PlugX: A Talisman to Behold

By Max Kersten, Marc Elias, Leandro Velasco, and Alexandre Mundo Alguacil · March 28, 2022

For over a decade, the PlugX malware has been observed internationally with different variants found around the world. This blog covers a PlugX variant that we have named Talisman, a name we based on comparisons with other PlugX variants, and its rather long life since it first emerged in 2008. First, the malware's technical details will be discussed, after which the infrastructure, attribution, and victimology will be covered.

Executive Summary

Talisman is a newly discovered PlugX variant which follows the usual execution process by abusing a signed and benign binary which loads a modified DLL to execute shellcode. The shellcode is used to decrypt the PlugX malware which then serves as a backdoor with plug-in capabilities. Unlike other versions, the malware's internal configuration's signature is different, as well as other minor changes within the code.

We want to mention that a change within the PlugX malware alone does not mean a new threat actor has emerged. The PlugX source code has allegedly circulated online already. This also means that not all PlugX samples are necessarily tied to Chinese actors, although it a prevalent tool in their kit. In the case of Talisman, there is more evidence which points towards a Chinese state-backed actor than a simple change in the malware's codebase, such as the overlaps in the used infrastructure, which is also present in Recorded Future's research. Based on this, Trellix attributes this campaign with medium confidence to the Chinese state-backed RedFoxtrot group.

The victims were in South Asia in the Telecommunication and Defense sectors, and align with China's geopolitical interests. One such initiative is the Belt and Road Initiative, via which China aims to establish strong social economical relationships across Europe, Asia, and Africa via trade.

Technical details

Within the analysis of the PlugX Talisman variant, we will reference the THOR variant of PlugX, which was discovered by Unit42, as well as an earlier version of PlugX documented by DrWeb. These articles describe other versions of PlugX and were of help to us during the analysis of this variant to both understand and compare the different iterations of the malware. Talisman has some differences with other PlugX versions. In the coming sections, we will highlight interesting segments of the malware. Below, a visual overview of the malware's stages is given.

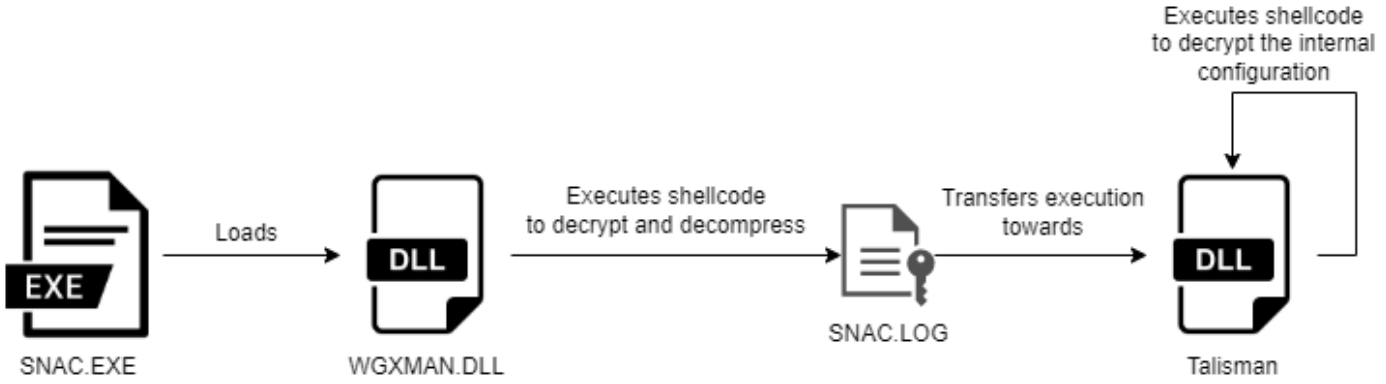


Figure 1. Talisman PlugX execution flow

Stage 1 – The signed and benign executable

Filename	SNAC.EXE
SHA-1	dc40970a3c8f03866e0b700460d3b1f7afa6a433
SHA-256	c09ff32519f112674bd5f4b1687feadf18844c5423e6f28df8be50eb9503e606
MD-5	8e886df3cb6160188f9748f14f249063

The first stage of the malware is a benign executable which is used to evade the prying eyes of security products as valid signatures often help to indicate the trustworthiness of a binary. The signed executables in this campaign have been created by security companies. The sole purpose of the first stage is to load a DLL which has been modified by the attacker. Sideloading a DLL is a commonly seen technique in various PlugX variants, as is also described on the respective MITRE ATT&CK page. Lastly, a third binary file, containing the encrypted Talisman payload, is decrypted by the DLL to complete the full chain of execution.

In the sample we are analyzing, the legitimate executable has the name SNAC.exe, the Talisman DLL Loader is named WGXMAN.DLL, and the encrypted and compressed Talisman payload is named SNAC.LOG.

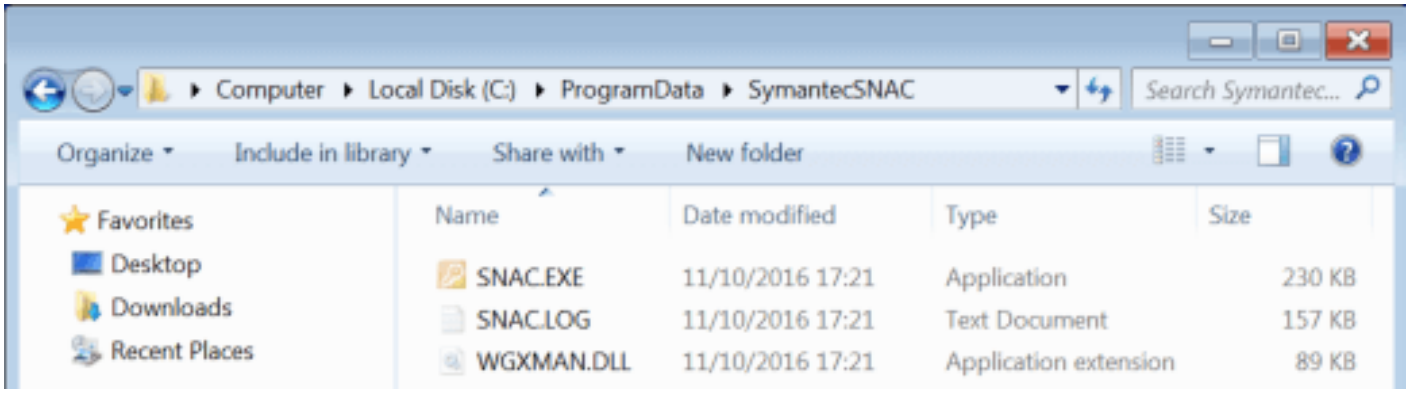


Figure 2. Talisman execution files

Most Talisman PlugX samples we analyzed consist of three-file long execution chains abusing the DLL sideloading technique, which is consistent with the tactics, techniques, and procedures of Chinese state-sponsored threat actors that use this type of execution to launch their malware to evade detection by security solutions.

The table below lists the different observed filenames for the legitimate executables, the Talisman DLL loader and the encrypted Talisman in the analyzed samples.

Legitimate executable	Talisman DLL Loader	Encrypted Talisman
SNAC.EXE	WGXMAN.DLL	SNAC.LOG
RasTls.exe	RasTls.dll	RasTls.dll.res
msvsct.exe	TmDbgLog.dll	TmDbgLog.dll.obj
msrers.exe	TmDbgLog.dll	TmDbgLog.dll.tsc

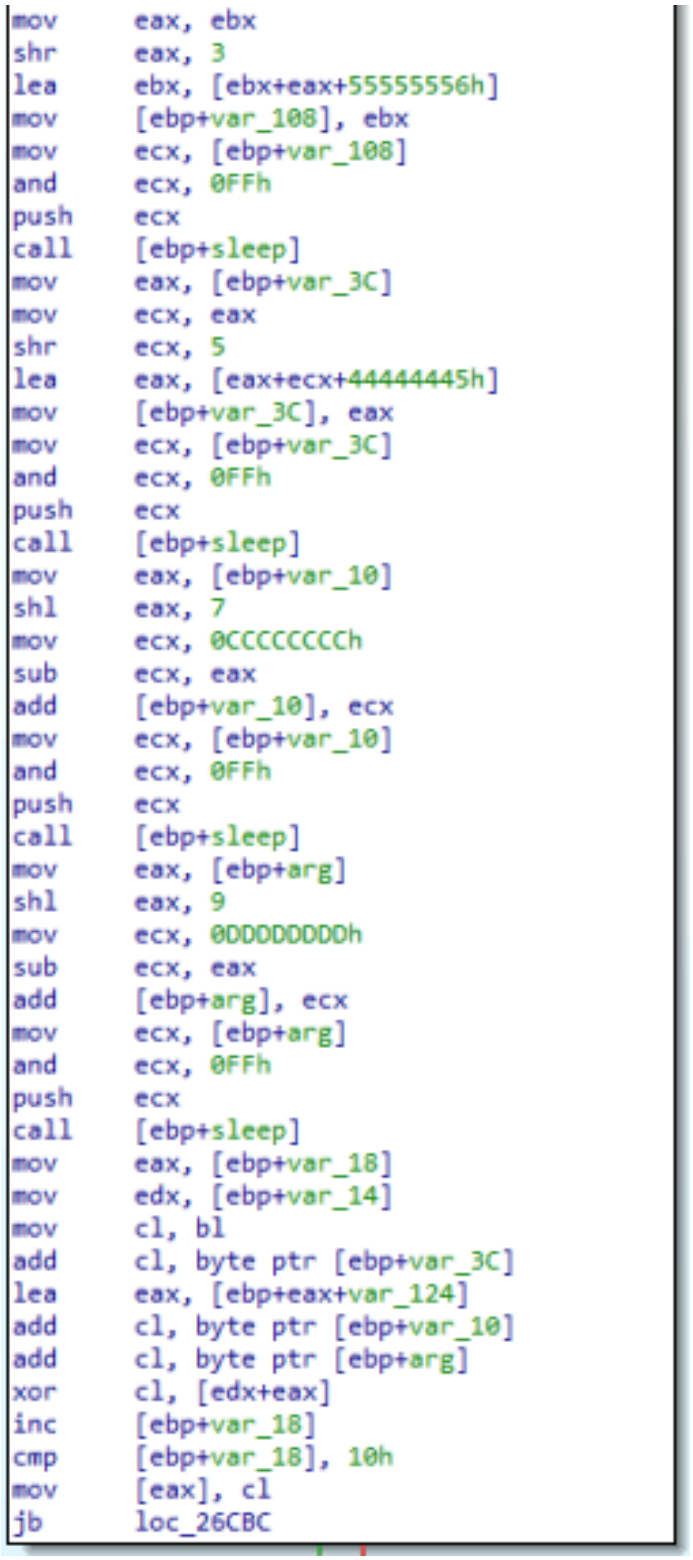
In addition to the filenames, we detected another type of execution chain consisting of a self-extracting SFX RAR file with the name “sys.exe” that drops the three related RasTls files to disk and executes them.

Stage 2 – The modified DLL

Filename	WGXMAN.DLL
SHA-1	ef3e558ecb313a74eeafca3f99b7d4e038e11516
SHA-256	1c0cf69bce6fb6ec59be3044d35d3a130acddbbf9288d7bc58b7bb87c0a4fb97
MD-5	b4f12a7be68d71f9645b789ccdc20561

The DLL is loaded by the benign executable, as it normally would. In this case, the DLL has only a single purpose: execute shellcode to decrypt a fake log file. The shellcode is called in the “DllMain” function the moment the DLL is attached to the executable. Once decrypted, a new PE file is uncovered, which is PlugX’ main component, to which the execution is then transferred.

The configuration decryption routine that is used within PlugX’ modified DLL to decrypt Talisman differs from both aforementioned samples. The decryption routine is a piece of shellcode which can be launched as-is, as it resolves all the required functions before using them. The decryption logic differs as the used decryption constants have been altered, which can be seen in the screenshot below.



```
mov     eax, ebx
shr     eax, 3
lea     ebx, [ebx+eax+55555556h]
mov     [ebp+var_108], ebx
mov     ecx, [ebp+var_108]
and     ecx, 0FFh
push    ecx
call    [ebp+sleep]
mov     eax, [ebp+var_3C]
mov     ecx, eax
shr     ecx, 5
lea     eax, [eax+ecx+44444445h]
mov     [ebp+var_3C], eax
mov     ecx, [ebp+var_3C]
and     ecx, 0FFh
push    ecx
call    [ebp+sleep]
mov     eax, [ebp+var_10]
shl     eax, 7
mov     ecx, 0CCCCCCCCh
sub     ecx, eax
add     [ebp+var_10], ecx
mov     ecx, [ebp+var_10]
and     ecx, 0FFh
push    ecx
call    [ebp+sleep]
mov     eax, [ebp+arg]
shl     eax, 9
mov     ecx, 0DDDDDDDDh
sub     ecx, eax
add     [ebp+arg], ecx
mov     ecx, [ebp+arg]
and     ecx, 0FFh
push    ecx
call    [ebp+sleep]
mov     eax, [ebp+var_18]
mov     edx, [ebp+var_14]
mov     cl, bl
add     cl, byte ptr [ebp+var_3C]
lea     eax, [ebp+eax+var_124]
add     cl, byte ptr [ebp+var_10]
add     cl, byte ptr [ebp+arg]
xor     cl, [edx+eax]
inc     [ebp+var_18]
cmp     [ebp+var_18], 10h
mov     [eax], cl
jb      loc_26CBC
```

Figure 3. The decryption routine to decrypt Talisman

Contrary to the other samples, the shellcode's decryption loop contains 5 sleep calls, which slows the decryption down, albeit barely noticeably. The decrypted buffer is then decompressed using "RtlDecompressBuffer", after which the PE module is accessible in-memory.

Filename	SNAC.LOG
SHA-1	2294ecbbb065c517bd0e01f3f01aabd0a0402f5a
SHA-256	6dc98a3c771f9f20d099e2d64995564dd083be9ac6ed9586a6e57c20ebd4176c
MD-5	60cb70545fbe3c96a0f82eeb54940553

Filename - (decrypted and decompressed SNAC.LOG)	
SHA-1	80e5fd86127de526be75ef42ebc390fb0d559791
SHA-256	344fc6c3211e169593ab1345a5cfa9bcb46a4604fe61ab212c9316c0d72b0865
MD-5	c6c6162cca729c4da879879b126d27c0

This section will deep dive into the used obfuscation techniques, briefly cover the execution flow, and provide insight into the used decryption routines, as well as the internally used structures within the malware.

Obfuscation techniques

The used obfuscation techniques attempt to make the analysis harder, but one can quite easily deobfuscate the sample. The used Windows API functions are loaded dynamically, based on the CRC32 hash of the function name. Note that the name's terminating null byte must be included when hashing the function name.

The image below shows a wrapper function for WSAIoctl, which is resolved based on the given hash, as can be seen in the third function argument. Additionally, the second argument is a handle to the module which contains the given function. It returns a pointer to the requested function, which can then be invoked as a function.

```
FunctionFromModuleHandleByCRC32Hash = TalismanWSAIOCTLGlobalVar;
if ( !TalismanWSAIOCTLGlobalVar )
{
  hModule = TalismanWS2_32BaseAddressModuleGlobalVar;
  if ( !TalismanWS2_32BaseAddressModuleGlobalVar )
  {
    hModule = TalismanGetModuleHandleByName("ws2_32");
    TalismanWS2_32BaseAddressModuleGlobalVar = hModule;
  }
  FunctionFromModuleHandleByCRC32Hash = TalismanGetFunctionFromModuleHandleByCRC32Hash(0, hModule, 0x49F9B50B);
  TalismanWSAIOCTLGlobalVar = FunctionFromModuleHandleByCRC32Hash;
}
return FunctionFromModuleHandleByCRC32Hash(a1, 0x98000004, a2, 12, 0, 0, a3, 0, 0);
```

Figure 4. Dynamic function resolving based on API hashing, as seen within Talisman

Secondly, some of the internally used strings are encrypted, but not all. The string encryption is based on simple XOR cipher, where the encrypted string is first created on the stack. The characters of the encrypted string are then decrypted and stored at the location of the encrypted character. Once the decryption loop has finished, the decrypted string will be stored on the stack. The image below shows the decryption of a string. Note that both ascii and wide strings are used within the program, which can cause some confusion when decrypting the strings with the wrong encoding.

```
s_Global_DelSelf_8_8X[0] = 0x55615596;
s_Global_DelSelf_8_8X[1] = 0x5573555E;
s_Global_DelSelf_8_8X[2] = 0x55615574;
s_Global_DelSelf_8_8X[3] = 0x55995571;
s_Global_DelSelf_8_8X[4] = 0x55615578;
s_Global_DelSelf_8_8X[5] = 0x55785582;
s_Global_DelSelf_8_8X[6] = 0x55775561;
s_Global_DelSelf_8_8X[7] = 0x5538553D;
s_Global_DelSelf_8_8X[8] = 0x551F552D;
s_Global_DelSelf_8_8X[9] = 0x558D552D;
qmemcpy(&s_Global_DelSelf_8_8X[10], "<UUUUU", 6);
for ( i = 0; i < 0x2E; ++i )
  *(s_Global_DelSelf_8_8X + i) = ((*(s_Global_DelSelf_8_8X + i) + 0x22) ^ 0x33) - 0x44;
```

Figure 5. Stack strings, as seen within Talisman

The qmemcpy function is an assumption that is made by IDA, since the assembly view contains mov instructions for the complete encrypted string. The loop that follows uses addition, bitwise exclusive or, and subtraction operators to decrypt the string, one byte at a time. The string decryption keys are the same for all encrypted strings within the binary. Note that not all strings within the sample are encrypted.

PlugX execution flow

Upon executing, the main function will resolve some functions, as these are used later. Additionally, the malware adjusts the SeDebugPrivilege and SeTcbPrivilege tokens of its own process. These two tokens ensure that the malware can, respectively, debug processes which aren't owned by the current process' owner, and the malware can create access tokens as if they are made by any user. Both instances allow the malware to act as the SYSTEM user, allowing free roam around the machine. The malware then creates its main thread, which is named "bootProc".

The screenshot below shows the pseudo code of the token adjustment, and the creation of the main thread. Note the plaintext "bootProc" string, as well as the token names, are prime examples to show that not all strings within the sample are encrypted.

```
strcpy(s_bootProc, "bootProc");
s_bootProc[9] = 0;
ShowWindow(0, 0);                                   // ShowWindow(null, SW_HIDE)
v1 = TalismanPointerToMemoryToKeepTheEventHandleGlobalVar;
if ( !TalismanPointerToMemoryToKeepTheEventHandleGlobalVar )
{
  if ( TalismanLocalAllocFunction(0x18u) )
    v1 = TalismanGetIfIsNeededCreateEventWAndCallItFunction();
  else
    v1 = 0;
  TalismanPointerToMemoryToKeepTheEventHandleGlobalVar = v1;
}
v1[1] = a1;
TalismanGetOwnProcessHandleAndTokenAndAdjustTokenPrivilegesFunction(L"SeDebugPrivilege");
TalismanGetOwnProcessHandleAndTokenAndAdjustTokenPrivilegesFunction(L"SeTcbPrivilege");
if ( !TalismanPointerToVarToKeepInitializeCriticalSectionHandleGlobalVar )
{
  if ( TalismanLocalAllocFunction(0x24u) )
    v2 = TalismanGetIfIsNeededInitializeCriticalSectionAndCallItFunction();
  else
    v2 = 0;
  TalismanPointerToVarToKeepInitializeCriticalSectionHandleGlobalVar = v2;
}
return TalismanPrepareToCreateThreadAndResumeItFunction(
         s_bootProc,
         TalismanManageCriticalFunctionForMalwareFunction,
         0);
```

Figure 6. Set-up prior to the main thread creation within Talisman

The first noteworthy action the main thread performs, is the unloading of the modified DLL from memory, minimising the amount of malicious artefacts in-memory. The rest of the malware's execution is in-line with other PlugX variants' behaviour, one example being the number of command-line arguments which decide the persistence method, as well as the persistence methods themselves: varying from the creation of a key in the register's start-up folder, the creation of a scheduled task, or the creation of a service.

Additionally, the malware connects with the command-and-control server, as is stored in the internal configuration structure. The usage of plug-ins is possible, much like other PlugX variants, if the correct signature is present in the plug-in. The following sections will highlight interesting segments of the sample.

Internal structures

A piece of shellcode is used to decrypt Talisman, a pointer to which is stored within a custom structure within the binary. The shellcode resolves all functions it requires and handles the decryption and the decompression of the given data blob. Additionally, the MZ-header and PE-header are verified using the offsets of 0 and 0x4550 respectively, which correspond to fields named e_magic and e_lfanew. The entry point of the newly decrypted PE file is stored in the custom structure as well. The image below shows the checks and structure field assignments in the pseudocode overview.

```
255  if ( pDecryptedPE->e_magic != 0x5A4D )
256    return 13;
257  pPE = (pDecryptedPE + pDecryptedPE->e_lfanew);
258  v19 = pPE->Signature == 0x4550;
259  v90 = pPE;
260  if ( !v19 )
261    return 15;
262  if ( (pPE->FileHeader.Characteristics & 0x2000) == 0 )
263    return 16;
264  configStruct = (v93)(0, pPE->OptionalHeader.SizeOfImage, 4096, 64);
265  if ( !configStruct )
266    return 17;
267  shellcodeSize_ = shellcodeSize_fromArg;
268  pEncryptedPlugX_ = pEncryptedPlugX_fromArg;
269  configStruct->pShellcodeBase = pShellcodeBase_fromArg;
270  encryptedPlugXSize_ = encryptedPlugXSize_fromArg;
271  configStruct->shellcodeSize = shellcodeSize_;
272  config_ = config_fromArg;
273  configStruct->pEncryptedPlugX = pEncryptedPlugX_;
274  configSize_ = configSize_fromArg;
275  configStruct->encryptedPlugXSize = encryptedPlugXSize_;
276  configStruct->config = config_;
277  configStruct->configSize = configSize_;
278  configStruct->signature = 0xCF455089;
279  configStruct->pPlugXEntryPoint = configStruct + pPE->OptionalHeader.AddressOfEntryPoint;
```

Figure 7. Internal configuration allocation

Our findings regarding the custom structure differ when comparing them to the analysis of DrWeb. Since the sample where DrWeb refers to is unavailable, we could not compare the two samples. In 2014, Takahiro Haruyama and Hiroshi Suzuki presented “I Know You Want Me - Unplugging PlugX” at Black Hat Asia. On slide 10, one can see the shellarg structure matches our findings. The image below shows DrWeb’s version of the structure, as well as the version that is found within Talisman.

```
struct shellarg
{
  DWORD signature;
  DWORD dword_0;
  DWORD dword_1;
  DWORD p_shellcode;
  DWORD shellcode_size;
  DWORD config;
  DWORD config_size;
};

struct shellarg_talisman
{
  DWORD signature;
  DWORD pShellcodeBase;
  DWORD shellcodeSize;
  DWORD pEncryptedPlugX;
  DWORD encryptedPlugXSize;
  DWORD config;
  DWORD configSize;
  DWORD pPlugXEntryPoint
};
```

Figure 8. Two versions of the internal structure which is used when loading Talisman

Within the custom structure, the signature field often corresponds to PLUG. In other variants, a different value is used, such as the THOR version that Unit42 wrote about. The Talisman variant uses the constant value 0xCF455089 as a signature, which decodes to the Chinese characters “蛟禒” using the UTF16-LE encoding and translates to “crotch” in English. Other versions are named based on their signature but given Talisman’s rude signature word we opted for a different name.

Config decryption

Before decrypting the configuration, the malware will check for the presence of the Talisman signature and ensure that the configuration length matches. In the observed Talisman variants, the configuration size equals 0x1924 bytes, as passed by the “shellarg_talisman” structure, which is located at the start of the binary.

```
if ( shellarg_talisman->signature == 0xCF455089 )
{
  config = shellarg_talisman->config;
  if ( *config == config[1] || shellarg_talisman->configSize != 0x1924 )
    return TalismanGetMemsetAndCallItFunction();
```

Figure 9. The length and signature check of the config

If those checks are met, Talisman will call the decryption routine to decrypt the configuration with the well-known PlugX algorithm, as is shown in the image below.

```
mov     edi, eax
shr     edi, 3
lea     eax, [eax+edi-11111111h]
mov     edi, ecx
shr     edi, 5
lea     ecx, [ecx+edi-22222222h]
mov     edi, edx
shl     edi, 7
mov     ebx, 33333333h
sub     ebx, edi
mov     edi, [ebp+var_4]
add     edx, ebx
shl     edi, 9
mov     ebx, 44444444h
sub     ebx, edi
add     [ebp+var_4], ebx
lea     ebx, [ecx+eax]
add     bl, dl
add     bl, byte ptr [ebp+var_4]
mov     [ebp+arg_0], edx
mov     edx, [ebp+var_8]
xor     bl, [edx+esi]
inc     esi
dec     [ebp+configSize]
mov     [esi-1], bl
jnz     short continueLoop
pop     ebx
```

Figure 10. The decryption routine

The encrypted configuration is fetched from “SNAC.LOG”, which is decrypted by the second stage DLL, rather than the Talisman PlugX binary. This is to ensure that the full execution chain (the benign executable, the modified DLL, and the encrypted SNAC.LOG file) are executed. If this is not the case, the malware will crash.

The Talisman PlugX configuration has a size of 0x1924 bytes and contains all the necessary values and information to properly run the executable. The following fields are contained in the Talisman configuration: a list of control panels (i.e., freewula.strangled.net), the target process that will inject (i.e., %SystemRoot%\system32\nslookup.exe), the malware home directory (i.e., %ALLUSERSPROFILE%\SymantecSNAC), the persist name (i.e., SymantecSNAC), the service display name (i.e., SymantecSNAC), the campaign id (i.e., TEST) and the mutex name (i.e., Global\Restart0).

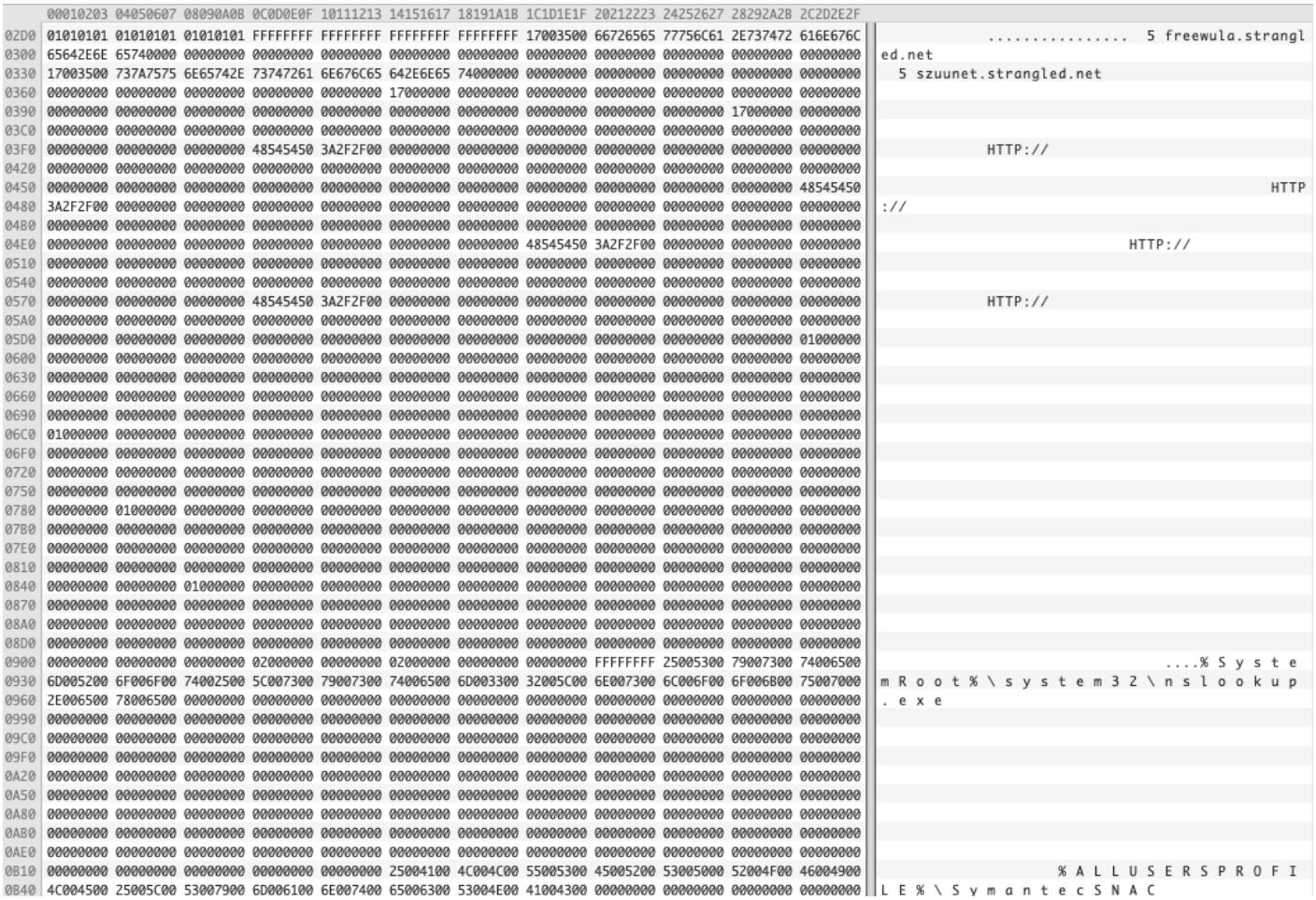


Figure 11. Talisman configuration

The aforementioned config fields are very similar as the ones reported in the analysis performed by DrWeb. Analyzing some of the different Talisman recovered configurations, we observed the following embedded campaign ids:

- TEST
- RT
- aop-1

We don't exactly know how the "campaign_id" field is used by the actors, but we presume it may be used to identify the victims, or the malware version in their operations.

Persistence

Talisman has, much like the original PlugX version, several ways of persisting itself. Depending on the number of command-line arguments, and the internal configuration, the sample may opt to not persist using one of the following methods:

- No persistence
- A scheduled task
- An auto-starting service

Note that the service is only created if administrative permissions are available to the sample. If this is not the case, the auto-run registry key is created instead.

Plug-ins

Talisman has, much like other PlugX variants, the ability to use plug-ins. Some of these are embedded by default, as also described by the United States' CISA: Disk, Nethood, Netstat, Option, PortMap, RegEdit, Service, Shell, SQL, and Telnet. The plug-in names are relatively self-explanatory, as their name indicates the type of activities that can be performed with them. The screenshot below shows the first seven of the embedded plug-ins, as listed above.

```
strcpy(plugin_name_holder, "Disk");
plugin_name_holder[5] = 0;
Function11 = TalismanOpenObjectNamedPIProcessPIDToReadFunction11(v13);
if ( Function11 )
  (*Function11)(-1, 0, 0x81020325, TalismanPrepareToManageDiskCommandFunction, plugin_name_holder);
TalismanPrepareToControlKeyboardAndMoreFunction();
strcpy(plugin_name_holder, "Nethood");
v37 = 0;
Function10 = TalismanOpenObjectNamedPIProcessPIDToReadFunction10(v13);
if ( Function10 )
  (*Function10)(-1, 5, 0x81020213, TalismanPrepareToManageNethoodCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "Netstat");
v37 = 0;
Function9 = TalismanOpenObjectNamedPIProcessPIDToReadFunction9(v13);
if ( Function9 )
  (*Function9)(-1, 4, 0x81020215, TalismanPrepareToManageNetstatCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "Option");
plugin_name_holder[7] = 0;
Function8 = TalismanOpenObjectNamedPIProcessPIDToReadFunction8(v13);
if ( Function8 )
  (*Function8)(-1, 6, 0x81020128, TalismanPrepareToManageOptionCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "PortMap");
v37 = 0;
Function7 = TalismanOpenObjectNamedPIProcessPIDToReadFunction7(v13);
if ( Function7 )
  (*Function7)(-1, 7, 0x81020325, TalismanPrepareToManagePortmapCommandFunction, plugin_name_holder);
TalismanCheckIfIsProcessCommandFunction();
strcpy(plugin_name_holder, "RegEdit");
v37 = 0;
Function6 = TalismanOpenObjectNamedPIProcessPIDToReadFunction6(v13);
if ( Function6 )
  (*Function6)(-1, 3, 0x81020315, TalismanPrepareToManageRegeditCommandFunction, plugin_name_holder);
TalismanManageCursorsAndDisplayAndMoreFunction();
strcpy(plugin_name_holder, "Service");
v37 = 0;
```

Figure 12. A list of embedded plug-ins

Often, the third argument of the plug-in's function (named "FunctionN" in the screenshot above, where "N" is a number) indicates the date of the plugin's creation when viewed as a hexadecimal value. This holds true in Talisman's case as well, albeit with a minor change: the digits of the year are inverted per byte. As an example, the "Service" plug-in's date equals "0x81020315". The first two bytes are "81 02", which equal "20 18" when inverted per byte, marking 2018 as a year. The specific date for this plugin would be 3 March 2018. The table below provides the dates for all embedded plug-ins.

Plug-in	Date (dd-mm-yyyy)
Disk	25-03-2018
Nethood	13-02-2018
Netstat	15-02-2018
Option	28-01-2018
PortMap	25-03-2018
RegEdit	15-03-2018

Service	17-01-2018
Shell	05-03-2018
SQL	23-03-2018
Telnet	25-02-2018

A small change like this can indicate that the actor has access to the source code of the malware, rather than a builder with an executable stub, as these generally do not provide the option for such granular changes.

Infrastructure

During our research we analysed three different Talisman PlugX samples, from which we extracted the command-and-control servers. This allowed us to cluster the actors' infrastructure. We found that one of the samples (SHA-256: 6dc98a3c771f9f20d099e2d64995564dd083be9ac6ed9586a6e57c20ebd4176c) connects to "dhsg123[.]jkub[.]com" as the command-and-control domain. This overlaps with the RedFoxtrot group, on which Recorded Future already reported.

Moreover, in the sample identified by the SHA256 hash fdada5ba799bd9f5270b218cfad543d99fde3eb7898fd9e3ee79603b643b3c48, the command-and-control domain is "final[.]staticd[.]dynamic-dns[.]net", which resolved to 158[.]247[.]204[.]191. Pivoting on that IP, we identified two PCShare samples communicating to that IP, both of which use the same injection method into the Remote Desktop shared clipboard (RDPclip.exe). PCShare is an open-source backdoor, which has been leveraged by Chinese actors, as documented by BlackBerry. Additionally, the same mutex (being "78de65b0701f3c9238a37") is used as the ones reported by Recorded Future.

```
strcpy(plugin_name_holder, "Disk");
plugin_name_holder[5] = 0;
Function11 = TalismanOpenObjectNamedPIProcessPIDToReadFunction11(v13);
if ( Function11 )
  (*Function11)(-1, 0, 0x81020325, TalismanPrepareToManageDiskCommandFunction, plugin_name_holder);
TalismanPrepareToControlKeyboardAndMoreFunction();
strcpy(plugin_name_holder, "Nethood");
v37 = 0;
Function10 = TalismanOpenObjectNamedPIProcessPIDToReadFunction10(v13);
if ( Function10 )
  (*Function10)(-1, 5, 0x81020213, TalismanPrepareToManageNethoodCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "Netstat");
v37 = 0;
Function9 = TalismanOpenObjectNamedPIProcessPIDToReadFunction9(v13);
if ( Function9 )
  (*Function9)(-1, 4, 0x81020215, TalismanPrepareToManageNetstatCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "Option");
plugin_name_holder[7] = 0;
Function8 = TalismanOpenObjectNamedPIProcessPIDToReadFunction8(v13);
if ( Function8 )
  (*Function8)(-1, 6, 0x81020128, TalismanPrepareToManageOptionCommandFunction, plugin_name_holder);
strcpy(plugin_name_holder, "PortMap");
v37 = 0;
Function7 = TalismanOpenObjectNamedPIProcessPIDToReadFunction7(v13);
if ( Function7 )
  (*Function7)(-1, 7, 0x81020325, TalismanPrepareToManagePortmapCommandFunction, plugin_name_holder);
TalismanCheckIfIsProcessCommandFunction();
strcpy(plugin_name_holder, "RegEdit");
v37 = 0;
Function6 = TalismanOpenObjectNamedPIProcessPIDToReadFunction6(v13);
if ( Function6 )
  (*Function6)(-1, 3, 0x81020315, TalismanPrepareToManageRegeditCommandFunction, plugin_name_holder);
TalismanManageCursorsAndDisplayAndMoreFunction();
strcpy(plugin_name_holder, "Service");
v37 = 0;
```

Figure 13. Talisman PlugX and PCShare connection to RedFoxtrot infrastructure

One interesting note on the TTPs employed by the actors is that unused, parked, or decommissioned domains are set to resolve to localhost (127.0.0.1), or public services such as Google (8.8.8.8) or Cloudflare (1.1.1.11, note that the “11” at the end is not a typo).

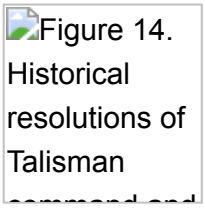


Figure 14. Historical resolutions of Talisman command and control domain

All the domains we’ve detected are using Dynamic DNS providers. The main reason to use a dynamic DNS provider is to quickly and easily change the IP address where the domain resolves. An additional advantage for the actor is the lack of WHOIS contact data regarding dynamic DNS subdomains, thus providing operational security. The main providers that the actors used to register the domains are Afraid FreeDNS, ChangeIP, and Dynu.

Interestingly, the vast majority of the threat actors’ infrastructure was hosted on virtual private servers of Choopa/Vultr company (AS 20473 - AS-CHOOPA) although we’ve seen some command-and-control servers hosted on Digital Ocean’s infrastructure (AS 14061 - DIGITALOCEAN-ASN).

We are sharing the full list of indicators of compromise regarding the Talisman PlugX command and control servers and the infrastructure of the actor in Appendix A of this blog.

Attribution & Victimology

We have observed the Talisman malware in a campaign which targets the telecommunication and defense sectors in South Asia. Moreover, the targets line up with the Chinese efforts to protect the Belt and Road Initiative, a program that aims to establish strong socioeconomically relationships across Europe, Asia, and Africa.

The South Asian region is of strategic interest for China for various reasons. Firstly, the stability in South Asia is a global concern, making it a logical focus area for all superpowers, including China. Secondly, companies in the telecommunication and defense sectors provide unique insights for attackers and are often targeted. Thirdly, given the military presence of multiple superpowers in the region, and the geographic location in relation to China, this area can be considered a priority for Chinese sponsored groups.

PlugX has been associated with various Chinese actors in recent years. This fact raises the question if the malware's code base is shared among different Chinese state-backed groups. On the other hand, the alleged leak of the PlugX v1 builder, as reported by Airbus in 2015, indicates that not all occurrences of PlugX are necessarily tied to Chinese actors.

However, based on our analysis of the infrastructure overlaps with Recorded Future's related research, we assess with medium confidence that the Talisman PlugX variant discussed in this blog is used by RedFoxtrot/Nomad Panda, a Chinese state sponsored actor. The victimology is in-line with the targets of this group as well. We have no evidence that this variant is exclusively used by this threat actor, instead we believe that Talisman could be shared among different Chinese groups to carry out their operations.

Conclusion

In this blog we have analysed the different steps the infamous PlugX RAT follows to start execution and avoid detection. Moreover, we highlighted several interesting characteristics of a new variant that we dubbed Talisman. During the technical analysis, we extracted various network artifacts that allowed us to not only find new samples, but also analyse the infrastructure overlap with a known Chinese state-backed group. This information, together with the analysis of the victims where we observed this PlugX variant, allowed us to attribute this campaign with medium confidence to the RedFoxtrot APT group, which is otherwise known as Nomad Panda.

Appendix A – Indicators of compromise

Talisman PlugX All execution files

e71d355dec64cbf8f02a754bf0585437ce48f7b68108cb642fb202393cd1ef90

0a00204517283c9a8d1e2d1a8743249c14de0edcec4a8292500083437735663c

45c944889a482ae2e0e0a8e260c3be737cb612c8804164badef61e8a8713b92f

f6b939dcc97c1c43f1c616174f936b6ef19c5ccc872a1a0ef14f2989cf11b02b

Talisman PlugX Loader

1c0cf69bce6fb6ec59be3044d35d3a130acddbbf9288d7bc58b7bb87c0a4fb97

ad48650c6ab73e2f94b706e28a1b17b2ff1af1864380edc79642df3a47e579bb

46cd5079a69d9a68029e37f2680f44b7ba71c2b1eecf4894c2a8b293d5f768f10

0468005682c814e7a5f07f3554e9fadbb2d2ba7527dcaee9a1a456f244c49ddb

Talisman PlugX Encrypted Payloads

a072133a68891a37076cd1eaf1abb1b0bf9443488d4c6b9530e490f246008dba

6dc98a3c771f9f20d099e2d64995564dd083be9ac6ed9586a6e57c20ebd4176c

fdada5ba799bd9f5270b218cfad543d99fde3eb7898fd9e3ee79603b643b3c48

37b3fb9aa12277f355bbb334c82b41e4155836cf3a1b83e543ce53da9d429e2f

fe18adaec076ffce63da6a2a024ce99b8a55bc40a1f06ed556e0997ba6b6d716

3c5d08f20a7bd04b1e6866344af59bec2152ec3542f2eae0c7925555e670676e

Talisman PlugX Encrypted Configuration

f44ede464f752ea3aa3595f8137945a4dee7298c8155c39f366aad05b125ac8b

PCShare

3f6102bd9add588b4df9b1523e40bb124af36a729037b8c3f2261563e4fa4be9

785ac72b10fd9cf98b5e2a40dc607e1ff735fcd8192bf71747755c963c764e2d

Mutex

Global\ReStart0

Global\DelSelf(00000000) (where the zeros are the process ID in hexadecimal format, prepended with zeros to ensure 8 digits are used)

PDB

c:\bld_area\SESAgent70\snac_build\bin.ira\WGXMAN.pdb

Domains

freewula.strangled[.]net

szuunet.strangled[.]net

dhsg123.jkub[.]com

final.staticd.dynamic-dns[.]net

oprblemoyo.kozow[.]com

asd.powergame.0077.x24hr[.]com

w.asd3.as.amazon-corp.wikaba[.]com

randomanalyze.freetcp[.]com

darkpapa.chickenkiller[.]com

miche.justdied[.]com

Domains

209[.]97[.]166[.]143

149[.]28[.]139[.]86

159[.]65[.]152[.]7

143[.]110[.]242[.]139

158[.]247[.]204[.]191

143[.]110[.]250[.]149

202[.]182[.]111[.]249

207[.]148[.]119[.]147

149[.]28[.]128[.]117

159[.]65[.]147[.]83

143[.]110[.]241[.]54

157[.]245[.]111[.]30

207[.]148[.]64[.]239

45[.]76[.]188[.]118

45[.]77[.]16[.]91

Appendix B – MITRE ATT&CK Techniques

Within this campaign, we have observed the following MITRE ATT&CK techniques.

T1071	Application Layer Protocol	HTTP/DNS requests are used in the C&C traffic
T1059	Command and Scripting Interpreter	A reverse shell can be made by PlugX
T1543	Create or Modify	One persistence option is a system service

	System Process	
T1140	Deobfuscate/Decode Files or Information	The API hashing and encrypted stack strings are obfuscation types. The decrypted Talisman payload is decompressed before it is used
T1574	Hijack Execution Flow	The sideloading of the described DLL
T1056	Input Capture	Keylogging capabilities
T1036	Masquerading	The registered task/service pretends to be benign by name
T1106	Modify Registry	The runkey which is made when persisting via the registry
T1106	Native API	Windows API functions are called directly
T1095	Non-Application Layer Protocol	PlugX can work directly with TCP/UDP packets
T1057	Process Discovery	Iterates over all processes
T1012	Query Registry	Queries the registry to check for values
T1113	Screen Capture	Can capture the screen of the victim
T1049	System Network Connections Discovery	Possible via the embedded “netstat” module