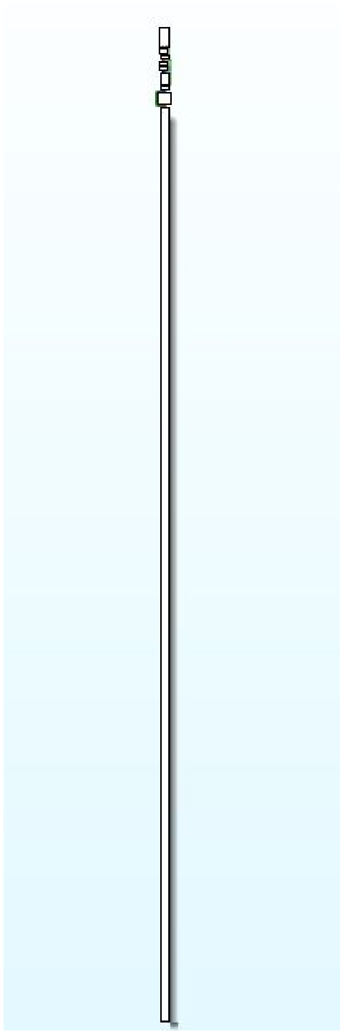
We have the key (the stack string), as well as a nulled IV, so we can go ahead and decrypt the data using something like CyberChef - though as that only supports 8 byte keys, we can use PyCryptoDome to decrypt the data. As the IV is nulled, we can assume it is using the ECB mode of encryption - though we can easily swap over to CBC if that fails, but as you can see, it was successful.

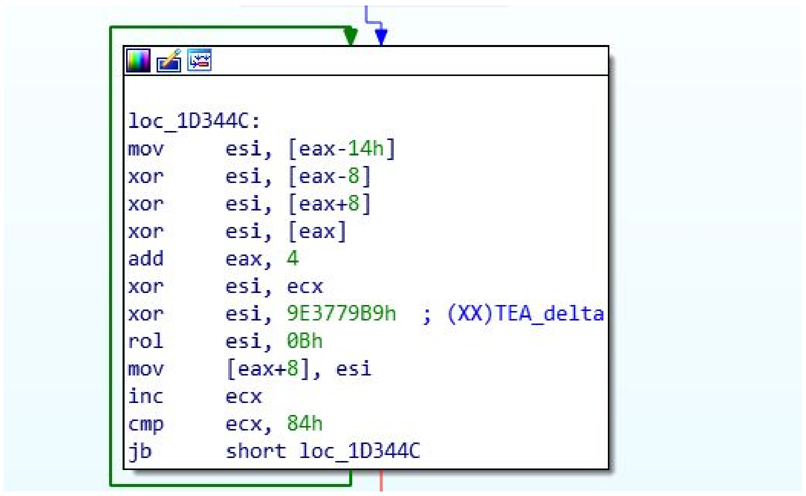


```
>>> from Crypto.Cipher import Blowfish
>>> key = "\x07\x54\x08\xd5\x67\x85\x9e\xe2\xaf\xe3\xc6\xa9\x8f\x3b\x0c\x5b"
>>> crypted = "\xbf\x87\x44\xb3\x8f\x94\x40\x7e\x91\x1c\x66\x5f\xea\xf0\xf1\xf1\x9e\x91\x93\x61\x9f\x2a\xa7\x92\x37\x76\
x3b\xe8\x4c\xfb\xbe\x19\xce\x6f\x6c\xdc\xcf\xee\x25\x67\x46\x79\x36\x07\x57\xec\x09\xed\x3c\xfc\x1b\xd0\x44\xbd\xa6\x25"

>>> cipher = Blowfish.new(key, Blowfish.MODE_ECB)
>>> cipher.decrypt(crypted)
'195.2.240.119:50055/50056;Enterprise Mailing Service\x00\x00\x00\x00'
>>>
```

Serpent - Block Cipher

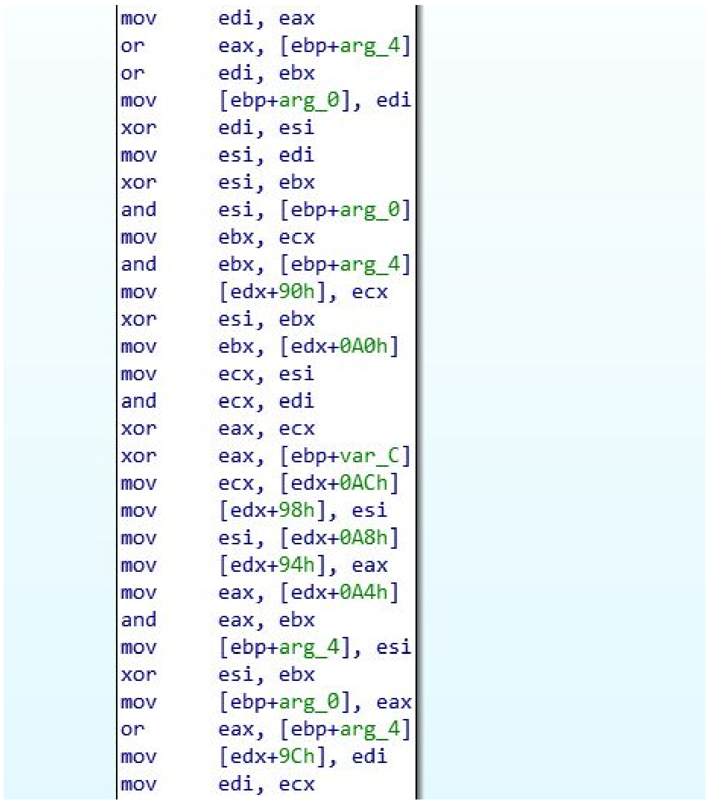
The Serpent algorithm is a very interesting algorithm, especially when it comes to recognizing it. There are very few, if any, constants in this algorithm, unless you consider the structure a constant. The easiest way to recognize this cipher in my opinion is through the sheer scale of it, and how there are no jumps at all in the main code blocks.
The Serpent encryption algorithm is another symmetric key block cipher, that encrypts/decrypts data in chunks of 16, and supports 128, 192, and 256 bit key lengths. It was developed as a candidate for the Advanced-Encryption-Standard.
Serpent is known to be used in ISFB/Gozi, to encrypt packets sent to and received from the C2 server. As mentioned previously, one of the key identifiers of the Serpent algorithm (especially in ISFB), is the lack of jumps in the key initialization function and the encryption function itself. Using Find-Crypt on a sample of ISFB will return one result for an encryption algorithm, which is the Tiny-Encryption-Algorithm. This is not unique to Serpent, and so it can lead to some confusion as to what you are dealing with.





```
loc_1D344C:
mov     esi, [eax-14h]
xor     esi, [eax-8]
xor     esi, [eax+8]
xor     esi, [eax]
add     eax, 4
xor     esi, ecx
xor     esi, 9E3779B9h  ; (XX)TEA_delta
rol     esi, 0Bh
mov     [eax+8], esi
inc     ecx
cmp     ecx, 84h
jb      short loc_1D344C
```

When you take a look at the actual byte operations of the algorithm, you'll notice they are mainly **XOR**, **AND**, **MOV**, and **OR** instructions. There is nothing really unique in the blocks of code aside from the feature that it is all one block of code, rather than something like AES which has several loops, different code execution jumps, etc. Plus the fact it uses a constant found in the TEA algorithm, yet it is extremely large, is a definite giveaway that the Serpent algorithm is being used.



```
mov     edi, eax
or      eax, [ebp+arg_4]
or      edi, ebx
mov     [ebp+arg_0], edi
xor     edi, esi
mov     esi, edi
xor     esi, ebx
and     esi, [ebp+arg_0]
mov     ebx, ecx
and     ebx, [ebp+arg_4]
mov     [edx+90h], ecx
xor     esi, ebx
mov     ebx, [edx+0A0h]
mov     ecx, esi
and     ecx, edi
xor     eax, ecx
xor     eax, [ebp+var_C]
mov     ecx, [edx+0ACh]
mov     [edx+98h], esi
mov     esi, [edx+0A8h]
mov     [edx+94h], eax
mov     eax, [edx+0A4h]
and     eax, ebx
mov     [ebp+arg_4], esi
xor     esi, ebx
mov     [ebp+arg_0], eax
or      eax, [ebp+arg_4]
mov     [edx+9Ch], edi
mov     edi, ecx
```

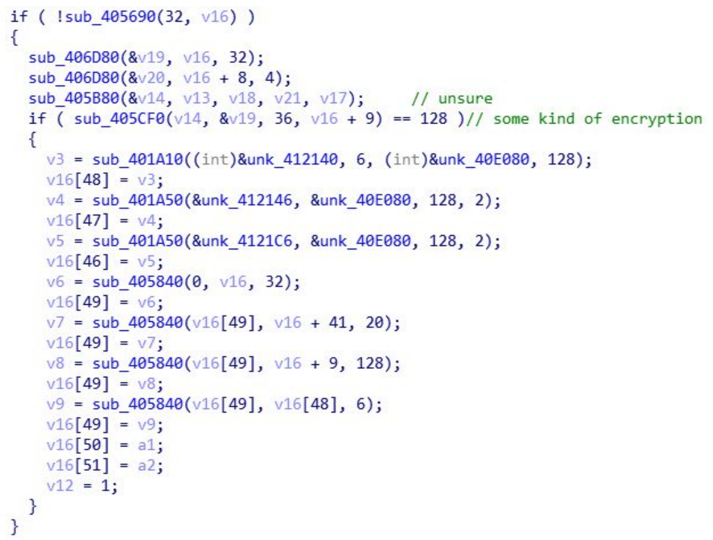
Furthermore, Serpent supports both ECB and CBC for encryption. In the case of ISFB, Serpent-CBC is used, with an IV set to 16 null bytes.

Asymmetric Algorithms

RSA

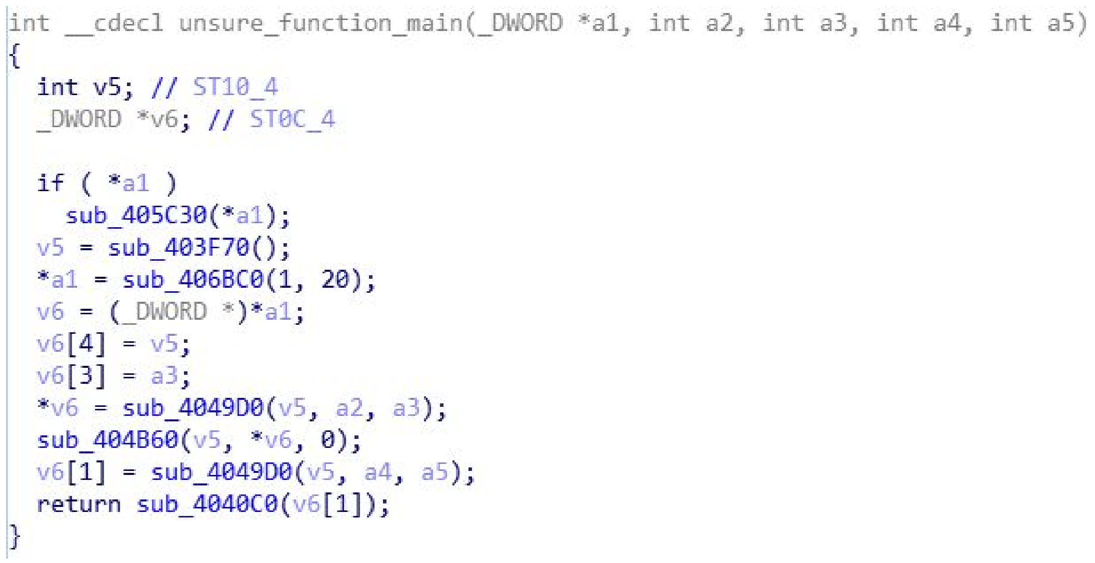
The RSA algorithm is a very popular asymmetric key encryption algorithm, with a public and private key pair. The public key is commonly used to encrypt data, and then only the private key is able to decrypt the data. This makes it a common choice for ransomware, as it allows attackers to make files only decryptable with their private key. While RSA is not used to encrypt the file itself, it is used to encrypt a symmetric key that is used to encrypt the file - algorithms such as AES and Salsa20 are common choices for ransomware. Dridex also uses RSA in a similar fashion, in order to encrypt the RC4 key used to encrypt a network packet.
As with other algorithms, there are 2 main methods of utilizing them: WinCrypt API, and code implementations. WinCrypt API is a favourite for a lot of malware authors as it is simple, reliable, and secure, whereas implementing algorithms yourself takes time, probably aren't as secure, or reliable. As a result, most of the implemented algorithms you will come across will be direct copies of open source software containing that algorithm - like the one we will be investigating now, inside the Dharma ransomware.

In this case we will assume we have located a function that seems to generate or load some data into memory, which is encrypted in the following function. We can link it to what seems to be another encryption algorithm linked to the main part of the ransomware, so at this point I assume I'm dealing with some kind of key generation/loading function. Other than that we don't have much to go on, so we might as well jump into the function!



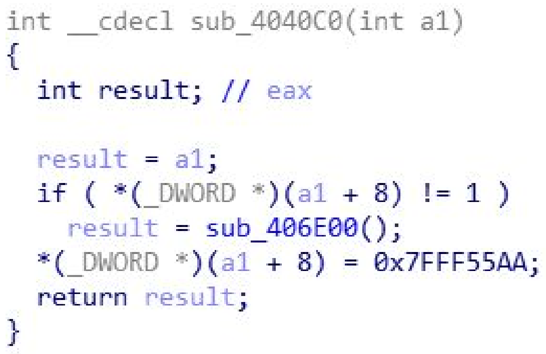
```
if ( !sub_405690(32, v16) )
{
  sub_406D80(&v19, v16, 32);
  sub_406D80(&v20, v16 + 8, 4);
  sub_405B80(&v14, v13, v18, v21, v17);      // unsure
  if ( sub_405CF0(v14, &v19, 36, v16 + 9) == 128 )// some kind of encryption
  {
    v3 = sub_401A10((int)&unk_412140, 6, (int)&unk_40E080, 128);
    v16[48] = v3;
    v4 = sub_401A50(&unk_412146, &unk_40E080, 128, 2);
    v16[47] = v4;
    v5 = sub_401A50(&unk_4121C6, &unk_40E080, 128, 2);
    v16[46] = v5;
    v6 = sub_405840(0, v16, 32);
    v16[49] = v6;
    v7 = sub_405840(v16[49], v16 + 41, 20);
    v16[49] = v7;
    v8 = sub_405840(v16[49], v16 + 9, 128);
    v16[49] = v8;
    v9 = sub_405840(v16[49], v16[48], 6);
    v16[49] = v9;
    v16[50] = a1;
    v16[51] = a2;
    v12 = 1;
  }
}
```

Diving into the function, we can see there isn't much to offer. Some kind of struct is present, along with calls to other subroutines. As a result, we need to start looking through these functions to see if there is anything we can use to narrow down what we're looking at.



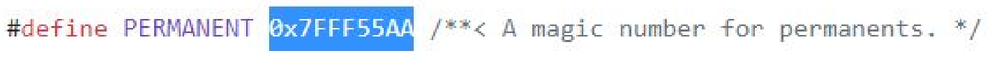
```
int __cdecl unsure_function_main(_DWORD *a1, int a2, int a3, int a4, int a5)
{
  int v5; // ST10_4
  _DWORD *v6; // ST0C_4

  if ( *a1 )
    sub_405C30(*a1);
  v5 = sub_403F70();
  *a1 = sub_406BC0(1, 20);
  v6 = (_DWORD *)*a1;
  v6[4] = v5;
  v6[3] = a3;
  *v6 = sub_4049D0(v5, a2, a3);
  sub_404B60(v5, *v6, 0);
  v6[1] = sub_4049D0(v5, a4, a5);
  return sub_4040C0(v6[1]);
}
```

Sure enough, looking at the very last function (we can see that the function will return whatever is returned from that function, so it might be useful to check out first), we can find an interesting value that might be a constant of some kind: **0x7FFF55AA**. This gives us something to base our research on.

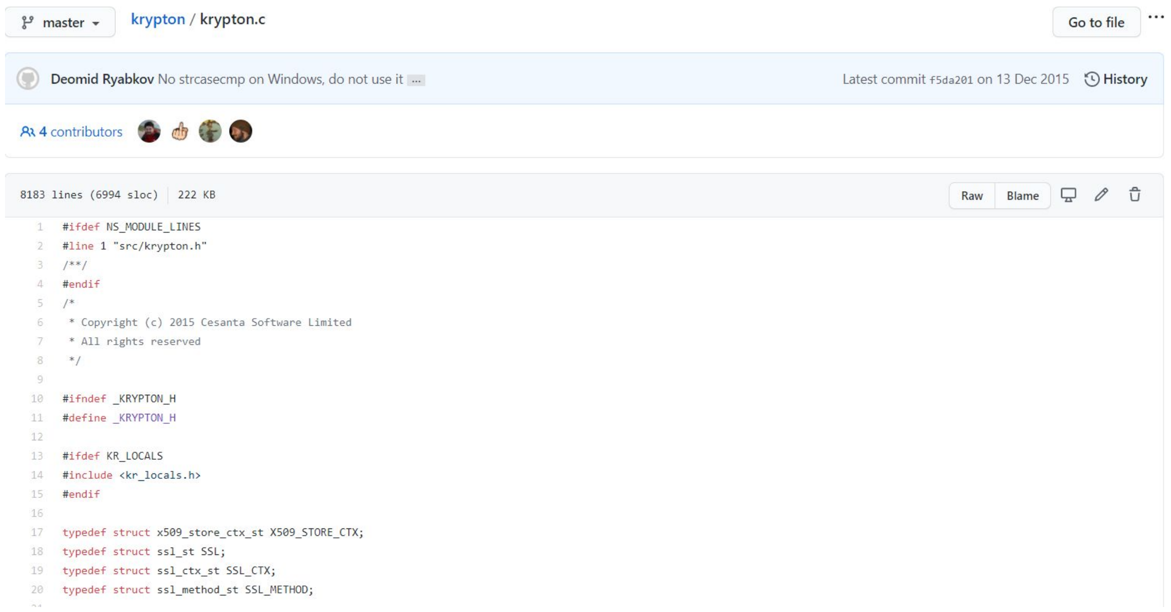


```
int __cdecl sub_4040C0(int a1)
{
  int result; // eax

  result = a1;
  if ( *(_DWORD *)(a1 + 8) != 1 )
    result = sub_406E00();
  *(_DWORD *)(a1 + 8) = 0x7FFF55AA;
  return result;
}
```

Sure enough, after a quick search for the constant, we can discover a library called **Krypton**, which contains a reference to this hex value under the name **PERMANENT**.

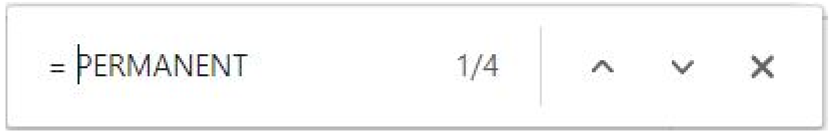


```
#define PERMANENT 0x7FFF55AA /**< A magic number for permanents. */
```

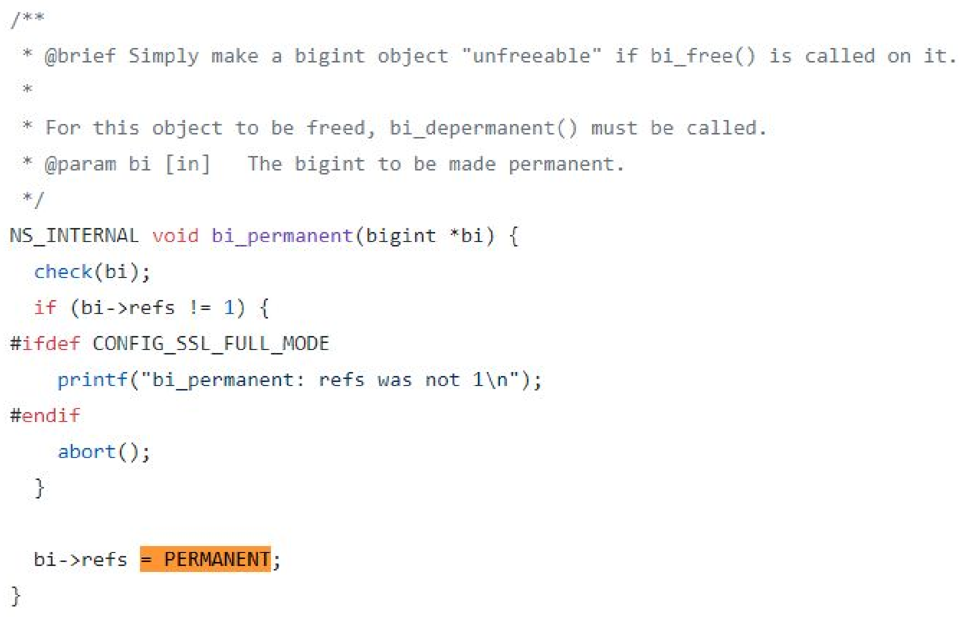

master
krypton / krypton.c
Go to file
Deomid Ryabkov No strcasecmp on Windows, do not use it
Latest commit f5da201 on 13 Dec 2015
History
4 contributors
8183 lines (6994 sloc) 222 KB
Raw
Blame

```c
#ifdef NS_MODULE_LINES
#line 1 "src/krypton.h"
/**/
#endif
/*
 * Copyright (c) 2015 Cesanta Software Limited
 * All rights reserved
 */

#ifndef _KRYPTON_H
#define _KRYPTON_H

#ifdef KR_LOCALS
#include <kr_locals.h>
#endif

typedef struct x509_store_ctx_st X509_STORE_CTX;
typedef struct ssl_st SSL;
typedef struct ssl_ctx_st SSL_CTX;
typedef struct ssl_method_st SSL_METHOD;
```

This specific library contains multiple algorithms, so we will need to narrow down our searching. Using one of the greatest tools available to you on a web browser (CTRL+F), we can search for the string PERMANENT, which yields 44 results. However, if you remember the code we looked at, it was assigning (a1 + 8) the hex value we discovered, and by simply adding the **=** sign to our query, we can narrow our results down to just 4.

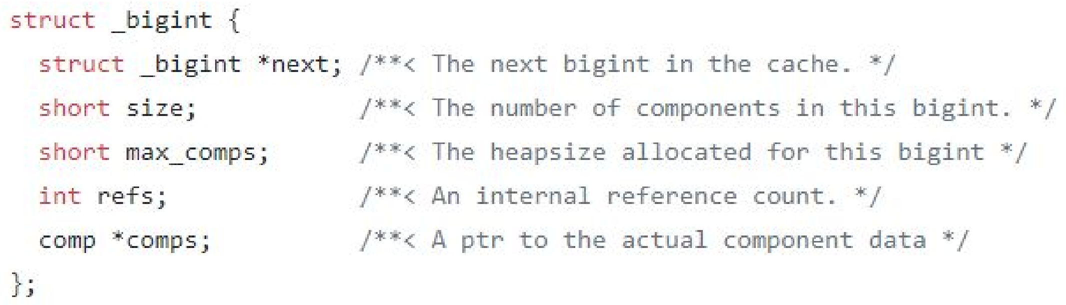


Now it's a case of checking these 4 results to see if any are in a function that looks remotely like the decompiled function. Sure enough, we can find the value PERMANENT being moved into **bi->refs** if **bi->refs** doesn't equal 1, which is pretty much an exact match to our decompiled function. So, now the plan is to locate the **bigint** struct layout, add it into IDA, and clean up our function so that it resembles the source code!



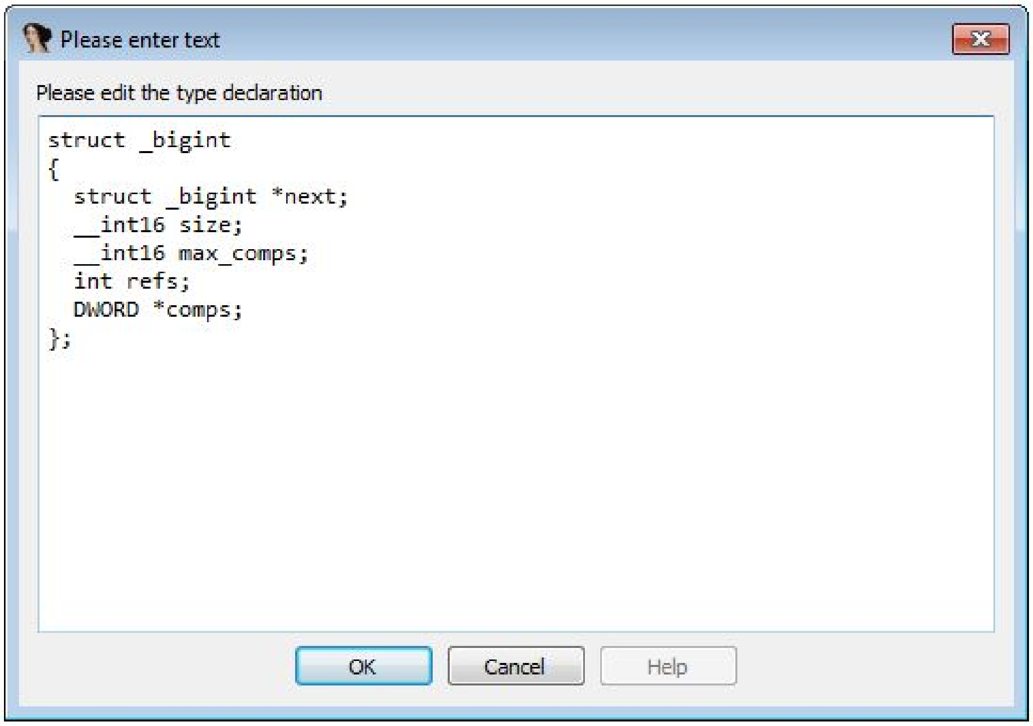
```
/**
 * @brief Simply make a bigint object "unfreeable" if bi_free() is called on it.
 *
 * For this object to be freed, bi_depermanent() must be called.
 * @param bi [in]   The bigint to be made permanent.
 */
NS_INTERNAL void bi_permanent(bigint *bi) {
  check(bi);
  if (bi->refs != 1) {
#ifdef CONFIG_SSL_FULL_MODE
    printf("bi_permanent: refs was not 1\n");
#endif
    abort();
  }

  bi->refs = PERMANENT;
}
```

As we have access to the source it is very simple to locate the structure. As you can see below, it is nicely commented for us in case we wanted to dive in deeper. This lines up with what we can see in IDA too. The struct pointer is most likely the size of a **DWORD**, and then there are 2 **short** variables, which take up 4 bytes in total. **(bigint + 8)** would therefore point to the **refs** variable.



```
struct _bigint {
  struct _bigint *next; /**< The next bigint in the cache. */
  short size;           /**< The number of components in this bigint. */
  short max_comps;      /**< The heapsize allocated for this bigint */
  int refs;             /**< An internal reference count. */
  comp *comps;          /**< A ptr to the actual component data */
};
```

Opening the **Local Types** window in IDA, we can add a new local type and create the struct that you can see in the image below. In my case, **short** was automatically converted to **__int16**, and I set the **comps** variable to a **DWORD** as that is what it is defined as.

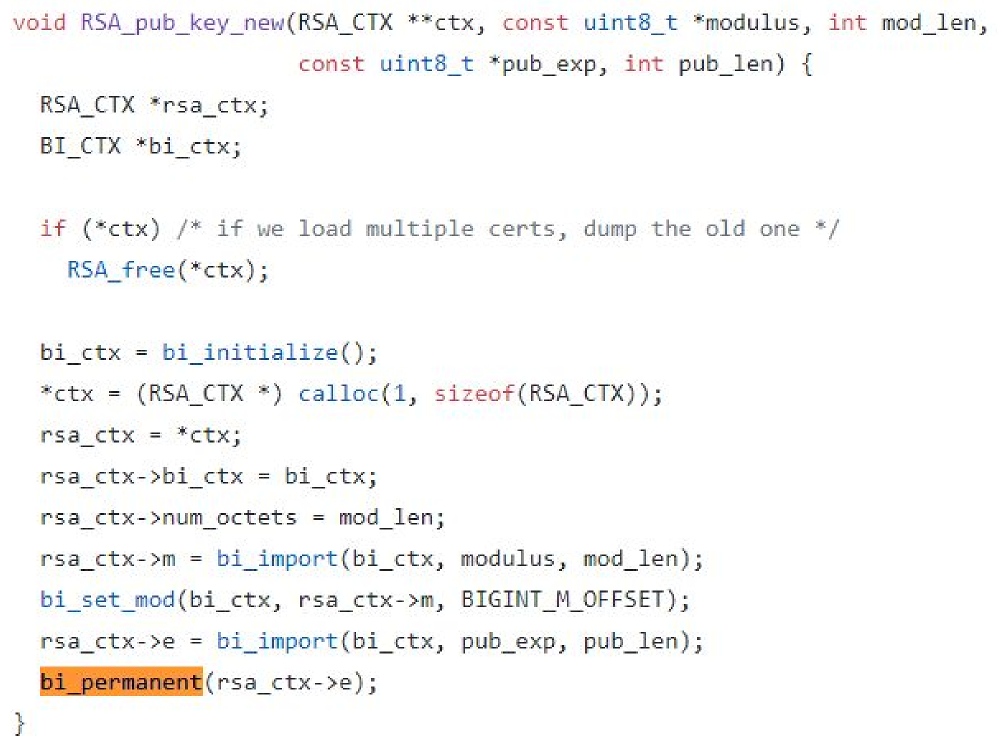


```
struct _bigint
{
  struct _bigint *next;
  __int16 size;
  __int16 max_comps;
  int refs;
  DWORD *comps;
};
```

After setting up the struct, we can declare that **a1** is a **_bigint** structure, and IDA will automatically clear up the function and display it almost identically to the source code implementation - we don't see the *#ifdef* comments as that is removed upon compilation. While we know what it is, we still don't know what it does, so time to work backwards from this!

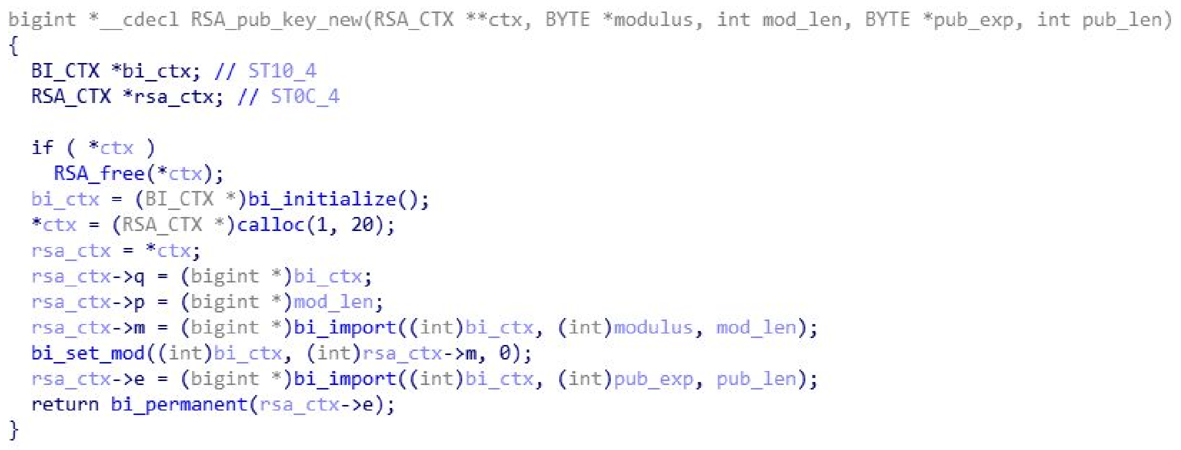
```
_bigint *__cdecl bi_permanent(_bigint *bi)
{
  _bigint *result; // eax

  result = bi;
  if ( bi->refs != 1 )
    result = (_bigint *)abort();
  bi->refs = 0x7FFF55AA;
  return result;
}
```

Remember the decompiled function we found **bi_permanent()** in? Well we know it was only called once in that function, so we can jump back to GitHub and search for instances the function is called. Soon enough, we can locate a function it is called once in, which is eerily similar to the function we discovered in IDA. All we need to do now is add some more local types (**RSA_CTX**), rename some functions, and we should have an almost identical function!



```
void RSA_pub_key_new(RSA_CTX **ctx, const uint8_t *modulus, int mod_len,
                     const uint8_t *pub_exp, int pub_len) {
  RSA_CTX *rsa_ctx;
  BI_CTX *bi_ctx;

  if (*ctx) /* if we load multiple certs, dump the old one */
    RSA_free(*ctx);

  bi_ctx = bi_initialize();
  *ctx = (RSA_CTX *) calloc(1, sizeof(RSA_CTX));
  rsa_ctx = *ctx;
  rsa_ctx->bi_ctx = bi_ctx;
  rsa_ctx->num_octets = mod_len;
  rsa_ctx->m = bi_import(bi_ctx, modulus, mod_len);
  bi_set_mod(bi_ctx, rsa_ctx->m, BIGINT_M_OFFSET);
  rsa_ctx->e = bi_import(bi_ctx, pub_exp, pub_len);
  bi_permanent(rsa_ctx->e);
}
```

After adding the local types, we can clean up the function resulting in the image you see below. While it is not exact due to some issues with the created types (potentially sizing issues), it is clear to see the similarity between the decompiled function and the source code.



```
bigint *__cdecl RSA_pub_key_new(RSA_CTX **ctx, BYTE *modulus, int mod_len, BYTE *pub_exp, int pub_len)
{
  BI_CTX *bi_ctx; // ST10_4
  RSA_CTX *rsa_ctx; // ST0C_4

  if ( *ctx )
    RSA_free(*ctx);
  bi_ctx = (BI_CTX *)bi_initialize();
  *ctx = (RSA_CTX *)calloc(1, 20);
  rsa_ctx = *ctx;
  rsa_ctx->q = (bigint *)bi_ctx;
  rsa_ctx->p = (bigint *)mod_len;
  rsa_ctx->m = (bigint *)bi_import((int)bi_ctx, (int)modulus, mod_len);
  bi_set_mod((int)bi_ctx, (int)rsa_ctx->m, 0);
  rsa_ctx->e = (bigint *)bi_import((int)bi_ctx, (int)pub_exp, pub_len);
  return bi_permanent(rsa_ctx->e);
}
```

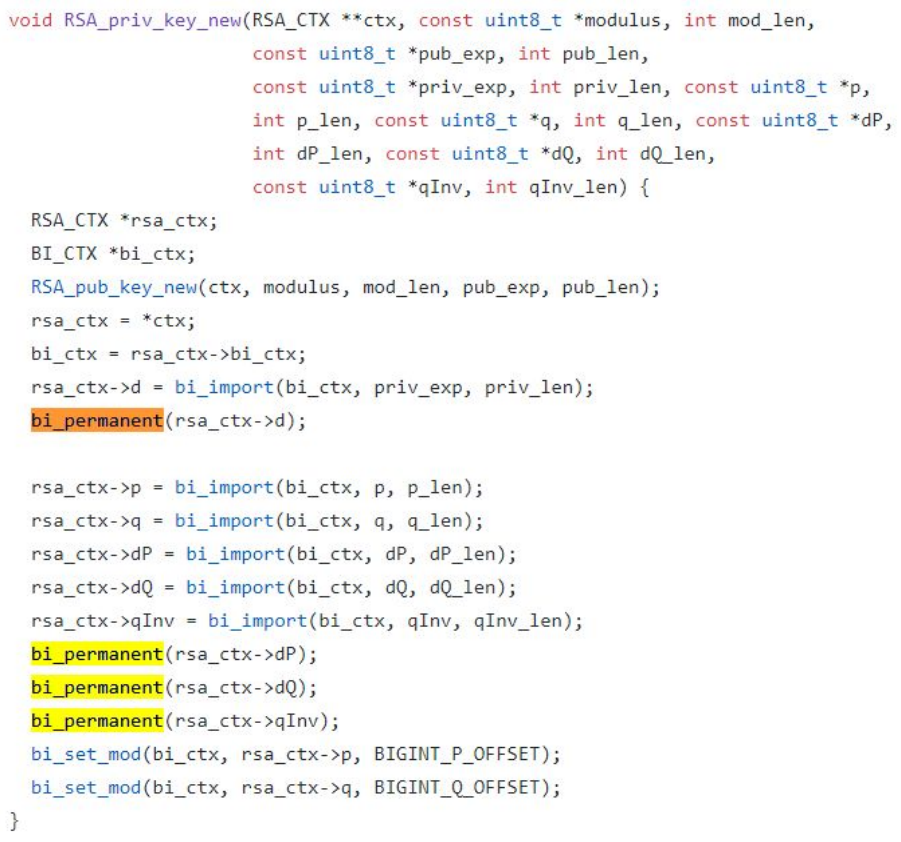
If you want to replicate this by yourself, I've added the local type structures below so you can add them directly into IDA, and hopefully speed up your analysis of this ransomware!

```
struct bigint
{
   struct bigint *next;
   __int16 size;
   __int16 max_comps;
   int refs;
   DWORD *comps;
};
```

```
struct BI_CTX
{
   _bigint *active_list;
   _bigint *free_list;
   _bigint *bi_radix;
   _bigint *bi_mod[3];
   _bigint *bi_normalised_mod[3];
   _bigint **g;
   int window;
   int active_count;
   int free_count;
   BYTE mod_offset;
};
```

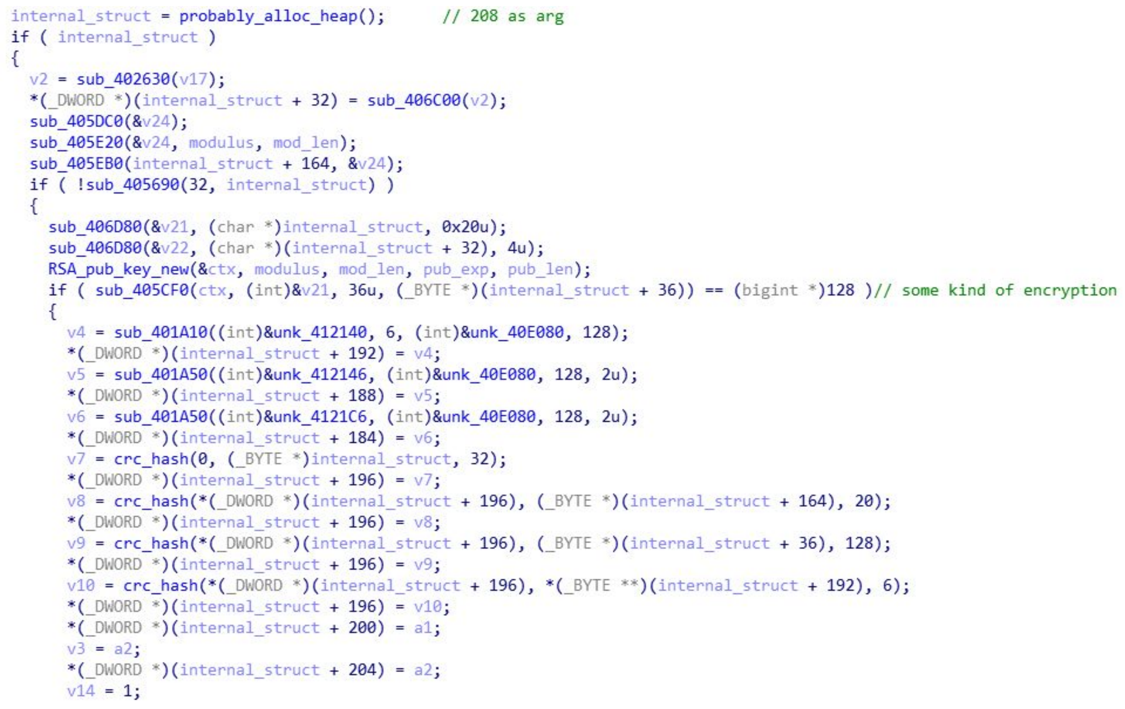
```
struct RSA_CTX
{
   _bigint *m;
   _bigint *e;
   _bigint *d;
   _bigint *p;
   _bigint *q;
   _bigint *dP;
   _bigint *dQ;
   _bigint *qInv;
   int num_octets;
   BI_CTX *bi_ctx;
};
```

We are also able to find the **bi_permanent()** function being used in a **RSA_priv_key_new()** function, however none of the cross references in IDA look similar to the function, so it is most likely used in the decryptor, rather than the ransomware itself.



```
void RSA_priv_key_new(RSA_CTX **ctx, const uint8_t *modulus, int mod_len,
                      const uint8_t *pub_exp, int pub_len,
                      const uint8_t *priv_exp, int priv_len, const uint8_t *p,
                      int p_len, const uint8_t *q, int q_len, const uint8_t *dP,
                      int dP_len, const uint8_t *dQ, int dQ_len,
                      const uint8_t *qInv, int qInv_len) {
  RSA_CTX *rsa_ctx;
  BI_CTX *bi_ctx;
  RSA_pub_key_new(ctx, modulus, mod_len, pub_exp, pub_len);
  rsa_ctx = *ctx;
  bi_ctx = rsa_ctx->bi_ctx;
  rsa_ctx->d = bi_import(bi_ctx, priv_exp, priv_len);
  bi_permanent(rsa_ctx->d);

  rsa_ctx->p = bi_import(bi_ctx, p, p_len);
  rsa_ctx->q = bi_import(bi_ctx, q, q_len);
  rsa_ctx->dP = bi_import(bi_ctx, dP, dP_len);
  rsa_ctx->dQ = bi_import(bi_ctx, dQ, dQ_len);
  rsa_ctx->qInv = bi_import(bi_ctx, qInv, qInv_len);
  bi_permanent(rsa_ctx->dP);
  bi_permanent(rsa_ctx->dQ);
  bi_permanent(rsa_ctx->qInv);
  bi_set_mod(bi_ctx, rsa_ctx->p, BIGINT_P_OFFSET);
  bi_set_mod(bi_ctx, rsa_ctx->q, BIGINT_Q_OFFSET);
}
```

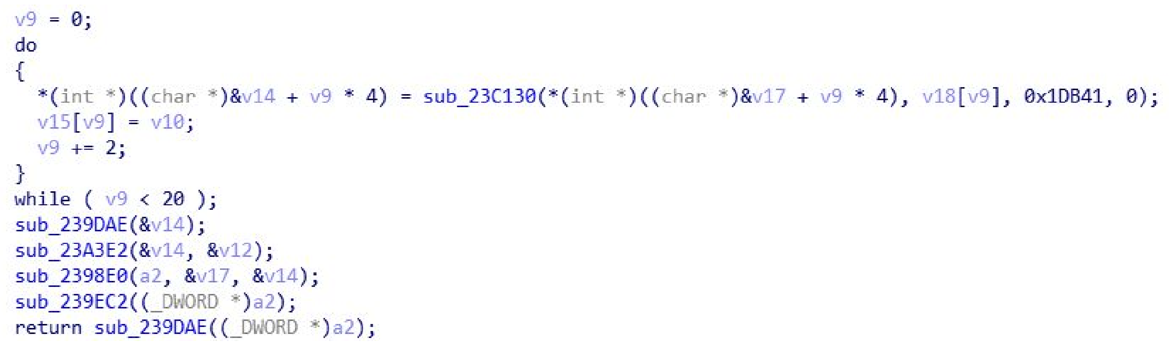
Backing out of the **RSA_pub_key_new()** function, we return to the initial function we looked at. Renaming some variables as we now know the arguments for the public key loading function, we can see that the variable **ctx** is passed into the encryption function, along with a variable that seems to be an internal structure. Without diving into it too deeply, it seems like the RSA_CTX structure is encrypted, and stored in internal_struct->offset_36, which we could set up as a local type and name **public_key**. At this point we know RSA is used, a public key is loaded, and potentially used later on. We can now continue with static analysis to narrow down where the key is used, or jump to dynamic analysis to be a lot more efficient in discovering what is going on.



```
internal_struct = probably_alloc_heap();      // 208 as arg
if ( internal_struct )
{
  v2 = sub_402630(v17);
  *(_DWORD *)(internal_struct + 32) = sub_406C00(v2);
  sub_405DC0(&v24);
  sub_405E20(&v24, modulus, mod_len);
  sub_405EB0(internal_struct + 164, &v24);
  if ( !sub_405690(32, internal_struct) )
  {
    sub_406D80(&v21, (char *)internal_struct, 0x20u);
    sub_406D80(&v22, (char *)(internal_struct + 32), 4u);
    RSA_pub_key_new(&ctx, modulus, mod_len, pub_exp, pub_len);
    if ( sub_405CF0(ctx, (int)&v21, 36u, (_BYTE *)(internal_struct + 36)) == (bigint *)128 )// some kind of encryption
    {
      v4 = sub_401A10((int)&unk_412140, 6, (int)&unk_40E080, 128);
      *(_DWORD *)(internal_struct + 192) = v4;
      v5 = sub_401A50((int)&unk_412146, (int)&unk_40E080, 128, 2u);
      *(_DWORD *)(internal_struct + 188) = v5;
      v6 = sub_401A50((int)&unk_4121C6, (int)&unk_40E080, 128, 2u);
      *(_DWORD *)(internal_struct + 184) = v6;
      v7 = crc_hash(0, (_BYTE *)internal_struct, 32);
      *(_DWORD *)(internal_struct + 196) = v7;
      v8 = crc_hash(*(_DWORD *)(internal_struct + 196), (_BYTE *)(internal_struct + 164), 20);
      *(_DWORD *)(internal_struct + 196) = v8;
      v9 = crc_hash(*(_DWORD *)(internal_struct + 196), (_BYTE *)(internal_struct + 36), 128);
      *(_DWORD *)(internal_struct + 196) = v9;
      v10 = crc_hash(*(_DWORD *)(internal_struct + 196), *(_BYTE **)(internal_struct + 192), 6);
      *(_DWORD *)(internal_struct + 196) = v10;
      *(_DWORD *)(internal_struct + 200) = a1;
      v3 = a2;
      *(_DWORD *)(internal_struct + 204) = a2;
      v14 = 1;
```

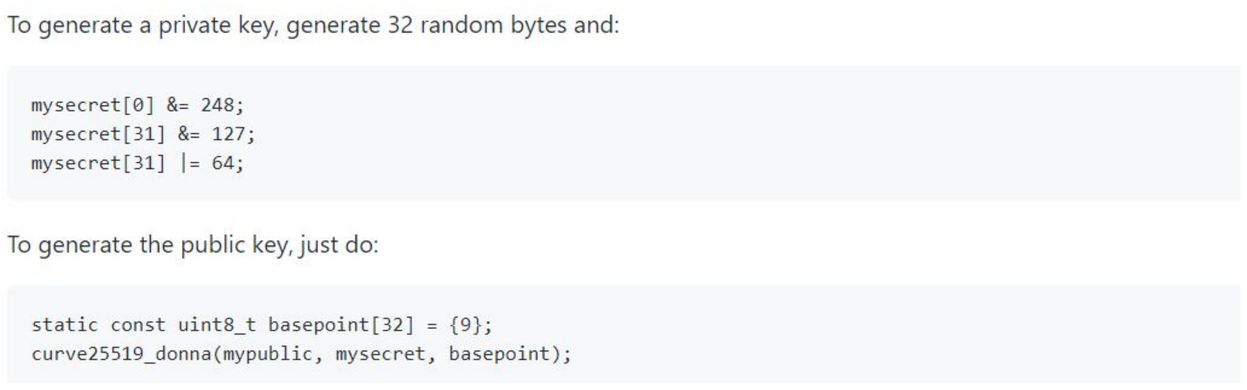
ECC

Elliptic-Curve Cryptography is a similar approach to public-key encryption, based on the elliptic curve theory. It can be used to generate faster, smaller, and more efficient keys through an elliptic curve equation, rather than dealing with prime numbers like the RSA algorithm does. ECC is typically a lot harder to crack than RSA, which could be why more ransomware actors are turning to ECC as a replacement for RSA.
There are multiple different variants of ECC, including something called Elliptic-curve Diffie–Hellman (ECDH) which allows the creation of a shared key based on a public key from one user, and a private key from another. This can be used to encrypt data alongside a symmetric algorithm, such as Salsa20. Once again, we will be looking at REvil as an example of this. As a shared key can be created from a public/private key pair, an attacker can provide a public key, and a private key can be generated on the users system, making it perfect for secure ransomware.
Unlike AES and Salsa20, it is usually very difficult to determine what ECDH algorithm is being used, and requires a lot of searching for possible constants. However, as ECDH algorithms are extremely difficult to write correctly, if a sample of malware is using an algorithm and you can determine which algorithm it is using, you will usually be able to find some open source code the attackers had taken it from. In the case of REvil, we can locate an interesting value that only appears once in the entire binary: **0x1DB41**. In this case it is usually easier to search for it as an integer rather than a hex value, as it is more likely to appear in some source code.



```
v9 = 0;
do
{
  *(int *)((char *)&v14 + v9 * 4) = sub_23C130(*(int *)((char *)&v17 + v9 * 4), v18[v9], 0x1DB41, 0);
  v15[v9] = v10;
  v9 += 2;
}
while ( v9 < 20 );
sub_239DAE(&v14);
sub_23A3E2(&v14, &v12);
sub_2398E0(a2, &v17, &v14);
sub_239EC2((_DWORD *)a2);
return sub_239DAE((_DWORD *)a2);
```

Searching for this value, we can find different mentions to **Curve25519**. Searching for source code implementations of this curve algorithm leads us to **Curve25519-donna**, which luckily has a **Github Repository** with some instructions on how to use it. Specifically, how to generate a private key and public key. Take note of the fact a private key can be 32 random bytes, and that there is no need for a specific format unlike RSA.



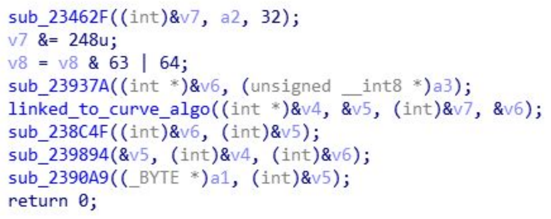
To generate a private key, generate 32 random bytes and:

```
mysecret[0] &= 248;
mysecret[31] &= 127;
mysecret[31] |= 64;
```

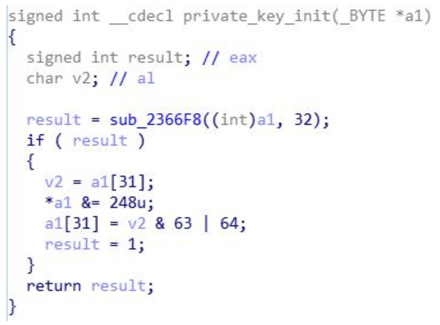
To generate the public key, just do:

```
static const uint8_t basepoint[32] = {9};
curve25519_donna(mypublic, mysecret, basepoint);
```

Luckily, we are able to search for the value 248 in the code, and find an **AND** operation utilizing it. From there, we can make our way back through the code, eventually pointing us to the private key initialization function, and the subroutine responsible for generating the public key.

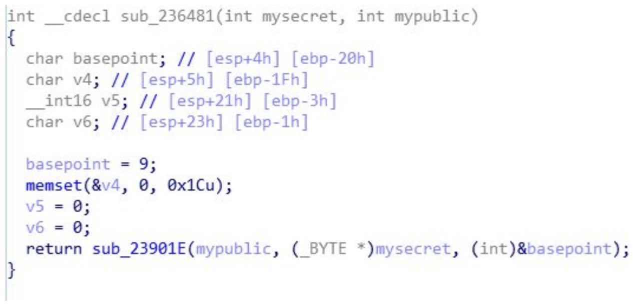


```
sub_23462F((int)&v7, a2, 32);
v7 &= 248u;
v8 = v8 & 63 | 64;
sub_23937A((int *)&v6, (unsigned __int8 *)a3);
linked_to_curve_algo((int *)&v4, &v5, (int)&v7, &v6);
sub_238C4F((int)&v6, (int)&v5);
sub_239894(&v5, (int)&v4, (int)&v6);
sub_2390A9((_BYTE *)a1, (int)&v5);
return 0;
```



```
signed int __cdecl private_key_init(_BYTE *a1)
{
  signed int result; // eax
  char v2; // al

  result = sub_2366F8((int)a1, 32);
  if ( result )
  {
    v2 = a1[31];
    *a1 &= 248u;
    a1[31] = v2 & 63 | 64;
    result = 1;
  }
  return result;
}
```

In the case of REvil, and probably many other ransomware families, the code is a direct copy from the Github we discovered, allowing us to correctly comment functions to get a great overview on how key generation, file encryption, and any other operations work.



```
int __cdecl sub_236481(int mysecret, int mypublic)
{
  char basepoint; // [esp+4h] [ebp-20h]
  char v4; // [esp+5h] [ebp-1Fh]
  __int16 v5; // [esp+21h] [ebp-3h]
  char v6; // [esp+23h] [ebp-1h]

  basepoint = 9;
  memset(&v4, 0, 0x1Cu);
  v5 = 0;
  v6 = 0;
  return sub_23901E(mypublic, (_BYTE *)mysecret, (int)&basepoint);
}
```