Technical Analysis of TransferLoader | ThreatLabz

By ThreatLabz

Published: 2025-05-14 · Archived: 2026-04-10 02:31:57 UTC

This section describes TransferLoader's functionality as well as payloads embedded in the malware. These payloads are likely developed by the same threat actor and share code similarities. ThreatLabz has identified TransferLoader samples with the following embedded payloads:

- Downloader - Downloads and executes a payload from a command-and-control (C2) server.
- Backdoor loader - Executes the backdoor module and manages its configuration data.
- Backdoor - executes commands specified by the C2.

Another embedded payload discovered in TransferLoader was the ransomware family known as Morpheus. Even though the ransomware's code shares a few similarities with the aforementioned payloads (e.g. string decryption routines), the link between them is less clear. Therefore, an analysis of Morpheus has been omitted in this section.

Anti-analysis

Anti-VM/anti-debug

TransferLoader and its payloads contain the following anti-analysis methods:

- The loader retrieves its own filename and checks if it includes a specified-hardcoded substring (for example the substring `ess`) followed by the character `_`.
- Certain variants of the loader require two or more command-line parameters to proceed with the execution.
- All components leverage the `BeingDebugged` field in the Process Environment Block (PEB) to detect a debugging session.
- Windows APIs are dynamically resolved using a hashing algorithm.
- There are junk code blocks in certain functions of the loader (see figure below). The inserted code block never executes, but this addition is enough to break the decompilation process of IDA.

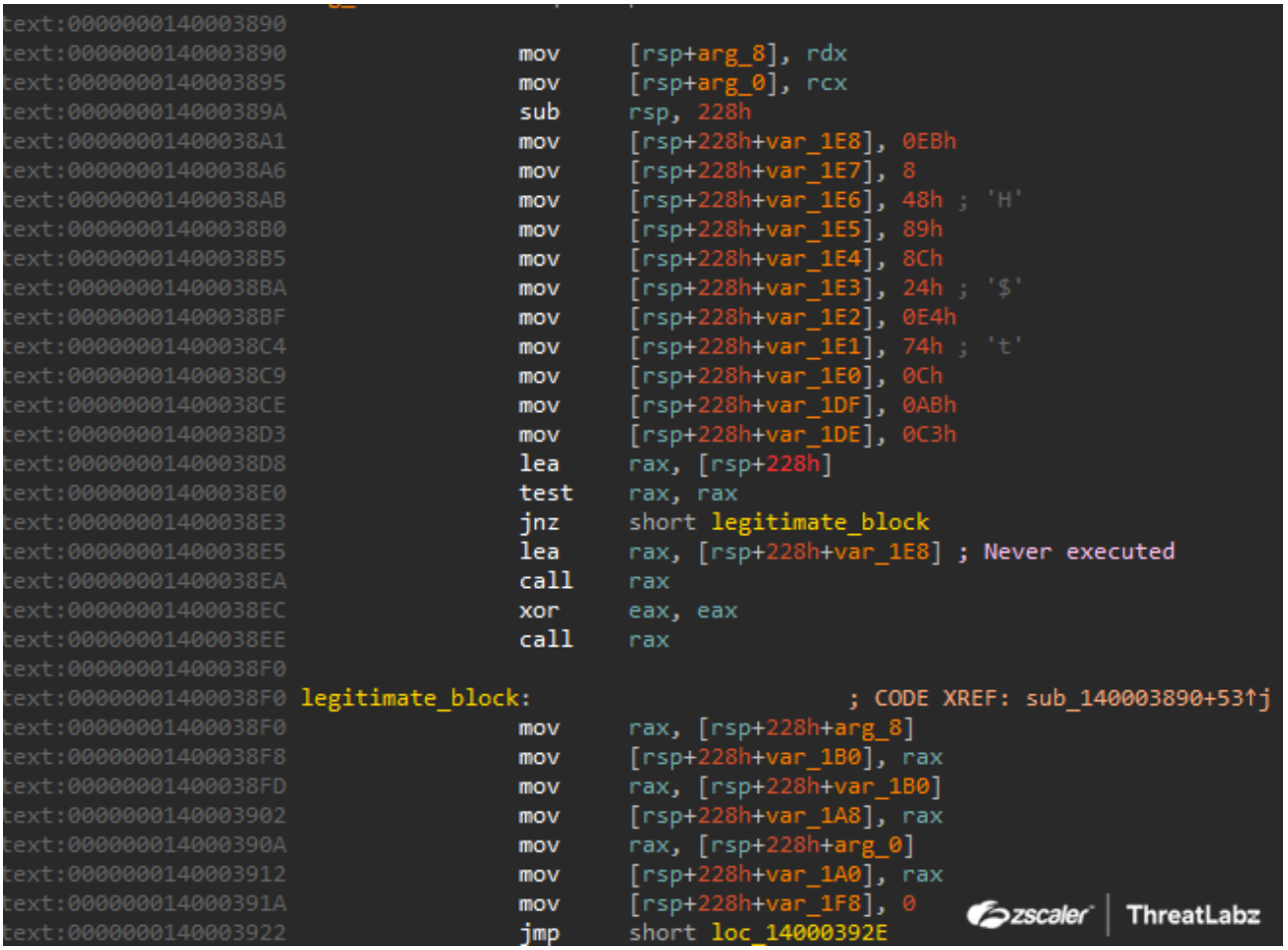

Figure 1: Example of TransferLoader junk code block.

String encryption

Each TransferLoader component decrypts strings at runtime using a bitwise-XOR operation. Every string is paired with a unique 8-byte key for decryption. The encrypted string data is pushed on the stack and decrypted, as shown in the figure below.

```
encrypted_string[0] = 0xE0;
encrypted_string[1] = 0xE0;
encrypted_string[2] = 0x24;
encrypted_string[3] = 0x33;
encrypted_string[4] = 0x1A;
encrypted_string[5] = 0xA1;
memset(&encrypted_string[6], 0, 2uLL);
xor_key = 0x7EFFA11A3315E0BCLL;
index = 0;
while ( index < 6 )
{
  for ( n = 0; n < 8uLL && index < 6; ++n )
  {
    // \1
    decrypted_character = (xor_key >> (8 * n)) ^ encrypted_string[index];
    decrypted_string[index++] = decrypted_character;
  }
}
```


Figure 2: TransferLoader runtime string decryption example.

Code obfuscation

All the components we analyzed employed code obfuscation using two different methods.

Obfuscation method 1

The first method reads the current address of the block and subtracts a hardcoded offset from this address. The result is the next block address to jump/execute as shown in the figure below.

```
e:000000140016888 current_block_address = rax
e:000000140016888                 lea     current_block_address, loc_140016892
e:000000140016892
e:000000140016892 loc_140016892:                          ; DATA XREF: start+13CEBto
e:000000140016892                 sub     current_block_address, 13CE1h
e:000000140016898                 mov     [rsp+1F8h+var_1F0], current_block_address
e:00000014001689D                 pop     current_block_address
e:00000014001689E                 retn
```


Figure 3: TransferLoader obfuscated control flow.

We observed this first method only in TransferLoader (not its embedded components) and the control flow can be recovered using the IDAPython script available in our GitHub repository.

Obfuscation method 2

The second method uses a more advanced approach and the developers make use of this method in the embedded payloads within TransferLoader (unlike the first method which is only used in TransferLoader itself).

The obfuscation process starts by saving all registers (including the RFLAGS and SIMD registers) and then allocating stack space for further operations/storage. Then, any local function variables and parameters are stored in the SIMD registers and the obfuscated instructions are executed. Each obfuscated instruction has its own handler. For example, the code block in the figure below shows how the obfuscator obtains the constant value 0x1000 and then stores the value in the SIMD register XMM4. The stored value is used at a later stage of the execution as a function parameter. Furthermore, the instructions for inserting a value into a SIMD register contain junk instructions to hinder analysis. In addition, it is important to highlight that before executing a handler, the obfuscator saves the offset of the next block address to execute as soon as the handler has executed.

```
mov     r9, 1000h
sub     rdx, 4          ; push 0x1000 in our stack
mov     [rdx], r9d
xor     r9, r9
mov     r9d, [rdx]
sub     rdx, 4
mov     [rdx], r9       ; now mov as a QWORD
mov     r9, [rdx]
add     rdx, 8          ; Restore our stack pointer
mov     r15, r9
shl     r15, 0
shr     r15, 3Eh
movq    r11, xmm4
ror     r11, 3Eh
shr     r11, 2
shl     r11, 2
or      r11, r15
rol     r11, 3Eh
pinsrq  xmm4, r11, 0
mov     r15, r9
shl     r15, 2
shr     r15, 2Ah
movq    r11, xmm4
ror     r11, 28h
shr     r11, 16h
shl     r11, 16h
or      r11, r15
rol     r11, 28h
pinsrq  xmm4, r11, 0
mov     r13, r9
shl     r13, 1Eh
shr     r13, 2Ch
movq    r15, xmm4
ror     r15, 0Eh
shr     r15, 14h
shl     r15, 14h
or      r15, r13
rol     r15, 0Eh
pinsrq  xmm4, r15, 0
mov     r11, r9
shl     r11, 18h
shr     r11, 3Ah
movq    r15, xmm4
ror     r15, 22h
shr     r15, 6
shl     r15, 6
or      r15, r11
rol     r15, 22h
pinsrq  xmm4, r15, 0
mov     r11, r9
shl     r11, 32h
shr     r11, 32h
movq    r13, xmm4
ror     r13, 0
shr     r13, 0Eh
shl     r13, 0Eh
or      r13, r11
rol     r13, 0
pinsrq  xmm4, r13, 0
mov     rsi, [r8]       ; Get the offset for the next block address
lea     r8, [r8+8]
lea     rbp, [r14+rsi]  ; rbp = image_base + next_block_address_offset
```

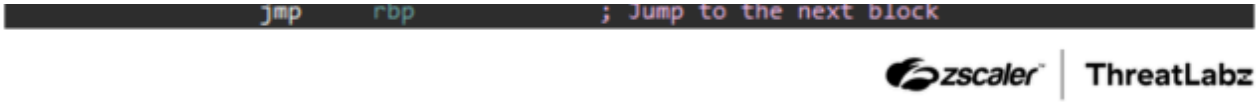

Figure 4: TransferLoader obfuscation handler.

After executing the obfuscated handlers, the obfuscator resets the state of the registers. At this stage, any pushed values from the obfuscated handlers are placed into the registers.

Despite the fact that the obfuscator handles a small set of instructions (e.g., the instruction `XOR REG, REG` is not handled), the threat actors can obscure the modification or assignment of local variables and code structures.

ANALYST NOTE: Even though the second obfuscation approach might appear to be a Virtual Machine (VM) protection, the obfuscator does not use any kind of virtualization and/or bytecodes.

TransferLoader

TransferLoader is a malware loader that appears to be the first part in the execution chain of subsequent modules and components. ThreatLabz has observed different variants of TransferLoader with minor functionality differences.

TransferLoader decrypts and executes an embedded payload with a straightforward decryption process, which is summarized as follows:

1. TransferLoader retrieves the data from the PE section with the encrypted payload. The name of this section might vary from sample to sample. For example, we observed the section names *.dbg* and *.secenc*.

2. Next, TransferLoader decrypts two strings: a 16-byte AES key and a custom Base32 charset.

3. TransferLoader calculates a 1-byte key from the custom Base32 charset and adds the calculated key to each encrypted byte of the target section.

```
 out = bytearray()
key = 0xFFFF
for i in range(len(base32_charset)):
   if input[i]
```

4. The resulting output from the previous operation is a Base32 string, which is then decoded using Base32 with the custom Base32 charset.

5. As a final step, TransferLoader performs a bitwise-XOR operation with the 16-byte AES key and the decoded data. The result is then decrypted using the AES-CBC decryption algorithm (with the aforementioned AES key).

Despite the simplicity of the decryption process described above, the developers have modified the AES algorithm to make the automation of the decryption process more tedious. Specifically, during the AES key expansion, the developers chose to use only the first element of the round constant array (`r_con`).

Note that older variants of TransferLoader use either Base32 decoding or AES decryption for the final payload, but not both of them at the same time.

Downloader

The downloader component is the most common embedded payload of TransferLoader that ThreatLabz has observed so far. The primary goal of the downloader is to download an additional payload from a C2 server and execute a decoy file.

The downloader sends an HTTPS GET request to the server to retrieve the payload. The HTTP request uses the following HTTP headers:

- `User-Agent: Microsoft Edge/1.0` (not used across all samples identified by ThreatLabz).
- `X-Custom-Header: xxx`

The downloader decrypts the received payload using a bitwise-XOR operation with the hardcoded 4-byte key 0xFFFFFFFF. The Python code below replicates the decryption routine.

```
 out = bytearray()
payload_data = b""
key = 0xffffffff
for b in payload_data:
   out.append((b ^ key) & 0xff)
   key -= 1
```

After executing the decrypted payload, the downloader attempts to resolve an export from the downloaded payload using its custom hashing algorithm. The DLL export should match the hash value 0xF78B59DF7AED203F. If successful, the downloader will execute the resolved export function.

After completing the operation above, the downloader opens a PDF decoy file. If the export function of the downloaded file was executed successfully, but did not return any data, then the downloader restarts the current Windows Explorer (explorer.exe) instance. The decoy document is either embedded in the binary or the downloader will create an invalid PDF file consisting of 8-bytes of junk data.

Backdoor loader

The backdoor module has its own dedicated loader, which handles any requests for reading or modifying configuration data.

The backdoor loader expects to reside either in the memory space of an Explorer instance (explorer.exe) or WordPad (wordpad.exe). In the second case (WordPad), the backdoor loader performs additional actions including the following:

- Hooks the Windows API `ShowWindow`. The hooked code enters an infinite sleep loop.
- Deletes the CSLID registry key SOFTWARE\Classes\CLSID\{F5078F32-C551-11D3-89B9-0000F81FE221}, which is the XML DOM Document 3.0 object.

- Uses the registry key SOFTWARE\Classes\CLSID\{1BAC8681-2965-4FFC-92D1-170CA4099E01}, which represents the object `Windows.System.User.ProxyStubFactory`, to add persistence on the compromised host. This method is also known as Component Object Model (COM) hijacking.

ANALYST NOTE: The backdoor loader attempts to create the file `defender.user.tmp` under the Windows temporary directory. If the backdoor loader fails to do this, the execution stops. We were not able to confirm the purpose of this operation.

The backdoor loader is also used for retrieving and updating the configuration of the backdoor. The table below describes the configuration elements that are stored in the registry key SOFTWARE\Microsoft\Phone\Config\.

Name	**Description**
rmi	C2 server (up to 64 bytes in length).
to	Timeout/sleep value.
id	Network encryption key (16-bytes).

Table 1: TransferLoader backdoor configuration fields.

For the backdoor module to update or retrieve the configuration information, the backdoor loader creates a named pipe using a hardcoded name (in our samples, the name is set to `nice`). The supported pipe commands (see the table below) and the packet structures for incoming/outgoing packets are described below.

Command ID	**Description**
2	Retrieves the network key from the configuration.
3	Retrieves the C2 server from the configuration.
4	Retrieves sleep/timeout value from the configuration.
5	Updates sleep/timeout value.

Command ID	Description
6	Updates the C2 server.
7	Stores a PE file in the registry under the name *md* and then executes it in memory.
8	Self-remove. Removes any files/registry keys associated with the infection and exits.

Table 2: TransferLoader pipe commands.

```
 #pragma pack(1)
struct get_data_pipe_packet
{
uint8_t result; // Set either to 1 (true) or 0 (false) to indicate if the requested operation was successful.
void* command_parameter_pointer;
};
#pragma pack(1)
struct update_data_pipe_packet
{
uint8_t cmd_id;
void* command_parameter_pointer;
DWORD sizeof_command_parameter; // Used only for command ID 7.
};
```

The pipe communication is encrypted using a bitwise-XOR operation with a 4-byte hardcoded XOR key (e.g., 0x534110E6) as shown below:

```
 initial_key = 0x534110E6
out = bytearray()
for c in data:
    key = initial_key + len(data)
    out.append((c ^ key) & 0xff)
```

As a final step, the backdoor loader executes the backdoor from the registry key SOFTWARE\Microsoft\Phone\Config\md after decrypting it using the same XOR decryption algorithm provided above.

Backdoor

The backdoor module is the core orchestrator for receiving and executing commands from the C2 server as well as for updating its own configuration data.

During the initialization phase, the backdoor requests the configuration data from its loader. If the configuration data is absent, the execution does not proceed and the malware exits. After reading and setting the configuration information, the backdoor attempts to start the initial communication with the malicious C2 server.

Interestingly, the backdoor uses the InterPlanetary File System (IPFS) decentralized network if it is unable to establish a successful connection with the C2 server. Specifically, the backdoor sends a request to the embedded IPFS URL and expects a new C2 server in return. Using this feature, the threat actors can redirect compromised hosts to a new C2 server in case of a server takedown.

The backdoor module supports both HTTPS and raw TCP communication methods. TCP is only used when the backdoor cannot initialize an HTTP session with the server. As for the HTTP request headers, the backdoor uses the user agent `My Agent/1.0` along with the custom header `X-Edge-key: none`.

Furthermore, the backdoor stores both the session's information and configuration data in the following structure:

```
 #pragma pack(1)
struct info
{
 uint8_t connection_mode;
 uint8_t  network_key[16];
 uint8_t  cnc[64];
 uint16_t  port;
 uintptr_t connection_handle; // Socket or HTTP request handle
 HINTERNET session_handle;
 HINTERNET request_handle;
};
```

As soon as a connection with the server is established, the backdoor requests a command. Each packet received has its own header with a size of 43-bytes followed by command parameters with a size up to 4,058 bytes. For clarity, we provided the C structures below, which represent the network packet.

```
 #pragma pack(1)
struct packet_header
{
 uint32_t encoded_length;
 uint32_t unknown;
 uint8_t cmd_id; // 2 for TCP, 3 for HTTP
 uint8_t connection_mode;
 uint32_t random_number;
 uint32_t  header_checksum;
 uint32_t  unknown_2; // Set when receiving a HTTP response
 uint8_t output_network_key[16];
};
```

```
#pragma pack(1)
struct command_network_packet
{
   packet_header header;
   uint8_t command_data[4058];
};
```

Before parsing the incoming network packet, the backdoor verifies the integrity of the packet by calculating the checksum value of the header. The Python code below replicates the checksum algorithm.

```
 checksum = 0xFFFFFFFF
for c in header_data:
    checksum ^= ( (checksum * c) ^ len(header_data)) & 0xffffffff
```

If the packet received passes the checksum validation, then the backdoor decrypts it using a custom stream-cipher, which has been replicated in Python below.

```
 def buggy_blocks_swap(data: bytes) -> bytearray:
    data = bytearray(data)
    div_size = len(data) // 2

    for iterator in range(div_size):
        read_char = data[iterator]
        data[iterator] = data[div_size - iterator - 1]
        data[div_size - iterator - 1] = read_char
    return data

def encrypt(key: bytes, input_data: bytes) -> bytearray:
    out = bytearray()
    for i in range(len(input_data)):
        out.append( input_data[i] ^ key[i % 16])
    out = buggy_blocks_swap(data=out)
    div_s = len(out) // 2
    for i in range(div_s):
        out[i + div_s] ^= out[i]
    return out


def decrypt(key: bytes, data: bytes) -> bytearray:
    data = bytearray(data)
    div_s = len(data) // 2
    for i in range(div_s):
        data[i + div_s] ^= data[i]
    reversed_data = buggy_blocks_swap(data=data)
    for i in range(len(reversed_data )):
```

```
        reversed_data[i] =  data[i] ^ key[i % 16]
    return reversed_data
```

ANALYST NOTE: Part of the cipher process is to swap the input data blocks. However, the implementation contains a bug in the loop's control variable and the data remains intact.

When the backdoor sends a request to the server, it first encrypts the header of the packet (excluding the network key at the end) and then the command's parameter data (if no data is present, then the encryption is applied to null bytes). Overall, the backdoor supports the network commands described in the table below.

Command ID	**Description**
0	Do nothing.
1	Executes a remote shell command and sends the output to the C2 server.
2	Reads a file from the compromised host.
3	Writes a file to the compromised host.
4	Executes a command/file without storing any output of the operation.
5	Updates the C2 server.
6	Updates the timeout/sleep value of the configuration.
7	Updates the network encryption key.
8	Collects information about the compromised host. This includes the username, hostname, NETBIOS name, Windows version and the access rights of the current user. The representative structure is: `struct host_info`

Command ID	Description

	`{`
	`uint8_t username[100];`
	`uint8_t hostname[100];`
	`uint8_t netbios_name[100];`
	`uint8_t windows_version[16];`
	`uint8_t user_rights;`
	`};`

9	Stops execution and starts the self-remove process from the backdoor loader side.

Table 3: TransferLoader backdoor network commands.

As shown in the table above, the backdoor reports any command's output. The outgoing network packet follows the same structure as the one provided above. The key difference is the handling of any error outputs. Specifically, the backdoor uses the following offsets in the network packet for logging error codes and messages.

- 0x2b - Includes an error string with a maximum size of 256 bytes.
- 0x27 - A DWORD value, which has the error code obtained from the Windows API function GetLastError.

Moreover, before sending the report packet, the backdoor applies an extra encryption layer by encrypting the entire packet again.
