HijackLoader Updates

zscaler.com/blogs/security-research/hijackloader-updates

Muhammed Irfan V A

Zscaler Blog

Get the latest Zscaler blog updates in your inbox

Subscribe

Security Research

Introduction

HijackLoader (a.k.a. IDAT Loader) is a malware loader initially spotted in 2023 that is capable of using a variety of modules for code injection and execution. It uses a modular architecture, a feature that most loaders do not have – which we discussed in a previous HijackLoader blog. ThreatLabz researchers recently analyzed a new HijackLoader sample that has updated evasion techniques. These enhancements aim to increase the malware's stealthiness, thereby remaining undetected for longer periods of time. HijackLoader now includes modules to add an exclusion for Windows Defender Antivirus, bypass User Account Control (UAC), evade inline API hooking that is often used by security software for detection, and employ process hollowing.

In addition, HijackLoader's delivery method involves the use of a PNG image, which is decrypted and parsed to load the next stage of the attack. ThreatLabz observed HijackLoader being used to drop multiple malware families, including Amadey, Lumma Stealer, Racoon Stealer v2, and Remcos RAT.

In this blog post, we discuss HijackLoader updates and provide a Python script to extract the malware configuration and modules from HijackLoader samples. Additionally, we delve into the malware families deployed by HijackLoader from March 2024 to April 2024.

Key Takeaways

- HijackLoader is a modular malware loader that is used to deliver second stage payloads including Amadey, Lumma Stealer, Racoon Stealer v2, and Remcos RAT.
- HijackLoader decrypts and parses a PNG image to load the next stage.
- HijackLoader now contains the following new modules: `modCreateProcess`, `modCreateProcess64`, `WDDATA`, `modUAC`, `modUAC64`, `modWriteFile`, and `modWriteFile64`.
- HijackLoader has additional features like dynamic API resolution, blocklist process checking, and user mode hook evasion using Heaven's Gate.
- ThreatLabz researchers created a Python script to decrypt and decompress the second stage and extract all HijackLoader modules.

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`.

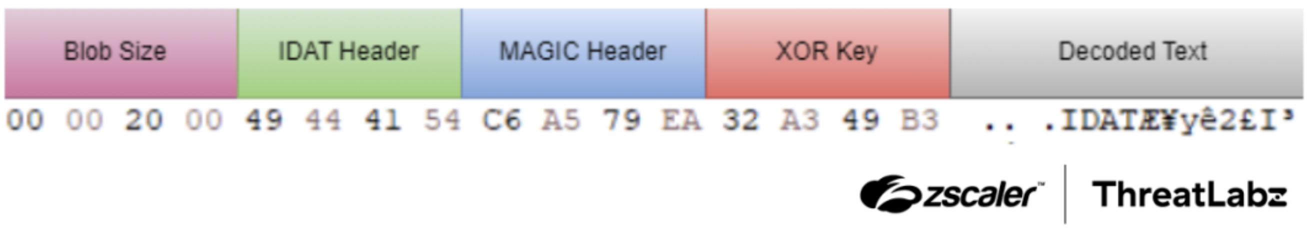

Figure 3: The format of the IDAT header.

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`	This module takes a pointer to an array as an argument. This array contains `lpCommandLine`, `dwCreationFlags`, `lpStartupInfo`, and `lpProcessInformation` which is used to create the new process.
`9757C10F`	`modCreateProcess64`	This module is the 64-bit version of `modCreateProcess`.

Hash Value	Module Name	Description
3B2859F5	modUAC	This module also takes a pointer to an array as an argument. It is used to bypass UAC using the CMSTPLUA COM interface. If a UAC bypass is required, there will be an additional module called UACDATA containing additional data required to perform the UAC bypass.
7366BCF3	modUAC64	This module is the 64-bit version of modUAC.
4F7A1A39	modWriteFile	This module also takes a pointer to an array as an argument. This array contains an XOR key, source address, destination address, and data size. The source data is XOR'ed with the key and copied to the destination address. The destination address (which contains the decrypted data) and data size are given as arguments to WriteFile.
1C549B37	modWriteFile64	This module is the 64-bit version of modWriteFile.
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.

Malware Delivery

In March 2024, ThreatLabz researchers analyzed around 50 samples of HijackLoader with an embedded PNG to identify which families are currently distributed by HijackLoader.

HijackLoader was used to distribute the following malware:

- **Amadey:** Trojan that collects data about the victim's system and is capable of loading other malware. Amadey emerged as the most prevalent family delivered by HijackLoader, comprising 52.9% of observed instances—a notable margin above other malware families.
- **Lumma Stealer (aka LummaC2 Stealer):** Information stealer that steals data from items like cryptocurrency wallets, steam accounts, KeePass, FileZilla, and browser extensions.
- **Racoon Stealer v2:** Information stealer that steals data such as saved passwords, cookies, auto-fill data, and cryptocurrency wallets.
- **Remcos:** Remote Access Trojan (RAT) used to gain backdoor access to a victim's system.
- **Meta Stealer:** Information stealer that targets browsers, cryptocurrency wallets, wallet extensions and steam accounts, and shares many similarities with Redline Stealer.
- **Rhadamanthys:** Information stealer that targets wallets, emails, note-keeping apps, and messengers.

The figure below illustrates the distribution of malware families delivered by HijackLoader.

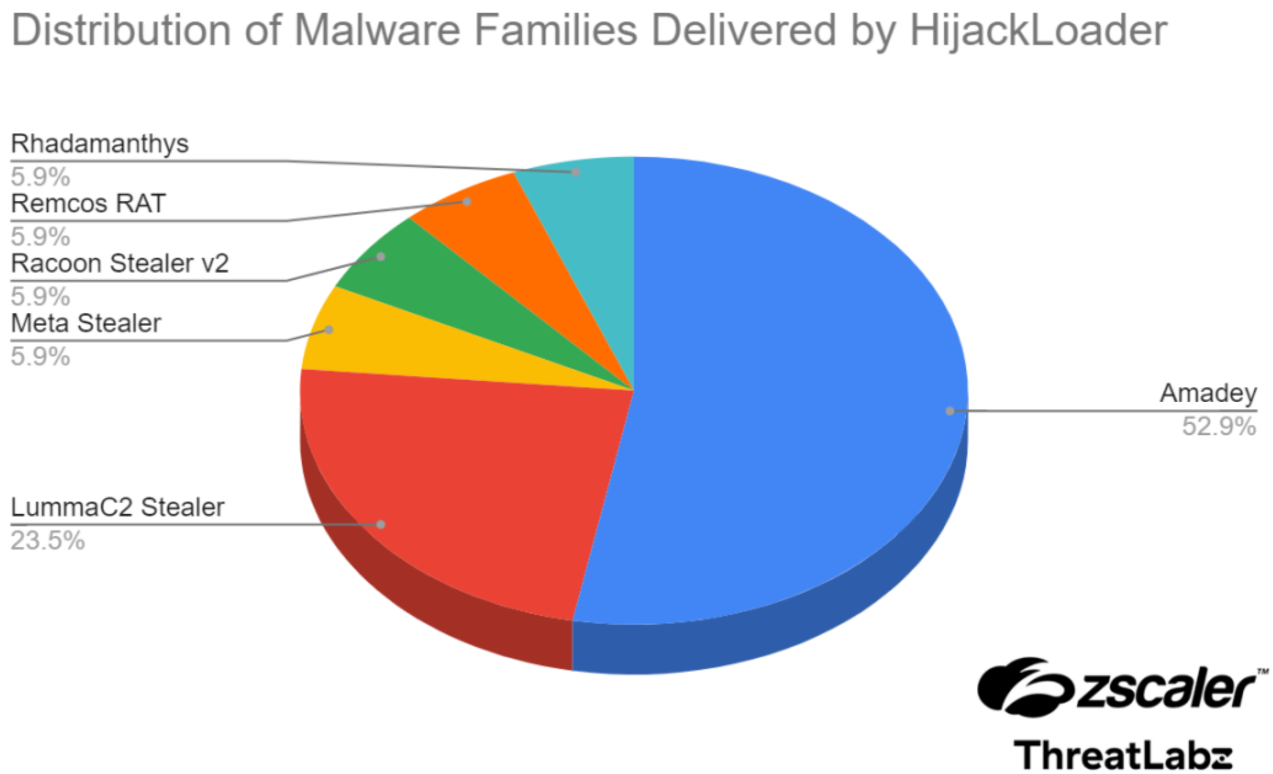

Figure 6: A pie chart showing the different malware families distributed by HijackLoader.

Conclusion

HijackLoader has emerged as a significant threat, delivering multiple malware families such as Amadey, Lumma Stealer, Racoon Stealer v2, and Remcos RAT. Among these, Amadey has been the most commonly delivered family by HijackLoader. The loading of the second stage involves the use of an embedded PNG image or PNG image downloaded from the web, which is decrypted and parsed to load the `ti` module. Additionally, new modules have been integrated into HijackLoader, enhancing its capabilities and making it even more robust.

To assist the research community in analyzing HijackLoader, we have created a Python script that is available in our GitHub repository. The script enables the decryption and decompression of the second stage, providing access to HijackLoader's modules.

ThreatLabz is actively monitoring this campaign and ensuring that Zscaler customers are protected from cybersecurity threats.

Zscaler Coverage

Zscaler's multilayered cloud security platform detects indicators related to HijackLoader at various levels. The screenshot below depicts the Zscaler Cloud Sandbox, showing detection details for HijackLoader.

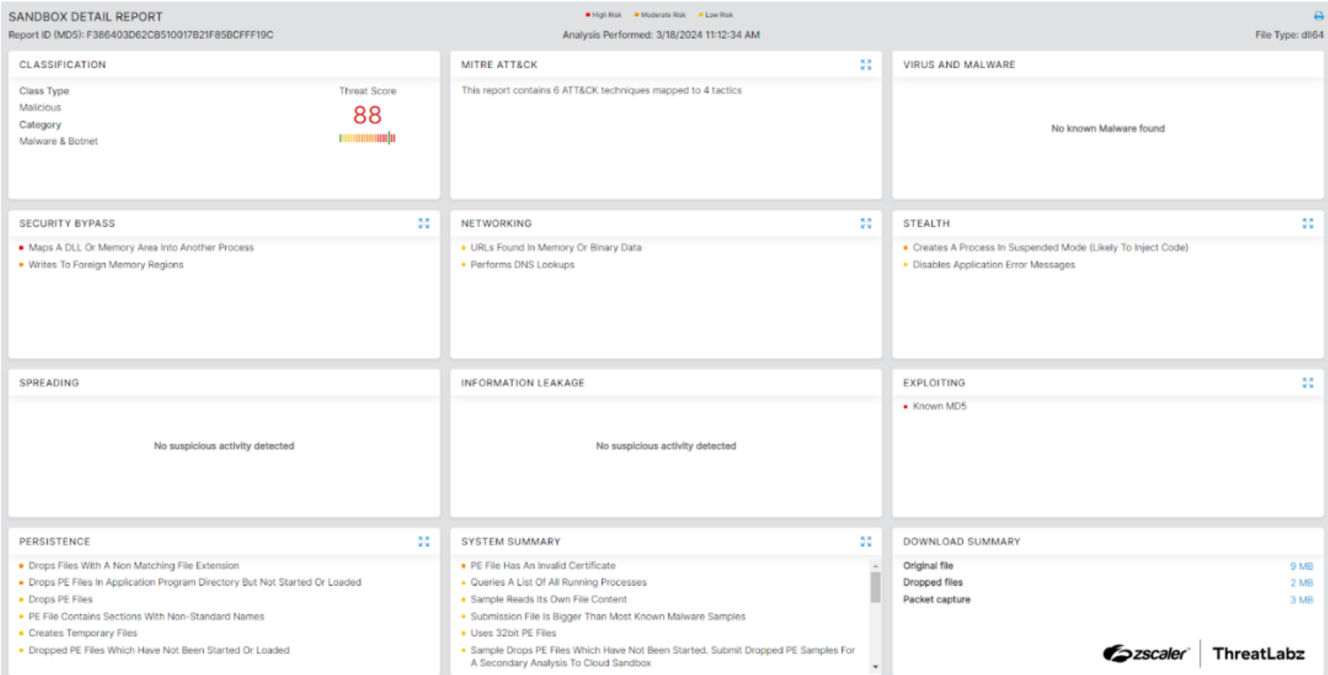

Figure 7: Zscaler Cloud Sandbox report

In addition to sandbox detections, Zscaler's multilayered cloud security platform detects indicators related to HijackLoader at various levels with the following threat names:

Indicators Of Compromise (IOCs)

Host indicators

Type	Indicator(s)	Description
SHA256	• 7a8db5d75ca30164236d2474a4719046a7814a4411cf703ffb702bf6319939d7 • d95e82392d720911f7eb5d8856b8ccd2427e51645975cdf8081560c2f6967ffb' • fcadcee5388fa2e6d4061c7621bf268cb3d156cb879314fa2f518d15f5fa2aa2 • f37b158b3b3c6ef9f6fe08d0056915fc7e5a220d1dabb6a2b62364ae54dca0f1 • e0a4f1c878f20e70143b358ddaa28242bac56be709b5702f3ad656341c54fb76 • cf42af2bdcec387df84ba7f8467bbcdad9719df2c524b6c9b7fffa55cfdc8844 • c215c0838b1f8081a11ff3050d12fcfe67f14442ed2e18398f0c26c47931df44 • 9b15cb2782f953090caf76efe974c4ef8a5f28df3dbb3eff135d44306d80c29c • 56fd2541a36680249ec670d07a5682d2ef5a343d1feccbcf2c3da86bd546af85 • 1fbf01b3cb97fda61a065891f03dca7ed9187a4c1d0e8c5f24ef0001884a54da	HijackLoader malware which uses an embedded PNG to load the next stage.

Network indicators

Type	Indicator	Description
URL	hxxp://discussiowardder[.]website/api	LummaStealer C2

MITRE ATT&CK Techniques

ID	Technique Name
TA0002	Execution
T1547.001	• Boot or Logon Autostart Execution • Registry Run Keys / Startup Folder
T1548.001	Abuse Elevation Control Mechanism
T1027.007	Dynamic API Resolution
T1140	Deobfuscate/Decode Files or Information
T1055	Process Injection
T1620	Reflective Code Loading
T1562.001	Impair Defenses: Disable or Modify Tools
T1057	Process Discovery

Appendix

Visit our GitHub repository to access the Python script to aid in your malware analysis.

Thank you for reading

Was this post useful?

Yes, very!Not really