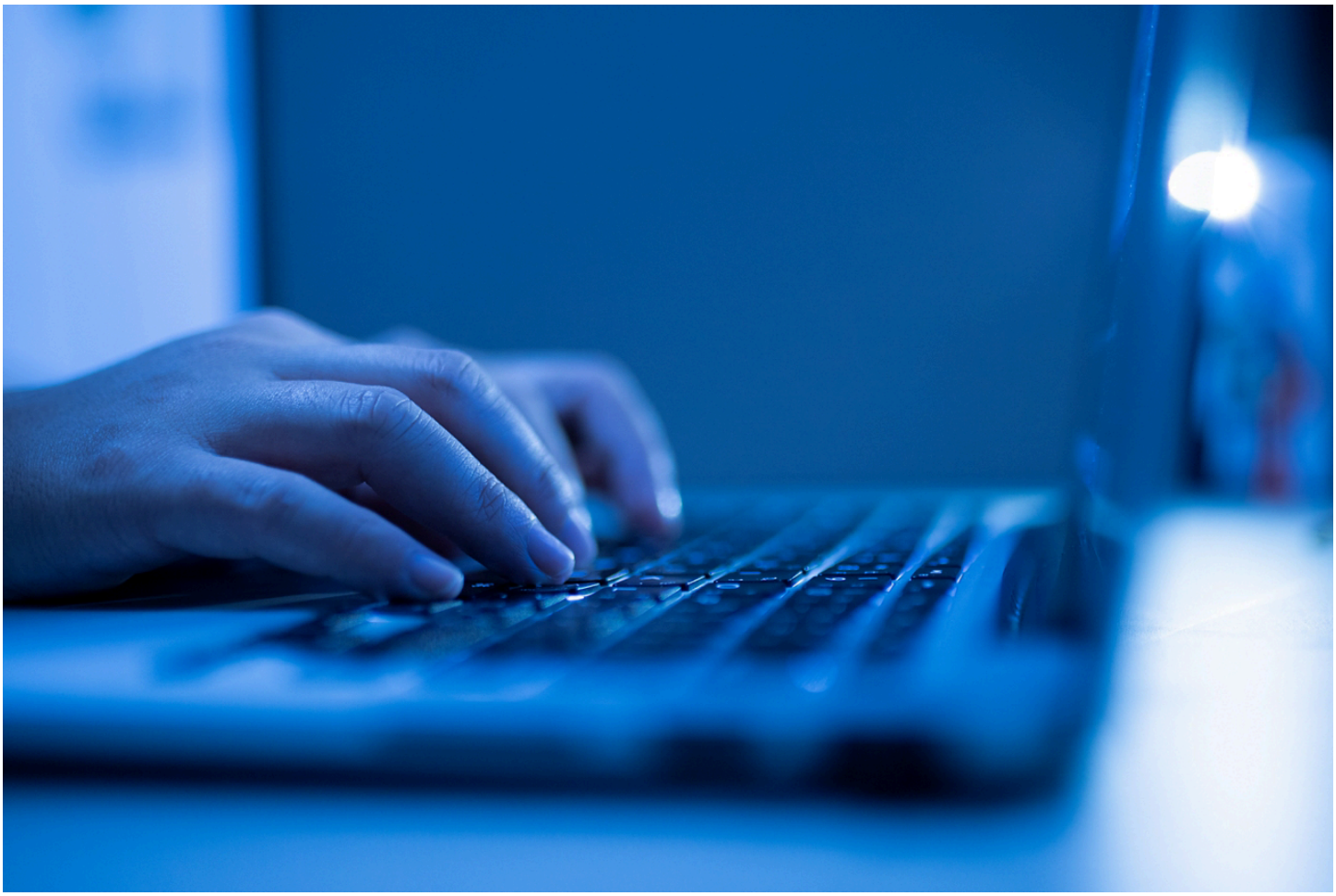
X-Force Threat Analysis Report: LATAM baited into the delivery of PureHVNC

Melissa Frydrych, Raymond Joseph Alfonso ⋮



Authors

Melissa Frydrych-Dean

Threat Hunt Researcher

IBM

Raymond Joseph Alfonso

Malware Reverse Engineer

IBM X-Force

Between August and October 2025, IBM X-Force observed several emails targeting likely Colombian, Spanish-speaking individuals with themes relating to the Attorney General's office of Colombia. The emails entice the user to download an "official document" from the judicial information system, which starts the infection chain of executing a Hijackloader executable that leads to the PureHVNC Remote Access Trojan (RAT).

Threat type

- Phishing

Background

Between August and October 2025, X-Force observed several emails targeting users likely residing in Colombia with emails imitating the Attorney General's office of Colombia with official document downloads. The emails aim to use Hijackloader to deliver several payloads, including PureHVNC. Hijackloader itself has not been widely used in campaigns targeting users within Latin America (LATAM), and previously, there were no observable campaigns by X-Force where LATAM users have been targeted to deliver PureHVNC. In 2024, there are details of Hijackloader being used to load RemcosRAT in campaigns targeting CrowdStrike customers, likely from LATAM countries (based on Spanish filenames and instructions). The delivery of PureHVNC RAT is interesting in that X-Force has not previously observed any campaigns where PureHVNC was delivered to Spanish-speaking users. PureHVNC RAT is part of a set of tools sold by PureCoder. The malicious tools are readily for sale on the dark web on underground forums, as well as on Telegram.

Analysis

Overview

Users are presented with an email purporting to be an official correspondence related to the Attorney General's office of Colombia. The email states that a lawsuit has been filed by a former employee, and is being processed before the labor courts. Attached to the email is an SVG file, which is opened by the victim in Google Drive. In most instances, the document preview is visible and is ready for download by clicking on the download button. In one instance, the victim was presented with a "Couldn't preview file" and a download button, which opened the file in Google Drive. In any case, while in Google Drive, clicking anywhere on the document will download a ZIP archive file, and the victim is now presented with a "Download Complete" page containing a password such as "KC4SX87". The ZIP file contains several additional files, one being an executable file for which the user needs the password in order to execute it if clicked. Clicking on the EXE file will initiate the infection chain, whereby Hijackloader is used to deploy several different payloads, including PureHVNC.

Infection chain example for PureHVNC

- 02 BOLETA FISCAL.exe (javaw.exe) → JLI.dll → MSTH7EN.DLL → Sumhand.zam → Plagkeg.zk → PureHVNC → sofiavergara[.]duckdns[.]org

Malware stage 1: DLL side-loading

Hijackloader uses a technique called DLL side-loading, which abuses the search order Windows uses to locate required libraries to execute a malicious DLL. Hijackloader uses a legitimate *javaw.exe* file that has been renamed with a judiciary-themed name (02 BOLETA FISCAL.exe). Since one of the dependencies of javaw.exe is JLI.dll, Hijackloader places a modified version of JLI.dll in the same directory. When the renamed *javaw.exe* is launched, the operating system also loads the malicious DLL from the local directory.

The primary function of the malicious JLI.dll is to load the 2nd stage payload, MSTH7EN.dll. It does this by calling the *LoadLibraryW()* API, which loads MSTH7EN.dll into the process's address space. The API call returns the image base address of the newly loaded DLL. This address is then added to a specific offset to calculate the entry point of the malicious code in MSTH7EN.dll.

```
if ( v25 ) {   *v25 += v25;   *(v25 - 117) += v26;   v27 = v42;   v28 = *v26;   do   {       ++v26;       *v27++ = v28;       v28 = *v26;   }   while ( *v26 );   *v27 = 0;   LibraryA = LoadLibraryA(v42); //Load MSTH7EN.DLL   if ( LibraryA )     LOBYTE(LibraryA) = ((LibraryA + 31934))(); //Image base of MSTH7EN.dll + 31934 (malicious code offset)   }
```

Malware stage 2: Loading phase

The second-stage payload begins with initialization. To avoid detection, it dynamically loads and resolves all necessary libraries and APIs. Once complete, it verifies that the current working directory matches the Hijackloader's location, ensuring the third-stage payload can be referenced and loaded properly.

The third-stage payload contains an encrypted malware configuration with the following components:

- Key to decrypt the configuration
- Size of the encrypted configuration
- The encrypted configuration data

Upon decryption, the malware configuration contains information, such as the following:

- The name of the DLL to perform DLL hollowing on
- The size of the shellcode for DLL hollowing
- The offset of the malicious code within the shellcode
- Hashes of process names that, if found, will delay the malware's execution
- The first four bytes of the shellcode, used for validation
- A search string used to locate the beginning of the shellcode

The shellcode is then loaded into **vssapi.dll**, which is the DLL specified in the malware's configuration. This is done by calling *VirtualProtect()* to change the memory protection of the DLL's *.text* section

to *PAGE_EXECUTE_READWRITE*. Finally, the shellcode is copied to this writable address, and the flow of execution is transferred to it.

The shellcode acts as a loader, but first, it hashes running process names in the system and compares them to the values specified in the malware configuration. If a match is found, the malware uses the *NtDelayExecution()* API to stall its own execution.

Next, it reads the content of **Plagkeg.zk**. The content of this file is another encrypted malware configuration and HijackLoader's modules. The data is split into multiple chunks, with the initial chunk containing the following information:

- The size of the encrypted data
- Marker ("IDAT")
- A value (0xC6A579EA) used for checking the starting bytes of the shellcode
- The key for decrypting the data

The subsequent chunks follow this structure:

- The size of the shellcode chunk
- Marker ("IDAT")
- The encrypted bytes

To assemble these chunks, HijackLoader iterates through the encrypted data searching for the "*????IDAT*" pattern, where the question marks act as wildcards. Once a match is found, it checks if the four bytes immediately following the pattern are equal to *0xC6A579EA*. This confirms that the initial chunk has been found, which is important because it contains the total size of the shellcode and the decryption key. If the value matches, HijackLoader stores the shellcode bytes into a buffer. The process is repeated for all subsequent chunks, with their shellcode bytes being appended to the same buffer, until no more matching patterns are found.

Once done, the buffer containing the encrypted shellcode is decrypted using an XOR cipher and then decompressed using the *LZNT1* algorithm. The result is a structure that contains various information, such as the final payload, the module structure, etc.

Malware stage 3: ti64 - main module

HijackLoader's functionality is divided into modules. Some contain executable code, while others are simply information used for reference. An example of this is the **COPYLIST** module, which contains the list of filenames related to this variant of HijackLoader. As per Trellix's report, some variants of HijackLoader support up to 40 modules, but the sample analyzed for this report only supports 35. Not all modules are executed, and their use depends on flags specified in the malware configuration.

The table below summarizes the name of each module and its purpose:

HijackLoader loops through these structures and converts each module name to a hash using a custom algorithm. Once the match for the “ti64” module is found, it calculates a pointer to the module’s code by adding the offset of the data to the base of the module data array. This pointer is then returned and used as a reference to shellcode of “ti64”.

Next, the malware performs another DLL hollowing operation to inject the “ti64” module’s shellcode. The target is a DLL specified in the previously decrypted configuration, which in this case is **pla.dll**.

Module Name	Hash	Purpose
AVDATA	0x78B783CA	Contains hashes of security product-related processes
ESAL ESAL64	0x757C9405 0x6364A15B	Cleans the in-memory data of hijackloader and executes the final payload
ESLDR ESLDR64	0xE7794E15 0x4FA01AC5	Used to inject and execute shellcode related to HijackLoader
ESWR ESWR64	0x93EB1CB1 0xAE2762	Clears out the shellcode data and executes the rshell module
FIXED	0x699D0C82	Legitimate PE file used for injecting code into its process
LauncherLdr64	0xF4F141C2	Decrypts configuration files that are stored on the disk
modCreateProcess modCreateProcess64	0x696F778F 0x9B0B7E4B	Used to execute a file
modTask modTask64	0x3115355E 0x9BFAF2D3	Creates persistence using scheduled task
modUAC modUAC64	0xC64EBFDA 0xC97832F9	Used for privilege escalation
modWriteFile modWriteFile64	0xFCE82FC1 0x90415081	Handles file creation on disk
rshell rshell64	0x74984889 0x7B37E907	Executes the final payload
ti ti64	0x3EE477F1 0x2AB77DB8	Serves as the main shellcode that executes all the other modules
TinyCallProxy TinyCallProxy64	0x455CBBC3 0x5515DCEA	Acts as a proxy to execute API calls
tinystub tinystub64	0x4EACE798 0x6E874E5A	Contains dummy executable file, which is used for patching during the final payload execution process
tinyutilitymodule.dll tinyutilitymodule64.dll	0xA1D724FC 0xA0077EA3	Overwrites the PE headers of a specified file with null bytes
SM	0xD8222145	Contains the name of the system DLL used in call stack spoofing or shellcode injection
COPYLIST	0x1AE7700A	A list of files names for copying or deletion

CUSTOMINJECT	0x6703F815	Contains a legitimate executable file which is used for injecting code into its process memory. The process is created in a custom path specified by the **CUSTOMINJECTPATH** module
CUSTOMINJECTPATH	0x192A4446	Contains a file path used to create the legitimate file in the **CUSTOMINJECT** module
X64L	0xCB5B9F3F	Module that is injected into a process to serve as an injection proxy
WDUACDATA	0x4D75088D	Contains the string used for executing commands via *cmd*
WDDATA	0xB718A6AE	Contains a PowerShell command to add a Windows Defender Antivirus exclusion
PERSDATA	0xA2E0AB5D	Contains the configuration used by the **modTask** module to create scheduled tasks
MUTEX	0x1999709F	Contains the name of mutex to check

Privilege escalation

The modUAC module, similar to the other modules, uses **TinycallProxy** to call APIs. If the first DWORD of the **UACDATA** module is 2, it uses the “runas” to elevate its privilege. Otherwise, it uses the *CMSTPLUA COM interface* to bypass UAC.

Evasion

Indirect API calling

In some variants, HijackLoader uses a technique called “*stack spoofing*“ to mask the origin of API and system calls. It does this by using the base pointer register (EBP) to navigate the stack, following the chain of EBP pointers to retrieve the return address from each stack frame. If a return address is not within the .text section of ntdll.dll or kernelbase.dll, HijackLoader stores it for later. This process is repeated until the stack limit is reached or until three consecutive return addresses are found within those system libraries.

Next, it performs call stack spoofing by overwriting the saved, legitimate return addresses with fake ones. Each fake address is generated by selecting a random export from a DLL specified by the **SM** module (in this case, dcd9.dll) and adding a random offset, ensuring the final pointer lands within that module’s *.text* section. *Heaven’s Gate* is then used to perform the syscall. Immediately after the call completes, the original stack addresses are restored.

More recent variants, however, use a different technique. Instead of stack spoofing, HijackLoader loads the target DLL specified by the SM module via *LoadLibraryW()*. It then saves the code from a random offset within that DLL to a temporary buffer and replaces it with the **TinyCallProxy64** module's shellcode, which is designed to call the specified API. Once the call is finished, the original, clean code is restored.

HijackLoader uses these techniques for a select number of functions that are likely to be monitored by AV software, such as *ZwProtectVirtualMemory* and *ZwGetContextThread*.

```
__int64 __fastcall sub_7FF87A86D3B0( GlobalContext_0 *a1, __int64 hash_of_function, __int64 a3, __int64 a4, __int64 a5, __int64 a6, __int64 a7) { Indirect_SYSCALL *syscall_struct; // [rsp+40h] [rbp-28h] __int64 (__fastcall *pAPIFunc)(__int64, __int64, __int64, __int64, __int64); // [rsp+48h] [rbp-20h] syscall_struct = sub_7FF87A86B470(a1, hash_of_function); if ( !syscall_struct ) return 0xFFFFFFFFLL; pAPIFunc = (a1->ntdll_image_base + syscall_struct->api_rva); if ( a1->GlobalContext_1 ) return mw_indirect_api_call(a1->GlobalContext_1, pAPIFunc, a3, a4, a5, a6, a7); else return pAPIFunc(a3, a4, a5, a6, a7); } __int64 __fastcall mw_indirect_api_call( GlobalContext_1 *TinyCallProxy64, __int64 pAPIFunc, __int64 a3, __int64 a4, __int64 NtClose, __int64 a6, __int64 a7) { _BYTE *shellcodeAddress; // [rsp+40h] [rbp-58h] unsigned int v9; // [rsp+48h] [rbp-50h] BYREF int v10; // [rsp+4Ch] [rbp-4Ch] BYREF unsigned int v11; // [rsp+50h] [rbp-48h] _BYTE *clean_code; // [rsp+58h] [rbp-40h] unsigned int v13; // [rsp+60h] [rbp-38h] void (__fastcall *FlushInstructionCache)(__int64, _BYTE *, _QWORD); // [rsp+68h] [rbp-30h] unsigned __int64 random_address; // [rsp+70h] [rbp-28h] __int64 (__fastcall *pShellcodeAddress)(__int64, unsigned __int64, _QWORD, __int64, __int64, __int64, __int64, __int64); // [rsp+78h] [rbp-20h] void (__fastcall *v17)(__int64, _BYTE *, _QWORD); // [rsp+80h] [rbp-18h] v13 = 5; v11 = 0; shellcodeAddress = mw_pick_random_address(TinyCallProxy64);// d3d9.dll address v9 = 0; v10 = 0; if ( !(TinyCallProxy64->VirtualProtect)(shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size), 64LL, &v9) ) return -1LL; clean_code = (TinyCallProxy64->malloc)(LODWORD(TinyCallProxy64->shellcode_size)); wrapper_memcpy(clean_code, shellcodeAddress, TinyCallProxy64->shellcode_size); wrapper_memcpy(shellcodeAddress, TinyCallProxy64->shellcode, TinyCallProxy64->shellcode_size); (TinyCallProxy64->VirtualProtect)(shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size), v9, &v9); if ( TinyCallProxy64->FlushInstructionCache ) { FlushInstructionCache = TinyCallProxy64->FlushInstructionCache; FlushInstructionCache(-1LL, shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size)); } pShellcodeAddress = shellcodeAddress; random_address = mw_pick_random_address(TinyCallProxy64); v11 = pShellcodeAddress(pAPIFunc, random_address, v13, a3, a4, a5, a6, a7); if ( (TinyCallProxy64->VirtualProtect)(shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size), 0x40LL, &v10) ) wrapper_memcpy(shellcodeAddress, clean_code, TinyCallProxy64->shellcode_size); (TinyCallProxy64->VirtualProtect)(shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size), 32LL, &v10); if ( clean_code ) (TinyCallProxy64->free)(clean_code); if ( TinyCallProxy64->FlushInstructionCache ) { v17 = TinyCallProxy64->FlushInstructionCache; v17(-1LL, shellcodeAddress, LODWORD(TinyCallProxy64->shellcode_size)); } return v11; }
```

ANTIVM

Technique	Description

Time-based anti-debugging check	Uses a timing-based evasion technique by measuring the latency of the cpuid instruction. It wraps the cpuid call with rdtsc instructions inside a loop, and if the execution time exceeds a specified threshold, it detects the presence of a debugger or virtual machine.
Hypervisor check	Performs a standard anti-VM check by executing the cpuid instruction and checking the “hypervisor bit” (bit 31) in the returned ECX register. If this bit is set to 1, it indicates the presence of a hypervisor.
Vendor ID check	Performs an anti-VM check by querying the hypervisor information leaf (0x40000000). A return value in EAX that is greater than or equal to 0x40000000 indicates the presence of active hypervisor-specific CPUID leaves.
Checks total RAM	Performs an anti-sandbox check by querying the total physical RAM. It calls NtQuerySystemInformation to calculate the total memory in gigabytes (by right-shifting the byte count by 30) and terminates if the result is below 4GB.
Checks number of processors	Performs an anti-sandbox check by querying the number of CPU cores. It calls NtQuerySystemInformation to get the NumberOfProcessors and compares it against the value specified in the configuration of the ANTIVM module.
Username checking	Compares the current user’s username to the specified value in the ANTIVM module.
Computer name checking	Checks if the computer name consists of only numbers.
Checks current working directory	Checks if the current module path is on the desktop.

A failed anti-virtualization check results in process termination via a call to *ZwTerminateProcess()*.

Unhooking NTDLLs

The unhooking routine compares the *.text* section of the currently loaded ntdll.dll against a clean, mapped copy. It scans for call (0xE8) and jmp (0xE9) instructions and detects a hook if the instruction type or destination address differs between the two versions. If a hook is found, the malware patches the in-memory ntdll.dll by restoring the original, clean bytes.

```
for ( i = 0; ; ++i ) {   result = v5[6];     if ( i >= result )       break;     function_rva = *(v7 + 4LL * *(v6 + 2LL * i));
  if ( *(a1->ntdll_image_base + function_rva) != *(a2->clean_ntdll_buffer + function_rva) )// check if the ntdll
```

functions are hooked // This is done by comparing the first byte of the fuinction. { if (check_if_valid_address(a1, a1->ntdll_image_base, function_rva)) mw_clean_dll(a1, a2, function_rva, a1->ntdll_image_base, a2->clean_ntdll_buffer); } } return result; }

Persistence

HijackLoader's persistence mechanism is also controlled by its configuration. The behavior is dictated by a flag:

- LNK Shortcut (Flag 1)**:** If the flag is set to 1, the HijackLoader creates an LNK file pointing to its own executable path. This shortcut is then moved into the user's startup folder to ensure execution on logon.
- Scheduled Task (Flag 3)**:** If the flag is set to 3, it creates a scheduled task using the **modTask** module.

In addition to these flags, HijackLoader can create another persistence mechanism by checking for a **PERSDATA** module. This module contains the necessary configuration data, such as the task name, to create a second scheduled task.

Injection methods

Injection Type	Description
If the file to inject is a DLLor injection flags is less than 0x3	the final payload will be executed under the same process, so the DLL payload will be mapped in the hollowed DLL.
If the final payload is not a .NET/CLR file, injection flags 0x20 is false and 0x80 is true	Hides the **rshell** payload in a dummy **tinystub** PE using a rolled-back NTFS transaction. It then maps this hidden PE into a suspended process (**FIXED**), where the **ESWR** module hijacks the main thread's context to execute the **rshell** code.
If the final payload is not a .NET/CLR file, injection flags 0x20 and 0x80 are both false	The **FIXED** module is dropped to disk and created as a suspended process. The **ESWR** module is then used to trigger the **rshell** payload's execution within the **FIXED** process.
The injection flags 0x100 is set to true and 0x20 is false	Injects **rshell** into a suspended legitimate system executable (e.g., MSBuild.exe) located by parsing the .NET header for the CLR path. The payload is patched in-memory before being executed via thread context hijacking and clears its own PE headers.
Injection flags 0x4 and 0x80 are both **true**.	Conditionally drops the **FIXED** module, then stores the **rshell** payload in a rolled-back transacted file (**tinystub**). It injects this into the suspended **FIXED** process via section mapping. Execution is

	triggered via thread context hijacking, followed by optionally erases its PE header.
Injection flags 0x4 is true and 0x80 is false.	HijackLoader launches a suspended process, creates and maps a new memory section directly within it, and then writes the patched **rshell** module into this section. Execution is triggered by hijacking the main thread's context to run the **rshell** code.
Injection flags 0x4 is falseand 0x10 is true.	Performs Process Hollowing by launching its **FIXED** module, wiping its main memory section, and then copying in the payload. It writes the "MZ" header in two separate calls. Finally, it injects the patched **rshell** module, modifies the PEB, and optionally erases the payload's PE header.
Injection type is set to 4	Injects the main payload and **rshell** module via section mapping. A section is created and populated locally with the patched rshell and payload, then mapped into a suspended target process (a system native binary or **CUSTOMINJECT** module). Execution is triggered by hijacking the main thread's context to point to the **rshell** entry point.

Conclusion

Users within LATAM regions are increasingly targets of emails impersonating government or judicial entities, with themes often creating a sense of urgency. X-Force observes campaigns that routinely involve an embedded link or ZIP attachments that lead victims to malicious downloaders. Between August and October 2025, X-Force observed several emails targeting users likely residing in Colombia with emails imitating the Attorney General's office of Colombia with official document downloads. Hijackloader is a modular malware with evasion and persistence mechanisms, primarily delivered to users as a ZIP or RAR archive file. The archives contain a malicious DLL that is sideloaded and used to deliver additional payloads. These emails, likely a part of a single campaign, are significant in that the actors utilize the Hijackloader to deliver PureHVNC RAT, a combination not previously observed by X-Force.

Recommendations

- Enable the display of file extensions.
- Examine the operational need to allow traffic to and from DuckDNS domains.
- Take caution in opening email attachments and clicking on embedded links from untrusted or unknown sources.
- Hunt for processes, network traffic and IoCs detailed in this report.

- Install, update, and configure endpoint security software.
- Monitor endpoint rules

Indicators of compromise

Indicator	Indicator Type	Context
troquelesmyj[@]gmail.com	Email	Sender email
nuevos777[.]duckdns[.]org	Domain	C2 Domain
7octubredc[.]duckdns[.]org	Domain	C2 Domain
dckis13[.]duckdns[.]org	Domain	C2 Domain
dckis7[.]duckdns[.]org	Domain	C2 Domain
enviopago[.]mysynology[.]net	Domain	C2 Domain
maximo26[.]duckdns[.]org	Domain	C2 Domain
sofiavergara[.]duckdns[.]org	Domain	C2 Domain
hxxps[:]//drive[.]google[.]com /file/d/1haApB_GMwZb83nw1 YPdIDTLMtksRjkh/view?pli=1	URL	SVG Host
hxxps[:]//drive[.]google[.]com/ file/d/1wzunPhL33jq_ZQug6k0 3hgxi4Eu57VfN/view? usp=sharing	URL	SVG Host
e7120d45ee357f30cb602c0d93 ed8d366f4b11c251c2a3cd4753c5 508c3b15e5	SHA256	ZIP
7e64102405459192813541448c8 fbadc481997a2065f26c848f1e35 94ca404c9	SHA256	RAR
14becb3a9663128543e1868d09 611bd30a2b64c655dfb407a727a 7f2d0fb8b7e	SHA256	Hijackloader
57c49cff3e71bc75641c78a5a72d 8509007a18032510f607c042053 c9d280511	SHA256	Hijackloader
7c3d9ad3f1bd890e3552dc6709 3e161395d4e1fab79ec745220af1 e19a279722	SHA256	Hijackloader
ce42377d3d26853fd1718f69341 c0631208138490decc8e71a5622 df5e9e1f59	SHA256	Hijackloader
a0e4979b4e4a706286438d48f 0e21b0d92cc7bd40c1c3ea5b98 72089aaec0124	SHA256	Hijackloader
6d93a486e077858b75eb814e 9a7bda181189d5833adce7cec7 5775cfda03f514	SHA256	Hijackloader

bdca9849d7263d508b7ed4db bf86bd628932b117b45933cb28 a7e78171d05cdd	SHA256	Hijackloader
1ae61edf35127264d329b7c0e2 bddb7077e34cc5f9417de86ab 6d2d65bad4b4f	SHA256	Hijackloader
2ec31a8a36d73fa8354a7ac0c 39506dbe12638a0dc1b900f5 7620b8d53ae987f	SHA256	Hijackloader
776bbaa44c7788e0ccd5945 d583de9473b6246c4490669 2cb0a52e6329cb213a	SHA256	Hijackloader
9e9997b54da0c633ffcf0a4fb 94e67b482cf7a89522d1b254 778d0c6c22c70ee	SHA256	Hijackloader
b2f733b67f1ef06d9e5ce76d3 cc848f6e7e3ec2d0c363c76d 5175c6cf85f979b	SHA256	Hijackloader
c93e70d20ba2948a6a8a013 df68e5c4d14d59e5f549417d 1a76833bd1c8efd22	SHA256	Hijackloader
d550a2a327394148c0c3d05 df2fe0156783fc313b4038e45 4f9aa2cb2f0f2090	SHA256	Hijackloader
e668ca17fcdfa818aac35f1206 4d10a0288d7d9c6b688966b 695125b760567d6	SHA256	Hijackloader
fe6d0ee45a70359008b2916 e5116c411a955978b5694cc4 57683ab7b26590e47	SHA256	Hijackloader
977f2f18ff13c93406c5702f83 c04a9412760e02028aefc7c1c b7d6f2797a9b5	SHA256	Hijackloader
768ca38878c5bb15650343ce 49292315a9834eaf62fad1442 2d52510c3787228	SHA256	Hijackloader
47245b7d2d8cb6b92308deb 80399e0273193d5bca39da8 5a6b2a87a109d18d85	SHA256	Hijackloader
4484b0ac51536890301a0e6 573b962e069e31abc4c0c6f0 f6fc1bf66bf588a93	SHA256	Hijackloader
0113d9f3d93069a29458b3b4 c33610aae03961014df60a9e8 59f3104086d886a	SHA256	Hijackloader
22d474e729d600dcd84ce139 f6208ce3e3390693afa7b52b0 615174fca6d0fe2	SHA256	Hijackloader

2cbfc482e27a2240a48d2fb6f6f740ff0f08598f83ae643a507c6f12a865dc28	SHA256	Hijackloader
96ee786c5b6167c0f0f770efbace25e97d61e127ef7f58a879b6cf4b57e202c3	SHA256	Hijackloader
33d0c63777882c9ec514be062612a56fdb1f291fcb6676c49480d3cd4501c508	SHA256	PureHVNC
afecefa6d9bd1e6d1c92144209eda320e1fe0f196ffa8e8bc114e7d3a25503f6	SHA256	PureHVNC
85641c8fb94e8e4c5202152dcbb2bb26646529290d984988ecb72e18d63c9bc5	SHA256	PureHVNC
1bf3a1cf9bc7eded0b8994d44cf2b801bf12bc72dc23fb337ddd3a64ac235782	SHA256	PureHVNC

IBM X-Force Premier Threat Intelligence is now integrated with OpenCTI by Filigran, delivering actionable threat intelligence about this threat activity and more. Access insights on threat actors, malware, and industry risks. Install the X-Force OpenCTI Connector to enhance detection and response, strengthening your cybersecurity with IBM X-Force's expertise. Get a 30-Day X-Force Premier Threat Intelligence trial today!