Zloader Analysis | ThreatLabz

By ThreatLabz

Published: 2024-01-19 · Archived: 2026-04-05 18:20:45 UTC

Technical Analysis

In the following sections, we dive into the technical details surrounding Zloader's new updates to their anti-analysis techniques, embedded configuration, DGA, and network encryption.

Anti-analysis techniques

Zloader uses a combination of API import hashing, junk code, a filename check, and string obfuscation. The following sections analyze each technique.

Imports and API resolution

The newest Zloader samples only import a few functions from the kernel32 library. The remaining imports are resolved at runtime using checksums to obfuscate the functions that are used. This technique, already present in older versions, changes its implementation, adding an XOR constant which changes between samples. Python code that replicates the API hashing algorithm is shown below.

```
def calculate_checksum(func_name, xor_constant):
   checksum = 0
   for element in func_name.upper():
       checksum = 16 * checksum - (0 - (ord(element)+1))
       if checksum & 0xf0000000 != 0:
           checksum = ((((checksum & 0xf0000000) >> 24) ^ checksum) & 0xfffffff)
   return checksum ^ xor_constant
```

Code sample available on GitHub.

Junk code

Similar to previous versions, Zloader uses custom obfuscation. The new version of Zloader adds junk code that consists of various arithmetic operations, as shown in Figure 1 below.

```
__int64 __fastcall malware_resolve_int(int a1)
{
  int v3; // [rsp+24h] [rbp-Ch]
  int v4; // [rsp+28h] [rbp-8h]
  int v5; // [rsp+28h] [rbp-8h]
  int v6; // [rsp+2Ch] [rbp-4h]
  int v7; // [rsp+2Ch] [rbp-4h]

  v3 = a1 ^ 0x34420CDD;
  v4 = a1 ^ 0x172;
  v6 = sub_140016430(a1 ^ 0x172u, a1);
  if ( a1 == v3 || a1 != v4 )
  {
    v4 = v6 + a1 + a1 - v3;
    v6 = v4 + 849;
  }
  v5 = v4 & v6;
  v7 = v5 & 0x200;
  if ( a1 == v3 || a1 <= v3 && a1 != v5 )
  {
    v5 = 914 * v7;
    v7 = v3 | (914 * v7);
  }
  if ( a1 == v5 && ((sub_1400180D0(a1, v3) & 1) != 0 || a1 == v5) )
    v5 = v3 ^ (v7 * a1 - 637);
  dword_14002B268 = v5;
  return (unsigned int)v3;
}
```


Figure 1. Example Zloader 2.1 junk code

In Figure 1, the instructions inside the red box are the junk code.

Anti-sandbox

Each Zloader sample expects to be executed with a specific filename. If the filename does not match what the sample expects, it will not execute further. This could evade malware sandboxes that rename sample files. Figure 2 shows an example of a Zloader sample that expects its filename to be **CodeForge.exe**.

```
proc_addr_ptr = malware_getProcAddr((__int64)"CodeForge.exe");
if ( proc_addr_ptr )
{
  if ( (malware_Init_alt_api_tables() & 1) != 0 )
  {
```


Figure 2. Example of Zloader's anti-analysis filename check

ThreatLabz has observed Zloader use the following filenames:

- CodeForge.exe
- CyberMesh.exe
- EpsilonApp.exe
- FusionBeacon.exe
- FusionEcho.exe
- IonBeacon.dll

- IonPulse.exe
- KineticaSurge.dll
- QuantumDraw.exe
- SpectraKinetic.exe
- UltraApp.exe

String obfuscation

Similar to prior versions, Zloader implements a string obfuscation algorithm for some of the malware's important strings such as registry paths, DLL names, and the DGA's top-level domain (TLD) using XOR with a hardcoded key. Python code that replicates the string obfuscation algorithm is shown below:

```
def str_deobfuscate(enc_bin, enc_key):
 res = ''
 for i, element in enumerate(enc_bin):
   res += chr( ((element ^ 0xff) & (enc_key[i % len(enc_key)])) | (~(enc_key[i % len(enc_key)]) & element))
 return res
```

Code sample available on GitHub.

The encryption key differs between samples and is also hardcoded in the *.rdata* section as shown in Figure 3 below.

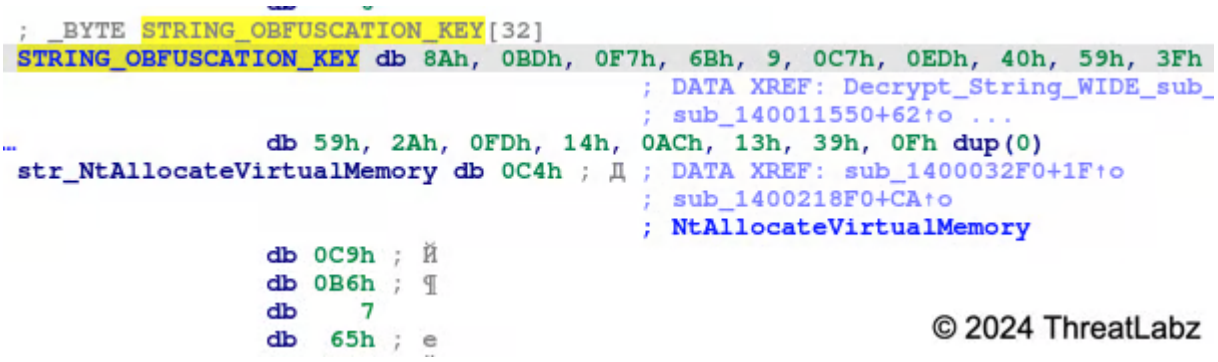

Figure 3. Example string obfuscation key used by Zloader

A list of Zloader's obfuscated strings is shown in the Appendix.

Static configuration encryption and structure

The Zloader static configuration is still encrypted using RC4 with a hardcoded alphanumeric key, but the structure is slightly different. The botnet ID, campaign name, and command-and-control servers (C2s) are set at fixed offsets, in addition to an RSA public key that replaces the old RC4 key that was used for network encryption. ThreatLabz has observed 15 unique new Zloader samples and all of them have the same RSA public key, likely indicating there is currently only a single threat actor using the malware.

An example Zloader static configuration is shown below.

```
 00000000  00 00 00 00 42 69 6e 67  5f 4d 6f 64 35 00 00 00  |....Bing_Mod5...|
00000010  00 00 00 00 00 00 00 00  00 4d 31 00 00 00 00 00  |.........M1.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 68 74  |..............ht|
00000030  74 70 73 3a 2f 2f 61 64  73 6c 73 74 69 63 6b 65  |tps://adslsticke|
00000040  72 68 69 2e 77 6f 72 6c  64 00 00 00 00 00 00 00  |rhi.world.......|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000002b0  00 00 00 00 00 00 00 00  02 00 00 00 01 00 00 00  |................|
000002c0  2d 2d 2d 2d 2d 42 45 47  49 4e 20 50 55 42 4c 49  |-----BEGIN PUBLI|
000002d0  43 20 4b 45 59 2d 2d 2d  2d 2d 0a 4d 49 47 66 4d  |C KEY-----.MIGfM|
000002e0  41 30 47 43 53 71 47 53  49 62 33 44 51 45 42 41  |A0GCSqGSIb3DQEBA|
000002f0  51 55 41 41 34 47 4e 41  44 43 42 69 51 4b 42 67  |QUAA4GNADCBiQKBg|
00000300  51 44 4b 47 41 4f 57 56  6b 69 6b 71 45 37 54 79  |QDKGAOWVkikqE7Ty|
00000310  4b 49 4d 74 57 49 38 64  46 73 61 0a 6c 65 54 61  |KIMtWI8dFsa.leTa|
00000320  4a 4e 58 4d 4a 4e 49 50  6e 52 45 2f 66 47 43 7a  |JNXMJNIPnRE/fGCz|
00000330  71 72 56 2b 72 74 59 33  2b 65 78 34 4d 43 48 45  |qrV+rtY3+ex4MCHE|
00000340  74 71 32 56 77 70 70 74  68 66 30 52 67 6c 76 38  |tq2Vwppthf0Rglv8|
00000350  4f 69 57 67 4b 6c 65 72  49 4e 35 50 0a 36 4e 45  |OiWgKlerIN5P.6NE|
00000360  79 43 66 49 73 46 59 55  4d 44 66 6c 64 51 54 46  |yCfIsFYUMDfldQTF|
00000370  30 33 56 45 53 38 47 42  49 76 48 71 35 53 6a 6c  |03VES8GBIvHq5Sjl|
00000380  49 7a 37 6c 61 77 75 77  66 64 6a 64 45 6b 61 48  |Iz7lawuwfdjdEkaH|
00000390  66 4f 6d 6d 75 39 73 72  72 61 66 74 6b 0a 49 39  |fOmmu9srraftk.I9|
000003a0  67 5a 4f 38 57 52 51 67  59 31 75 4e 64 73 58 77  |gZO8WRQgY1uNdsXw|
000003b0  49 44 41 51 41 42 0a 2d  2d 2d 2d 2d 45 4e 44 20  |IDAQAB.-----END |
000003c0  50 55 42 4c 49 43 20 4b  45 59 2d 2d 2d 2d 2d 0a  |PUBLIC KEY-----.|
000003d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000003fc
```

Domain generation algorithm

When the primary C2 server is not available, Zloader reverts to a DGA. The DGA algorithm has changed in the latest version and no longer contains a different seed per botnet. Python code that replicates Zloader's new DGA algorithm is shown below.

```
 import time
from datetime import datetime, timedelta

def uint32(val):
    return val & 0xffffffff

def get_dga_time():
    now = datetime.now()
    ts = time.time()
```

```
    utc_offset = (datetime.fromtimestamp(ts) - datetime.utcfromtimestamp(ts)).total_seconds() / 3600
    midnight = now.replace(hour=0, minute=0, second=0, microsecond=0)
    midnight = midnight + timedelta(hours=utc_offset)
    return int(midnight.timestamp())


def generate_zloader_dga_domains():
    domains = []
    t = get_dga_time()
    for i in range(32): # number of domains to generate
        domain = ""
        for j in range(20): # domain name length
            v = uint32(ord('a') + (t % 25 ))
            t = uint32(t + v)
            t = (t >> 24) & ((t >> 24) ^ 0xFFFFFF00) | uint32(t
```

Code sample available on GitHub.

The code generates 32 domains per day by using the local system time at midnight (converted to UTC) as a seed. Each of the DGA domains have a length of 20 characters followed by the ".com" TLD.

Network communications

Zloader continues to use HTTP POST requests to communicate with its C2 server. However, the network encryption is now using 1,024-bit RSA with RC4 and the Zeus "visual encryption" algorithms. Zloader uses the custom Zeus BinStorage format where the first 128 bytes are the RSA encrypted RC4 key (32 random bytes) and, the remaining bytes are encrypted with the RC4 key and visual encryption as shown in Figure 4:

```
00211C32D3860 B4 C4 88 58 F2 59 CA 1D C8 2A 30 73 9E 11 37 A7 ´Ä.XòYÊ.È*0s..7§
00211C32D3870 6E A1 6D 82 CE F4 AE 8C 92 7A 81 8A 12 58 EF 11 n¡m.Îô®..z...Xï.
00211C32D3880 CB 63 19 98 2C 8D 9C 98 07 14 81 1C 66 FF 94 AE Ëc..,.......fÿ.®
00211C32D3890 47 66 76 C8 F3 1F EE 89 C3 7D 59 F4 56 DD 78 4E GfvÈó.î.Ã}YôVÝxN
00211C32D38A0 A1 30 16 43 34 C2 BE C6 1B 97 45 0F B2 42 7C 96 ¡0.C4Â¾Æ..E.²B|.
00211C32D38B0 67 48 E1 76 1A D8 41 A8 66 2D 25 2C 06 1B 1A ED gHáv.ØA¨f-%,...í
00211C32D38C0 5C 19 2D 77 89 79 13 CB B4 80 4B 6A 6A 1F 26 12 \.-w.y.Ë´.Kjj.&.
00211C32D38D0 43 E3 C1 35 6E 55 22 3B B1 CF C4 8E 29 98 42 BC CãÁ5nU";±ÏÄ.).B¼
00211C32D38E0 DA CE 5D CA 20 39 B2 23 F6 69 FB 76 9B E3 CA AD ÚÎ]Ê 9²#öiûv.ãÊ.
00211C32D38F0 2C 16 47 14 B1 00 00 00 00 00 00 00 05 00 00 00 ,.G.±...........
00211C32D3900 81 42 11 AE E4 F6 1C 4E D0 08 69 2F 7D 67 4E D4 .B.®äö.NÐ.i/}gNÔ
00211C32D3910 12 27 00 00 00 00 00 00 09 00 00 00 09 00 00 00 .'..............
00211C32D3920 42 69 6E 67 5F 4D 6F 64 35 29 27 00 00 00 00 00 Bing_Mod5)'.....
00211C32D3930 00 02 00 00 00 02 00 00 00 4D 31 11 27 00 00 00 .........M1.'...
00211C32D3940 00 00 00 1E 00 00 00 1E 00 00 00 XX XX XX XX XX ...........XXXXX
00211C32D3950 XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XXXXXXXXXXXXXXXX
00211C32D3960 XX XX XX XX XX XX XX XX XX 13 27 00 00 00 00 00 XXXXXXXXX.'.....
00211C32D3970 00 04 00 00 00 04 00 00 00 00 07 01 02 16 27 00 ..............'.
00211C32D3980 00 00 00 00 00 04 00 00 00 04 00 00 00 01 00 00 ................
```


Figure 4. Zloader BinStorage object for a *hello* message (prior to encryption)

The Zeus BinStorage structure uses an ID integer value to represent the information stored, followed by the length and data. The BinStorage ID values in this example are shown in Table 1.

Value (Decimal)	Value (Hexadecimal)	Description
10002	0x2712	Botnet ID
10025	0x2729	Campaign ID
10001	0x2711	Bot ID
10003	0x2713	Malware version
10006	0x2716	Unknown flag (set to 0x1)

Table 1. Zloader BinStorage *hello* message fields

ThreatLabz has observed samples containing the following botnet IDs:

- Bing_Mod2
- Bing_Mod3
- Bing_Mod4
- Bing_Mod5

All of the campaign IDs have been set to the value **M1**.

Explore more Zscaler blogs
