Technical Analysis of Xloader Versions 6 and 7 P2 | ThreatLabz

By ThreatLabz

Published: 2025-02-13 · Archived: 2026-04-05 14:30:24 UTC

C2 decryption

Decoy C2 servers

Xloader shares many characteristics as Formbook, its predecessor, including the use of a *decoy C2 list* and a *real C2 server*, which are encrypted differently and stored separately within the binary. The purpose of the decoys is to generate network traffic to legitimate domains to disguise real C2 traffic. This approach has been used by other malware families in the past such as Pushdo. Note that the so-called decoy list can also include actual C2 servers, but for simplicity, we'll continue to refer to these as the "decoy list" and "real" C2 server in this blog, since the former still primarily contains legitimate domains.

The figure below shows a high-level description of the process that Xloader uses to decrypt the decoy C2s.

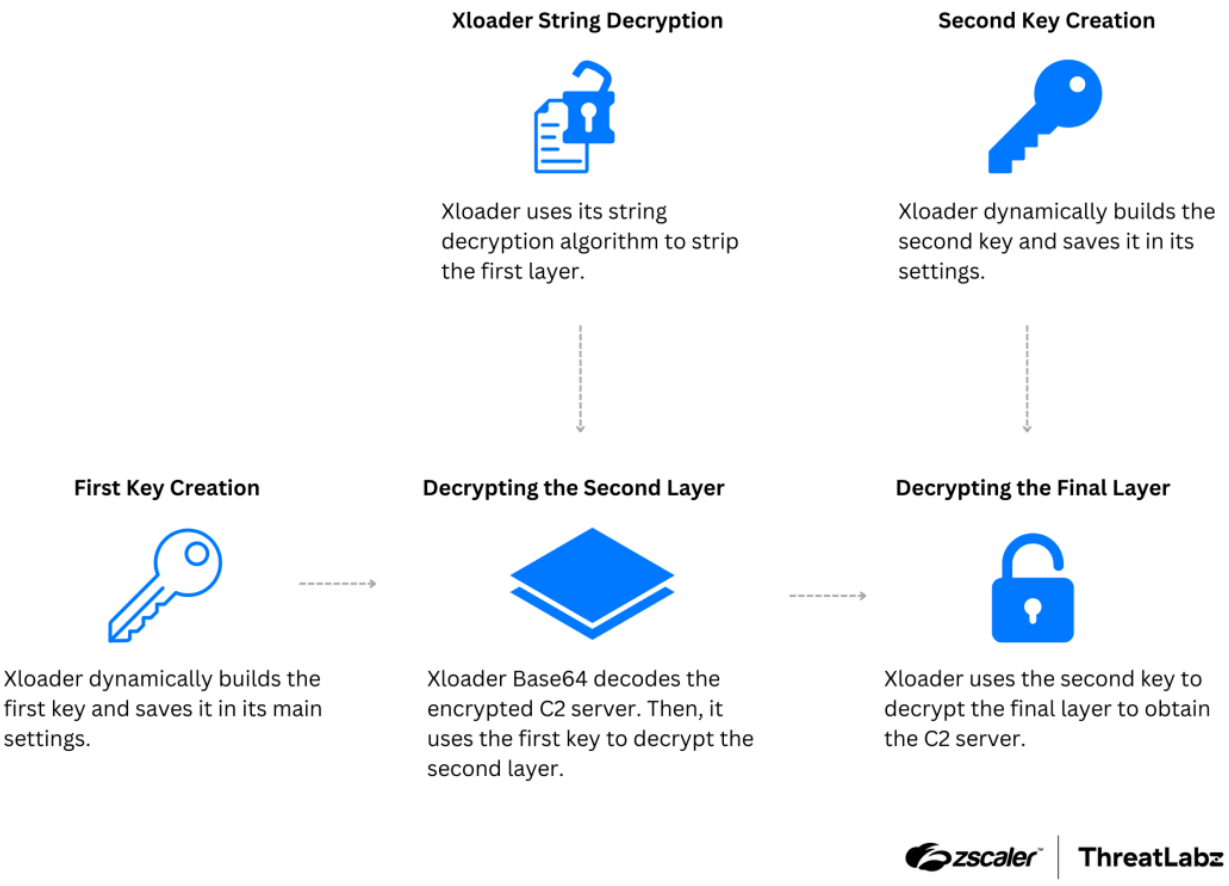

Figure 1: The functions that decrypt the decoy C2 servers in Xloader 6.2.

The Xloader decoy C2s are encrypted with three layers. The keys needed for decryption are generated by various functions within the malware code and are stored in global configuration structures as described below.

The first decryption key for the decoy C2 is constructed dynamically by one of the encrypted `NOPUSHEBP` functions. Five DWORDs are combined to construct an initial 20-byte seed. This seed is then XOR'ed with a hardcoded DWORD XOR key and an additional hardcoded 1-byte XOR key. The resulting 20-byte key is stored in the global configuration structure.

Similarly, the second key of the decoy C2 is generated by another encrypted `NOPUSHEBP` function. Once again, 5 DWORDs are initialized on the stack and XOR'ed with a DWORD XOR key retrieved from the global configuration structure. This DWORD XOR key was previously calculated and stored in the global configuration structure by another function.

The list of decoy C2s is stored among the encrypted strings with indexes that typically range from 1 to 63 (inclusive). Another function implements the process to retrieve and decrypt a specific decoy C2 server based on its index using Xloader's standard string encryption algorithm that we described in Part 1 of this blog series. The result of removing this first layer is the encrypted second layer, which is a Base64 encoded string.

The second layer is Base64 decoded and decrypted with Xloader's RC4 and subtraction algorithm using the first key XOR'ed with the index of the decoy C2. The third and final layer uses Xloader's RC4 and subtraction algorithm using the second key.

Below is a Python implementation of Xloader's decoy C2 decryption algorithm:

```
 # Get the necessary seeds and xor keys from the binary
rc4_key_1_seed = get_rc4_key_1_seed()
rc4_key_1_xor = get_rc4_key_1_xor()
rc4_key_2_seed = get_rc4_key_2_seed()
rc4_key_2_xor = get_rc4_key_2_xor()

# Calculate final keys
decoy_C2s_key_1 = xor(rc4_key_1_seed, rc4_key_1_xor)
decoy_C2s_key_2 = xor(rc4_key_2_seed, rc4_key_2_xor)

# Decrypt the decoy C2
enc_C2 = decrypt_encrypted_string_by_index(target_C2_index)
b64dec = base64.b64decode(enc_C2)
key1 = xor(decoy_C2s_key_1, target_C2_index)
dec = rc4_sub(b64dec, key1)
decrypted_c2 = rc4_sub(dec, decoy_C2s_key_2)
```

Legitimate C2 servers

Following the C2 decoy list, is another encrypted string located at index `64`. This encrypted string contains the real Xloader C2, which is decrypted using a similar algorithm but with different keys. First, the encrypted string at index `64` is retrieved and decrypted. After decryption, the result is Base64 decoded, and a new key is dynamically built as follows:

1. A 20-byte seed is constructed.
2. This seed is XOR'ed with a hardcoded 1-byte XOR key.
3. The seed is then XOR'ed with a hardcoded DWORD XOR key.
4. Finally, the seed is XOR'ed with another hardcoded 1-byte XOR key.

The resulting 20-byte key is then used to decrypt the first encryption layer of the real C2 using RC4 and subtraction.

The next encryption layer of the real C2 is decrypted using another function:

1. A 20-byte seed is constructed.
2. This seed is then XOR'ed with a hardcoded 1-byte XOR key.
3. The resulting key is used to decrypt the final RC4 and subtraction layer of the real C2.

Below is a Python implementation for decrypting Xloader's real C2:

```
 # Get the necessary seeds and xor keys from the binary
rc4_key_1_seed = get_rc4_key_1_seed()
rc4_key_1_xor1byte = get_rc4_key_1_xor1byte()
rc4_key_1_xor4bytes = get_rc4_key_1_xor4bytes()
rc4_key_2_seed = get_rc4_key_2_seed()
rc4_key_2_xor = get_rc4_key_2_xor()

# Calculate the final keys
real_C2_key = xor(rc4_key_1_seed, rc4_key_1_xor1byte)
real_C2_key = xor(real_C2_key, rc4_key_1_xor4bytes)
real_C2_key_2 = xor(rc4_key_2_seed, rc4_key_2_xor)

# Decrypt the real C2
string_64 = decrypt_encrypted_string_by_index(64)
b64dec = base64.b64decode(string_64)
dec = rc4_sub(b64dec, real_C2_key)
dec = rc4_sub(dec, real_C2_key_2)
if dec.startswith(b'www'):
   return dec.decode()
```

Note that all of the real C2 servers observed by ThreatLabz (after decryption) start with a *www* subdomain. In contrast, the decoy C2 servers embedded in the malware do not start with a *www* subdomain, but that prefix is added by Xloader prior to establishing network communications.

C2 URL path

In Xloader versions 6 and earlier, the real C2 string included a domain and a path. However, each decoy C2 string only consisted of a domain. Therefore, Xloader appended the real C2's path to each decoy domain prior to generating network traffic.

In Xloader version 7.5, each decoy C2 and real C2 has its own URL path, and the decryption function now includes an additional argument to return either the domain or the path of the decrypted C2 server. This process uses a 20-byte key combined with the C2's index to decrypt each 4 character C2 path via Xloader's RC4 and subtraction algorithm.

Network protocol

We previously described Xloader's network protocol and a new encryption layer that was added to the malware's registration packet. In Xloader 4.3, we discovered that there was a bug that caused the registration packet to be truncated because of the improper placement of a NULL character. However, this issue has since been resolved in version 6, with the packet now formatted correctly.

Explore more Zscaler blogs
