Technical Analysis of Xloader Versions 6 and 7 P1 | ThreatLabz

By ThreatLabz

Published: 2025-01-27 · Archived: 2026-04-05 15:02:37 UTC

In the following sections, we provide a detailed analysis of Xloader, focusing on the malware's behavior, obfuscation, and anti-analysis techniques.

Behavior

Persistence

Xloader establishes persistence by making a copy of itself in a subdirectory under `%APPDATA%` (or `%PROGRAMFILES%` if the user has sufficient privileges) with the following format:

- `%APPDATA%\\.exe`

Xloader then adds an entry in the Windows registry, either under the `Run` key or, if that fails, under the `Policies` key as follows:

- `\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\`
- `\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run`

Xloader's entry is placed in either the `HKCU` (`HKEY_CURRENT_USER`) or HKLM (`HKEY_LOCAL_MACHINE`) registry hive, based on the user's privileges. The name of the entry is randomly generated with 5-12 uppercase alphanumeric characters. The registry value will then be set to the path of the Xloader executable in the `%APPDATA%` or `%PROGRAMFILES%` directory.

Process injection

The figure below is a high-level view of how Xloader injects into multiple processes to evade antivirus and endpoint security software.

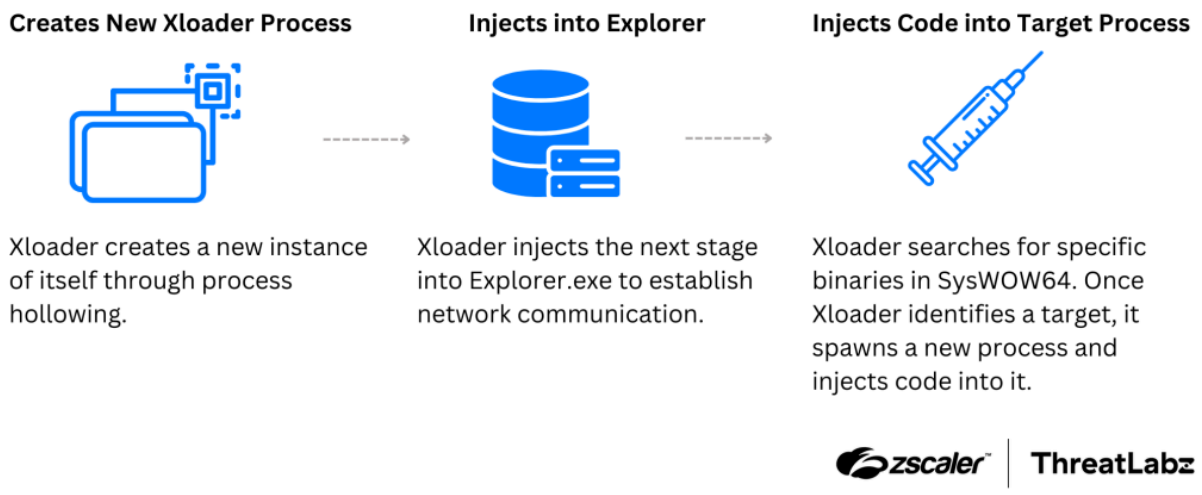

Figure 1: A high-level view of how Xloader injects malicious code into target processes.

Xloader first creates a new instance of its own executable through process hollowing. Next, Xloader injects the next stage into the `explorer.exe` process to establish network communication. In this case, Xloader uses the asynchronous procedure call (APC) queue technique to inject an x64 shellcode. Xloader uses the native API `NtQueueApcThread` instead of higher-level APIs, likely to evade antivirus hooks. The original and hollowed Xloader processes will then terminate.

Finally, Xloader launches an executable file in the SysWOW64 Windows directory that will also be targeted for code injection. Xloader uses a combination of the Windows API functions including `CreateProcessInternal`, `NtCreateSection`, `NtMapViewOfSection`, and `NtResumeThread` to inject code into the remote target process. The code injected into `explorer.exe` and the code injected in the SysWOW64 process run concurrently and use shared memory sections to communicate.

The list of target executable filenames in the SysWOW64 directory varies across different samples and versions. In one sample of Xloader version 6.2, the list of executables included the following:

- `SearchFilterHost.exe`
- `Isv.exe`
- `UserAccountControlSettings.exe`
- `systeminfo.exe`
- `SyncHost.exe`
- `print.exe`
- `sdiagnhost.exe`
- `fixmapi.exe`
- `msiexec.exe`
- `takeown.exe`
- `systray.exe`
- `net1.exe`

In a sample from Xloader 7.5, the following target executable filenames were identified:

- `nslookup.exe`
- `srdelayed.exe`
- `makecab.exe`
- `setx.exe`
- `runonce.exe`
- `auditpol.exe`
- `notepad.exe`
- `setupugc.exe`
- `AtBroker.exe`
- `RMActivate_isv.exe`
- `mountvol.exe`
- `dfrgui.exe`

Xloader will choose the first entry in the executable filename list. However, if the executable is not found in the Windows SysWOW64 directory, it will then proceed to the next entry in the list until one of the executables is located.

Code obfuscation

In the following sections, we examine Xloader's custom encryption and obfuscation layers.

Previous versions of Xloader had two types of encrypted functions that we refer to as the following:

- `NOPUSHEBP` : The function's entire code is encrypted.
- `PUSHEBP` : These functions start with the well-known *push ebp* prologue followed by encrypted code.

Many of Xloader's encryption layers continue to use an encryption algorithm that uses a combination of RC4 and two rounds of adjacent byte subtraction. However, in Xloader versions 6 and 7, an additional encryption layer has been added to the `NOPUSHEBP` functions as shown in the figure below.

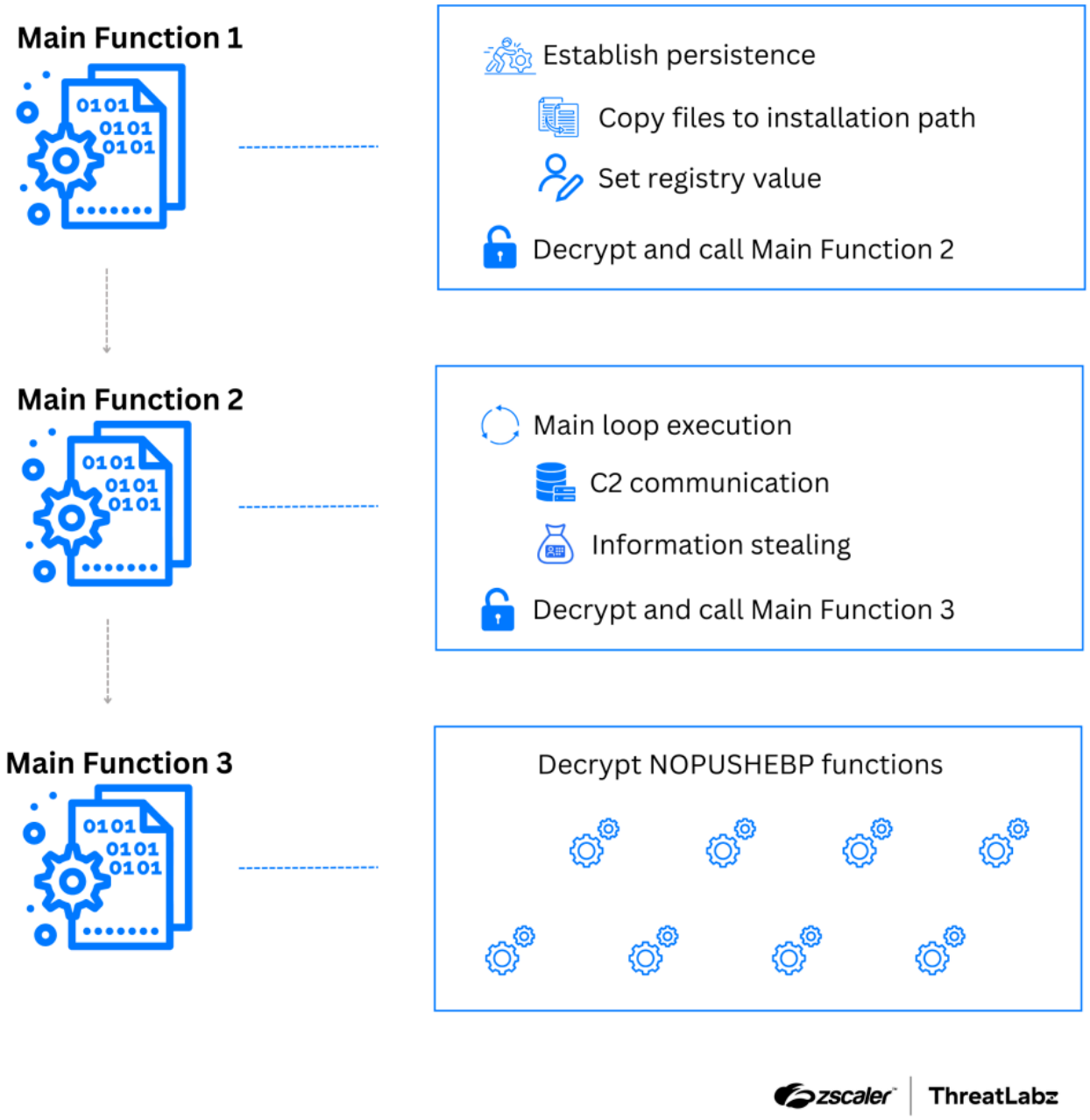

Figure 2: A high-level diagram for Xloader versions 6 and 7, which leverage three main functions to decrypt and execute critical parts of code.

Xloader executes three main functions: *Main Function 1*, *Main Function 2*, and *Main Function 3*. Each function implements a decryption routine for subsequent encrypted functions. The most noteworthy of these is Main Function 3, which itself is dynamically decrypted through multiple layers, and responsible for decrypting a new additional encryption layer on top of the `NOPUSHEBP` encrypted functions.

Main Function 1

Main Function 1 is responsible for establishing persistence on the victim's computer by copying itself to the appropriate directories and modifying the necessary registry keys explained in the previous section. In addition,

Main Function 1 decrypts and calls Main Function 2 as explained below.

Main Function 1 uses an egg-hunting technique that searches for two DWORD ID values to locate the memory address range of the encrypted Main Function 2. Note that all of these DWORD values in Xloader are calculated dynamically at runtime by performing an XOR operation with hardcoded values to evade static signature-based detection. The first layer of the code is decrypted using Xloader's RC4 and subtraction algorithm using a 20-byte key stored in the malware's global configuration. Once the first layer is decrypted, another DWORD value is used to delineate the end of the second encryption layer. The key to decrypt the second layer is the same DWORD value used to mark the beginning of the first encryption layer (padded with zeros until the length is 20).

Once the code is decrypted, the function prologue bytes `55 8B EC` are written at the beginning of the decrypted code and 4 no-op (NOP) opcodes are written at the end.

A Python implementation of the Main Function 1 decryption code is shown below:

```
 # The malware keeps a common 0x14 len key at configobj + 0x410
# This offset could change from one sample to another
id_find_encode = get_hardcoded_id_find_encode(binary)
key_layer1 = keys['keys_0x14_stored_in_configobj'][0x410]
id_end_layer1 = get_hardcoded_id_end_layer1(binary)
id_end_layer2 = get_hardcoded_id_end_layer2(binary)
key_layer2 = padding(id_find_encode)
if id_find_encode in binary:
   encode_base = binary.index(id_find_encode) + 4
   if id_end_layer1 in binary and id_end_layer2 in binary:
       # Decrypt layer 1
       end_layer1 = binary[encode_base:].index(id_end_layer1) + encode_base
       decode_layer1 = rc4_sub(binary[encode_base:end_layer1], key_layer1)

       # Add layer 1 decrypted code
       binary = binary[:encode_base] + decode_layer1 + binary[end_layer1:]
       # Decrypt layer 2
       end_layer2 = binary[encode_base:].index(id_end_layer2) + encode_base
       decode_layer2 = rc4_sub(binary[encode_base:end_layer2], key_layer2)
       # Add layer 2 decrypted code
       binary = binary[:encode_base] + decode_layer2 + binary[end_layer2:]
       # Add the function prologue and NOPs
       start = encode_base
       end = end_layer1
       if end_layer2 > end_layer1:
           end = end_layer2
       decrypted_code = binary[start:end]
       decrypted_function = b"\x55\x8B\xEC" + decrypted_code + b"\x90\x90\x90\x90"
       return decrypted_function
```

Main Function 2

Main Function 2 executes Xloader's main loop that calls the functions that communicate with the C2 server and steal data from the victim's computer. The function also decrypts and calls Main Function 3.

The decryption code in Main Function 2 is similar to Main Function 1 with two layers decrypted by Xloader's RC4 and subtraction algorithm. The first layer is decrypted using the key stored in the global configuration and the second layer is decrypted using a 20-byte key that is constructed dynamically (rather than using the DWORD ID value used to mark the beginning of the first encryption layer).

Main Function 3

As mentioned previously, Main Function 3 is primarily responsible for decrypting a new encryption layer on top of `NOPUSHEBP` functions. Main Function 3 uses another egg-hunting technique to search for these encrypted blocks. For example, in the sample `66ebf028ab0f226b6e4c6b17cec00102b1255a4e59b6ae7b32b062a903135cc9`, Xloader leverages 7 DWORD IDs used to find the beginning of the encrypted blocks, and another 7 IDs used to locate the end of the encrypted blocks. Each block is decrypted using Xloader's RC4 and subtraction algorithm with a key obtained from the malware's global configuration. Note that after each block is decrypted, the code is still encrypted in the `NOPUSHEBP` functions.

Encrypted NOPUSHEBP functions

The decryption process of `NOPUSHEBP` functions remains the same as previous versions with the only exception being how the IDs and decryption keys are dynamically calculated.

The following code shows a Python implementation of Xloader's `NOPUSHEBP` function decryption:

```
 # Find the required seeds and xor keys from the binary
nopushebp_tag1_seed = get_nopushebp_tag1_seed(binary)
nopushebp_tag2_seed = get_nopushebp_tag2_seed(binary)
xor_key_tags = get_xor_key_tags(binary)
nopushebp_key_seed = get_nopushebp_key_seed(binary)
nopushebp_key_xor = get_nopushebp_key_xor(binary)
# Calculate the limit tags and key
nopushebp_tag1 = xor(nopushebp_tag1_seed, xor_key_tags)
nopushebp_tag2 = xor(nopushebp_tag2_seed, xor_key_tags)
nopushebp_key = xor(nopushebp_key_seed, nopushebp_key_xor)
# Decrypt the function
if nopushebp_tag1 in binary and nopushebp_tag2 in binary:
   enc = binary.split(nopushebp_tag1)[1].split(nopushebp_tag2)[0]
   decrypted_function = b"\x55\x8B\xEC" + \
                        rc4_sub(enc, nopushebp_key) + \
                        b"\x90\x90\x90\x90"
```

Version 7.5 of Xloader introduced a slight modification to decrypt `NOPUSHEBP` functions with the construction of the RC4 keys and some functions contain multiple layers of encryption.

Encrypted PUSHEBP functions

Xloader's `PUSHEBP` functions are very similar to prior versions, with additional complexities added in version 7.5 that leverage XOR operations to derive the RC4 key.

Code encryption on API calls

Xloader now encrypts its own code before calling critical APIs, like `ZwSetThreadContext`, involved in process injection, as shown in the figure below.

```
ALGOS_GetNextInstruction_address_pop_push_eax_ret();
enc_limit = v5 - (unsigned int)configObj->ptr_0x403463_or_INJECTED_BASE;
for ( i = 0; i < enc_limit; ++i )// encrypt blocks of the malware code before calling API
  *(_BYTE *)(i + ptr_0x403463_or_INJECTED_BASE) ^= *(&v15 + i % 0x14);
if ( enc_limit_ins )
{
  enc_limit_2 = v14 - (enc_limit_ins - (unsigned int)configObj->ptr_0x403463_or_INJECTED_BASE);
  for ( j = 0; j < enc_limit_2; ++j )
    *(_BYTE *)(j + enc_limit_ins) ^= *(&v15 + j % 0x14);
}
v13 = ptr_ZwSetThreadContext(arg_api_1, arg_api_2);// Call API
                //
for ( k = 0; k < enc_limit; ++k )// restore the original code
  *(_BYTE *)(k + ptr_0x403463_or_INJECTED_BASE) ^= *(&v15 + k % 0x14);
if ( enc_limit_ins )
{
  for ( m = 0; m < enc_limit_2; ++m )
    *(_BYTE *)(m + enc_limit_ins) ^= *(&v15 + m % 0x14);
}
```


Figure 3: Code protection technique prior to calling `ZwSetThreadContext` in Xloader 6.2.

Once the API returns, the original code is decrypted again. This is likely a mechanism designed to thwart analysis platforms that generate memory dumps when specific system calls are executed (for example, memory allocation, process creation, etc.), since the important parts of the code will remain encrypted. A similar technique is implemented in modern versions of SmokeLoader's stager component, which decrypts code blocks when needed and re-encrypts them after use.

Data obfuscation

Previous versions of Xloader stored all critical information (for example, the encrypted strings and the encryption keys) in a set of static encrypted data blocks that we called `PUSHEBP` data blocks because the encrypted data was preceded by a *push ebp* prologue.

In Xloader versions 6 and 7, encrypted `PUSHEBP` data blocks no longer exist. When the malware requires a string, key, seed, or constant, the value is constructed dynamically. As a result, there are no hardcoded keys in the malware code.

String obfuscation

In previous versions of Xloader, the malware's encrypted strings were contained in an encrypted `PUSHEBP` data block. In the new versions, there are dedicated functions that build, decrypt, and return each string, as shown in the figure below.

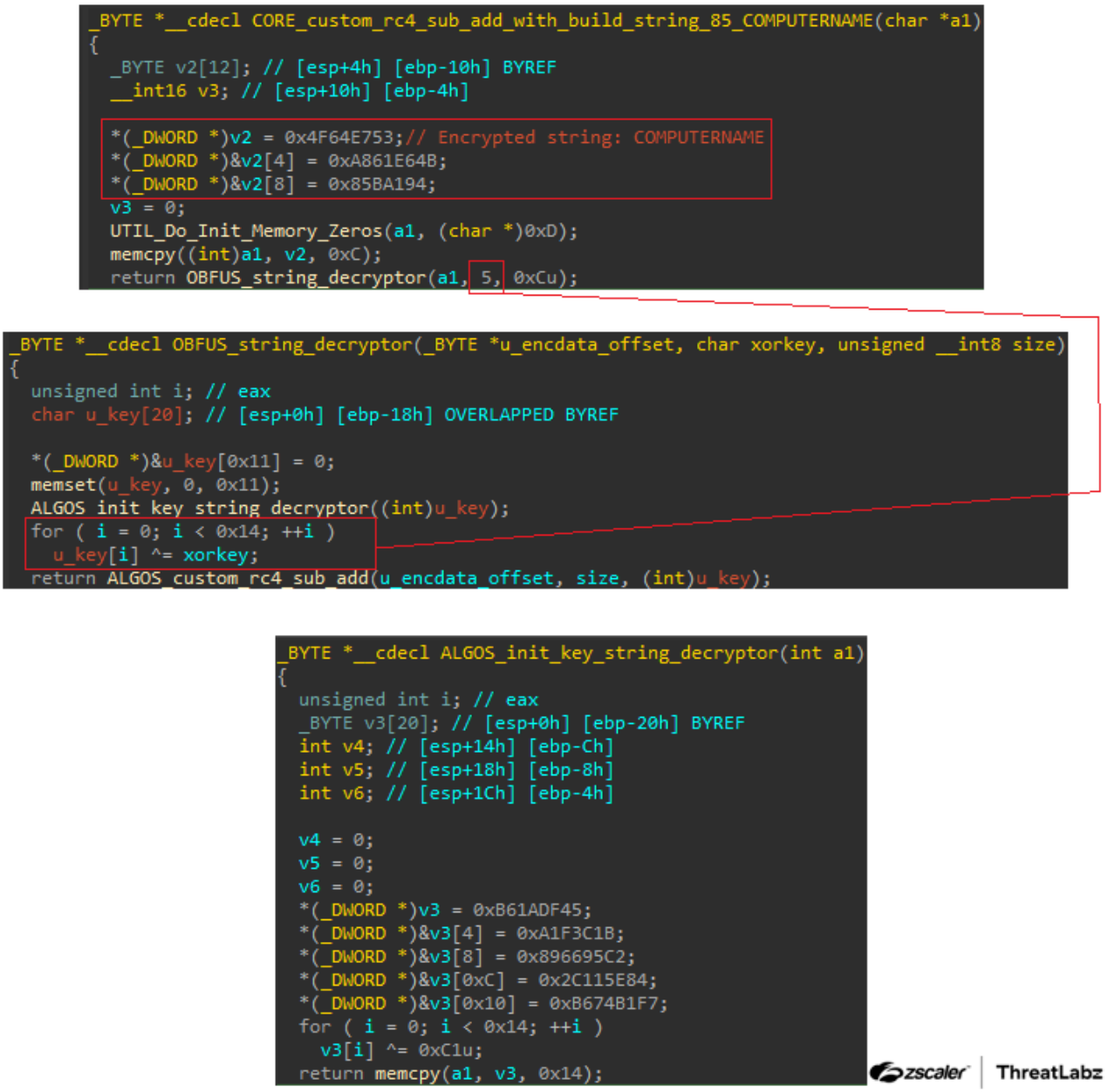

Figure 4: Function to decrypt the `COMPUTERNAME` string in Xloader 6.2.

All of these functions operate by pushing the encrypted string on the stack. Xloader then calls a function that initializes a 20-byte key. The key is then used to decrypt the string using Xloader's RC4 and subtraction algorithm.

Stack strings

In addition to the encrypted strings described above, Xloader builds some plaintext strings on the stack through `NOPUSHEBP` functions. Since the code itself in these functions is encrypted and protected, the strings are too.

API obfuscation

Earlier versions of Xloader used two encrypted `PUSHEBP` data blocks that contained tables of 32-bit cyclic redundancy check (CRC32) values for Windows API function names. In versions 6 and 7, these `PUSHEBP` data blocks have been removed. Now, each CRC32 value is created by its own specific code block.

There are two main types of functions that calculate the CRC32 values for Windows API function names. One function requires a seed and an encrypted CRC32, while the other function only requires an encrypted CRC32 value. These functions are described below.

The first API resolution function dynamically builds each API CRC32 value from two parameters. The functions maintain a consistent code structure with two parameters consisting of a 1-byte XOR key and an encrypted CRC32 DWORD value as shown in the figure below.

```
int __cdecl ALGOS_HASH_GET_ID_custom_rc4_sub_add_with_init_key_0x14_with_xor_1(int u_encdata_offset, char xor_key)
{
  char v2; // cl
  unsigned int i; // eax
  char u_key[20];
  memset(u_key, 0, 0x11);
  *(_DWORD *)&u_key[0x11] = 0;
  ALGOS_init_key_0x14_with_xor_1((int)u_key);
  v2 = xor_key;
  for ( i = 0; i < 0x14; ++i )
    u_key[i] ^= v2;
  ALGOS_custom_rc4_sub_add(&u_encdata_offset, 4u, (int)u_key);
  return u_encdata_offset;
}
```

```
_BYTE *__cdecl ALGOS_init_key_0x14_with_xor_1(int outbuf)
{
  unsigned int i; // eax
  _BYTE tmp[20]; // [esp+0h] [ebp-20h] BYREF
  int v4; // [esp+14h] [ebp-Ch]
  int v5; // [esp+18h] [ebp-8h]
  int v6; // [esp+1Ch] [ebp-4h]

  v4 = 0;
  v5 = 0;
  v6 = 0;
  *(_DWORD *)tmp = 0x8F132B15;
  *(_DWORD *)&tmp[4] = 0x6003EB74;
  *(_DWORD *)&tmp[8] = 0xEA48088E;
  *(_DWORD *)&tmp[0xC] = 0x7D89618F;
  *(_DWORD *)&tmp[0x10] = 0xC1A6A49E;
  for ( i = 0; i < 0x14; ++i )
    tmp[i] ^= 0x48u;
  return memcpy(outbuf, tmp, 0x14);
}
```


Figure 5: Algorithm to derive the API hash using a seed and an encrypted CRC32 value in Xloader 6.2.

In the example above, an XOR operation is performed with a 20-byte seed (`15 2B 13 8F 74 EB 03 60 8E 08 48 EA 8F 61 89 7D 9E A4 A6 C1`) and a hardcoded byte (`0x48`). Another XOR operation is performed using the result and the 1-byte XOR key provided to the function. This generates the decryption key, which is then used by Xloader's RC4 and subtraction algorithm to decrypt the API function name's CRC32 value.

In other cases, the API CRC32 value is calculated with inline code (instead of a dedicated function). In these cases, only the RC4 encrypted CRC32 value is provided as an argument to a function that builds a 20-byte RC4 key dynamically (with 5 DWORDs encoded by a single XOR key). Xloader's RC4 with subtraction algorithm is then used to decrypt the final API CRC32 value.

Xloader then computes the CRC32 value for each export function name (converted to lowercase) and the previously calculated CRC32 to locate the address of each required API function.

NTDLL hook evasion

Xloader versions 6 and 7 load a copy of ntdll and call the library's API functions through the copy instead of the original library. This ensures that if breakpoints are set on the exported functions of the library or if a monitoring tool attempts to perform hooks, they will be unable to properly trace Xloader's behavior. This technique is used by other malware families including SmokeLoader.

Xloader obfuscation evolution

The table below outlines the obfuscation techniques used in Formbook and various versions of Xloader.

Technique	**V2**	**V4**	**V6/V7**
RC4 with subtraction encryption	Yes	Yes	Yes
Custom lookup table decryption algorithm	Yes	Yes	No
PUSHEBP data blocks	Yes	Yes	No
PUSHEBP code blocks with plaintext limit IDs	Yes	No	No
PUSHEBP. code blocks with encrypted limit IDs	No	Yes	Yes
PUSHEBP decryption with XOR key passed by caller	No	No	No/Yes
NOPUSHEBP code blocks Key from PUSHEBP data blocks	No	Yes	No
NOPUSHEBP code blocks with egg-hunting	No	No	Yes

Technique	**V2**	**V4**	**V6/V7**
NOPUSHEBP code blocks with dynamic key	No	No	Yes
NOPUSHEBP code blocks Two RC4 with subtraction layers Key 1 built dynamically Key 2 from global config	No	No	No/Yes
Strings in PUSHEBP data blocks	Yes	Yes	No
Stack-based string obfuscation	No	No	Yes
Decoy C2s stored as encrypted strings	Yes	Yes	Yes
Decoy C2s encrypted with additional layers	Yes	Yes	Yes
Real C2 in PUSHEBP data blocks	Yes	Yes	No
Real C2 among the encrypted strings	No	No	Yes
C2 keys in data blocks	Yes	Yes	No
C2 keys built dynamically	No	No	Yes
API CRC values in PUSHEBP data blocks	Yes	Yes	No

Technique	V2	V4	V6/V7
API CRCs encrypted with RC4 and subtraction	No	No	Yes

Table 1: Comparison of Xloader obfuscation techniques by version.

As the table above demonstrates, Xloader continues to add new layers of encryption and obfuscation with each new release to complicate manual and automated analysis. Xloader now contains multiple layers of encryption for code, strings, constants, and API hashes. Furthermore, the Xloader author has implemented additional measures over time to decrypt critical information only when necessary, and scattered code and data across multiple sections. Xloader's encryption algorithms have also been changed significantly over time from a custom encryption algorithm that used a lookup table to an algorithm that leverages RC4 and subtraction. These modifications are designed to better evade detection by endpoint security software and stay one step ahead.

Explore more Zscaler blogs
