Mustang Panda: ToneShell and StarProxy | ThreatLabz

By Sudeep Singh, ThreatLabz

Published: 2025-04-16 · Archived: 2026-04-05 15:44:32 UTC

Technical Analysis

Mustang Panda packs their tools in archive files hosted on a staging server. All the tools detected by ThreatLabz utilized DLL sideloading to execute malicious payloads to evade endpoint detection products.

We will first examine new versions of a known Mustang Panda tool known as ToneShell. ToneShell is designed to download malicious payloads and execute attacker-specified commands on target machines.

ToneShell

ToneShell is one of Mustang Panda's most frequently used malware tools. Since the core functionality has already been publicly documented, this analysis focuses solely on the recent changes worth highlighting. This section examines three variants of ToneShell: Variants 1 and 3 were discovered on Mustang Panda’s staging server, while Variant 2 was identified through a third-party malware repository bundled in a ZIP file instead of a RAR file.

All three ToneShell variants were found in archive files, which include a legitimate executable (EXE) file and a DLL file (ToneShelll) which is sideloaded.

ToneShell Variant	Archive Filename	Executable Name	DLL Name
1	`cf.rar`	`mrender.exe`	`libcef.dll`
2	`ru.zip`	`FastVD.exe`	`LogMeIn.dll`
3	`zz.rar`	`gpgconf.exe`	`libgcrypt-20.dll`

Table 1: Example ToneShell variant filenames.

Seed generation

Random number generators are utilized in multiple sections of ToneShell’s code for two primary purposes:

- As a fallback mechanism to generate a 16-byte GUID if `CoCreateGuid()` fails (as described in the following section).

- To generate a rolling XOR key, which encrypts and decrypts data exchanged between the victim's machine and the C2 server.

Each random number generator requires a seed, which ToneShell derives using various methods. These methods are listed in the table below.

Variant	Description
Variant 1	Derives the seed by calling `GetTickCount` and adding a constant value to it.
Variant 2	Derives the seed by calling `GetTickCount` twice and multiplying the return values together.
Variant 3	Derives the Unix epoch timestamp from the Windows FILETIME format using the code shown below.

Table 2: ToneShell random number seed generation.

```
seed = (*(_QWORD *)&SystemTimeAsFileTime - 116444736000000000i64) / 10000000;
```

Creation of GUID file

Each ToneShell variant observed in-the-wild generates a GUID, or a value derived from the GUID, and writes it to a file on the filesystem. This GUID is used to uniquely identify the infected machine. Recent variants display subtle changes in how this file is created. The table below shows how the three different ToneShell variants create the GUID file.

Variant	Description
Variant 1	1. Calls CoCreateGuid to create a 16-byte GUID. ◦ If this fails, an array of 16 bytes is created, and populated with random bytes using a linear congruential generator (LCG). 2. The generated 16-byte value is written to a file (`C:\Users\public\description.ini`).

Variant	Description
Variant 2	1. Calls CoCreateGuid to create a 16-byte GUID. ◦ If this fails, an array of 16 bytes is created, and populated with random bytes using a linear congruential generator (LCG). 2. Hashes the GUID to generate a 32-bit hash using a custom hash algorithm. 3. The 32-bit hash value is then written to a file (`C:\ProgramData\bcryptprimitive.appcore.tbi`).
Variant 3	1. Calls CoCreateGuid to create a 16-byte GUID. ◦ If this fails, the malware creates an array of 16 random bytes, using the `rand()` function. 2. Randomly generates a value up to 64KB that is used to determine the length of an array, using a custom LCG. 3. Creates and populates the variable length array with randomly generated bytes, using the `rand()` function. 4. Writes the length of the array, the GUID, and the randomly generated array to a file (`%temp%\cryption_event.ini`) using the structure shown below.

Table 3: Shows how each ToneShell variant creates the GUID file.

```
 struct GUIDFILE
{
        DWORD total_length;
        BYTE guid[16];
        DWORD length_of_random_byte_array; // 64KB max size
        BYTE random_byte_array[];
}
```

Rolling XOR key

ToneShell employs a rolling XOR key to encrypt and decrypt network traffic exchanged with its C2 server. This XOR key is generated using an LCG, seeded by values derived through methods previously described. The XOR key size varies across ToneShell variants, with Team T5 documenting sizes ranging from 0x20 to 0x200.

- Variant 1 uses a 0x100-byte XOR key
- Variant 2 uses a 0x100-byte XOR key
- Variant 3 uses a 0x200 byte XOR key

This aligns with Mustang Panda's tactic of rapidly iterating and modifying their tools, likely to evade detection.

FakeTLS header

FakeTLS headers are a key feature of ToneShell, which helps disguise the malware's network activity. They are used to mimic the TLS protocol in network traffic exchanged between the infected machine and the C2 server.

Earlier ToneShell variants utilized the FakeTLS header with the bytes `0x17 0x03 0x03`, corresponding to TLSv1.2. However, newer variants have introduced the FakeTLS header bytes `0x17 0x03 0x04`, to spoof TLSv1.3. Interestingly, in Variant 2, the TLSv1.3 FakeTLS header was used for sending beacons, while the server continued responding with the older TLSv1.2 header.

This modification appears to be an attempt by the threat actor to evade network signature-based detection systems that rely on identifying specific FakeTLS header patterns.

C2 commands

ToneShell variants continue to utilize a custom TCP-based protocol. Compared to earlier versions, these newer variants focus primarily on executing payloads received from the C2, reflecting Mustang Panda's pattern of iterating their tools, likely to evade detection while adopting a more minimalist design for remote shells.

Variant 1 is a very minimal backdoor, which receives files and acts as a reverse shell. The table below shows the C2 commands supported by Variant 1.

Command Number	Description
1	Receives and processes the next command from the C2 server.
2	Creates an empty file at the C2-specified file path and stores the file handle.
3	Appends the bytes received to an open file handle (from Command 2).
4	Appends the bytes received to an open file handle (from Command 2), closes the file handle, and frees up the file path string in memory thereafter.

Command Number	Description
5	Creates a reverse shell (cmd.exe), redirecting both stdin and stdout. ToneShell keeps track of this subprocess using an ID received from the C2 server. A background thread is created to continually send the output from the subprocess back to the C2 server.
6	Finds the subprocess with the given ID and writes to its standard input.
7	Terminates the subprocess with the given ID.

Table 4: C2 commands supported by Variant 1 of ToneShell.

Variant 2 specifically includes functionality to download DLLs from the C2 and execute them within victim processes through DLL injection. The table below shows the C2 commands supported by Variant 2.

Command Number	Description
1	Receives and processes the next command from the C2 server.
2	Pauses operation for 3 minutes.
3	Terminates the current process.
4	Retrieves the name of the infected machine.
5	Identifies usernames that are not built-in accounts.
6	Specifies the file path to create a new file (used in Command 8).

Command Number	Description
7	Defines the size of the file to be downloaded (used in Command 8).
8	Creates and writes a DLL file likely used in conjunction with Command 10.
9	Updates the configuration for the executable path used in shellcode injection (Command 10). Defaults to `C:\WINDOWS\system32\svchost.exe`.
10	This command creates a child process using the victim executable path specified by command 9. Command 10 also supports an option via a parameter to perform token impersonation when creating the process. ToneShell then decodes shellcode that is hardcoded into the malware with randomized padding bytes, and specific arguments, including `dwGUIDHash` (the 32-bit GUID hash), `wUnkSig` (an unknown signature), `dwC2IPAddress` (the IP address of the C2), and `szFilePath` (the path to the DLL that will be injected into the child process). The patched shellcode is then written to the child process, which will load the specified DLL by invoking `LoadLibraryA(szFilePath)`. The first ordinal of this DLL is resolved by using `GetProcAddress(hModule, 1)` and the ordinal is called with the respective arguments: `dwGUIDHash`, `wUnkSig`, and `dwC2IPAddress`.

Table 5: C2 commands supported by Variant 2 of ToneShell.

ToneShell Variant 3 includes functionality to download files and create a subprocess that redirects standard input (stdin), standard output (stdout), and standard error (stderr) streams to the C2 server. The table below shows the C2 commands supported by Variant 3.

Command Number	Description
3	Creates an empty file at the C2-specified file path and stores the file handle.
4	Opens an existing file at the C2-specified file path, moves to the end of the file, and stores the file handle.

Command Number	Description
5	Appends bytes received from the C2 to an open file handle (from Command 3).
6	Deletes the file used in commands 3, 4, or 5.
7	Executes a subprocess using the command-line received from the C2, redirecting the stdin, stderr, and stdout subprocess.
8	Transfers bytes received from the C2 to the stdin subprocess and reads its output.
9	Retrieves the output of the subprocess.
10	Terminates the subprocess and closes all associated handles.

Table 6: C2 commands supported by Variant 3 of ToneShell.

StarProxy

ThreatLabz uncovered a new lateral movement tool associated with Mustang Panda's operations, which we named *StarProxy*. This tool was found within a RAR archive hosted on the group's staging server with the name `client.rar`. The archive contains two files: a legitimate, signed binary (`IsoBurner.exe`) and a malicious DLL (`StarBurn.dll`), which is the StarProxy tool. The StarProxy DLL is sideloaded when the `IsoBurner.exe` file is executed.

Once active, StarProxy allows attackers to proxy traffic between infected devices and their C2 servers. StarProxy achieves this by utilizing TCP sockets to communicate with the C2 server via the FakeTLS protocol, encrypting all exchanged data with a custom XOR-based encryption algorithm. Additionally, the tool uses command-line arguments to specify the IP address and port for communication, enabling attackers to relay data through compromised machines. The figure below illustrates how StarProxy proxies traffic.

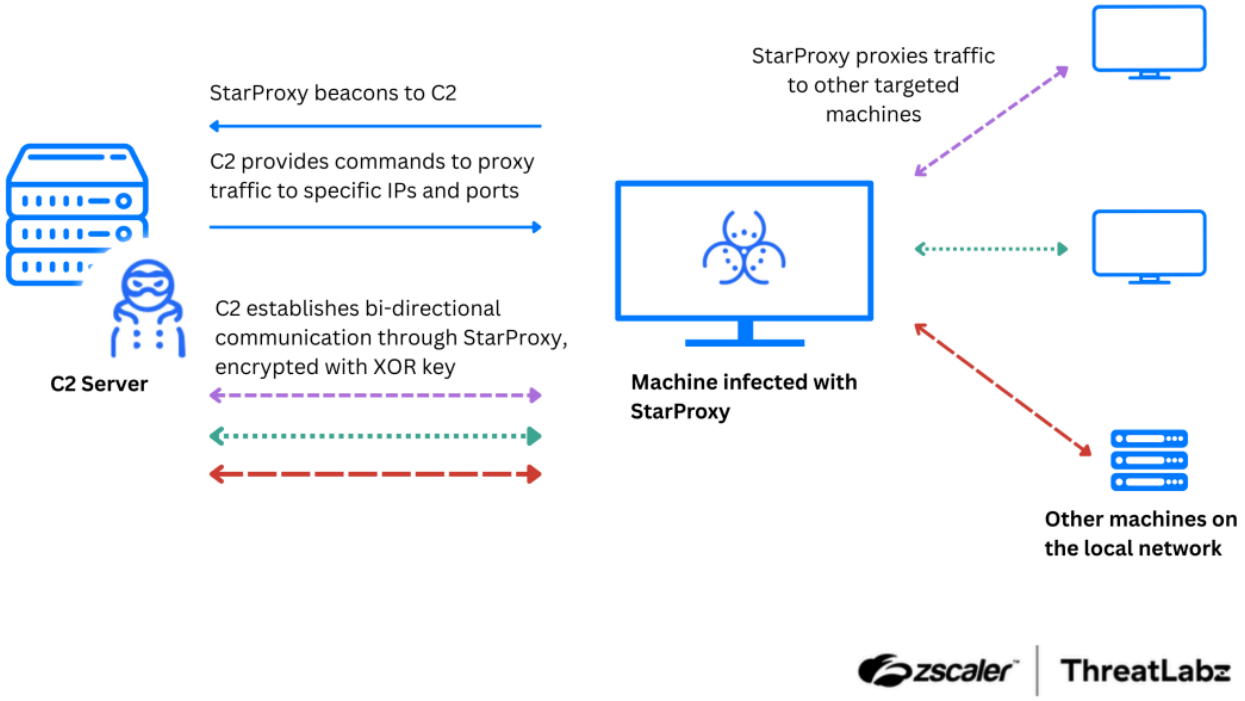

Figure 1: High-level diagram of StarProxy activity.

Given the features of the malware, and the use of command-line arguments, Mustang Panda likely uses StarProxy as a post-compromise tool to access systems that are not reachable directly over the Internet.

Initialization

The StarProxy DLL contains a malicious export function named *StarBurn_UpStartEx*, which expects two command-line arguments: a C2 IP address and a port number. Upon execution, the DLL attempts to connect to the specified C2 IP address. If the connection fails, StarProxy will keep retrying until it succeeds, waiting for one second in between attempts.

Beaconing

Once a connection to the C2 server is successfully established, StarProxy beacons to the C2 to receive commands. If the C2 server signals that there are no commands, StarProxy waits for one second before beaconing again. If the C2 server signals that there are further commands to execute, StarProxy makes additional requests to the C2 server to retrieve the commands and execute them.

Packet encryption and decryption

All messages exchanged between the client and the C2 server are encrypted using two hardcoded 0x100-byte XOR keys. Details of the XOR keys and the encryption algorithm are available on the ThreatLabz GitHub page.

C2 protocol

Request Header

All messages sent by StarProxy to the C2 server are prefixed with a request header (`SEND_HEADER`), which includes the fields below.

```
 struct SEND_HEADER
{
   BYTE fake_tls_header[3]; // 0x17 0x03 0x03.
   WORD message_size;       // Size of data after the FakeTLS header.
   DWORD zero;              // Zero bytes (start of encrypted data with a hardcoded key).
   WORD message_type;       // C2 message type.
   WORD buffer_size;        // Buffer size specific to each message type.
}
```

Response header

All messages sent by the C2 to StarProxy are prefixed with a response header (`RECV_HEADER`), which includes the fields below.

```
 struct RECV_HEADER
{
   BYTE fake_tls_header[3]; // 0x17 0x03 0x03.
   WORD buffer_size;        // Size of the message body (RECV_BODY).
}
```

Notably, the FakeTLS header in these messages is designed to impersonate the TLSv1.2 protocol similar to the earlier ToneShell variants.

Response body (`RECV_BODY`)

The `RECV_HEADER` is followed by the `RECV_BODY` , which includes the command number and data specific to the corresponding command. The generic structure of the `RECV_BODY` is shown below.

```
 struct RECV_BODY
{
   DWORD unused1;
   WORD command_number;  // The ID of the command to be executed.
   WORD unused2;
   BYTE body[];          // Body data (variable length array).
}
```

Beacon message

As mentioned in the beaconing section above, StarProxy sends a beacon message (`CMD_0_MESSAGE`) at regular intervals to check for commands to be executed. The structure of the beacon message is shown in the table below.

```
 struct BEACON_MESSAGE
{
   SEND_HEADER send_header; // message_type = 0 to indicate beacon message.
   DWORD hash_of_rand_arr;  // The hash of a randomly generated array with  variable length, computed using a cu
   DWORD size_of_rand_arr;  // Number of bytes occupied by the random array. This is always a multiple of four.
   BYTE rand_arr[];         // A variable-length random array, likely used to verify the correct decryption of r
}
```

StarProxy uses the following convoluted process to generate the array of random bytes:

- Calls `GetLocalTime()` to retrieve a `SystemTime` structure.
- Adds up all members of the `SystemTime` structure to generate a seed.
- The seed is used to initialize a pseudo-random number generator (PRNG) using `srand()`.
- Generates a random number between 1 and 16. This value will be the variable length (in bytes) of the array containing 32-bit integers.
- The array is filled with randomly generated 32-bit integers, which are then used to generate a 32-bit hash.

In response to the beacon message, StarProxy expects the following response format from the C2:

```
 struct BEACON_MESSAGE_RESPONSE
{
   RECV_HEADER recv_header;   // Contains the FakeTLS header and message size.
   DWORD unused;
   DWORD has_command;         // 0: command data present; any other value: no data present.
}
```

C2 command handler

The StarProxy client supports 5 command IDs (`RECV_BODY.wCmdNum`). The table below describes each command ID and its purpose.

Command ID	Description
1	Command 1 appears to be a ping command.
2	Command 2 also appears to be a ping command, implemented exactly as command 1. The purpose of this duplicated command is unclear.

Command ID	Description
3	Command 3 directs StarProxy to set up a TCP connection to a target IP or domain. Once the connection is established, StarProxy sends the newly created socket descriptor to the C2. StarProxy then sets up a background thread to receive data from the target and forward it back to the C2. Command 3 is used in conjunction with command 4 to establish two-way communication with the target. While this sample of StarProxy is hardcoded to set up a TCP connection, StarProxy contains code which supports both UDP and TCP connections.
4	Forwards message to specified socket descriptor.
5	Closes the socket descriptor specified by the StarProxy C2 server.

Table 7: Lists StarProxy commands.

The purpose of the duplicate command handlers 1 and 2 is not clear, but could suggest that StarProxy is still under development.

Explore more Zscaler blogs
