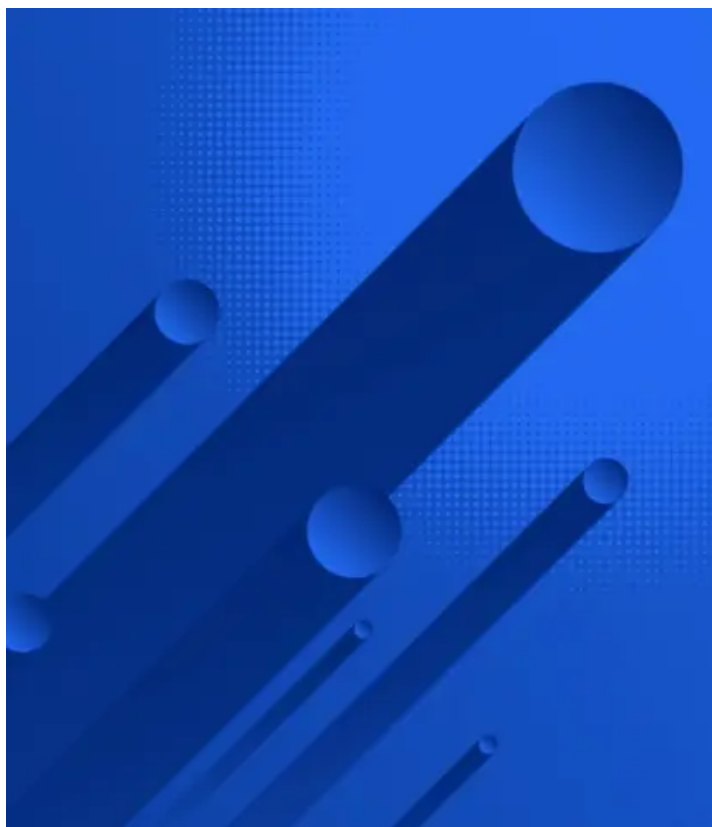


Technical Analysis of Xloader's Code Obfuscation in Version 4.3

zscaler.com/blogs/security-research/technical-analysis-xloaders-code-obfuscation-version-43

Javier Vicente, Brett Stone-Gross, Nikolaos Pantazopoulos



Key Points

- Xloader is a popular information stealing malware family that is the successor to Formbook.
- In early 2020, Formbook was rebranded as Xloader and the threat actors moved to a malware-as-a-service (MaaS) business model, renting C2 infrastructure to customers.
- Xloader implements different obfuscation methods and several encryption layers to protect critical parts of code and data from analysis.
- The developers behind this malware family continue to update the code with improved obfuscation and encryption layers with each new version that is released.
- In January 2023, Zscaler ThreatLabz identified a new variant of Xloader that identifies itself as version 4.3 with several modifications including additional obfuscation.

Introduction

Xloader is a rebranded version of the Formbook information stealing malware, which has been sold in criminal forums since 2016. The threat actors behind this malware family have been updating and improving the code regularly. In early 2020, the malware was rebranded as Xloader. In early 2022, the threat actors released Xloader version 2.9, which introduced significant improvements to obfuscate the malware code and data including the list of command-and-control (C2) servers. In October 2022, ThreatLabz identified a new Xloader version labeled as 3.9. In January 2023, the threat actors released Xloader version 4.3. Across

Xloader versions, the group has modified the malware's obfuscation techniques including adding numerous layers of encryption with code that recursively decrypts other blocks of code until reaching the core functionality that decrypts the most sensitive data (also encrypted with multiple layers).

This blog post analyzes the encryption algorithms used by Xloader to decrypt the most critical parts of the code and the most important parameters of the malware's configuration. The analysis is performed on the latest version of Xloader 4.3. ThreatLabz has also created an IDA Python script to decrypt Xloader's code and data to facilitate analysis.

Technical Analysis

Basic Algorithms and Structures

Formbook and Xloader have evolved along the years with new layers of obfuscation added in each new version. However, there is a set of basic algorithms that have been used since the first versions of Formbook. These algorithms are combined in different ways to decrypt other blocks of code and data. The primary algorithms that are shared between different versions of Xloader are the following:

- Custom RC4: an RC4-based algorithm with two additional layers based on subtraction operations.
- Custom buffer decryption algorithm: a custom algorithm used by Xloader, mainly used to decrypt the first encryption layer of the PUSHEBP data blocks (described in the following sections).
- Custom SHA1: a SHA1 hash is calculated and the result is reversed DWORD by DWORD.

There is also a large global data structure that is used to store important information. When Xloader is executed, this structure is allocated and initialized with information from PUSHEBP data blocks, or from hardcoded values in the code. This structure contains data and encryption keys that are used by other parts of the code. Previous blog posts have referred to this structure as the ConfigObj, with fields that are used to store flags, encryption parameters, pointers, etc. The most important offsets in the ConfigObj structure are identified in Table 1.

| Offset | Description | Size |
|--------|---|-------|
| 0x00 | Value 0xffffffff | 0x04 |
| 0x04 | Pointer to a second PE header used for process injection (e.g., explorer.exe) | 0x04 |
| 0x08 | Result of RtlGetProcessHeaps() | 0x04 |
| 0x48 | Branch ID – XLNG (XORed with 0x3c) | 0x04 |
| 0x90 | Pointer to an extended config (located in memory following the ConfigObj structure) | 0x04 |
| 0x2DC | Decrypted content of the PUSHEBP block 2, which is an array of API hashes | 0x220 |
| 0x510 | Array of library and process names hashes | 0x254 |

| Offset | Description | Size |
|--------|--|------|
| 0x828 | Seed of a random number generator (RNG) used by the malware | 0x4 |
| 0x83C | Flag indicating that Xloader has generated the parameters necessary for the communications with the C2 | 0x4 |
| 0x970 | The Xloader version number (XORed with 0x3c) | 0x4 |
| 0x990 | RC4 key used to decrypt other parameters | 0x14 |
| 0x9A4 | RC4 key used to decrypt other parameters (this key is the SHA1 of the decrypted content of the PUSHEDBP block 5) | 0x14 |
| 0xCE8 | RC4 key used to decrypt the C2s list | 0x14 |

Table 1. Important Xloader 4.3 ConfigObj fields.

Encrypted PUSHEDBP Data Blocks

Throughout the Xloader code there is a set of encrypted data blocks with the structure shown in Figure 1.

```

.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 ; 5
.text:0041E3D1 A1                  db     0A1h ; i
.text:0041E3D2 A9                  db     0A9h ; @
.text:0041E3D3 DD                  db     0DDh ; Y

```

Figure 1. Xloader PUSHEDBP encrypted block

This structure is designed to resemble the beginning of a function, but are in fact blocks of encrypted data such as encryption keys and encrypted strings. These data blocks are decrypted using a custom buffer decryption algorithm. Table 2 shows the PUSHEDBP encrypted blocks that were found in Xloader 4.3.

| PUSHEDBP Block Number | Description | Size |
|-----------------------|--|-------|
| PUSHEDBP Block 1 | Encrypted strings | 0xA82 |
| PUSHEDBP Block 2 | API CRCs | 0x222 |
| PUSHEDBP Block 3 | Encryption key involved in C2 communications | 0x15 |


```

.text:004178D3 00 03 00 FF FF FF
.text:004178D9 8D 86 04 12 00 00
.text:004178D9
.text:004178DF 51
.text:004178E0 50
.text:004178E1 89 96 D0 2C 00 00
.text:004178E7 C7 85 4C FF FF FF 31 83 9E C9
.text:004178E7
.text:004178F1 C7 85 50 FF FF FF C6 88 86 72
.text:004178FB C7 85 54 FF FF FF E9 00 BC 34
.text:00417905 C7 85 58 FF FF FF B4 AA D5 E4
.text:0041790F B3 A8
.text:00417911 C7 85 5C FF FF FF F5 31 9E A8
.text:0041791B C6 85 60 FF FF FF 00
.text:00417922 E8 89 48 00 00
mov     [ebp-93h], al
lea    eax, [esi+1204h] ; ConfigObjExt + 1204h - key to
; decrypt extra layer decoy c2
push   ecx
push   eax
mov     [esi+2CD0h], edx
mov     dword ptr [ebp-0B4h], 0C99E8331h ; key build in stack and
; copied to extended ConfigObj
mov     dword ptr [ebp-0B0h], 728688C6h
mov     dword ptr [ebp-0ACh], 348C00E9h
mov     dword ptr [ebp-0A8h], 0E4D5AAB4h
mov     bl, 0A8h
mov     dword ptr [ebp-0A4h], 0A89E31F5h
mov     byte ptr [ebp-0A0h], 0
call   memcpy
;

```

Figure 3. Decryptor of the PUSHEBP functions in Xloader version 2.9

In Xloader version 4.3, there are still PUSHEBP encrypted functions. However, the tags identifying the start and the end of the encrypted code have changed, and now they appear to be random bytes. Figure 4 shows an example of an encrypted function in Xloader 4.3.

```

EncryptedFunc1 proc near
push   ebp
mov     ebp, esp
sub     esp, 2C0h
push   ebx
push   esi
EncryptedFunc1 endp ; sp-analysis failed
;
-----
db     6
db     2Fh ;
db     34h ;
db     0DBh ;
db     35h ;
db     0E4h
db     0D7h
db     7
db     0EDh
db     0ECh
db     9
db     0C2h

```

Figure 4. Encrypted PUSHEBP function (Xloader version 4.3)

Figure 5 shows the code that decrypts a PUSHEBP function (e.g., the function DecryptCriticalCodeType1_Set_909090909090), which accepts two encrypted tags and an ID value. Inside the decryptor, another 0x14 byte key is constructed dynamically in the sub-function init_key_encrypted_funcs, XORed with a DWORD (XOR key 1) and XORed again with the ID value passed as an argument and another hardcoded DWORD (XOR key 2). The resulting 0x14 byte key will be used to decrypt the encrypted code using Xloader's custom RC4 algorithm. The same RC4 key is also used to decrypt the encrypted TAG1 and TAG2, which are passed as arguments to the decryptor to derive the starting and ending tags that delimit the encrypted PUSHEBP function.

```

int __cdecl call_EncryptedFunc1(_DWORD *a1, int a2)
{
    int result; // eax
    unsigned int v3[8]; // [esp+4h] [ebp-30h] BYREF
    int v4[2]; // [esp+24h] [ebp-10h] BYREF
    int v5[2]; // [esp+2Ch] [ebp-8h] BYREF

    memset(v3, 0, sizeof(v3));
    v4[0] = 0x4F5B484D;
    v4[1] = 0x3304;
    v5[0] = 0x61524238;
    v5[1] = 0x320C;

    result = DecryptCriticalCodeType1_Set_909090909090(
        (int)a1,
        0x195457E2,
        (unsigned __int8 *)v4,
        (unsigned __int8 *)v5,
        v3);
    if ( result )
        EncryptedFunc1(a1, a2);
    return result;
}

EncryptedFunc1 proc near
push    ebp
mov     ebp, esp
sub     esp, 2C0h
push   ebx
push   esi
endp ; sp-analysis failed

;
db 6
db 2Fh
db 34h
db 00Bh
db 35h
db 0E4h
db 007h
db 7
db 0EDh
db 0ECh
db 9
db 0C2h
;

int __cdecl DecryptCriticalCodeType1_Set_909090909090(
    int arg_ConfigObj,
    int arg_id,
    unsigned __int8 *arg_tag_1,
    unsigned __int8 *arg_tag_2,
    unsigned int *arg_zerobuf)
{
    int result; // eax
    int v6; // edi
    int v7; // esi
    int i; // edi
    unsigned int v9; // esi
    _BYTE *j; // edi
    int v11; // [esp+4h] [ebp-2B0h] BYREF
    char v12[572]; // [esp+8h] [ebp-2ACh] BYREF
    int v13; // [esp+244h] [ebp-70h]
    int v14; // [esp+2ACh] [ebp-8h]
    unsigned int v15; // [esp+2B0h] [ebp-4h]

    v11 = 0;
    Sub_41B790_Init_Memory(v12, 0, 0x2A4u);
    result = Sub_414330_Set_ConstantValue(arg_ConfigObj, &v11);
    if ( result )
    {
        v6 = v13 - 0x10;
        v14 = *( _DWORD *) (arg_ConfigObj + 4);
        v15 = v13 - 0x10;
        init_key_encrypted_funcs((int)(arg_zerobuf + 2));
        xor_array_5_dwords(arg_zerobuf + 2, arg_id ^ 0x18276D36);
        decrypt_type2_rc4based(arg_tag_1, 6u, (int)(arg_zerobuf + 2));
        decrypt_type2_rc4based(arg_tag_2, 6u, (int)(arg_zerobuf + 2));
        v7 = 0;
        if ( v6 )
            func_start_id == rc4(tag1, init_key ^ xor_key_1 ^ arg_id ^ xor_key_2)
        {
            for ( i = v14; !Sub_41BA20_Compare_AsciiString(arg_tag_1, i, 6u); ++i )
            {
                if ( ++v7 >= v15 )
                    return 0;
            }
            *( _DWORD *)i = 0x90909090;
            *( _WORD *) (i + 4) = 0x9090;
            v9 = 4;
            for ( j = ( _BYTE *) (i + 6); v9 < v15; ++v9 )
            {
                func_end_id == rc4(tag1, init_key ^ xor_key_1 ^ arg_id ^ xor_key_2)
                if ( Sub_41BA20_Compare_AsciiString(arg_tag_2, (int)&j[v9], 6u) )
                    break;
            }
            *( _DWORD *)&j[v9] = 0x90909090;
            *( _WORD *)&j[v9 + 4] = 0x9090;
            decrypt_type2_rc4based(j, v9, (int)(arg_zerobuf + 2));
            arg_zerobuf[1] = functioned int * i;
        }
    }
}

BYTE __cdecl init_key_encrypted_funcs(int a1)
{
    int v2[8]; // [esp+0h] [ebp-20h] BYREF

    memset(&v2[5], 0, 12);
    v2[0] = 0xC4E5764F;
    v2[1] = 0x9138BC64;
    v2[2] = 0x2225EF0A;
    v2[3] = 0xD476BF6C;
    v2[4] = 0x4616C850;
    xor_array_5_dwords(v2, 0x18276D36);
    return Sub_41B710_Transfer_Data(a1, v2, 0x14);
}

```

Figure 5. PUSHEDBP function decryption code (Xloader version 4.3)

After the code is decrypted, the delimiter tags are replaced by 90 90 90 90 90 90 (NOP) opcodes. Figure 6 shows an encrypted function before and after being decrypted.

```

EncryptedFunc1 proc near
    push    ebp
    mov     ebp, esp
    sub     esp, 2C0h
    push    ebx
    push    esi
EncryptedFunc1 endp ; sp-analysis failed
;
    db     6      ; id of the function
    db     2Fh   ; looking like random
    db     34h   ; bytes
    db     0DBh
    db     35h ;
    db     0E4h
    db     0D7h
    db     7     ; encrypted code
    db     0EDh
    db     0ECh
    db     9
    db     0C2h

EncryptedFunc1 proc near
    var_2C0 = dword ptr -2C0h
    var_2BC = byte ptr -2BCh
    var_2B4 = dword ptr -2B4h
    var_2B0 = dword ptr -2B0h
    var_2AC = dword ptr -2ACh
    var_2A0 = byte ptr -2A0h
    var_A0 = dword ptr -0A0h
    var_9C = dword ptr -9Ch
    var_80 = dword ptr -80h
    var_78 = byte ptr -78h
    var_34 = byte ptr -34h
    var_18 = byte ptr -18h
    var_14 = byte ptr -14h
    var_10 = dword ptr -10h
    var_C = word ptr -0Ch
    var_8 = dword ptr -8
    var_4 = dword ptr -4
    arg_0 = dword ptr 8
    arg_4 = dword ptr 0Ch

    push    ebp
    mov     ebp, esp
    sub     esp, 2C0h
    push    ebx
    push    esi
    nop
    nop
    nop
    nop
    nop
    nop
    push    2A4h
    lea    eax, [ebp+var_2BC]
    push    0
    nush   eax

```

Figure 6. Example PUSHEDBP function decrypted (Xloader version 4.3)

Encrypted NO-PUSHEBP Functions

In Xloader version 4.3, a new type of encrypted function without the push ebp / mov ebp esp preamble has also been introduced. The limits of the encrypted code are located by searching for two tags that identify the start and the end of the block. Figure 7 shows the code responsible for determining the limits of a NO-PUSHEBP encrypted function.

```

if ( arg_0 == 0xFAB122D4 )
tag_start = 0;
else
tag_start = (arg_0 + 0x54EDD2C) ^ 0x1FF13A12;
v31 = 0;
v32 = 0;
v33 = 0;
v26 = 0xC88E9C7F;
v27 = 0xEB5536D;
v28 = 0xA937A82D;
v29 = 0xB2A470BC;
v30 = 0xFB82ADF5;
if ( v34 )
{
// find start of encrypted code
while ( *(DWORD *)&encrypted_code[v2] != tag_start )
{
if ( ++v2 >= v34 )
goto LABEL_22;
}
arg_0 = 63;
UTIL_Do_Init_Memory((char *)&arg_0, (char *)4);
v8 = 0;
if ( arg_0 == 0xFAB122D4 )
tag_end = 0;
else
tag_end = (_BYTE *)((arg_0 + 0x54EDD2C) ^ 0x1488B7F2);
encrypted_code += v2;
v9 = encrypted_code + 4;
v10 = 0;
v11 = v34 - v2;
v34 -= v2;
if ( v34 )
{
while ( *(BYTE *)&v9[v10] != tag_end )
{
if ( ++v10 >= v11 )
goto LABEL_18;
}
v8 = v10;
}
// decrypt layer 1
OBFUS_decrypt_type2_rc4based(encrypted_code + 4, v8, (int)&v1->sha1_pushhebp_blk_5);
}
}

BYTE * __cdecl OBFUS_Decryptor_Using_pushhebp_blk5_6_customrc4_multiplelayers(ConfigObj *)
{
int pushhebp_blk_5; // eax
int pushhebp_blk_6; // eax
char dst[123]; // [esp+Ch] [ebp-E4h] BYREF
int sha1_pushhebp_blk_5[26]; // [esp+88h] [ebp-68h] BYREF

dst[0] = 0;
UTIL_Init_Memory(&dst[1], 0, 0x79u);
pushhebp_blk_5 = OBFUS_get_pushhebp_blk_5();
OBFUS_FF_EncodeFunction((int)dst, pushhebp_blk_5 + 2, 0x78u);
pushhebp_blk_6 = OBFUS_get_pushhebp_blk_6();
OBFUS_FF_EncodeFunction((int)&configObj->field_50C, pushhebp_blk_6 + 2, 0x310u);
OBFUS_build_key_xor_with_configobj_1b0((int)configObj, &configObj->field_990);
OBFUS_malware_sha1_init(sha1_pushhebp_blk_5);
OBFUS_sha_Stir_Value(sha1_pushhebp_blk_5, dst, 0x78);
OBFUS_sha_Mix_60h_Value((int)sha1_pushhebp_blk_5);
UTIL_Transfer_Data((int)&configObj->sha1_pushhebp_blk_5, sha1_pushhebp_blk_5, 0x14);
}

```

| | | | | |
|----------------|----|-------------------------|----|---------|
| .text:1041AB73 | 5A | EncryptedCodeNoPushEBP1 | db | 5Ah |
| .text:1041AB74 | 8D | | db | 8Dh |
| .text:1041AB75 | 09 | | db | 9 |
| .text:1041AB76 | AA | | db | 0AAh |
| .text:1041AB77 | D5 | | db | 0D5h |
| .text:1041AB78 | 63 | | db | 63h ; c |
| .text:1041AB79 | 46 | | db | 46h ; F |
| .text:1041AB7A | C8 | | db | 0C8h |
| .text:1041AB7B | 81 | | db | 81h |
| .text:1041AB7C | 84 | | db | 084h |
| .text:1041AB7D | B8 | | db | 0B8h |
| .text:1041AB7E | 55 | | db | 55h ; U |
| .text:1041AB7F | 7A | | db | 7Ah ; v |

Figure 7. NO-PUSHEBP decryption code limit identification and layer 1 decryption (Xloader version 4.3)

The custom Xloader RC4 algorithm is again used to decrypt the encrypted code with two layers and two different keys. The encryption key for the first layer is calculated in another function and stored in the global structure ConfigObj (the value is the result of Xloader's custom SHA1 algorithm of the decrypted content of the PUSHEDBP data block number 5). The encryption key for the second layer is built on the fly: an initial key is built on the stack and XORed with a DWORD (XOR key), producing the final key (Xloader never hardcodes exact values including for encryption keys and delimiter tags). Figure 8 shows the code involved in the decryption of the second layer for one of the encrypted NO-PUSHEBP functions.

```

_id_1 = Do_Init_Memory_3F_if_0xFAB122D4_return_0_else_xor_a1_54edd2c(0x24A7508);
v5 = v3 == 0x10;
v6 = v3 - 0x10;
init_key_nopushebp2[0] = 0x5B922B40;
init_key_nopushebp2[1] = 0xECDF2CC6;
init_key_nopushebp2[2] = 0x13483169;
init_key_nopushebp2[3] = 0x6592F893;
init_key_nopushebp2[4] = 0xA3A6A840;
memset(&init_key_nopushebp2[5], 0, 0xC);
if ( !v5 )
{
while ( *(DWORD *)v2 + v24 != id_1 )
{
if ( ++v2 >= v6 )
goto LABEL_7;
}
xor_array_5_dwords(init_key_nopushebp2, 0x873600);
_id_2 = Do_Init_Memory_3F_if_0xFAB122D4_return_0_else_xor_a1_54edd2c(0x1D27B6C4);
DecryptCriticalCodeType2_noPUSHEBPHdr_set_90909090ec8b55(
(_BYTE *)v2 + v24 + 4,
(int)init_key_nopushebp2,
id_2,
v6 - v2,
1);
}
EL_7:
result = (char *)((int (__cdecl *) (int *)) EncryptedCodeNoPushEBP2)(a1);
}

unsigned int __cdecl DecryptCriticalCodeType2_noPUSHEBPHdr_set_90909090ec8b55(
_BYTE *buf,
int key,
int id_2,
unsigned int a4,
int a5)
{
unsigned int v5; // esi
unsigned int v6; // eax
int v8[2]; // [esp+8h] [ebp-8h] BYREF

v5 = 0;
v6 = 0;
if ( a4 )
{
while ( *(DWORD *)&buf[v6] != id_2 )
{
if ( ++v6 >= a4 )
goto LABEL_6;
}
v5 = v6;
}
LABEL_6:
decrypt_type2_rc4based(buf, v5, key);
if ( a5 )
{
a4 = 0x90909090;
v8[0] = 0xEC8B55;
v8[1] = 0;
Sub_41B710_Transfer_Data((int)buf - 3, v8, 3);
Sub_41B710_Transfer_Data((int)&buf[v5], &a4, 4);
}
return v5;
}

```

Figure 8. NO-PUSHEBP layer 2 decryption (Xloader version 4.3)

After the code is decrypted, the tag before the encrypted code is replaced by the opcodes EC 8B 55 (push ebp / mov ebp esp function preamble). The tag after the encrypted code is replaced by 90 90 90 90 (NOP) opcodes.

Encrypted Configuration

The most important parameters of Xloader's configuration are stored in the PUSHEDB encrypted data blocks or calculated from hardcoded constants (that are also obfuscated).

Encrypted Strings

The encrypted strings in Xloader are stored in the PUSHEDB data block 1. All the PUSHEDB data blocks have to be decrypted with the custom buffer decryption algorithm as explained before. Once the block is decrypted, the result is a sequential list of items that have the following format:

```
struct encrypted_string {
    BYTE length;
    BYTE content[length];
}
```

Each string is decrypted with the custom Xloader RC4 algorithm and an encryption key stored at offset 0x990 in the ConfigObj. This RC4 key is generated in the function shown in Figure 9.

```
__BYTE *__cdecl OBFUS_GenKeys_ConfigObj(ConfigObj *configObj)
{
    int pushebp_blk_5; // eax
    int pushebp_blk_6; // eax
    char pushebp_blk_5_dec[123]; // [esp+Ch] [ebp-E4h] BYREF
    int cur_shal[26]; // [esp+88h] [ebp-68h] BYREF

    pushebp_blk_5_dec[0] = 0;
    UTIL_Init_Memory(&pushebp_blk_5_dec[1], 0, 0x79u);
    pushebp_blk_5 = OBFUS_get_pushebp_blk_5();
    OBFUS_FF_EncodeFunction((int)pushebp_blk_5_dec, pushebp_blk_5 + 2, 0x78u);
    pushebp_blk_6 = OBFUS_get_pushebp_blk_6();
    OBFUS_FF_EncodeFunction((int)&configObj->key_50c, pushebp_blk_6 + 2, 0x310u);
    OBFUS_build_key_xor_with_configobj_1b0((int)configObj, &configObj->key_990);
    OBFUS_malware_shal_init(cur_shal);
    OBFUS_sha_Stir_Value(cur_shal, pushebp_blk_5_dec, 0x78);
    OBFUS_sha_Mix_60h_Value((int)cur_shal);
    UTIL_Transfer_Data((int)&configObj->shal_pushebp_blk_5, cur_shal, 0x14);
    OBFUS_decrypt_type2_rc4based(&configObj->key_50c, 0x310u, (int)&configObj->shal_pushebp_blk_5);
    OBFUS_malware_shal_init(cur_shal);
    OBFUS_sha_Stir_Value(cur_shal, &configObj->key_990, 0x14);
    OBFUS_sha_Mix_60h_Value((int)cur_shal);
    OBFUS_decrypt_type2_rc4based(&configObj->key_50c, 0x310u, (int)cur_shal);
    OBFUS_malware_shal_init(cur_shal);
    OBFUS_sha_Stir_Value(cur_shal, &configObj->key_50c, 0x310);
    OBFUS_sha_Mix_50h_Value((int)cur_shal);
    return OBFUS_decrypt_type2_rc4based(&configObj->key_990, 0x14u, (int)cur_shal);
}

__DWORD *__cdecl OBFUS_build_key_xor_with_configobj_1b0(int a1, __DWORD *a2)
{
    __BYTE v3[4]; // [esp+4h] [ebp-20h] BYREF
    int v4; // [esp+8h] [ebp-1Ch]
    int v5; // [esp+Ch] [ebp-18h]
    int v6; // [esp+10h] [ebp-14h]
    int v7; // [esp+14h] [ebp-10h]
    int v8; // [esp+18h] [ebp-Ch]
    int v9; // [esp+1Ch] [ebp-8h]
    int v10; // [esp+20h] [ebp-4h]

    v8 = 0;
    v9 = 0;
    v10 = 0;
    *(_DWORD *)v3 = 0xE4011093;
    v4 = 0xB305D4FE;
    v5 = 0x5BF5E872;
    v6 = 0xFF05FC72;
    v7 = 0xD30E6C1F;
    UTIL_Transfer_Data((int)a2, v3, 0x14);
    return OBFUS_xor_array_5_dwords(a2, *(__DWORD *)v3);
}

.text:10418DB6 C7 86 B0 01 00 00 CC C0 FC 03 mov [esi+ConfigObj.field_1B0], 3FCC0CCh
```

Figure 9. Generation of the RC4 key for encrypted strings (Xloader version 4.3)

ThreatLabz has reproduced this algorithm to decrypt the encrypted strings in Xloader 4.3 in Python. The code is available in our GitHub repository [here](#).

Encrypted C2s

The Xloader configuration contains a C2 that is stored separately from another list of C2 domains. The C2 that is stored separately was thought to be Xloader's real C2 and the other C2s were used as decoys. However, in more recent versions of Xloader, real C2s are likely hidden among the list of decoy C2s. In fact, the author behind Xloader has made significant efforts to protect the list of C2s that were previously thought to be decoys.

Hardcoded C2

The code shown in Figure 10 is responsible for decrypting the hardcoded Xloader C2.

```

v10 = 0;
memset(v27, 0, sizeof(v27));
v19 = 0;
v11 = 0;
v8 = 0;
UTIL_Init_Memory(buf, 0, 0x27u);
UTIL_Init_Memory(MemoryTop, 0, 0x50u);
*(DWORD*)(a1 + 0x1B0) = 0x3FCC0CC;
v5 = OBFUS_get_pushebp_blk_4();
OBFUS_FF_EncodeFunction((int)&v19, v5 + 2, 0x14u);
OBFUS_build_key_xor_with_configobj_1b0(a1, &v11);
v6 = OBFUS_get_pushebp_blk_5();
OBFUS_FF_EncodeFunction((int)&v8, v6 + 2, 0x78u);
OBFUS_customrc4_with_sha1_decrypted_pushebp_blk1_as_key(&v8);
UTIL_Transfer_Data((int)v27, &v8, 22);
OBFUS_decrypt_type2_rc4based(v27, 0x16u, (int)&v11);
OBFUS_decrypt_type2_rc4based(v27, 0x16u, (int)&v19);
if ( !a3 )
    HIWORD(v28) = 0;
if ( *(DWORD*)v27 == '.www' )
    v3 = 4;
v7 = UTIL_strlen(&v27[v3]);
UTIL_Transfer_Data(a2, &v27[v3], v7);

```

Figure 10. Hardcoded C2 decryption (Xloader version 4.3)

The code in Figure 10 combines a set of operations based on Xloader's various encryption algorithms and the data stored in the PUSHEDBP data blocks to generate the encryption key necessary to decrypt the hardcoded C2 (which is stored in the PUSHEDBP data block 5).

C2 List

As previously mentioned, there is another list of C2s that may contain decoy C2s and real C2s. In Formbook and in earlier versions of Xloader, these were stored as an encrypted string with no additional layers of encryption. In Xloader 2.9, the developers introduced an additional custom RC4 layer and Base64 encoding for the C2 list as shown in Figure 11.

XLOADER version 2.9:

```
for ( i = 0; i < 0x1C40; i += 0x388 )
{
    if ( ConfigObjExt[0xB34] != v7 && ConfigObjExt[0x58E] != v7 )
    {
        memset(v26, '.');
        memset((char *)&v22, '@');
        v19 = 0;
        Sub_41B790_Init_Memory(v20, 0, 0x206u);
        memset((char *) (i + ConfigObjExt[0xAF7]), 0x388u);
        *(_DWORD *) (ConfigObjExt[0xAF7] + i + 0x2E4) = (((_BYTE)v7 - 1) * (unsigned __int8)sub_41C7A0()) & 1;
        v9 = strlen(&v24);
        memcpy(ConfigObjExt[0xAF7] + i + 840, &v24, v9);
        memcpy((int)v26, v32, 4);
        current_pos_decrypted_domain_table = &v26[strlen(v26)];
        StringsDecryptor2((int)ConfigObjExt, (int)&v22, (unsigned __int8)*(&rand_table + rand_table_index));
        v11 = base64_decode(current_pos_decrypted_domain_table, &v22);
        decrypt_type2_rc4based(current_pos_decrypted_domain_table, v11, (int)(ConfigObjExt + 0x481));
        v12 = strlen(v26);
        memcpy(ConfigObjExt[0xAF7] + i + 0x2F0, v26, v12);
    }
}
```

Additional layers

XLOADER version 2.5:

```
for ( i = 0; i < 0x1C40; i += 0x388 )
{
    if ( *(_DWORD *) (ConfigObjExt + 0x1170) != v7 )
    {
        domemset((int)v21, 46);
        memset(v18, 0, sizeof(v18));
        domemset(i + *(_DWORD *) (ConfigObjExt + 0x14A4), 0x388);
        *(_DWORD *) (*(_DWORD *) (ConfigObjExt + 0x14A4) + i + 0x40) = (((_BYTE)v7 - 1) * (unsigned __int8)TickCountWithoutApi()) & 1;
        v9 = strlen(v20);
        memcpy(*(_DWORD *) (ConfigObjExt + 0x14A4) + i + 135, v20, v9);
        memcpy((int)v21, v28, 4);
        v10 = strlen(v21);
        StringsDecryptor2(ConfigObjExt, (int)&v21[v10], (unsigned __int8)*(&v24 + v7));
        v11 = strlen(v21);
        memcpy(i + *(_DWORD *) (ConfigObjExt + 0x14A4), v21, v11);
    }
}
```

Figure 11. Additional encryption layer for the C2 list (Xloader version 2.9)

In Figure 11, the function StringsDecryptor2 decrypts the first layer of the encrypted strings. In version 2.9, an additional Base64 layer is decoded followed by a layer of custom RC4 decryption. In Xloader version 4.3, they have added an additional encryption layer to this C2 list. Figure 12 shows the code responsible for decrypting these new layers.

XLOADER version 4.3:

```

do
{
if ( configobj->field_2B68 != v8 )
{
if ( *((_BYTE *)stringID )
{
UTIL_Do_Init_Memory(MemoryTop, (char *)'.');
UTIL_Do_Init_Memory((char *)&output_stagel_domain, (char *)'@');
v21 = 0;
UTIL_Init_Memory(v22, 0, 0x206u);
UTIL_Do_Init_Memory((char *) (v10 + configobj->field_44E8), (char *)0x394);
*((_DWORD *) (v10 + configobj->field_44E8 + 0x38C) = (((_BYTE)v8 - 1) * (unsigned __int8)UTIL_RNG()) & 1;
v11 = UTIL_strlen(&v26);
UTIL_Transfer_Data(v10 + configobj->field_44E8 + 0x334, &v26, v11);
UTIL_Transfer_Data((int)MemoryTop, v36, 4);
output_stagel_domain_nobase64 = &MemoryTop[UTIL_strlen(MemoryTop)];
OBFUS_StringDecryptor2_with_extra_C2_layer(
configobj,
(int)&output_stagel_domain,
*((unsigned __int8 *)stringID++);
v13 = OBFUS_base64_decode(output_stagel_domain_nobase64, &output_stagel_domain);
OBFUS_decrypt_type2_rc4based(output_stagel_domain_nobase64, v13, (int)&configobj->key_1_decoy_C2s);
}
}
}

int __cdecl OBFUS_StringDecryptor2_with_extra_C2_layer(ConfigObj *configObj
{
unsigned __int8 buf; // [esp+4h] [ebp-104h] BYREF
char MemoryTop[259]; // [esp+5h] [ebp-103h] BYREF

buf = 0;
UTIL_Init_Memory(MemoryTop, 0, 0x103u);
OBFUS_StringDecryptor2(configObj, (int)&buf, string_id + 0x49);
return OBFUS_base64_customRC4_reencode_base64_hardcoded_key_for_C2s(
string_id + 0x49,
&buf,
output_stagel_domain,
string_id + 0x49);
}

v6[99] = 0x623048C;
v6[100] = 0x31F1BE8;
v6[101] = 0x109F8FB8;
v6[102] = 0x6398440;
v6[103] = 0x281E0CB;
v6[104] = 0x3840F83;
v6[105] = 0x18860EAB;
v6[106] = 0x12A6270;
v6[107] = 0x8BF8474;
v6[108] = 0x1B234A6;
v6[109] = 0x1960FF46;
v6[110] = 0x1969A55C;
v6[111] = 0x9AB58E;
v6[112] = 0x3E41B75;
v6[113] = 0x15CE18BF;
v6[114] = 0x1AE2648;
v6[115] = 0xF87FE46;
v6[116] = 0xF0AB58;
v6[117] = 0x281C30EF;
v6[118] = 0x2302BFE;
v6[119] = 0x2015A;
v6[120] = 0x15141D02;
v6[121] = 0x3110ACB5;
v6[122] = 0x4AE0C32;
v6[123] = 0x2D7C7818;
v6[124] = 0xCFF7C22;
v6[125] = 0x119A900;
v6[126] = 0x27D1A926;
v6[127] = 0x5C2A5E3;
v6[128] = 0x22A3C37;
v6[129] = 0x43E83A;
v6[130] = 0x9F8A0FC;
v6[131] = 0x25D9C61;
v6[132] = 0x843D12A;
v6[133] = 0x26B03512;
v6[134] = 0xCAD474E;
v6[135] = 0x3E71194;
v6[136] = 0x3F757E4;
BufOut = (_BYTE *)v6[a4_StringID];
return OBFUS_base64_customRC4_reencode_base64(cpBufOut, Buf

```

Additional decoy C2 encryption layer

Table of 4-bytes length keys. Each position in the table is a key for one of the encrypted strings

It decodes the first BASE64 layer, decrypts the first layer using the key in the table and custom RC4, and encodes in BASE64 again

Figure 12. New encryption layer for Xloader's C2 list (Xloader version 4.3)

In the new version, the C2 list is first Base64 decoded and a custom RC4 layer is decrypted. A table of 4 byte keys is built on the stack. Each position of the table corresponds to a C2. Once decrypted, this custom RC4 layer is Base64 encoded again. After the new additional decryption layer is complete, Xloader decrypts the same layers as version 2.9: decoding the Base64 layer again and decrypting an additional custom RC4 layer with a key stored in a sub-structure of the ConfigObj. The way that this key (for the last RC4 layer) is generated has also changed in Xloader 4.3. Figure 13 shows the code generating the RC4 key for the last encryption layer of the C2 list.

```

.text:1042B603 8B 75 08          mov     esi, [ebp+configobj]
.text:1042B606 8B BE 90 00 00 00  mov     edi, [esi+ConfigObj.configObj_Extended1]
.text:1042B60C 33 C0             xor     eax, eax
.text:1042B60E 89 85 40 FF FF FF  mov     [ebp+var_C0], eax
.text:1042B614 89 85 44 FF FF FF  mov     [ebp+var_BC], eax
.text:1042B61A 89 85 48 FF FF FF  mov     [ebp+var_B8], eax
.text:1042B620 C7 86 68 2B 00 00 0A 00 00 00  mov     dword ptr [esi+2B68h], 0Ah
.text:1042B62A C7 85 2C FF FF FF 3F 49 F3 9C  mov     dword ptr [ebp+var_build_key1_decoy_C2s], 9CF3493Fh
.text:1042B634 C7 85 30 FF FF FF 63 1B A2 7B  mov     [ebp+var_D0], 7BA21B63h
.text:1042B63E C7 85 34 FF FF FF B0 43 DA 07  mov     [ebp+var_CC], 7DA4380h
.text:1042B648 C7 85 38 FF FF FF 47 C1 ED 37  mov     [ebp+var_C8], 37EDC147h
.text:1042B652 C7 85 3C FF FF FF A4 21 23 E6  mov     [ebp+var_C4], 0E62321A4h
.text:1042B65C 8B 97 8C 20 00 00  mov     edx, [edi+208Ch]
.text:1042B662 52              push   edx
.text:1042B663 8D 85 2C FF FF FF  lea   eax, [ebp+var_build_key1_decoy_C2s]
.text:1042B669 50              push   eax
.text:1042B66A B3 A4          mov     bl, 0A4h
.text:1042B66C E8 E2 EF FE FF  call  OBFUS_xor_array_5_dwords
.text:1042B66C

.text:104169A5 C7 87 D8 03 00 00 DC E0 C9 22  mov     dword ptr [edi+3D8h], 22C9E0DCh
.text:104169AF C7 87 8C 00 00 00 9C 83 43 05  mov     dword ptr [edi+8Ch], 543839Ch; ConfigObjExtended = *(DWORD*)&ConfigObj[0x90]
.text:104169AF                                     ; edi = ConfigObjExtended + 0x2000
.text:104169AF                                     ; *(DWORD*)(edi + 0x8c) = 0x543839C -> key xor DECOY C2s key
.text:104169B9 C7 87 EC 04 00 00 90 85 68 09  mov     dword ptr [edi+4ECh], 9688590h

```

The key to decrypt the last decoy C2s' custom RC4 layer is built on the stack and xored with a DWORD previously kept in the ConfigObjExtended

Figure 13. Key generation for the final encryption layer of the C2 list (Xloader version 4.3)

As shown in Figure 13, the key is built on the stack and it is XORed with a value from the ConfigObj that was initialized previously in a different part of the code. Once this last layer is decrypted, the plaintext C2s are obtained.

Branch ID and Version Number

In previous versions, the Xloader branch ID and version number were sent in the registration packet to the C2. The format of the registration packet (before the last two RC4 layers) was the following:

XLNG Bot ID Version Number Operating System Base64(Username)

XLNG is the tag for the Xloader branch (FBNG was the branch ID for Formbook).

In Xloader version 4.3, the registration packet sent to the C2 includes an additional encryption layer as shown in Figure 14.

```

mov r11,qword ptr ds:[rsi+4520]
mov edx,dword ptr ss:[rbp+6F]
mov rcx,qword ptr ss:[rbp+67]
lea r8,qword ptr ds:[r12+r11+27A]
call DC53382
mov edx,dword ptr ss:[rbp+6F]
    
```

[rbp+67]:"PKT2:xkJoHTgUiBSbWary2TG/+6BC2yQ="

rc4

Figure 14. Xloader 4.3 Registration packet with additional PKT2 layer

This new encryption layer is marked with the tag PKT2. Communications are performed in the context of explorer.exe (previously injected). However, this registration packet is built in the first injected process (a hollow process) and copied to the context of explorer together with the rest of the injected code. That first injected process exits after injecting into explorer, so the registration packet under the last encryption layer marked with the PKT2 tag is no longer in plaintext after the first injected process terminates.

The PKT2 packet is built in one of the NO-PUSHEBP encrypted functions. That function is decrypted and executed in the context of the first injected process. The code first builds a string with the same format as the registration packet in previous Xloader versions as shown in Figure 15.

```

0015DD9C 58 4C 4E 47 3A 39 46 39 41 32 44 34 00 34 2E 33 XLNG:9F9A2D4.4.3
0015DDAC 3A 57 69 6E 64 6F 77 73 20 31 30 20 45 6E 74 65 :Windows 10 Ente
0015DDBC 72 70 72 69 73 65 20 78 36 34 3A 00 00 00 00 00 rprise x64:.....
0015DDCC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
    
```

Figure 15. Registration packet with the new PKT2 encryption layer

However, as we can see in Figure 15, Xloader 4.3 introduces a NULL character separating the bot ID and the malware version number. This added NULL byte is likely a coding error.

Figure 16 shows how the first registration packet is constructed (marked with XLNG tag) and encrypted with RC4 and encoded with Base64, and then concatenated to the PKT2 tag to generate the final registration packet.

```

mov     dword ptr [ebp+var_44], '2TKP'; PKT2
mov     [ebp+var_44+4], 3Ah; ':'
call    OBFUS_xor_array_5_dwords
push    5; int
lea     ecx, [ebp+var_44]
push    ecx; _BYTE *
lea     esi, [edi+50Ch]
push    esi; int
call    UTIL_Transfer_Data;
; arg_0:*Destination, arg_4:*Source, arg_8:ByteCount

mov     edx, [ebp+arg_0]
push    5; int
push    edx; _BYTE *
lea     eax, [ebp+var_42B+303h]
push    eax; int
call    UTIL_Transfer_Data;
; arg_0:*Destination, arg_4:*Source, arg_8:ByteCount

mov     ecx, [ebp+ptr_version_xored_3c]
xor     [ebp+var_42B+303h], 3Ch; decrypt branch
xor     [ebp+var_127], 3Ch
xor     [ebp+var_126], 3Ch
xor     [ebp+var_125], 3Ch
xor     [ebp+var_124], 3Ch
push    4; int
push    ecx; _BYTE *
lea     edx, [ebp+cp_version]
push    edx; int
call    UTIL_Transfer_Data;
; arg_0:*Destination, arg_4:*Source, arg_8:ByteCount

xor     byte ptr [ebp+cp_version], 3Ch; decrypt version
xor     byte ptr [ebp+cp_version+1], 3Ch
xor     byte ptr [ebp+cp_version+2], 3Ch
xor     byte ptr [ebp+cp_version+3], 3Ch
lea     eax, [ebp+var_42B+303h]
add     esp, 48h
push    eax; eax -> XLNG:9F9A2D4 \x00 4.3:Windows 10 Enterprise x64:
call    UTIL_strlen
push    eax; int
lea     ecx, [ebp+var_42B+303h]
push    ecx; _BYTE *
lea     edx, [ebp+var_42B+7Fh]
push    edx; int
mov     [ebp+arg_0], eax
call    UTIL_Transfer_Data;
; arg_0:*Destination, arg_4:*Source, arg_8:ByteCount

push    ebx
lea     eax, [ebp+var_42B+103h]
push    eax; eax -> YNRtaW4= (base64 'admin')
lea     ecx, [ebp+var_42B+7Fh]
push    ecx; ecx -> XLNG:9F9A2D4
call    UTIL_Transfer_data_after_set_null; Built string: "XLNG:9F9A2D4YNRtaW4="
mov     ebx, [ebp+arg_0]
add     ebx, [ebp+var_A4]
lea     ecx, [ebp+var_64]
push    ecx; rc4 key
lea     edx, [ebp+var_42B+7Fh]
push    ebx
push    edx; edx -> "XLNG:9F9A2D4YNRtaW4="
call    OBFUS_do_Rc4_InitByteTable
push    ebx
lea     eax, [ebp+var_42B+7Fh]
push    eax
lea     ecx, [edi+511h]
push    ecx
call    OBFUS_Encode_Base64;
; arg_0:controlBlock+0x14a8, arg_4:*AsciiString, arg_c:StringLength
push    esi; esi -> "PKT2:xxJoHTGuiBSbWary2TG/+6BC2yQ=" (registration packet)
call    UTIL_strlen

mov     esi, [eop+arg_0]
mov     edi, [esi+90h]
lea     eax, [esi+ConfigObj.version_xored_3c_byte1];
; Python>xor(b"\x08\x12\x0f\x06", b"\x3c\x08\x04\x03")
mov     cl, 6
push    81h; ByteSize
mov     [esi+ConfigObj.version_xored_3c_byte4], cl
mov     [ebp+ptr_version_xored_3c], eax
mov     byte ptr [eax], 8
lea     ecx, [esi+48h]
lea     ecx, [ebp+var_127]
push    ebx; InitData
push    ecx; MemoryTop
mov     word ptr [ebp+var_74], 3Ah; ':'; ':'
mov     dword ptr [ebp+var_74+2], ebx
mov     [ebp+var_6E], ebx
mov     [ebp+var_6A], ebx
mov     [ebp+var_66], bx
mov     word ptr [esi+ConfigObj.version_xored_3c_byte2], 0F12h
[ebp+arg_0], eax
mov     byte ptr [eax], 64h; 'd'
mov     dword ptr [esi+49h], 67B7270h
mov     [ebp+var_64], 32826C3Ch
mov     [ebp+var_60], 349C5D0Ch
mov     [ebp+var_5C], 80CCF0B1h
mov     [ebp+var_58], 0E55329D8h
mov     [ebp+var_54], 5D96F733h
mov     [ebp+var_50], ebx
mov     [ebp+var_4C], ebx
mov     [ebp+var_48], ebx
add     edi, 2000h

```

Figure 16. Xloader version 4.3 registration packet construction

However, because of the coding error previously mentioned (an extra NULL character after the bot ID) the final registration data contains just two fields for the XLNG branch and bot ID as shown below:

XLNG Bot ID

The PKT2 tag is then prepended with the RC4 and Base64 encoded data as follows:

PKT2 RC4_BASE64(registration_data)

As a result, the version_number, operating_system and user_name is never sent to the C2. This bug will likely be fixed in future versions of the malware.

Figure 16 also shows that the branch ID and version number are no longer hardcoded unlike previous versions, with the encrypted version number and branch ID decrypted with an XOR key (0x3c).

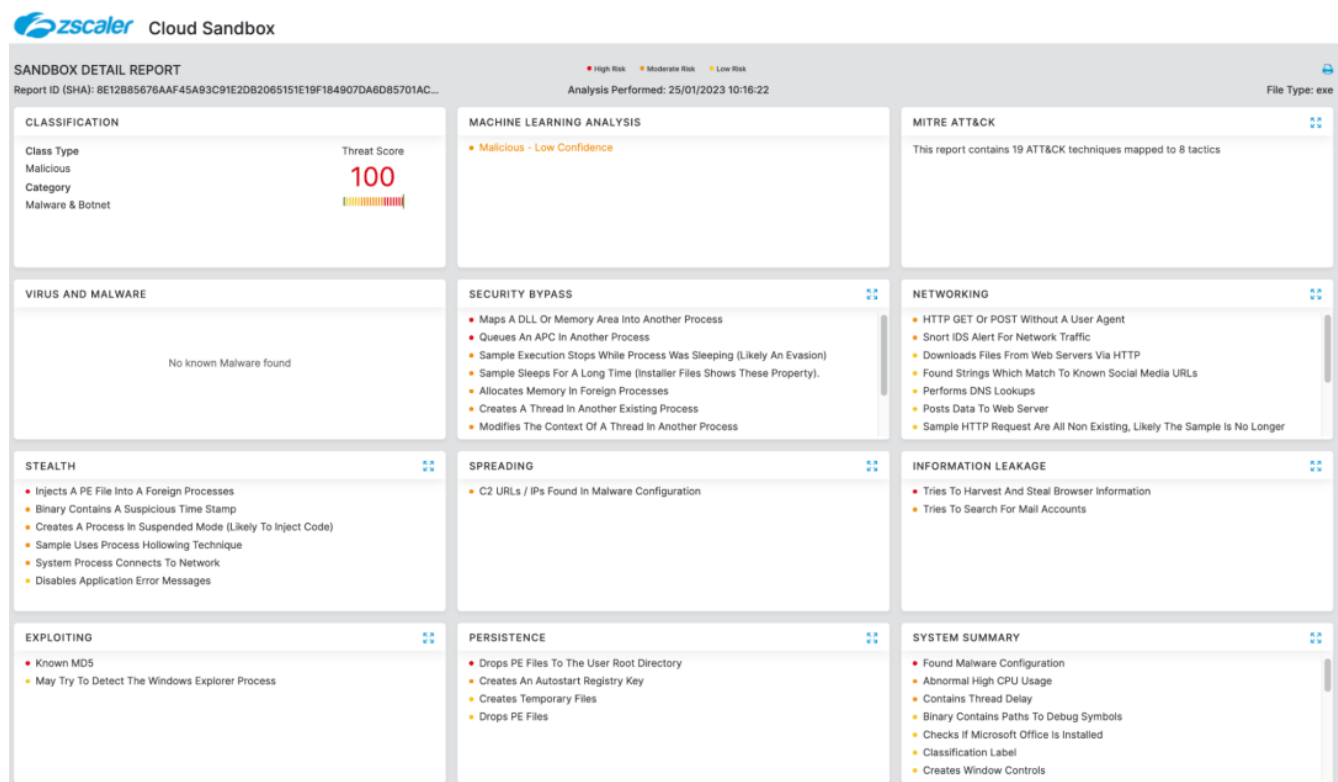
Tools

Zscaler ThreatLabz has developed an IDA script to decrypt the Xloader's encrypted code. The code is available in our GitHub tools repository [here](#).

Conclusion

Since its inception, the Formbook and Xloader malware families has been a prominent threat. The threat actors behind it continue to update and improve the malware code in an effort to hinder analysis. In the most recent version, the threat actors increased the level of complexity yet again with additional layers of encryption for critical parts of the code and important data. However, Zscaler researchers have been able to unravel these obfuscation layers and analyze the key components of the malware code.

Cloud of Sandbox Detection



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

Indicators of Compromise

| Variant | Version | SHA256 | Botnet |
|---------|---------|--|--------|
| XLoader | 4.3 | 9e1b4f2d408e187ca641c0c16269069d0acabe5ae15514418726fbc720b33731 | 6qne |

| Variant | Version | SHA256 | Botnet |
|---------|---------|--|--------|
| XLoader | 4.3 | f55ce0741ed4615bae5646c644b3a971323ac344b12693495d5749c688d5d489 | 6qne |
| XLoader | 4.3 | 3bd86f3906f59f627bf65664d2bfacf37a29dbaafae601baf5eeb544396f26c | 6qne |
| XLoader | 3.9 | 8e12b85676aaf45a93c91e2db2065151e19f184907da6d85701ac3b13d0e6052 | nvp4 |
| XLoader | 3.9 | 6a726fb5c93adbae0f3061b40b19745587c0114deb86bd72c90acdd69242cbe0 | nvp4 |

Network Indicators

| Type | Domain |
|--------------------|--------------------------|
| Harcoded C2 domain | jourmoe[.]com |
| C2 list domain | 060jinbo[.]com |
| C2 list domain | 10086253[.]vip |
| C2 list domain | 117ygh9x[.]com |
| C2 list domain | 365-8119[.]com |
| C2 list domain | 365heji[.]com |
| C2 list domain | 4tx[.]ru |
| C2 list domain | 667fm[.]com |
| C2 list domain | 991-touring[.]info |
| C2 list domain | abttt[.]win |
| C2 list domain | adacaranya[.]com |
| C2 list domain | allforfun[.]online |
| C2 list domain | allison2patrick[.]online |
| C2 list domain | applicationsdown[.]store |

| Type | Domain |
|----------------|-----------------------------|
| C2 list domain | apsocreto[.]online |
| C2 list domain | avdeeva[.]info |
| C2 list domain | betfury-platform[.]net |
| C2 list domain | bilpoinaat[.]net |
| C2 list domain | bocc[.]live |
| C2 list domain | bookinbournemouth[.]co[.]uk |
| C2 list domain | botanica-online[.]ru |
| C2 list domain | byfuture[.]biz |
| C2 list domain | canlicerrahi[.]xyz |
| C2 list domain | ceu84g[.]com |
| C2 list domain | chiyiqian[.]net |
| C2 list domain | christmatoy[.]com |
| C2 list domain | cinemamaxz[.]com |
| C2 list domain | coffeelectro[.]online |
| C2 list domain | dedmorozvidos[.]store |
| C2 list domain | difozaa[.]life |
| C2 list domain | dugebitv4[.]xyz |
| C2 list domain | eatgre[.]wiki |
| C2 list domain | expertponto[.]com |
| C2 list domain | farmanow[.]xyz |

| Type | Domain |
|----------------|-----------------------------|
| C2 list domain | flippingfrenzy[.]com |
| C2 list domain | g2fm[.]co[.]uk |
| C2 list domain | ginaandhipa[.]com |
| C2 list domain | graciesvoice[.]info |
| C2 list domain | guzmanmodels[.]com |
| C2 list domain | habka[.]online |
| C2 list domain | hal-skincare[.]com |
| C2 list domain | hiufouwnwk[.]shop |
| C2 list domain | hjiqa[.]com |
| C2 list domain | huifeng-tech[.]com |
| C2 list domain | identowel[.]com |
| C2 list domain | inigrey[.]com |
| C2 list domain | ituyiut[.]wang |
| C2 list domain | jimtrosper[.]com |
| C2 list domain | kajainterior[.]com |
| C2 list domain | loaddown[.]vip |
| C2 list domain | mgconsultantlogistics[.]com |
| C2 list domain | myif471ok9ipidk2kkl[.]xyz |
| C2 list domain | najdlegend1[.]com |
| C2 list domain | nnhuigou[.]com |

| Type | Domain |
|----------------|----------------------------|
| C2 list domain | ogeif[.]app |
| C2 list domain | poweroffer[.]net |
| C2 list domain | realtxt[.]co[.]uk |
| C2 list domain | seeword[.]site |
| C2 list domain | solutionsquik[.]net |
| C2 list domain | themas5erofssuepnse[.]cyou |
| C2 list domain | uevj[.]win |
| C2 list domain | vowlashes[.]co[.]uk |
| C2 list domain | wanknumbers[.]co[.]uk |
| C2 list domain | wsavxrg[.]shop |
| C2 list domain | zzaen[.]com |

References

Get the latest Zscaler blog updates in your inbox



By submitting the form, you are agreeing to our [privacy policy](#).