HijackLoader Updates | ThreatLabz

By Muhammed Irfan V A

Published: 2024-05-06 · Archived: 2026-04-05 15:26:38 UTC

Technical Analysis

The following sections focus on the new changes to HijackLoader.

First stage

The purpose of the loader's first stage is to decrypt and decompress the HijackLoader modules, including the second stage (`ti64` module for 64-bit processes and `ti` module for 32-bit processes), and execute the second stage.

The first stage resolves the APIs dynamically by walking the process environment block (PEB) and parsing the Portable Executable (PE) header. The loader uses the SDBM hashing algorithm below to resolve APIs.

```
def SDBMHash(apiName):
  finalHash = 0
  for i in apiName:
       finalHash = (finalHash*0x1003F) & 0xFFFFFFFF
       finalHash = (finalHash + ord(i)) & 0xFFFFFFFF
  return finalHash
```

Using the SDBM hash algorithm above, the loader resolves the WinHTTP APIs to check for an internet connection. This is achieved with the following URL:

```
https://apache.org/logos/res/incubator/default.png
```

The loader repeats this process until there is an internet connection. After that, HijackLoader decrypts embedded shellcode by performing a simple addition operation with a key. It then sets the execute permission using VirtualProtect for the decrypted shellcode and calls the start address of the shellcode.

Blocklist processes

The shellcode uses the SDBM hash algorithm again to resolve additional APIs. After that, the shellcode uses the `RtlGetNativeSystemInfo` API to check for blocklisted processes running on the system. In previous versions of HijackLoader, the loader checked for five processes related to antivirus applications. Now, the code only checks for two processes.

The names of current running processes are converted to lowercase letters and the SDBM hashing algorithm is compared with the hashes of the two processes. If these values match, execution is delayed using the `NtDelayExecution` API. Information for the two blocklisted processes can be found in the table below.

Hash Value	Process Name	Description
`5C7024B2`	`avgsvc.exe`	The `avgsvc.exe` file is a software component of AVG Internet Security.
`B03D4537`	Unknown	N/A

Table 1: Processes blocklisted by HijackLoader.

Note that the AVG process was also previously blocklisted by earlier versions of HijackLoader.

Second stage loading process

There are two methods that HijackLoader uses to load the second stage, both of which are embedded in the malware's configuration. HijackLoader retrieves a `DWORD` from the configuration at offset 0x110 and XOR's the value with a `DWORD` from the configuration at offset 0x28. Let's refer to the result of this XOR as the value "a".

The malware loads a copy of itself into memory using `GlobalAlloc` and `ReadFile`. Then it reads a `DWORD` from the configuration at offset 0xC and adds this value to the address where the malware file was loaded into memory, and then reads a `DWORD` from this address. Let's call this value "b".

- If "a" and "b" do not match, a PNG file is downloaded and used to load the second stage.
- If "a" and "b" match, an embedded PNG is used to load the second stage.

The image below shows an example of an embedded PNG when rendered using an image viewer.

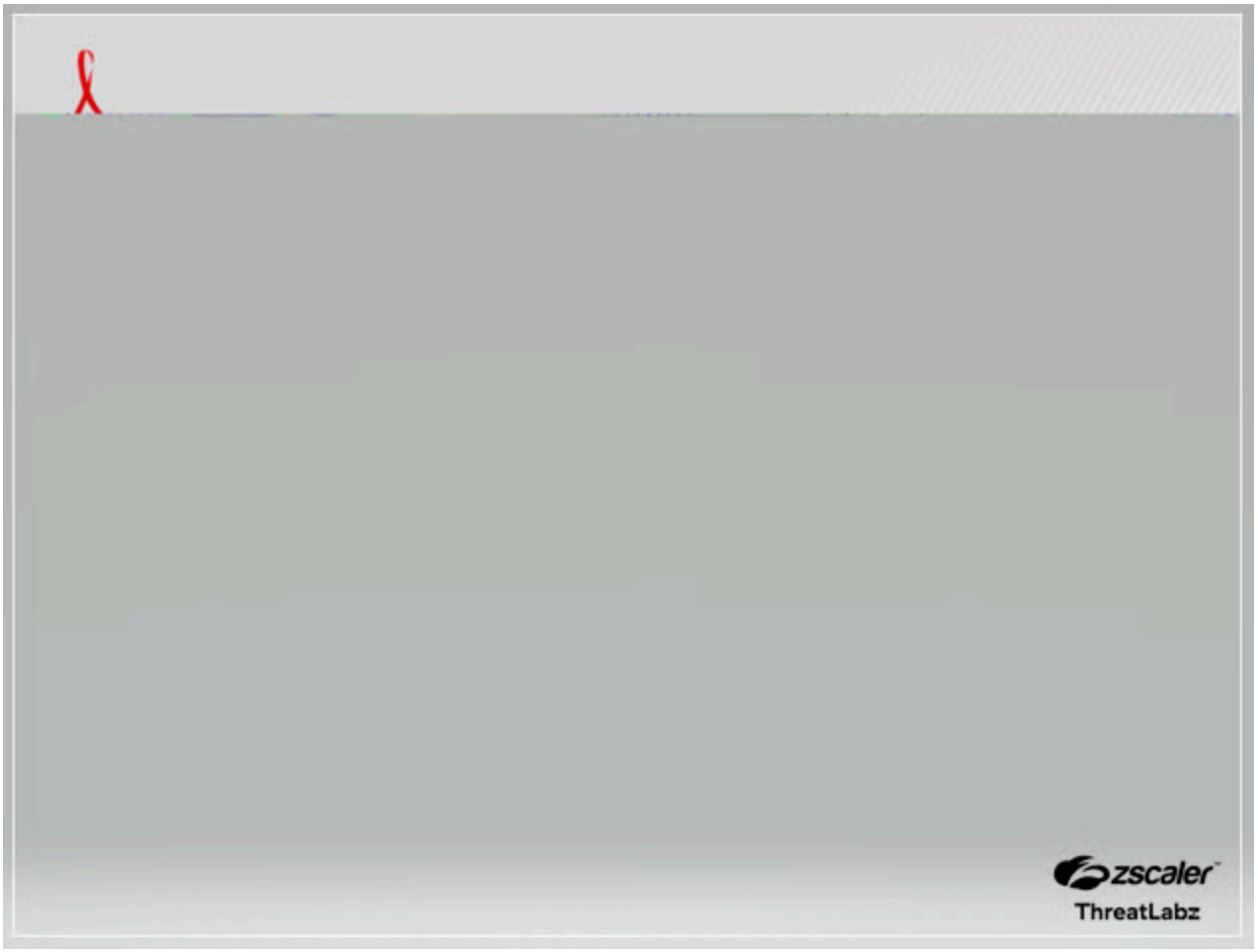

Figure 1: An embedded PNG image containing the encrypted modules used by HijackLoader in a PNG viewer.

PNG payload

The screenshot below displays the decompiled HijackLoader code that checks whether the PNG is embedded within the file or if it needs to be downloaded separately.

```
CurrentFileHandle = (*(code *)APIStruct->GetModuleHandleW)(0);
local_1b0 = APIStruct;
srand = APIResolve(msvcrt.dll,ConfigArray->srandHash,ConfigArray);
rand = APIResolve(msvcrt.dll,ConfigArray->randHash,ConfigArray);
GetComputerNameW = APIResolve(kernel32.dll,ConfigArray->GetComputerNamewHash,ConfigArray);
local_80 = 0;
FileContents = 0;
FileName = 0;
mw_getCurrentFileName(APIStruct,&FileName,0);
local_1dc = 0;
mw_ReadFile(FileName,APIStruct,&FileContents,&local_1dc);
VirtualProtect = (code *)APIResolve(kernel32.dll,ConfigArray->VirtualProtectHash,ConfigArray);
a = ConfigArray->xorVal1 ^ ConfigArray->xorVal2;
Offset = ConfigArray->OffsettoMatch;
b = (uint *)(Offset + FileContents);
iSWeborEmbedded = a == *b;
local_a0 = (uint *)(Offset + FileContents);
Pointer = local_a0 + 1;
local_ac = *local_a0;
local_b0 = *Pointer;
local_1e0 = 0;
local_1e8 = (int *)0x0;
local_1f0 = 0;
local_b4 = ConfigArray->field75_0x78;
local_b8 = ConfigArray->field71_0x68;
if ((bool)iSWeborEmbedded) {
  local_1e0 = *Pointer;
  mw_Stegnography(local_a0 + 2,local_1dc,local_1e0,&local_1f0,APIStruct,(int)ConfigArray);
}
else {
  local_c0 = mw_GlobalAlloc(APIStruct,local_ac);
  memCopy(local_c0,local_a0 + 2,local_ac);
  local_c4 = 0;
  local_c8 = 0;
  for (local_14 = 0; local_14 < local_ac; local_14 = local_14 + 4) {
    local_d0 = (uint *)(local_c0 + (int)local_14);
    *local_d0 = *local_d0 ^ local_b0;
  }
  local_d8 = local_c0;
  local_e0 = mw_GlobalAlloc(APIStruct,0);
```

Figure 2: The decompiled output of HijackLoader to find if the PNG is embedded or if it should be downloaded.

If the PNG is embedded, the malware starts searching for the PNG image with the following bytes: `49 44 41 54 C6 A5 79 EA`. This contains the IDAT header `49 44 41 54` and magic header `C6 A5 79 EA`.

The logic replicated below in Python shows how the malware finds the IDAT header followed by the magic header.

```
 checkFlag = 0
with open(malware_file, "rb") as input_file:
 input_file.seek(0)
 file = input_file.read()
 offset = 0
```

```
try:
 resultOffsetNextByte = file.index(b'\x49\x44\x41\x54\xC6\xA5\x79\xEA', offset + 1)
 print("Found Corect PNG Image")
 checkFlag = 1
except ValueError:
  print('Could not Find PNG with Correct Header')
```

The screenshot below shows the IDAT header followed by the magic header `C6 A5 79 EA`.

Figure 3: The format of the IDAT header.

If the PNG needs to be downloaded, the URL is decrypted from the configuration using a simple XOR cipher. Following that, the WinHTTP library is utilized to download the PNG from the decrypted URL.

There are multiple encrypted blobs in the PNG file. Each encrypted blob is stored in the format `Blob size:IDAT header`. Each of the encrypted blobs can be found by searching for the IDAT header. The size of each encrypted blob is parsed and data of this size following the header is appended to a new memory address. When the total size is reached, the encrypted data is decrypted using a simple XOR cipher with the key: `32 A3 49 B3` (which is a `DWORD` that follows the `MAGIC Header`). Then, the decrypted blob is decompressed using the LZNT1 algorithm.

This decompressed data contains the modules and configurations used to load the second stage. The offset at 0xF4 of the decompressed data contains a DLL name (in this specific case `pla.dll`) that is loaded into memory. The modules are then parsed to locate the ti module (by name) using the SDBM hashing algorithm. Then, the `ti` module is copied to the DLL's `BaseOfCode` field and is executed.

The screenshot below shows the decompiled output for the injection of the second stage.

```
  DataPrased = Data_parsed;
  local_a4 = 0;
  cmd.exeStringLength = mw_returnStringLength(Data_parsed + 0x90);
  if (cmd.exeStringLength == 0) {
    local_a4 = *(int *)(DataPrased + 0x160);
  }
  local_b0 = DataPrased + 0x3dd + local_a4 + *(int *)(DataPrased + 8);
  tiModuleSize = 0;
  tiModule = mw_gettiModule(CONFIG_DATA,local_b0,&tiModuleSize,CONFIG_DATA2[0x18],CONFIG_DATA2);
  DLLName = mw_WrapGlobalAlloc(CONFIG_DATA,0);
  memCpy2(DataPrased + 0xf4,DLLName);
                  /* PLA.dll */
  DLLName = getFilename(DLLName);
  DLLHandle = (undefined *)mw_LoadLib(CONFIG_DATA,DLLName);
  local_dc = mw_getElfanew(DLLHandle);
                  /* BaseofCodeOffset */
  BaseofCodeofDLL = (code *)(DLLHandle + *(int *)(local_dc + 0x2c));
  local_e0 = mw_WrapGlobalAlloc(CONFIG_DATA,tiModuleSize);
  memCopy(local_e0,BaseofCodeofDLL,tiModuleSize);
  local_70 = 0;
  (*VirtualProtect)(BaseofCodeofDLL,CONCAT44(CONFIG_DATA2[0x45],tiModuleSize),(DWORD)&local_70,
                    lpflOldProtect);
  memCopy(BaseofCodeofDLL,tiModule,tiModuleSize);
  (*VirtualProtect)(BaseofCodeofDLL,CONCAT44(local_70,tiModuleSize),(DWORD)&local_70,
                    in_stack_fffffaf4);
  local_e8 = BaseofCodeofDLL;
  local_f4 = CONFIG_DATA2[0x45];
  local_f0 = BaseofCodeofDLL;
  local_ec = tiModuleSize;
  (*BaseofCodeofDLL)();
}
return;
```

Figure 4: The decompiled output for the injection of the second stage.

Second stage

The main purpose of the second stage is to inject the main instrumentation module. To increase stealthiness, the second stage of the loader employs more anti-analysis techniques using multiple modules. The modules have features that include a UAC bypass, Windows Defender Antivirus exclusion, using Heaven's Gate to execute x64 direct syscalls, and process hollowing.

Modules

The malware developers have continued to create new modules. The table below shows the HijackLoader modules added since our last blog.

Hash Value	Module Name	Description
`8858AC11`	`modCreateProcess`	Before transferring control to the `modCreateProcess` module, the `ti` module is copied to a new address and XOR'ed with a

Hash Value	Module Name	Description
		performance counter value retrieved by calling the `QueryPerformanceCounter` API. The new address of the XOR'ed `ti` module and the XOR key are stored for later use. When control is transferred to the `modCreateProcess` module, HijackLoader begins by overwriting the entire `ti` module with zeros. HijackLoader then performs x86 call stack spoofing. Next, the `modCreateProcess` module copies the `TinycallProxy` module into the text section of the system DLL specified by the `SM` module and uses this copied `TinycallProxy` module to call `CreateProcess`. The parameters `lpCommandLine`, `dwCreationFlags`, `lpStartupInfo`, and `lpProcessInformation` are supplied by the `ti` module based on the requirements. After the API call, the return address pointers are patched with the actual return addresses, and the XOR'ed `ti` module is decrypted and restored to its original address. This technique helps obfuscate the origin of the API call.
`9757C10F`	`modCreateProcess64`	This module is the 64-bit version of `modCreateProcess`.
`3B2859F5`	`modUAC`	Handles the encryption and decryption of the `ti` module, performs x86 call stack spoofing, and utilizes the `TinycallProxy` module to call APIs, similar to the `modCreateProcess` module. If the first DWORD of the `UACDATA` module is two, `modUAC` elevates its privilege using the `runas` operation. Otherwise, `modUAC` bypasses UAC using the CMSTPLUA COM interface.
`7366BCF3`	`modUAC64`	This module is the 64-bit version of `modUAC`.
`4F7A1A39`	`modWriteFile`	Only handles the encryption and decryption of the `ti` module, similar to the `modCreateProcess` module. `modWriteFile` does not perform x86 call stack spoofing or use the `TinycallProxy` module to call APIs. `modWriteFile` uses `CreateFile` to obtain the file handle and `WriteFile` to write the content. The parameters `lpFileName` and `dwCreationDisposition` for `CreateFile`, and `lpBuffer` and `nNumberOfBytesToWrite` for `WriteFile`, are provided by the `ti` module based on the requirements.
`1C549B37`	`modWriteFile64`	This module is the 64-bit version of `modWriteFile`.

Hash Value	Module Name	Description
`1003C017`	`WDDATA`	This module contains a PowerShell command to add a Windows Defender Antivirus exclusion.

Table 2: The new modules added to HijackLoader.

ti module

The `ti` module is the first module called by the first stage. It dynamically resolves APIs by walking the PEB and parsing PE headers using CRC-32 as a hashing algorithm. It then checks if a mutex, whose name is resolved using CRC-32 hashing, is present on the system. If the mutex is present, the process terminates; otherwise, it continues execution.

Next, the ti module retrieves the environment variable at offset 0x45 from the decompressed data (`%APPDATA%` in our case), expands it, and checks if the current process is running under this directory. If not, it copies itself to this location, executes the copied file, and terminates itself.

Next, the loader uses the Heaven's Gate technique to bypass user mode hooks – this is further explored in the next section.

Bypass user mode hooks for injection

To bypass user mode hooks, the loader uses a combination of Heaven's Gate and a direct syscall technique. The screenshot below shows HijackLoader employing the Heaven’s Gate technique to execute an x64 direct syscall.

```
00004ac2 66 8e e0        MOV        FS,AX
00004ac5 89 65 f4        MOV        dword ptr [EBP + local_10],ESP
00004ac8 83 e4 f0        AND        ESP,0xfffffff0
00004acb 6a 33           PUSH       0x33
00004acd e8 00 00        CALL       LAB_00004ad2
         00 00

                     LAB_00004ad2                                    XREF[1]:     00004acd(j)
00004ad2 83 04 24 05     ADD        dword ptr [ESP]=>local_88,offset X64_Direct_Syscall
00004ad6 cb              RETF
```

Figure 5: HijackLoader using Heaven’s Gate to execute a x64 direct syscall.

Subsequently, the execution is returned to the 32-bit code. Following this, the next stage is injected. The process designated for injection is stored in the decompressed data at offset 0x90 (`%windir%\SysWOW64\cmd.exe` in our specific case). The `cmd.exe` is created as a child process, and the main instrumentation module is injected using

process hollowing. The main instrumentation module uses the same process mentioned in our previous blog to decrypt the final payload and execute it.

Explore more Zscaler blogs
