Analysis of Xloader's C2 Network Encryption

zscaler.com/blogs/security-research/analysis-xloaders-c2-network-encryption

Introduction

Xloader is an information stealing malware that is the successor to Formbook, which had been sold in hacking forums since early 2016. In October 2020, Formbook was rebranded as Xloader and some significant improvements were introduced, especially related to the command and control (C2) network encryption. With the arrival of Xloader, the malware authors also stopped selling the panel's code together with the malware executable. When Formbook was sold, a web-based command and control (C2) panel was given to customers, so they could self-manage their own botnets. In 2017, Formbook's panel source was leaked, and subsequently, the threat actor behind Xloader moved to a different business model. Rather than distributing a fully functional crimeware kit, Xloader C2 infrastructure is rented to customers. This malware-as-a-service (MaaS) business model is likely more profitable and makes piracy more difficult.

The capabilities of Xloader include the following:

- Steal credentials from web browsers and other applications
- Capture keystrokes
- Take screenshots
- Steal stored passwords
- Download and execute additional binaries
- Execute commands

Previous blog posts have analyzed various aspects of Formbook and Xloader's obfuscation. In this blog post, we perform a detailed analysis of Xloader's C2 network encryption and communication protocol. Note that Xloader is cross-platform with the ability to run on Microsoft Windows and MacOS. This analysis focuses specifically on the Windows version of Xloader.

Technical Analysis

Xloader and Formbook use HTTP to communicate with the C2 server. An HTTP GET query is sent as a form of registration. Afterwards, the malware makes HTTP POST requests to the C2 to exfiltrate information such as screenshots, stolen data, etc. In both cases, the GET parameters and the POST data share a similar format and are encrypted as shown in Figure 1. We will explain the encryption algorithms in the following sections.

```
DNS    80 Standard query 0x49f1 A www.erikahealth.info
DNS   110 Standard query response 0x49f1 A www.erikahealth.info CNAME erikahealth.info A 184.168.221.56
HTTP  221 GET /pw9/?jfGHNt=vDji614SJdfLu9VbK2jDv/mP+A8HYxb0NKAQaxCkCRxvJKadnaknVLlfe59KnFDAD5xVyQ==&UBk=D8Tp7BM HTTP/1.1 Continuation
DNS    76 Standard query 0x5f83 A www.artiyonq.com
DNS    92 Standard query response 0x5f83 A www.artiyonq.com A 63.250.45.114
HTTP  217 GET /pw9/?jfGHNt=BWMV6koueRSV8lNQPD+p9XeHTMELk/gzFj0TOmfsDWprmvNnKAVk8FYzBr6O/71AQzbzlA==&UBk=D8Tp7BM HTTP/1.1 Continuation
HTTP  556 HTTP/1.1 404 Not Found  (text/html)
HTTP  753 POST /pw9/ HTTP/1.1  (application/x-www-form-urlencoded)Continuation
HTTP  522 HTTP/1.1 404 Not Found  (text/html)
HTTP  592 POST /pw9/ HTTP/1.1  (application/x-www-form-urlencoded)Continuation
HTTP  522 HTTP/1.1 404 Not Found  (text/html)
HTTP  620 POST /pw9/ HTTP/1.1  (application/x-www-form-urlencoded)Continuation
HTTP  522 HTTP/1.1 404 Not Found  (text/html)
```

Figure 1. Xloader C2 communications capture

Decoy and Real C2 Servers

Throughout the Xloader malware there are multiple structures of encrypted blocks of data and code. These blocks are designed to confuse malware analysts and disassemblers by using the assembly instructions for a function prologue *push ebp* and *mov ebp, esp* as shown in Figure 2. We have named these structures PUSHEBP encrypted blocks. These blocks are decrypted using an RC4 based algorithm combined with an encoding layer and a custom virtual machine (VM).

```
.text:0041E3C4                       loc_41E3C4:
.text:0041E3C4
.text:0041E3C4 E8 00 00 00 00                    call    $+5
.text:0041E3C9 58                                pop     eax
.text:0041E3CA C3                                retn
.text:0041E3CB                       ; -----------------------------------
.text:0041E3CB 55                                push    ebp
.text:0041E3CC 8B EC                             mov     ebp, esp
.text:0041E3CC                       ; -----------------------------------
.text:0041E3CE 22                                db  22h ; "
.text:0041E3CF 35                                db  35h ; 5
.text:0041E3D0 8A                                db  8Ah ; Š
.text:0041E3D1 A1                                db 0A1h ; ¡
.text:0041E3D2 A9                                db 0A9h ; ©
.text:0041E3D3 DD                                db 0DDh ; Ý
```

Figure 2. Xloader PUSHEBP encrypted block

One of these PUSHEBP blocks contains encrypted strings, and a list of decoy C2s. These decoys are legitimate domains that have been added to mislead malware researchers and automated malware analysis systems. The real C2 server is stored separately and encrypted using another more complex scheme. The pseudocode responsible for decrypting the real C2 server is shown in Figure 3.

```
dec_real_c2[0] = 0;
key_rc4_1 = 0;
key_rc4_2 = 0;
dec_real_c2_layer1 = 0;
memset(&v10, 0, 0xC8u);
v5 = GetPUSHEBPEncContent8__keylen14();
VM_Decryptor((int)&key_rc4_1, v5 + 2, 0x14u);
v6 = GetPUSHEBPEncContent2__keylen14();
VM_Decryptor((int)&key_rc4_2, v6 + 2, 0x14u);
enc_real_c2 = GetPUSHEBPEncContent11___RealC2_();
VM_Decryptor((int)&dec_real_c2_layer1, enc_real_c2 + 2, 0xC7u);
DecryptorRc4_KeySha1EncStrings((int)&dec_real_c2_layer1);
memcpy((int)dec_real_c2, &v11, 0x19);
RC4_based_Decryptor(dec_real_c2, 0x19u, (int)&key_rc4_2);
RC4_based_Decryptor(dec_real_c2, 0x19u, (int)&key_rc4_1);
if ( !a3 )
  *(_WORD *)((char *)&v32 + 1) = 0;
if ( *(_DWORD *)dec_real_c2 == '.www' )
```

Get and decrypt two PUSHEBP blocks containing RC4 keys to be used later

Get and decrypt PUSHEBP block containing the encrypted real C2

Use the RC4 based decryptor to decrypt one more layer of the real C2. RC4 key is the SHA1 of the full block of encrypted strings (recovered also from PUSHEBP blocks)

Decrypt two last layers of the real C2, using the RC4 based algorithm and the two keys recovered in the beginning of the function

Figure 3. Xloader C2 decryption algorithm

In Figure 3, the *RC4_based_Decryptor* function consists of RC4 encryption (with a 0x14 byte key) with an additional two encoding layers as shown below:

```
################################################################

def decrypt_PUSHEBP_encrypted_function_block(self, rc4_key,
encdata):

  #backward / forward sub layer 1
  encdata =
self.decrypt_PUSHEBP_backward_forward_sub_layers(encdata)

  #rc4
  encdata = self.rc4(encdata, rc4_key)

  #backward / forward sub layer 2
  encdata =
self.decrypt_PUSHEBP_backward_forward_sub_layers(encdata)

  return encdata
```

The additional encoding layers consist of simple subtraction operations:

```
################################################################

 def decrypt_PUSHEBP_backward_forward_sub_layers(self, encdata):
   encdata = list(encdata)
   lencdata = len(encdata)
   #backward sub
   p1 = lencdata - 2
   counter = lencdata - 1
   while True:
     encdata[p1] = chr(0xff&(ord(encdata[p1]) - ord(encdata[p1 +
1])))
     p1 -= 1
     counter -= 1
     if not counter: break
   #forward sub
   p1 = 0
   counter = lencdata - 1
   while True:
     encdata[p1] = chr(0xff&(ord(encdata[p1]) - ord(encdata[p1 +
1])))
     p1 += 1
     counter -= 1
     if not counter: break
   return ''.join(encdata)
```

The VM_Decryptor function is another algorithm that is used by Xloader, which implements a custom virtual machine (VM). The following lines of Python reproduce the steps that Xloader performs to decrypt the real C2.

```
################################################################

# get blocks of enc strings
b1 = GetPUSHEBPBlock(1)
enc_strings_block = VM_Decryptor(b1)

# get rc4 key 1, 0x14 bytes
b8 = GetPUSHEBPBlock(8)
key_rc4_1 = VM_Decryptor(b8)

# get rc4 key 2, 014 bytes
b2 = GetPUSHEBPBlock(2)
key_rc4_2 = VM_Decryptor(b2)

# get the block containing enc real C2
b11 = GetPUSHEBPBlock(11)
enc_real_c2 = VM_Decryptor(b11)

# decrypt first layer of the real C2, use the RC4 based algorithm
and
# the SHA1 of the full block of encrypted strings
enc_real_c2 = RC4_based_Decryptor(enc_real_c2,
SHA1(enc_strings_block))

# decrypt the next layers of the real C2, use RC4 based algorithm
and
# the two RC4 key recovered previously from the PUSHEBP blocks
enc_real_c2 = RC4_based_Decryptor(enc_real_c2, key_rc4_1)
dec_real_c2 = RC4_based_Decryptor(enc_real_c2, key_rc4_2)

# the valid decrypted real c2 must start with www.
b_ok = is_www(dec_real_c2)
```

Once decrypted, the C2 URL has a format similar to www.domain.tld/botnet_id/.

The C2 communications occur with the decoy domains and the real C2 server, including sending stolen data from the victim. Thus, there is a possibility that a backup C2 can be hidden in the decoy C2 domains and be used as a fallback communication channel in the event that the primary C2 domain is taken down.

Formbook Communication Encryption Specific Details

In FormBook, the HTTP GET parameters (and POST data) were encrypted in four steps:

1. Using the domain and path of the real C2, an RC4 key was calculated in this way:
   Reverse_DWORDs(SHA1(<domain>/<cncpath>/))
2. The result was used as an RC4 key to encrypt the data
3. Once the data was RC4 encrypted, it was additionally encoded using *Base64*
4. Data sent via HTTP POST requests was formatted using the character substitution that is shown in Table 1.

Original Symbol	Replacement Symbol
+	-
/	_
=	.
+	~
/	(
=	)
+	<space>

Table 1. Formbook C2 Characters Substitution

Therefore, Formbook C2 communications could be easily decrypted by reversing the process since the C2 domain and path are known.

Xloader Communication Encryption Specific Details

The network encryption in XLoader is more complex. An additional RC4 layer was added to the process, with a convoluted algorithm that is used to derive this encryption key using the following steps:

1) To encrypt the HTTP network data, Xloader first calculates a key that we call *Key0Comm* as shown in Figure 4.

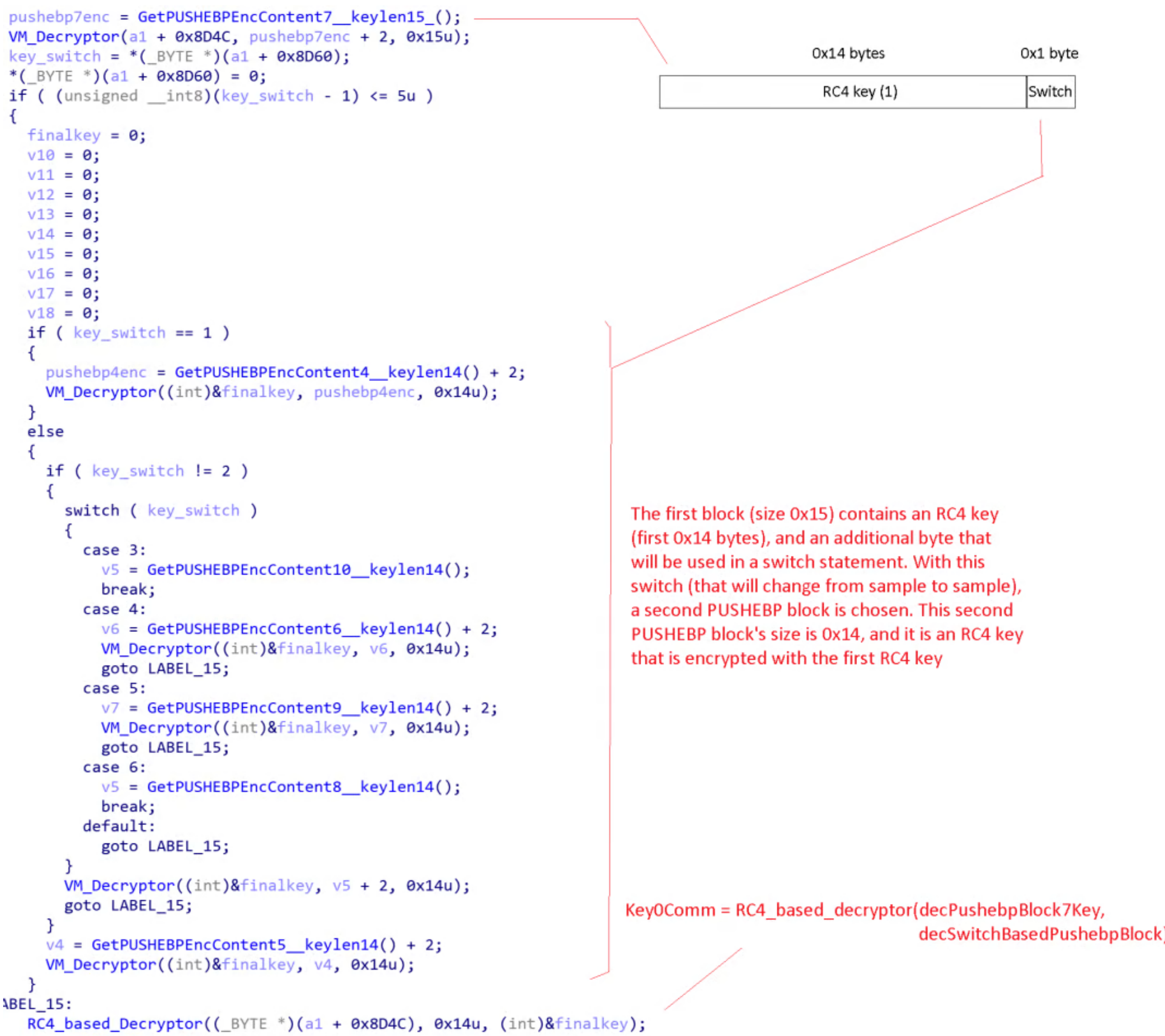


Figure 4. Xloader KeyComm0 Derivation

As we can see in Figure 4, the PUSHEBP block 7 is decrypted using the Xloader VM. This block, once decrypted, has a length of 0x15 bytes. The first 0x14 bytes are used as an RC4 key, and the last byte is used to choose and decrypt another PUSHEBP block (among the blocks 4, 5, 6, 8, 9 and 10) based on a switch statement. Thus the parameter *Key0Comm* in derived as follows:

Key0Comm = RC4_based_Decryptor(decPushebpBlock7Key[:0x14], decSwitchBasedPushebpBlock)

However, the order of the PUSHEBP blocks, and the associations between the switch and the block number, changes from one sample to another (i.e., the code of this function is randomized), even on the same versions of Xloader. Figure 5 shows a comparison of this function between two different Xloader v2.5 samples.

From sample 18B5783DE4068B6E8B7CD6EA20C89AF4 (Xloader 2.5)

```
if ( key_switch == 1 )
{
  pushebp4enc = GetPUSHEBPEncContent4__keylen14() + 2;
  VM_Decryptor((int)&finalkey, pushebp4enc, 0x14u);
}
else
{
  if ( key_switch != 2 )
  {
    switch ( key_switch )
    {
      case 3:
        v5 = GetPUSHEBPEncContent10__keylen14();
        break;
      case 4:
        v6 = GetPUSHEBPEncContent6__keylen14() + 2;
        VM_Decryptor((int)&finalkey, v6, 0x14u);
        goto LABEL_15;
      case 5:
        v7 = GetPUSHEBPEncContent9__keylen14() + 2;
        VM_Decryptor((int)&finalkey, v7, 0x14u);
        goto LABEL_15;
      case 6:
        v5 = GetPUSHEBPEncContent8__keylen14();
        break;
      default:
        goto LABEL_15;
    }
    VM_Decryptor((int)&finalkey, v5 + 2, 0x14u);
    goto LABEL_15;
  }
  v4 = GetPUSHEBPEncContent5__keylen14() + 2;
  VM_Decryptor((int)&finalkey, v4, 0x14u);
```

From sample F841C72B1C4CADC4C98903AD26A96A16 (Xloader 2.5)

```
if ( key_switch == 1 )
{
  pushebp6enc = GetPUSHEBPEncContent6__keylen14() + 2;
  VM_Decryptor(&finalkey, pushebp6enc, 0x14);
}
else
{
  if ( key_switch != 2 )
  {
    switch ( key_switch )
    {
      case 3:
        v5 = GetPUSHEBPEncContent8__keylen14();
        break;
      case 4:
        v6 = GetPUSHEBPEncContent5__keylen14() + 2;
        VM_Decryptor(&finalkey, v6, 0x14);
        goto LABEL_15;
      case 5:
        v7 = GetPUSHEBPEncContent4__keylen14() + 2;
        VM_Decryptor(&finalkey, v7, 0x14);
        goto LABEL_15;
      case 6:
        v5 = GetPUSHEBPEncContent10__keylen14();
        break;
      default:
        goto LABEL_15;
    }
    VM_Decryptor(&finalkey, v5 + 2, 0x14);
    goto LABEL_15;
  }
  v4 = GetPUSHEBPEncContent7__keylen14() + 2;
  VM_Decryptor(&finalkey, v4, 0x14);
```

Figure 5. Xloader KeyComm0 Function to Map the Switch to a Block

Table 2 shows how these switch statements map to different block IDs in these samples.

	Switch 1	Switch 2	Switch 3	Switch 4	Switch 5	Switch 6
Sample 1	Block 4	Block 5	Block 10	Block 6	Block 9	Block 8
Sample 2	Block 6	Block 7	Block 8	Block 5	Block 4	Block 10

Table 2. Xloader Block ID Mapping Example

In order to perform encryption for the C2 communications, the sample-specific table that maps these blocks must be known to derive the encryption key *Key0Comm*.

2) Next, another key that we refer to as *Key1Comm* is calculated using the same algorithm as Formbook:

Key1Comm = Reverse_DWORDs(SHA1(<domain>/<cncpath>/))

3) Finally, we need to calculate one last key, using the Xloader custom RC4-based decryption algorithm as follows:

Key2Comm = RC4based_Decryptor(Key0Comm, Key1Comm)

Having all three of these RC4 keys, we can encrypt and decrypt Xloader C2 communications. The packets are encrypted with two layers of standard RC4 using the keys Key2Comm and Key1Comm, as shown below:

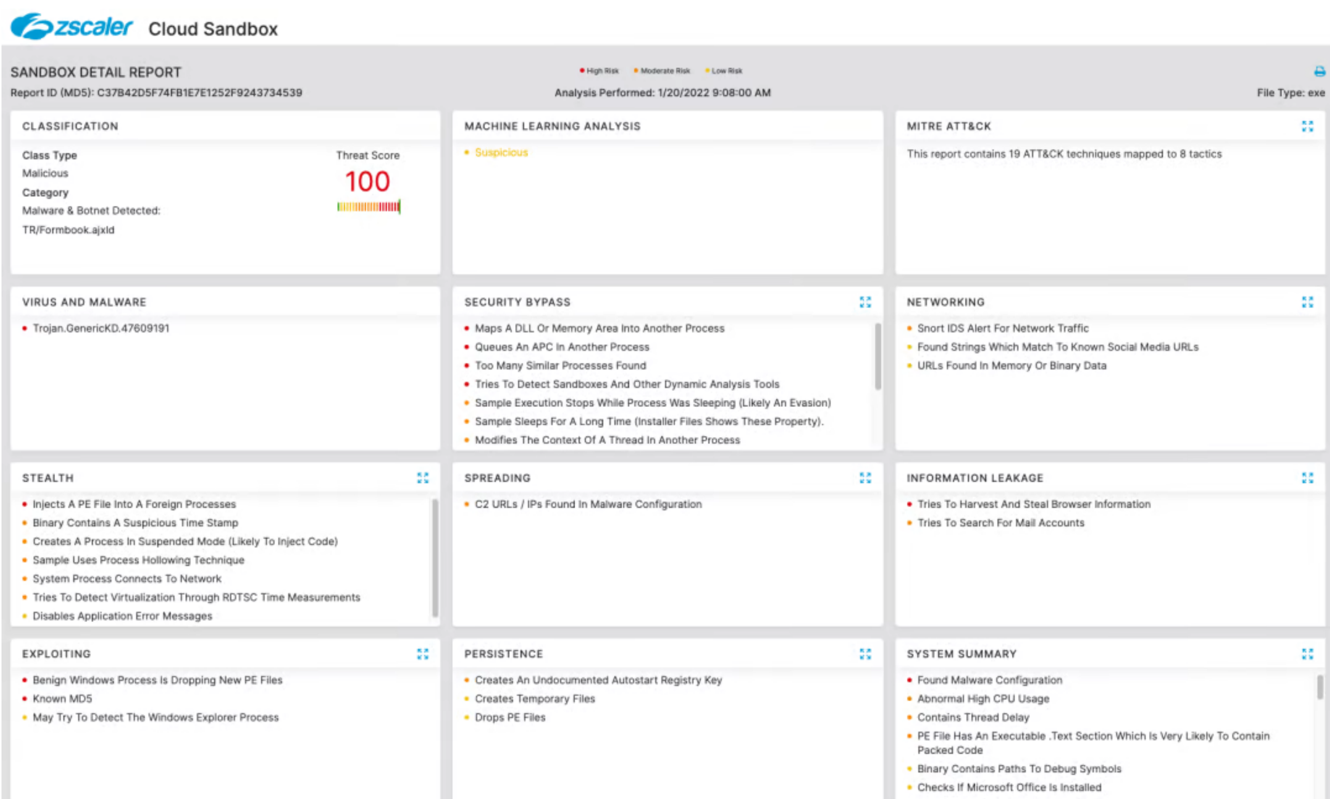
```
Key0Comm = <…from binary…>

c2 = "www.pc6888.com"
c2path = "htbn"

get1="xPeDUfwp=X/0PTsm65bsB0xA5p5tU+UuBoyxUJvYd1eRdC0qFrd+bv9rqN9yTTECZJTYp88Jb6Qhj
uA=="

Key1Comm = Reverse_DWORDs(SHA1(f"{c2}/{path}/"))
fake_var, encrypted_params = get1.split('=', 1)
sdec0 = b64_trans(encrypted_params)
sdec1 = base64.b64decode(sdec0)
Key2Comm = RC4_based_Decryptor(Key0Comm, Key1Comm)
sdec2 = rc4(sdec1, Key2Comm) #layers encrypted with standard rc4
sdec3 = rc4(sdec2, Key1Comm)
print(sdec3)
```

Xloader also further applies the *Base64* and character substitution described earlier for POST queries.

Conclusion

Xloader is a well-developed malware family that has numerous techniques to mislead researchers and hinder malware analysis including multiple layers of encryption and a custom virtual machine. Even though the authors abandoned the Formbook branch to focus on the rebranded Xloader, both strains are still quite active today. Formbook is still being used by threat actors using the leaked panel source code and self-managing the C2, while the original authors have continued to sell Xloader as MaaS, supporting and renting the servers infrastructure. Not surprisingly, it has been one of the most active threats in recent years.

Cloud Sandbox Detection



Zscaler's multilayered cloud security platform detects indicators at various levels, as shown below:

Indicators of Compromise

Variant	Version	SHA256	Real C2
Xloader	2.5	c60a64f8910005f98f6cd8c5787e4fe8c6580751a43bdbbd6a14af1ef6999b8f	http://www.finetipster[.]com/pvxz/

Xloader	2.5	2c78fa1d90fe76c14f0a642af43c560875054e342bbb144aa9ff8f0fdbb0670f	http://www.go2payme[.]com/snec/
Xloader	2.5	f3c3c0c49c037e7efa2fbef61995c1dc97cfe2887281ba4b687bdd6aa0a44e0a	http://www.pochi-owarai[.]com/hr8n/
Xloader	2.5	efd1897cf1232815bb1f1fbe8496804186d7c48c6bfa05b2dea6bd3bb0b67ed0	http://www.hosotructiep[.]online/bsz6/

References