XLoader/FormBook: Encryption Analysis and Malware Decryption

any.run/cybersecurity-blog/xloader-formbook-encryption-analysis-and-malware-decryption/

ANY.RUN February 28, 2023

Today ANY.RUN's malware analysts are happy to discuss the encryption algorithms of XLoader, also known as FormBook. And together we'll decrypt the stealer's strings and C2 servers.

Xloader is a stealer, the successor of FormBook. However, apart from the basic functionality, the unusual approaches to encryption and obfuscation of internal structures, code, and strings used in XLoader are also of interest. Let's take a detailed look at the encryption of strings, functions, and C2 decoys.

Encryption in XLoader

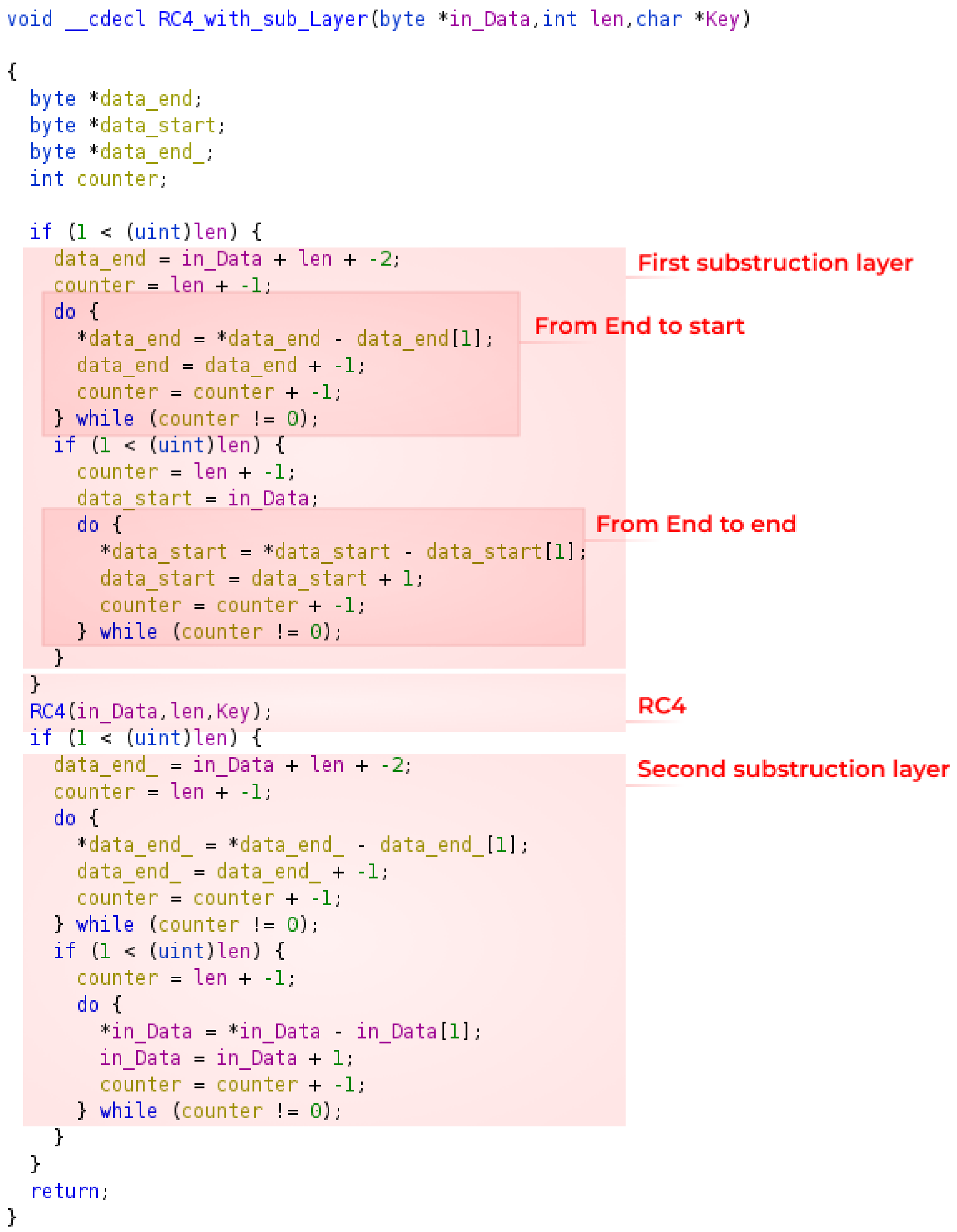
First, we should research 3 main cryptographic algorithms used in XLoader. These are the modified algorithms: RC4, SHA1, and Xloader's own algorithm based on a virtual machine.

The modified RC4 algorithm

The modified RC4 algorithm is a usual RC4 with additional layers of sequential subtraction before and after the RC4 call. In the code one layer of subtractions looks like this:

```
# transform 1
    for i in range(len(encbuf) - 1, 0, -1):
        encbuf[i-1] -= encbuf[i]

    # transform 2
    for i in range(0, len(encbuf) -1):
        encbuf[i] -= encbuf[i+1]
```

The ciphertext bytes are subtracted from each other in sequence from right to left. And then they go from left to right. In the XLoader code, it looks like this:



```
void __cdecl RC4_with_sub_Layer(byte *in_Data,int len,char *Key)

{
  byte *data_end;
  byte *data_start;
  byte *data_end_;
  int counter;

  if (1 < (uint)len) {
    data_end = in_Data + len + -2;
    counter = len + -1;
    do {
      *data_end = *data_end - data_end[1];
      data_end = data_end + -1;
      counter = counter + -1;
    } while (counter != 0);
    if (1 < (uint)len) {
      counter = len + -1;
      data_start = in_Data;
      do {
        *data_start = *data_start - data_start[1];
        data_start = data_start + 1;
        counter = counter + -1;
      } while (counter != 0);
    }
  }
  RC4(in_Data,len,Key);
  if (1 < (uint)len) {
    data_end_ = in_Data + len + -2;
    counter = len + -1;
    do {
      *data_end_ = *data_end_ - data_end_[1];
      data_end_ = data_end_ + -1;
      counter = counter + -1;
    } while (counter != 0);
    if (1 < (uint)len) {
      counter = len + -1;
      do {
        *in_Data = *in_Data - in_Data[1];
        in_Data = in_Data + 1;
        counter = counter + -1;
      } while (counter != 0);
    }
  }
  return;
}
```

Function performing RC4 encryption

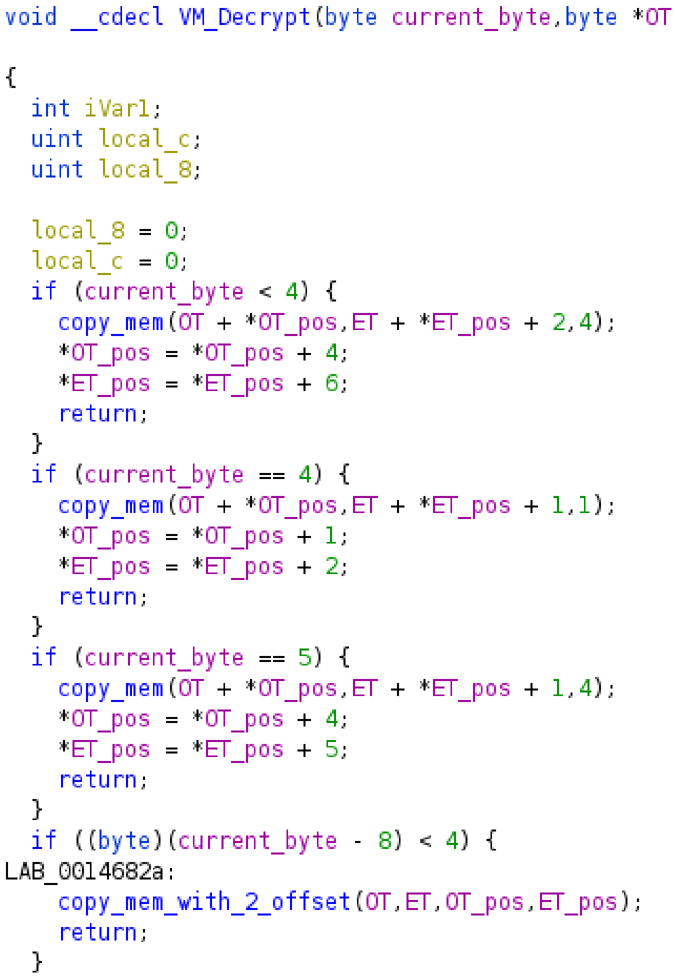
The modified SHA1 algorithm

The SHA1 modification is a regular SHA1, but every 4 bytes are inverted:

```
def reversed_dword_sha1(self, dat2hash):
      sha1Inst = SHA1.new()
      sha1Inst.update(dat2hash)
      hashed_data = sha1Inst.digest()
      result = b""
      for i in range(5):
          result += hashed_data[4*i:4*i+4][::-1]
      return result
```

Xloader’s own virtual machine algorithm

The last algorithm is a virtual machine that generates one to four bytes of plaintext, depending on the current byte of the ciphertext. Usually, this algorithm is used as an additional encryption layer, which will be discussed later. The entry of the VM decryption routine looks like this:

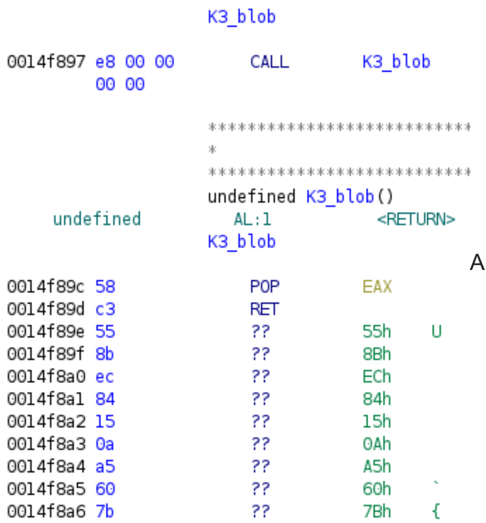


```
void __cdecl VM_Decrypt(byte current_byte,byte *OT,byte *ET,int *OT_pos,int *ET_pos)

{
  int iVar1;
  uint local_c;
  uint local_8;

  local_8 = 0;
  local_c = 0;
  if (current_byte < 4) {
    copy_mem(OT + *OT_pos,ET + *ET_pos + 2,4);
    *OT_pos = *OT_pos + 4;
    *ET_pos = *ET_pos + 6;
    return;
  }
  if (current_byte == 4) {
    copy_mem(OT + *OT_pos,ET + *ET_pos + 1,1);
    *OT_pos = *OT_pos + 1;
    *ET_pos = *ET_pos + 2;
    return;
  }
  if (current_byte == 5) {
    copy_mem(OT + *OT_pos,ET + *ET_pos + 1,4);
    *OT_pos = *OT_pos + 4;
    *ET_pos = *ET_pos + 5;
    return;
  }
  if ((byte)(current_byte - 8) < 4) {
LAB_0014682a:
    copy_mem_with_2_offset(OT,ET,OT_pos,ET_pos);
    return;
  }
```

An example of transformations in a virtual machine's decryption routine

Decrypting XLoader Strings

Next, let's investigate how string encryption works in XLoader. All byte arrays containing encrypted strings or key information are located in special kinds of blobs.

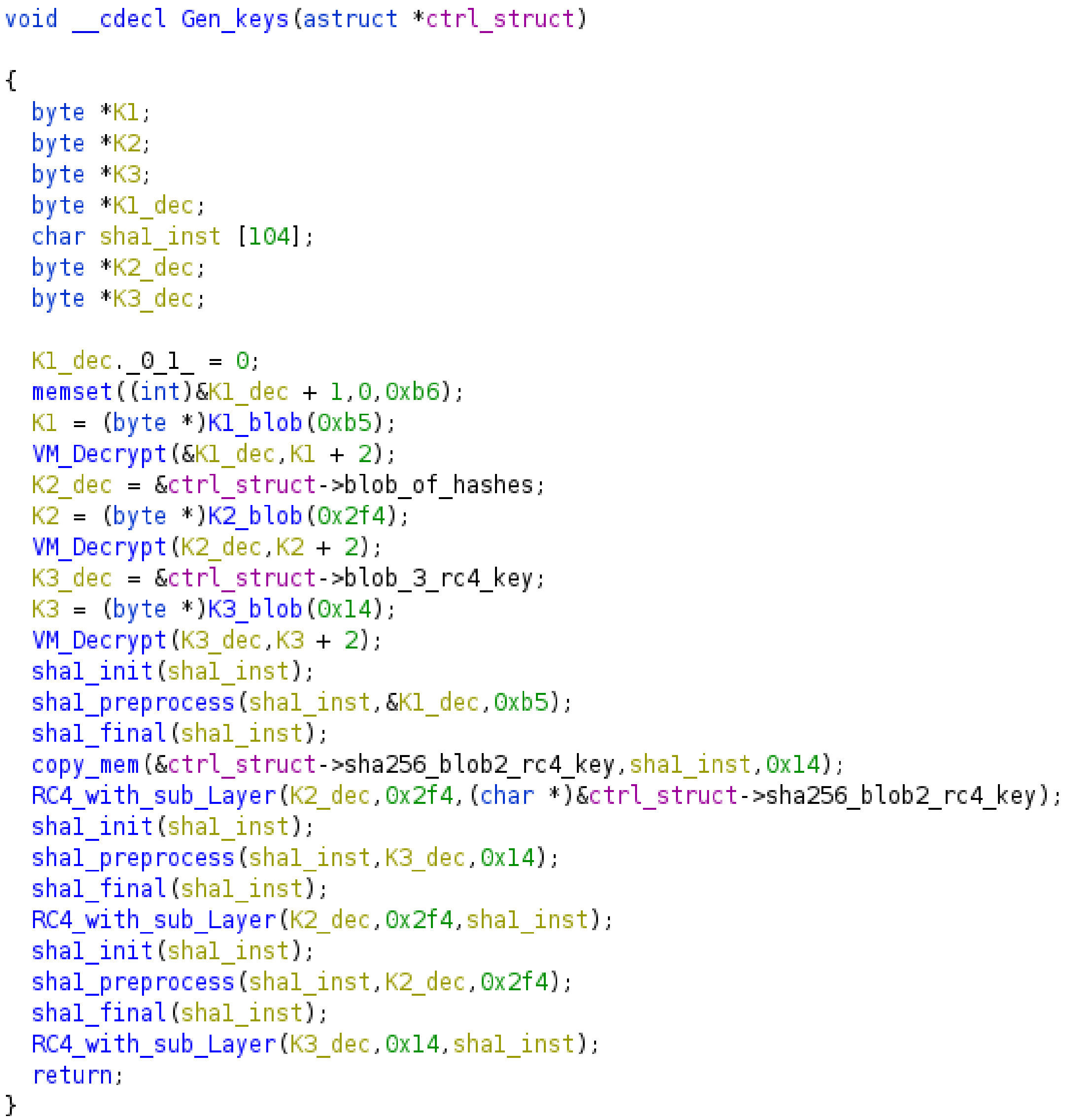


```
                         K3_blob

 0014f897 e8 00 00          CALL       K3_blob
          00 00

                         **************************
                         *
                         **************************
                         undefined K3_blob()
      undefined            AL:1           <RETURN>
                         K3_blob
 0014f89c 58                POP        EAX
 0014f89d c3                RET
 0014f89e 55                ??         55h    U
 0014f89f 8b                ??         8Bh
 0014f8a0 ec                ??         ECh
 0014f8a1 84                ??         84h
 0014f8a2 15                ??         15h
 0014f8a3 0a                ??         0Ah
 0014f8a4 a5                ??         A5h
 0014f8a5 60                ??         60h    `
 0014f8a6 7b                ??         7Bh    {
```

An example of a blob with encrypted data

As you can see in the screenshot above, this blob is a function that returns a pointer to itself, below this function are the bytes you are looking for.

In order to decrypt strings, first a key is generated. The key is generated from 3 parts, to which the above-described functions are applied.



```
void __cdecl Gen_keys(astruct *ctrl_struct)

{
  byte *K1;
  byte *K2;
  byte *K3;
  byte *K1_dec;
  char sha1_inst [104];
  byte *K2_dec;
  byte *K3_dec;

  K1_dec._0_1_ = 0;
  memset((int)&K1_dec + 1,0,0xb6);
  K1 = (byte *)K1_blob(0xb5);
  VM_Decrypt(&K1_dec,K1 + 2);
  K2_dec = &ctrl_struct->blob_of_hashes;
  K2 = (byte *)K2_blob(0x2f4);
  VM_Decrypt(K2_dec,K2 + 2);
  K3_dec = &ctrl_struct->blob_3_rc4_key;
  K3 = (byte *)K3_blob(0x14);
  VM_Decrypt(K3_dec,K3 + 2);
  sha1_init(sha1_inst);
  sha1_preprocess(sha1_inst,&K1_dec,0xb5);
  sha1_final(sha1_inst);
  copy_mem(&ctrl_struct->sha256_blob2_rc4_key,sha1_inst,0x14);
  RC4_with_sub_Layer(K2_dec,0x2f4,(char *)&ctrl_struct->sha256_blob2_rc4_key);
  sha1_init(sha1_inst);
  sha1_preprocess(sha1_inst,K3_dec,0x14);
  sha1_final(sha1_inst);
  RC4_with_sub_Layer(K2_dec,0x2f4,sha1_inst);
  sha1_init(sha1_inst);
  sha1_preprocess(sha1_inst,K2_dec,0x2f4);
  sha1_final(sha1_inst);
  RC4_with_sub_Layer(K3_dec,0x14,sha1_inst);
  return;
}
```

Key generation function

to decrypt strings

Here K1_blob, K2_blob, and K3_blob are functions that return data from the blocks described above, and the string length is an argument for them.

The functions VM_Decrypt, RC4_with_sub_Layer and sha1_* are modified algorithms that we discussed earlier.

Schematically, the key generation algorithm can be represented by the following diagram.

Scheme of key generation to decrypt strings

Here **E** and **K** are the data and the key that is fed to the input of the RC4 function, respectively, and **K1, K2,** and **K3** are the data obtained from the K1_blob, K2_blob, and K3_blob functions.

The strings themselves are also stored as a blob and are covered by two layers of encryption:

- VM_decrypt
- RC4 that uses the key obtained above.

At the same time, RC4 is not used for the whole blob at once.

After removing the first layer, the encrypted strings themselves are stored in the format:

encrypted string length – encrypted string

Consequently, to decrypt the strings, we need to loop through this structure and consistently decrypt all the strings.

Function for decrypting strings

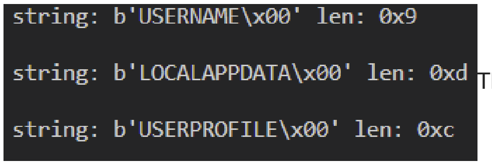
Function for decrypting strings

Below is an example of the encrypted data after stripping the first layer. Length/string pairs for the first 3 encrypted strings are highlighted in red.

The first 3 encrypted strings

The first 3 encrypted strings

The same strings after decryption:



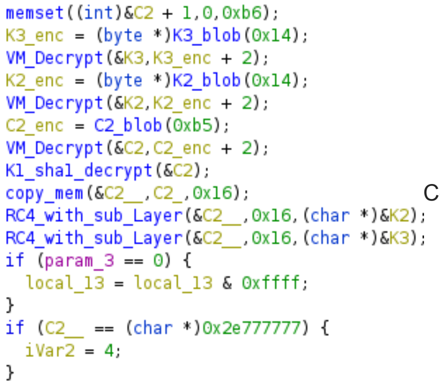
```
string: b'USERNAME\x00' len: 0x9

string: b'LOCALAPPDATA\x00' len: 0xd

string: b'USERPROFILE\x00' len: 0xc
```

The first 3 lines after decoding

Along with the encrypted strings, C2 decoys are also stored there. They are always located at the end of all decrypted strings, beginning and ending with the f-start and f-end strings.

Decrypting XLoader’s C2 Servers

Next, let’s see how the main C2 encryption works. The main C2 is located elsewhere in the code, so you can get it separately from the C2 decoys.



```
memset((int)&C2 + 1,0,0xb6);
K3_enc = (byte *)K3_blob(0x14);
VM_Decrypt(&K3,K3_enc + 2);
K2_enc = (byte *)K2_blob(0x14);
VM_Decrypt(&K2,K2_enc + 2);
C2_enc = C2_blob(0xb5);
VM_Decrypt(&C2,C2_enc + 2);
K1_sha1_decrypt(&C2);
copy_mem(&C2__,C2_,0x16);
RC4_with_sub_Layer(&C2__,0x16,(char *)&K2);
RC4_with_sub_Layer(&C2__,0x16,(char *)&K3);
if (param_3 == 0) {
  local_13 = local_13 & 0xffff;
}
if (C2__ == (char *)0x2e777777) {
  iVar2 = 4;
}
```

Code snippet demonstrating C2 decryption.

To decrypt it, as well as to decrypt the strings, 3 keys are used. The C2 decryption scheme is shown below:

- **EC2** is the encrypted C2
- **DC2** is the decrypted C2

The algorithm itself is a 3 times sequential application of the RC4 algorithm with 3 different keys.

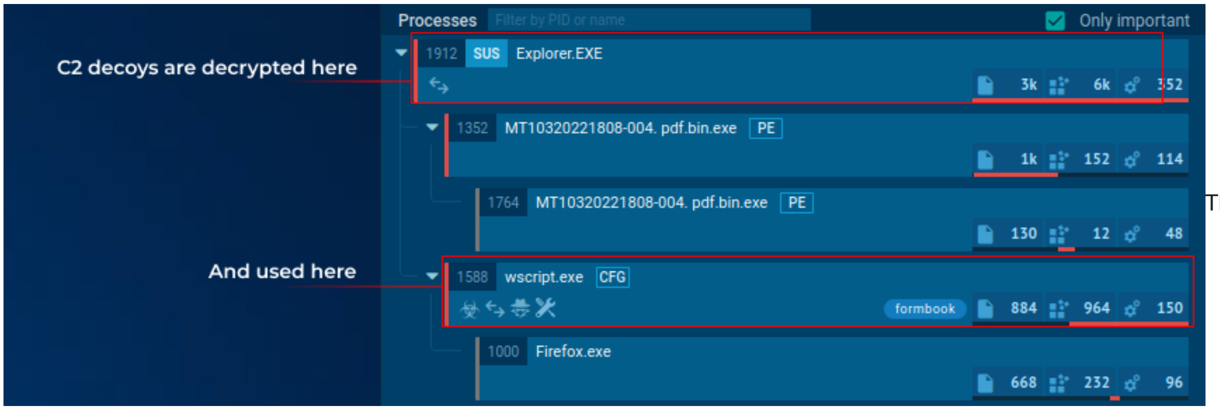
C2 decoys’ decryption scheme

C2 decoys’ decryption scheme

Also, in newer versions of XLoader C2 decoys, which usually lie along with all the other strings, turn out to be covered by an additional layer of encryption, and, at first glance, it is completely unclear where exactly the decryption of these strings occurs.

Since XLoader has several entry points, each responsible for different non-intersecting functionality, with many functions turning out to be encrypted.

The C2 decoys are decrypted inside the XLoader injected into Explorer.exe. And in this case, it is passed to netsh.exe, which also contains XLoader via APC injection.

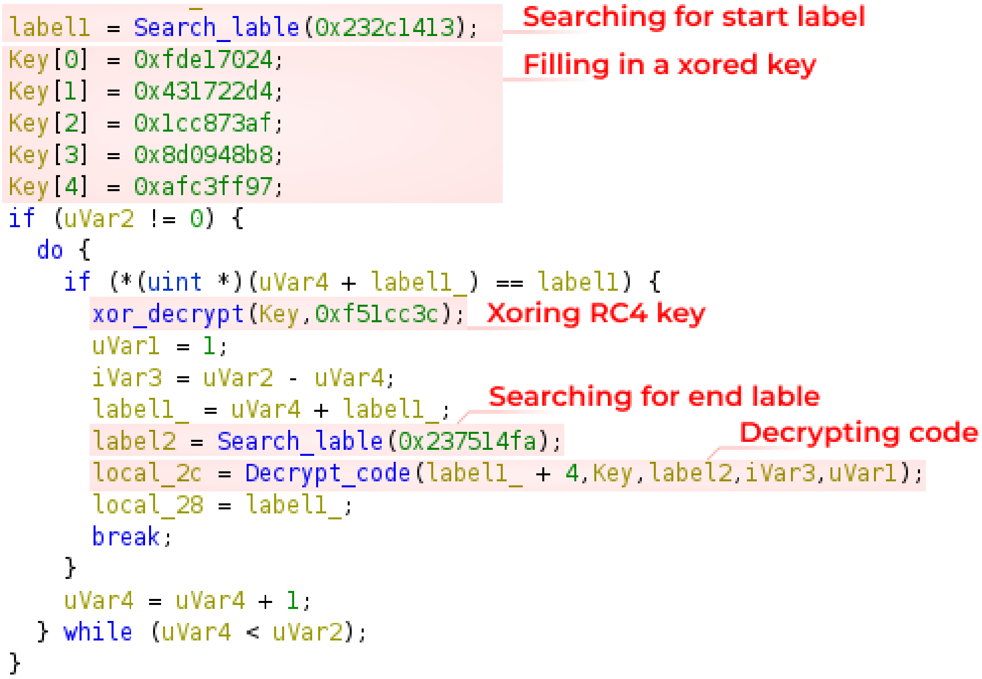


The C2 life cycle in different XLoader modules

In order to understand how a C2 decoy is encrypted, first of all, you need to understand how the functions are encrypted.

It's actually quite simple. RC4 is used as the encryption algorithm. This time, the key is hardcoded and written right in the code and then xored with the 4-byte gamma.

After that, you should find pointers to the start and end of the function. This is how you do it: a unique 4-byte value is placed at the beginning and end of each encrypted function. The XLoader looks for these values and gets the desired pointers.



```
label1 = Search_lable(0x232c1413);
Key[0] = 0xfde17024;
Key[1] = 0x431722d4;
Key[2] = 0x1cc873af;
Key[3] = 0x8d0948b8;
Key[4] = 0xafc3ff97;
if (uVar2 != 0) {
  do {
    if (*(uint *)(uVar4 + label1_) == label1) {
      xor_decrypt(Key,0xf51cc3c);
      uVar1 = 1;
      iVar3 = uVar2 - uVar4;
      label1_ = uVar4 + label1_;
      label2 = Search_lable(0x237514fa);
      local_2c = Decrypt_code(label1_ + 4,Key,label2,iVar3,uVar1);
      local_28 = label1_;
      break;
    }
    uVar4 = uVar4 + 1;
  } while (uVar4 < uVar2);
}
```

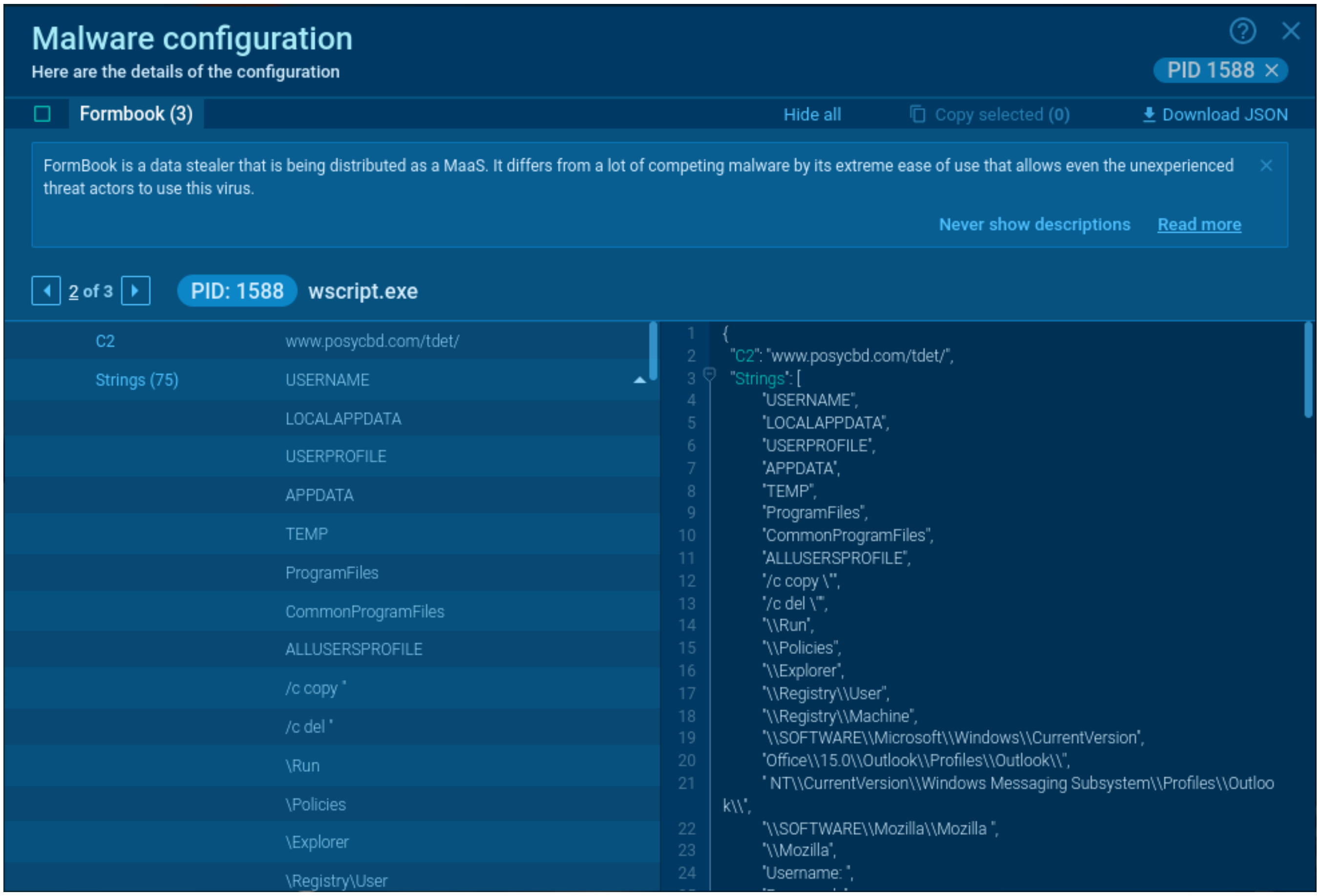
Code snippet demonstrating the decryption of the function

Then the function is decrypted, control is given to it, and it similarly searches for and decrypts the next function. This happens until the function with the main functionality is decrypted and executed. So, functions should be decrypted recursively.

The key to decrypting C2 decoys consists of 2 parts and is collected separately at two different exit points. One exit point gets the 20-byte protected key, and the second gets the 4-byte gamma to decrypt the key.

Example of extracted XLoader malware configuration

Applying the above algorithms we can extract the configuration from Xloader, including C2, C2 decoys, and strings. For your convenience, we have integrated automatic extraction of the Xloader configuration into ANY.RUN interactive sandbox — just run the sample and get all the IOCs in seconds.



Malware configuration

Here are the details of the configuration

FormBook is a data stealer that is being distributed as a MaaS. It differs from a lot of competing malware by its extreme ease of use that allows even the unexperienced threat actors to use this virus.

C2	www.posycbd.com/tdet/
Strings (75)	USERNAME
	LOCALAPPDATA
	USERPROFILE
	APPDATA
	TEMP
	ProgramFiles
	CommonProgramFiles
	ALLUSERSPROFILE
	/c copy "
	/c del "
	\Run
	\Policies
	\Explorer
	\Registry\User

```
{
 "C2": "www.posycbd.com/tdet/",
 "Strings": [
      "USERNAME",
      "LOCALAPPDATA",
      "USERPROFILE",
      "APPDATA",
      "TEMP",
      "ProgramFiles",
      "CommonProgramFiles",
      "ALLUSERSPROFILE",
      "/c copy \"",
      "/c del \"",
      "\\Run",
      "\\Policies",
      "\\Explorer",
      "\\Registry\\User",
      "\\Registry\\Machine",
      "\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion",
      "Office\\15.0\\Outlook\\Profiles\\Outlook\\",
      " NT\\CurrentVersion\\Windows Messaging Subsystem\\Profiles\\Outloo
k\\",
      "\\SOFTWARE\\Mozilla\\Mozilla ",
      "\\Mozilla",
      "Username: ",
```

Extracted malware configuration in ANY.RUN

Extracted malware configuration in ANY.RUN

Examples of successfully executed samples:

https://app.any.run/tasks/f6d29aa7-4054-44b6-b4cc-61684742da88/

https://app.any.run/tasks/aa804b50-2c11-447e-a5a9-709b83634aa0/

https://app.any.run/tasks/8bcca55f-99ae-4b8d-b60e-226562068d9a/

Sum it up

In this article, we discussed the encryption in xLoader stealer. It is based on both add-ons to existing algorithms and self-written algorithms.

The main tricky part of the decryption process is the key generation and the fact that the XLoader functionality is split into modules that can be run in different processes. Because of this, in order to extract strings, we have to decrypt the executable code, among other things.

Fortunately, ANY.RUN is already set up to detect this malware automatically, making the relevant configuration details just a click away.

If you want to read more content like this, check out our analysis of the Raccoon Stealer, CryptBot, or Orcus RAT.

Appendix

Analyzed files

Sample with new C2 decoys encryption

Title	Description
Name	MT10320221808-004. pdf.exe
MD5	b7127b3281dbd5f1ae76ea500db1ce6a
SHA1	6e7b8bdc554fe91eac7eef5b299158e6b2287c40
SHA256	726fd095c55cdab5860f8252050ebd2f3c3d8eace480f8422e52b3d4773b0d1c

Sample without C2 decoys encryption

Title	Description
Name	Transfer slip.exe
MD5	1b5393505847dcd181ebbc23def363ca
SHA1	830edb007222442aa5c0883b5a2368f8da32acd1
SHA256	27b2b539c061e496c1baa6ff071e6ce1042ae4d77d398fd954ae1a62f9ad3885

malware analysis
What do you think about this post?

14 answers

- Awful
- Average
- Great

No votes so far! Be the first to rate this post.

1 comments