HijackLoader | ThreatLabz

By Nikolaos Pantazopoulos

Published: 2023-09-08 · Archived: 2026-04-05 13:01:24 UTC

Modules

HijackLoader's modules assist with the code injection and execution process of the final payload. The table below shows the modules identified by ThreatLabz, along with the corresponding CRC32 values and functionality.

Table 2 - HijackLoader modules observed by ThreatLabz

CRC32	MODULE NAME	DESCRIPTION
0x78b783ca	AVDATA	Blocklist of security products' process names. The blocklist includes the CRC32 value of each process name.
0x757c9405	ESAL	Clears out the shellcode data and executes the final payload.
0x6364a15b	ESAL64	64-bit version of the *ESAL* module.
0xe7794e15	ESLDR	Assists with code injection of the main instrumentation shellcode.
0x4fa01ac5	ESLDR64	64-bit version of ESLDR module.
0x93eb1cb1	ESWR	Clears out the shellcode data and executes the *rshell* module.
0x699d0c82	FIXED	Legitimate executable file (e.g., QQPCMgr), which is used for injecting code into its process.
0xfea2e0eb	LauncherLdr	Decrypts the stored modules table file from disk. We have only seen the 64-bit version of this module being included.
0xf4f141c2	LauncherLdr64	64-bit version of LauncherLdr module.
0x74984889	rshell	Relocates, parses and executes the final payload.
0x7b37e907	rshell64	64-bit version of rshell module.
0x3ee477f1	ti	Executed after the first stage. Performs code injection for other stages and modules.
0x2ab77db8	ti64	64-bit version of *ti* module.

CRC32	MODULE NAME	DESCRIPTION
0x4eace798	tinystub	Empty executable file, which is used for patching during the final payload execution process.
0xa1d724fc	tinyutilitymodule.dll	Overwrites the PE headers of a specified file with null bytes.
0x263596ba	tinyutilitymodule64.dll	64-bit version of *tinyutilitymodule.dll* module.
0x1ae7700a	Unknown	Unknown module. Not included in any of the observed binaries. Based on the analyzed code, we assess that it includes a file path along with an optional parameter. The current running file is copied into this new location and executed along with the specified parameter.
N/A	Main instrumentation shellcode	Shellcode injected into the specified target process from the *ti* module. This module is responsible for executing the final payload.

Moreover, each module has a structure that holds information such as:

- Module name
- Module offset in the table
- Module size

Due to the quality of the code and because the rest of the structure fields are not being used, ThreatLabz was not able to identify the purpose for the rest of the structure members. However, ThreatLabz assesses that the following information might be included as well:

- Module timestamp
- Internal names (e.g., *rLdr64* for the *rshell* module)

From the modules mentioned above, our analysis focused on the *ti* and the main instrumentation shellcode modules since these two implement the majority of the core functionality for HijackLoader.

TI module

Anti-Analysis

The anti-analysis techniques are the same as presented in the previous section, but there are two notable differences:

1. The developers have included the Heaven's gate technique.
2. The process blocklist remains the same but the code behavior is different as can be seen in Table 3.

Table 3 - Process blocklist in the HijackLoader *ti* module

PROCESS NAME	DESCRIPTION
avastsvc	• Adds persistence by creating a shortcut (LNK) file and saving the current executable under a random filename in the Windows folder *%AppData%*. *NOTE: A flag, which is set in the decrypted payload at offset 4 is required too.* • Generates a random environment variable name with seed 0xE1ABD1C2 and a new random filename. This is used at a later stage for storing the modules table. • Writes to disk a legitimate executable file (the *FIXED* module) and uses it for code injection at a later stage.
avgsvc	Same as *avastsvc*
a2service	No behavioral code change.
wrsa	No behavioral code change.
msmpeng	No behavioral code change.

Execution Phase

The primary role of the *ti* module is to inject the main instrumentation module, which is responsible for loading the final stage.

The *ti* module executes the next stage by using one of the following methods:

- Re-executes the initial file from a new location and parameter (if the module with the CRC32 value 0x1ae7700a exists).
- Creates a new process (specified in the configuration), maps the hijacked file into it (e.g. *mshtml*), and injects the main instrumentation shellcode. In addition (depending on the configuration flags), it executes the *FIXED* module, which might be used later for code injection.
- HijackLoader might include an additional set of files, which can be used for DLL hijacking. In this case, it writes them to disk along with the encrypted payload (from stage 1) and executes the hijacked executable.
- Executes the *ESLDR* module, which injects the main instrumentation shellcode.

Main Instrumentation Module

The main instrumentation module contains the core functionality for loading and executing the final stage of the infection chain.

Anti-Analysis

The anti-analysis techniques remain the same with the previous aforementioned stages. However, one key difference is the deployment of the *AVDATA* module. The *AVDATA* module contains a set of process names and if

any of them are detected then the code behavior might change. The process names observed by ThreatLabz and their CRC32 values are presented in the table below.

Table 4 - Process name blocklist in the HijackLoader AVDATA module

CRC32	PRODUCT NAME	PROCESS NAME
0xb02ef94	Avast Antivirus	avastsvc.exe
0xc0bfbba0	ESET Smart Security	ekrn.exe
0x40cb21d3	Kaspersky AntiVirus	avp.exe
0xc0fe273f	Symantec Event Manager	ccsvchst.exe
0x9e0539f6	Norton 360	n360.exe
0xe6ef3ab	Avira	avguard.exe
0x8e9e8add	AVG Internet Security	avgsvc.exe
0x923d5594	AVG Internet Security	avgui.exe
0xce1599c2	BitDefender AntiVirus	vsserv.exe
0x83ed98a3	BitDefender AntiVirus	epsecurityservice.exe
0xd50dea99	TrendMicro AntiVirus	coreserviceshell.exe
0x2fba3706	McAfee Antivirus	mcshield.exe
0x1235ed11	McAfee Antivirus	mctray.exe
0x3a39ba4	Norton Internet Security	nis.exe
0xe981e279	Norton Internet Security	ns.exe
0x19e8fad2	BitDefender Antivirus	bdagent.exe
0x5f1c2fc2	Trend Micro Security	uiseagnt.exe
0xc68b2fd8	ByteFence Anti-Malware	bytefence.exe
0xefba2118	McAfee Security Scan Plus	mcuicnt.exe
0xfeb42b97	Internet Security Essentials	vkise.exe
0x6274fa64	Comodo Internet Security	cis.exe
0x4420ef23	Malwarebytes Anti-Malware	mbam.exe
0x31c100e7	360 Safe Guard	zhudongfangyu.exe

CRC32	PRODUCT NAME	PROCESS NAME
0x219b199a	360 Total Security	360tray.exe
0x64760001	N/A	Unknown
0x27873423	N/A	Unknown
0x8bdc7f5b	N/A	Unknown

Each process block in the *AVDATA* module has the following structure:

```
struct avdata_process_block
{
  unsigned int CRC32; // CRC32 value of process.
  unsigned char Execution_Type; // Code execution method for the final payload.
  unsigned char LNK_Persistence_Flag; // Adds persistence with an LNK shortcut file.
  unsigned char unknown; // Not used.
  unsigned char unknown_2; // Not used.
  int BITS_Persistence_Flag; // Adds persistence by creating a BITS job.
  int unknown_3; // Not used.
  int unknown_4; // Not used.
  int Injection_Type; // Defines what code injection method to use.
  int overwrite_pe_headers_with_junk_Flag; // Used during module deployment.
};
```

Persistence

As described in the structure above, persistence on the compromised host is established via the following methods:

- Creation of a BITS job, which points to the executable file.
- Creation of a shortcut file (LNK) in the Windows Startup folder. The shortcut's path is added in a new BITS job and points to the executable file.

Final payload decryption and execution

The embedded payload is decrypted using a bitwise XOR operation with the key being derived from the first 200 bytes. This can be easily represented in Python as follows:

```
enc_data = data[200:]
key = data[:200]
dec = bytearray()
for idx in range(0, len(enc_data), 4):
    dec_int = struct.unpack(" key[idx%200:idx%200 + 4])[0]
    dec.extend(dec_int.to_bytes((dec_int .bit_length() + 7) // 8, byteorder='little'))
```

HijackLoader's shellcode then proceeds with the injection or direct execution of the decrypted payload. The technique the shellcode uses depends on a number of different factors such as the payload's file type and a flag, which is stored in the settings and indicates the injection method to use. In the table below, we describe each case along with a description of the action taken.

Table 5 - HijackLoader Code Injection Methods

INJECTION TYPE	DESCRIPTION
DLL file type.	In the case of a DLL file type, the shellcode parses the PE file and leverages the *ESLR* module, which erases the shellcode data and executes directly the entry point of the DLL.
Code injection when the injection flag is set to 3 and PE relocation is required (Method 1).	Creates a process of the *FIXED* module and writes to disk a file, which includes various information such as: • Pointer to the address of the *rshell* module data. • Process and thread handles of the created process. • Pointer to the address of the final payload. Then it executes the *ESWR* module, which injects the *rshell* module into the created process. As a result, the rshell module reads the written file and therefore the data of the final payload. After relocating and parsing the file, HijackLoader executes the final payload.
Code injection when the injection flag is set to 3 and PE relocation is required (Method 2).	• Creates a process of the *FIXED* module, loads the *tinystub* module, and adds a new PE section to it. • The patched stub module is written to the disk. It is important to note that HijackLoader writes the data without including the "MZ" string. Instead, it delays the execution for a few seconds and then writes to the file the MZ signature byte-by-byte. • Finally, it proceeds with the execution of the final payload as described in the previous method.
Code injection when the injection flag is set to 3 and no PE relocation is required (Method 1).	The implementation is similar to the previous two cases. However, there are a few notable differences: • HijackLoader adds a new section in the final payload file. This new section has the *rshell* module. • In the case of a .NET file, HijackLoader creates an *msbuild* process instead of using the *FIXED* module. • The code searches for certain values in the *rshell* module and replaces them with the same values that are used in the file, which is written to disk (i.e., egg-hunting).

INJECTION TYPE	DESCRIPTION
	• In order to evade detection by security products, it writes random data to the injected process.
Code injection when the injection flag is set to 3 and no PE relocation is required (Method 2).	Same code injection technique as with the previous case, but the data injection takes place with timing delays.
Code injection when the injection flag is set to 3 with a .NET PE file.	• Creates an *msbuild* process, and injects the *rshell* module and the payload. • Searches for certain values in the *rshell* module and replaces them to point to the address of the decrypted payload along with the injection type and the payload size. ***NOTE:*** *No file is written on disk.* • Writes random data in the injected process.
Code injection when the injection flag is set to 4.	• Creates an *msbuild* process and injects the *rshell* module and the payload. • Searches for certain values in the *rshell* module and replaces them to point to the address of the decrypted payload along with the injection type and the payload size. ***NOTE:*** *No file is written on disk.* • Writes random data in the injected process.

Explore more Zscaler blogs
