

Malware Tales: Gootkit

Archived: 2026-04-05 17:55:35 UTC

Summary

1. *The Threat*
2. *Payload Delivery*
3. *Gootkit executable*
4. *Stage 1: Packed Gootkit*
5. *Stage 2: Gaining a foothold*
6. *Stage 3: Check-in phase*
7. *Last stage*
8. *Additional findings*
9. *Conclusions*

1.The Threat

Gootkit belongs to the category of Infostealers and Bankers therefore it aims to steal information available on infected machines and to hijack bank accounts to perform unwanted transactions.

It has been around at least since 2014 and it seems being actively distributed in many countries, including Italy.

Previous reports about this threat can be found following this link: [Malpedia](#)

Today we are going to dive into the analysis of a particular variant that came up the last week.

2.Payload Delivery

The infection vector is an email written in Italian. In this case adversaries used one of the most common social engineering techniques to trigger the user to open the attachment



Caro Pubblico

Il vostro ordine n 65423 è stato un successo ricevuta magazzino centro di esecuzione e pronto a consegna. Ma l'indirizzo nel tuo ordine è elencato con sbaglio e non possiamo consegnare l'ordine a te.

Numero d'ordine: # 65423

Data di consegna: 05.02.2019

Stato: consegnato al magazzino

Controlla il tuo ordine e contattaci se possibile.

CONTROLLA ORDINE - Scarica il file allegato alla lettera

Copyright © 2019 GLS. Tutti i diritti riservati.


[Unsubscribe](#)

The downloaded file is a heavily obfuscated Javascript file called "GLS_Notifica.js". If the user opens it, the native Javascript interpreter wscript.exe would be executed by default and it would perform the following HTTP request:

```
hxxp://redok.com.mx/tmp/337.php
```

The result is the download of a cabinet file that is an archive file which can be extracted natively by Windows. Inside there is a Portable Executable file that is saved into the **%TEMP%** folder ("C:\Users<username>\AppData\Local\Temp") and launched.

Javascripts downloaders are a common payload delivery because a little obfuscation can be enough to make them very difficult to be detected by antivirus engines.



5 engines detected this file

SHA-256 597f734c780f283d6cd7c7443700a65249b8350ec1ad58b8aa790a32f57eec4d
File name GLS_Notifica.js
File size 33.25 KB
Last analysis 2019-02-07 13:18:17 UTC

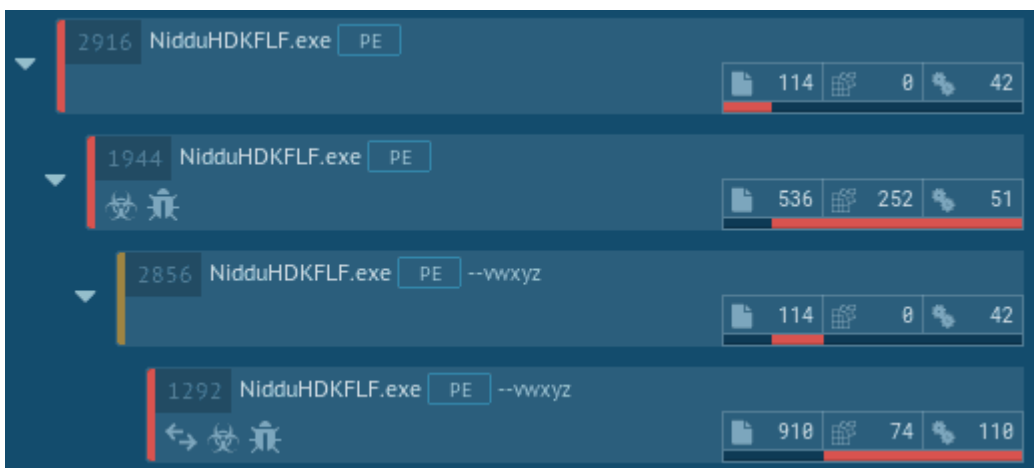
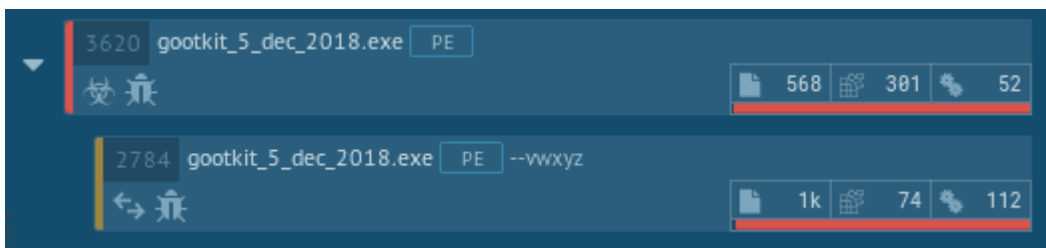
5 / 57

Detection Details Community

Ikarus	⚠ Trojan-Downloader.JS.Agent	Kaspersky	⚠ HEUR:Trojan-Downloader.Script.Generic
Rising	⚠ Trojan.ObfusJS/Heur!1.A4CA (CLASSIC)	Symantec	⚠ ISB.Downloaderigen60
ZoneAlarm	⚠ HEUR:Trojan-Downloader.Script.Generic	Ad-Aware	✔ Clean

3.Gootkit executable

First run of the sample in an automated environment revealed that something new was added in this version. As we can see in the following images, malware authors added a new layer of protection to the malicious agent. The comparison has been made with a variant spread during December of 2018 in Italy. images are from [AnyRun](#)



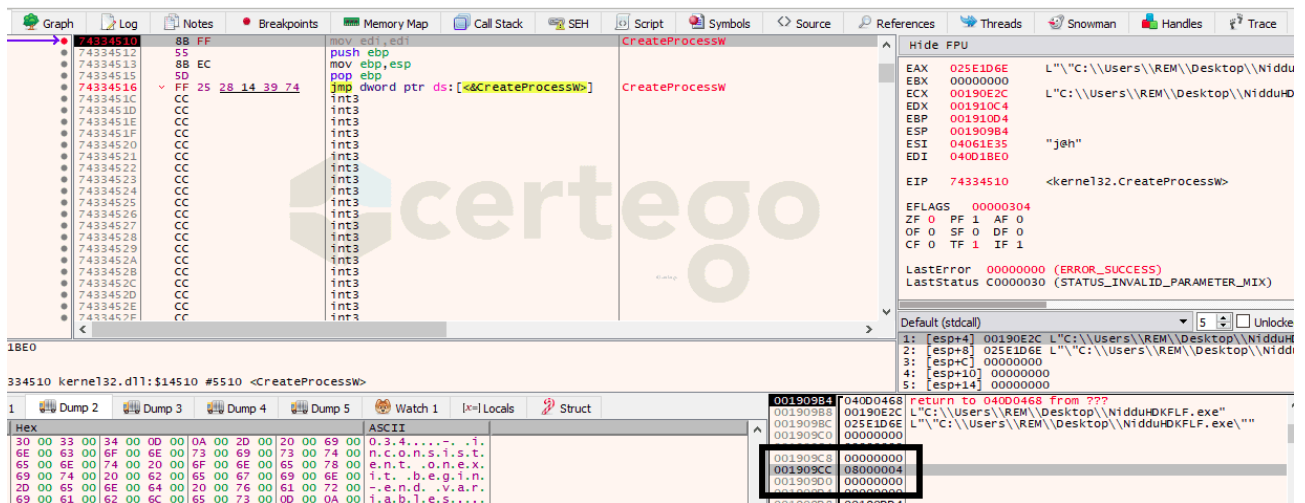
This means that the original program was “packed” with the aim to slow down reverse engineers and to make ineffective static analysis tools like Yara rules.

4.Stage 1: Packed Gootkit

In such cases, a malware analyst knows that he has to extract the original payload as fast as possible without losing time to try to understand the inner workings of this stage.

A great open-source tool exists which can resolve the problem in a matter of seconds. It's called **PE-Sieve** [GitHub](#). Even though it does not always work, in this case it can dump the unmapped version of the original executable because the malicious software uses a technique called **Process Hollowing** a.k.a. **RunPE**. This method consists in starting a new process in a suspended state, "hollowing" out the content of the process, replacing it with malicious code and, finally, resuming the thread.

In the image we can see that the 6th parameter of "CreateProcessW" was set to "4", indicating that the process will start in a suspended state.



We are talking about a well known technique that is easily detectable with the monitoring of the Windows API calls that are needed to perform the injection. But here comes the trick.

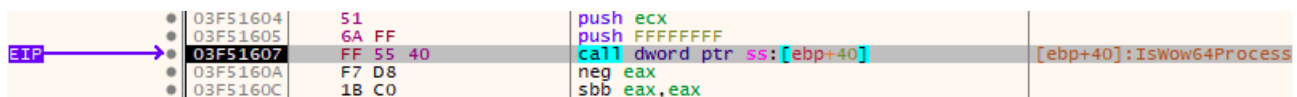
Following the flow of execution we couldn't find all the needed API calls: we got NtCreateProcess, NtGetContextThread, NtReadVirtualMemory and NtSetContextThread.

The most important ones that are used by monitoring applications to detect the technique were missing:

- NtUnmapViewOfSection to "hollow" the target process
- NtWriteVirtualMemory to write into the target process
- NtResumeThread to resume the suspended thread

Let's find out what's happening!

After some shellcode injections inside its memory space, the process executes a call to IsWow64Process API that is used by the application to understand if the process is running under the WOW64 environment ([Wiki](#)): this is a subsystem of the Windows OS that is able to run 32-bit applications, like this one, on 64-bit operating systems.



The result of this check is used to run two different paths of code but with the same scope: run one of the aforementioned missing API calls in the Kernel mode. This means that, in this way, classic user-level monitoring tools would not catch these calls and the RunPE technique would remain unnoticed.

Specifically, in case the process is running in a 32-bit environment, it would use the SYSENTER command to switch into the Kernel mode, while, on the contrary, it would use the SYSCALL command to perform the same operation.

03F513B0	8B D4	mov edx,esp
03F513B2	0F 34	sysenter
03F513B4	C3	ret

03FA1BA5	8B 45 08	mov eax,dword ptr ss:[ebp+8]
03FA1BA8	0F 05	syscall
03FA1BAA	89 45 D4	mov dword ptr ss:[ebp-2C],eax

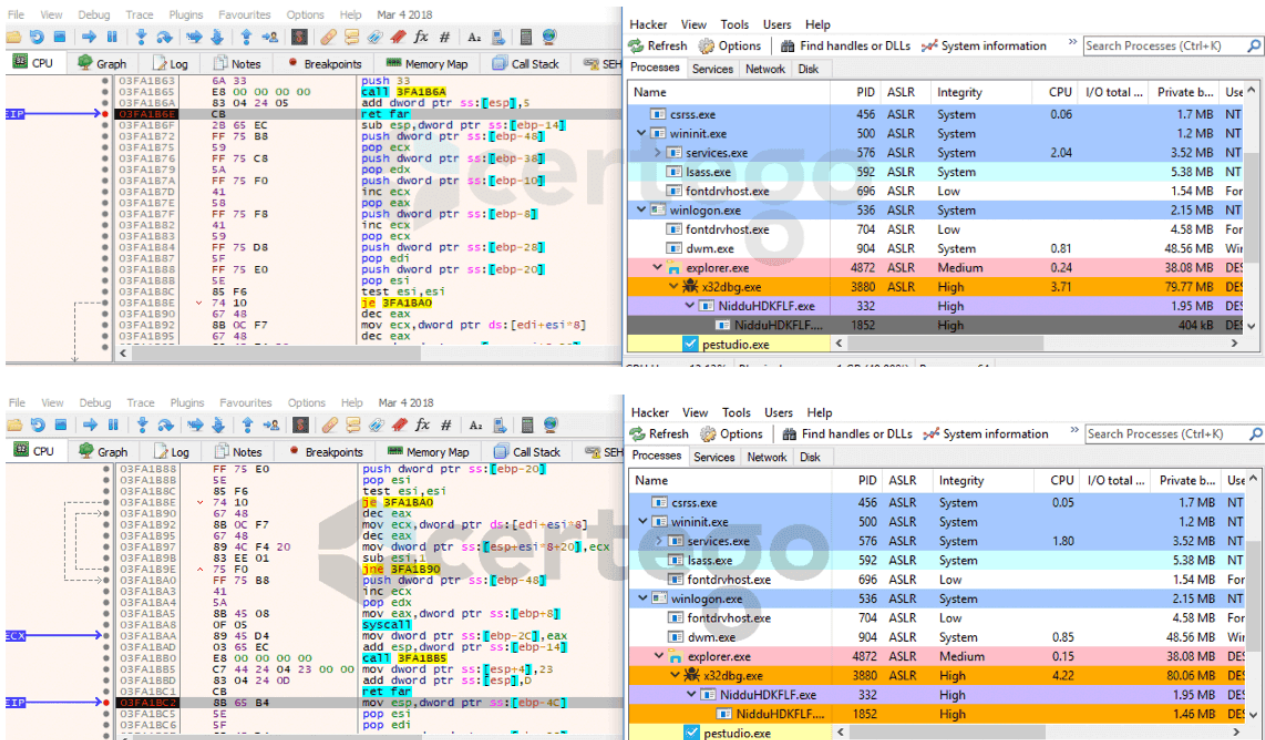
To complicate even further, the SYSCALL command can't be called in the context of a 32-bit application. This means that the executable needs to perform a "trick-into-the-trick" to execute this operation. We are talking about a technique known as *The Heaven's Gate*.

Practically, thanks to the RETF instruction, it's possible to change the code segment (CS) from 0x23 to 0x33, de facto enabling 64-bit mode on the running process.

In the following image we highlight the entrance and the exit of the "Gate" which contains the 64-bit code that performs the SYSCALL operation.

1b63:	6a 33	push 0x33
1b65:	e8 00 00 00 00	call 0x1b6a
1b6a:	83 04 24 05	add DWORD PTR [rsp],0x5
1b6e:	cb	retf
1b6f:	2b 65 ec	sub esp,DWORD PTR [rbp-0x14]
1b72:	ff 75 b8	push QWORD PTR [rbp-0x48]
1b75:	59	pop rcx
1b76:	ff 75 c8	push QWORD PTR [rbp-0x38]
1b79:	5a	pop rdx
1b7a:	ff 75 f0	push QWORD PTR [rbp-0x10]
1b7d:	41 58	pop r8
1b7f:	ff 75 f8	push QWORD PTR [rbp-0x8]
1b82:	41 59	pop r9
1b84:	ff 75 d8	push QWORD PTR [rbp-0x28]
1b87:	5f	pop rdi
1b88:	ff 75 e0	push QWORD PTR [rbp-0x20]
1b8b:	5e	pop rsi
1b8c:	85 f6	test esi,esi
1b8e:	74 10	je 0x1ba0
1b90:	67 48 8b 0c f7	mov rcx,QWORD PTR [edi+esi*8]
1b95:	67 48 89 4c f4 20	mov QWORD PTR [esp+esi*8+0x20],rcx
1b9b:	83 ee 01	sub esi,0x1
1b9e:	75 f0	jne 0x1b90
1ba0:	ff 75 b8	push QWORD PTR [rbp-0x48]
1ba3:	41 5a	pop r10
1ba5:	8b 45 08	mov eax,DWORD PTR [rbp+0x8]
1ba8:	0f 05	syscall
1baa:	89 45 d4	mov DWORD PTR [rbp-0x2c],eax
1bad:	03 65 ec	add esp,DWORD PTR [rbp-0x14]
1bb0:	e8 00 00 00 00	call 0x1bb5
1bb5:	c7 44 24 04 23 00 00	mov DWORD PTR [rsp+0x4],0x23
1bbc:	00	
1bbd:	83 04 24 0d	add DWORD PTR [rsp],0xd
1bc1:	cb	retf
1bc2:	8b 65 b4	mov esp,DWORD PTR [rbp-0x4c]
1bc5:	5e	pop rsi
1bc6:	5f	pop rdi

Instead, in this other image, we can see the process status before opening the gate (grey=suspended process) and after having closed it (orange=running process).



Also, Gootkit takes advantage of The Heaven’s Gate as an anti-debugging technique because the majority of commonly used debuggers can’t properly handle this situation, not allowing the analyst to follow the code of the Gate step-by-step.

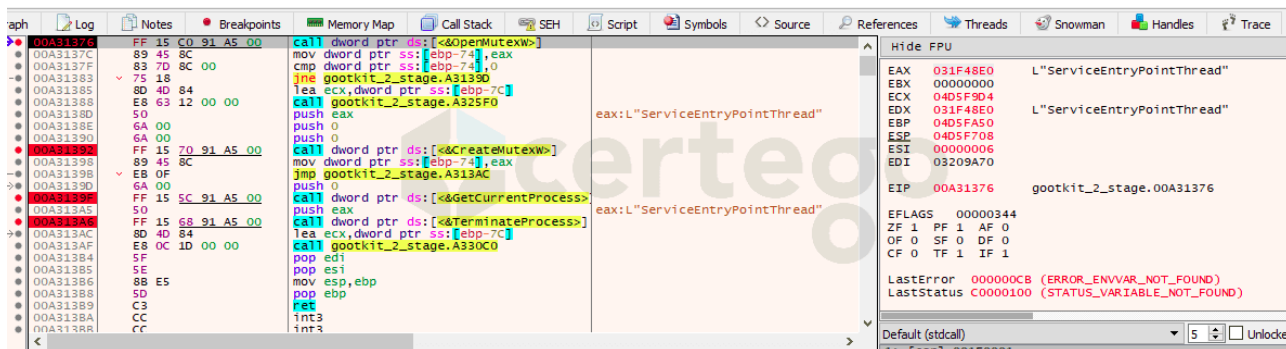
For further details, this method was deeply explained in this blog [MalwareBytes](#)

Going back to the point, the first stage resulted more complicated than expected because it pushed over the limits of obfuscation and stealthiness with the combination of various techniques.

5.Stage 2: Gaining a foothold

At this point we can proceed with the analysis of the unpacked Gootkit.

The very first considerable finding was the check for the existence of a **mutex** object named “ServiceEntryPointThread”. If it exists, the process would terminate itself.



But how mutexes works? Mutexes are used as a locking mechanism to serialize access to a resource on the system. Malware sometimes uses it as an “infection marker” to avoid to infect the same machine twice. The fascinating thing about mutexes is that they are a double-edged weapon: security analysts could install the mutex in advance to vaccinate endpoint. ([ref: Zeltser blog](#)).

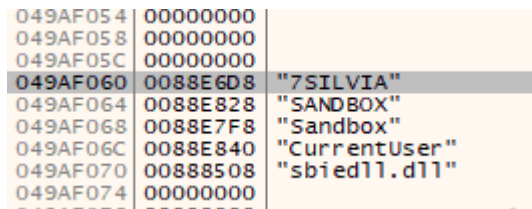
This means that this is a great indicator of compromise that we can use not only to detect the infection but also to prevent it.

Moving on, we found that malware authors implemented a lot of checks to understand if the malware is running inside a virtual environment. Some of them are:

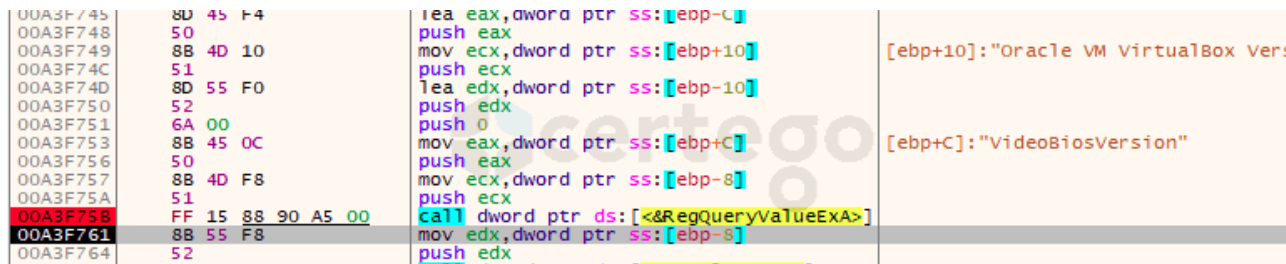
- It checks if the registry key “HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString” contains the word “Xeon”



- it checks if the computer name is “7SILVIA” or “SANDBOX”, if the username is “CurrentUser” or “Sandbox” or if “sbiedll.dll” has been loaded.



- it checks if “HKLM\HARDWARE\Description\System\VideoBiosVersion” contains the word “VirtualBox”



- it checks “HKLM\Software\Microsoft\Windows\CurrentVersion\SystemBiosVersion” for the string “VBOX”

In the case one of this check fails, the program would execute a Sleep operation in a infinite cycle in the attempt to thwart automated sandbox execution.

```

00A3F5F7 8D 4D F8      lea ecx,dword ptr ss:[ebp-8]
00A3F5FA E8 71 32 FF FF call gootkit_2_stage.A32870
00A3F5FF 50           push eax
00A3F600 6A 00       push 0
00A3F602 FF 15 AC 91 A5 00 call dword ptr ds:[<&GetProcAddress>]
00A3F608 85 C0       test eax,eax
00A3F60A 75 0D       jne gootkit_2_stage.A3F619
00A3F60C 68 10 27 00 00 push 2710
00A3F611 FF 15 E4 91 A5 00 call dword ptr ds:[<&Sleep>]
00A3F617 EB DE       jmp gootkit_2_stage.A3F5F7
00A3F619 8D 4D F8      lea ecx,dword ptr ss:[ebp-8]
00A3F61C E8 EF 32 FF FF call gootkit_2_stage.A32910
    
```

After that, we encountered the implementation of a particular persistence mechanism that it seems Gootkit has been using for many months: it’s already documented in various blog posts, for ex. ReaQta blog.

Briefly, the infostealer generates a INF file with the same filename of itself.

```

00A38A8E FF 15 3C 92 A5 00 call dword ptr ds:[<&CreateFile>]
00A38A94 89 45 FC     mov dword ptr ss:[ebp-4],eax
00A38A97 83 7D FC FF  cmp dword ptr ss:[ebp-4],FFFFFFFF
00A38A9B 75 42       jne gootkit_2_stage.A38ADF
00A38A9D FF 15 8C 91 A5 00 call dword ptr ds:[<&GetLastError>]
00A38AA3 89 45 F8     mov dword ptr ss:[ebp-3],eax
00A38AA6 83 7D F8 20  cmp dword ptr ss:[ebp-3],20
00A38AAA 75 33       jne gootkit_2_stage.A38ADF
00A38AAC 8B 45 14     mov eax,dword ptr ss:[ebp+14]
    
```

Content of the INF file:

```

[Version]
signature = "$CHICAGO$"
AdvancedINF = 2.5, "You need a new version of advpack.dll"

[DefaultInstall]
RunPreSetupCommands = qwpmkzhnsnhgtlqyemydppcpui:2

[qwpmkzhnsnhgtlqyemydppcpui]
%TEMP%\NidduHDKFLF.exe
    
```

Then it creates 3 different registry keys (“Count”, “Path1” and “Section1”) inside “HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs” with the purpose to allow the threat to execute on reboot.

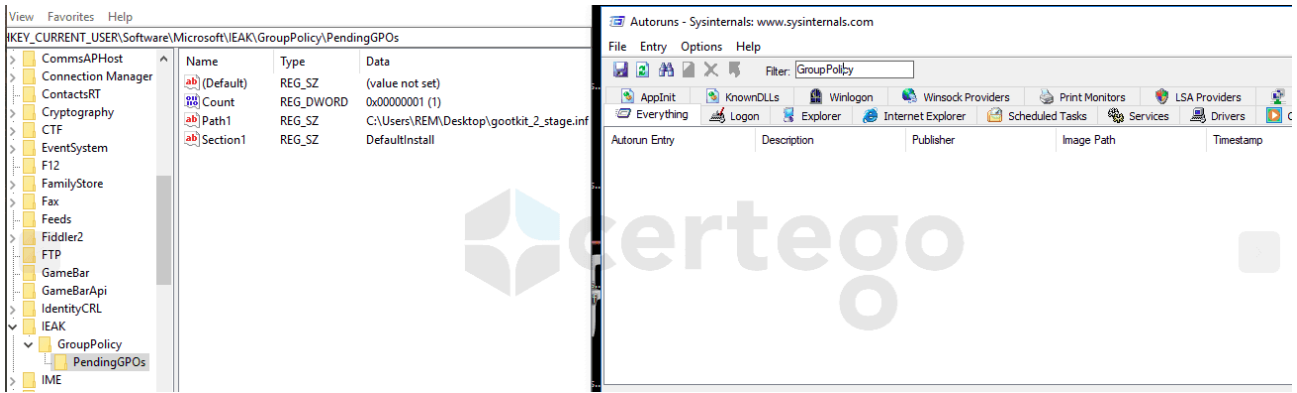
```

00A3C594 FF 15 64 90 A5 00 call dword ptr ds:[<&RegCreateKeyEx>]
00A3C597 89 45 FC     mov dword ptr ss:[ebp-4],eax
00A3C57A 75 18       jne gootkit_2_stage.A3C597
00A3C57C 8D 4D F8     lea ecx,dword ptr ss:[ebp-8]
00A3C57F 51         push ecx
00A3C580 8B 55 10     mov edx,dword ptr ss:[ebp+10]
00A3C583 52         push edx
00A3C584 6A 00       push 0
00A3C586 8B 45 0C     mov eax,dword ptr ss:[ebp+C]
00A3C589 50         push eax
00A3C58A 8B 4D 08     mov ecx,dword ptr ss:[ebp+8]
00A3C58D 51         push ecx
    
```

It seems that this technique was reported to be used **only** by Gootkit.

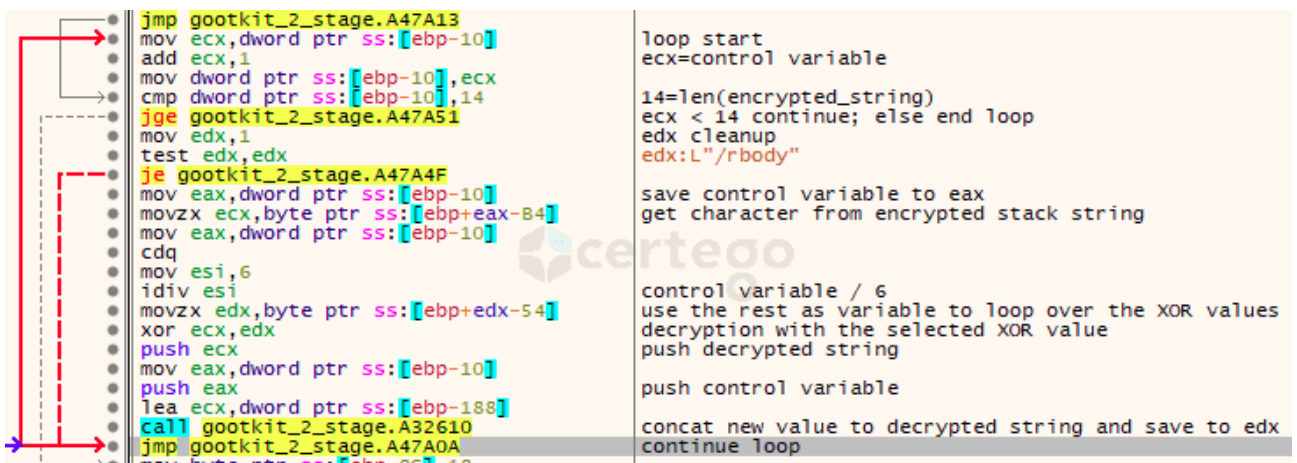
Famous security tools still can’t detect this mechanism even if it has been used for months.

For example, the famous SysInternal Autoruns tool, that should be able to show all the programs that are configured to run on system bootup or login, fails the detection of this persistence method.



Stepping through the code, we noticed that, at runtime, Gootkit decrypts the strings it uses with a custom algorithm to evade static analysis detection of anomalous behaviour.

It's a combination of "stack strings", XOR commands and the modulo operation.



An exhaustive explanation of the decryption routine can be found here: [link](#)

Skipping further, eventually there's a call to "CreateProcessW" to start a new instance of Gootkit with the following parameter: --vwxyz

6.Stage 3: Check-in phase

Quickly we found out that executing the malware with the cited parameter allows us to skip all the previous anti-analysis controls to get into the part of the code that starts to contact the Command & Control Server.

The first check-in to home is done to the following URL via HTTPS:

The response that arrives from the previous connection contains the final stage of Gootkit, configured to work properly on the infected machine.

The malware dynamically loaded “RtlDecompressBuffer” call to use it to decompress the payload; then, it injected into an area of the current process memory.

Afterwards the flow of execution is transferred to the start of the injected code.

The final payload is a DLL file that is bigger than 5MB because it contains the Node.js engine which is probably needed to run some embedded javascript files. At this time we decided to stop our analysis and leave the rest to future work.

8. Additional findings

While debugging, we noticed that Gootkit does not check only if a parameter called “--vwxyz” was passed to the command line. Also it checks if other 3 parameters:

- **--reinstall**

```
EAX    00000014
EBX    00000000
ECX    0000006C    '1'
EDX    02FA5390    L"--reinstall"
EBP    04C1F994
ESP    04C1F950
ESI    00000005
EDI    02FA9A98
```

- **--service**

```
EAX    00000014
EBX    00000000
ECX    00000000
EDX    0341BC68    L" --service"
EBP    050FF684
ESP    050FF640
ESI    00000006
EDI    03429A98
```

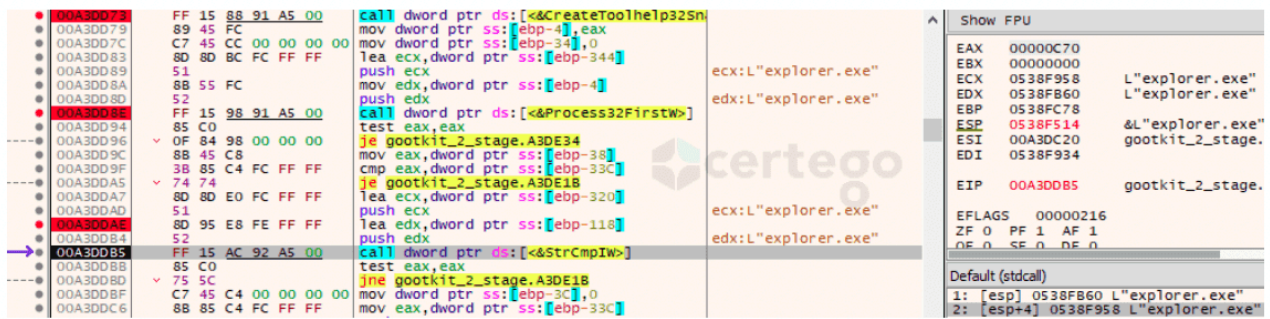
- **-test**

```
EAX    00000009
EBX    00000000
ECX    00000000
EDX    03428998    L"-test"
EBP    050FF684
ESP    050FF64C
ESI    00000005
EDI    03429A98
```

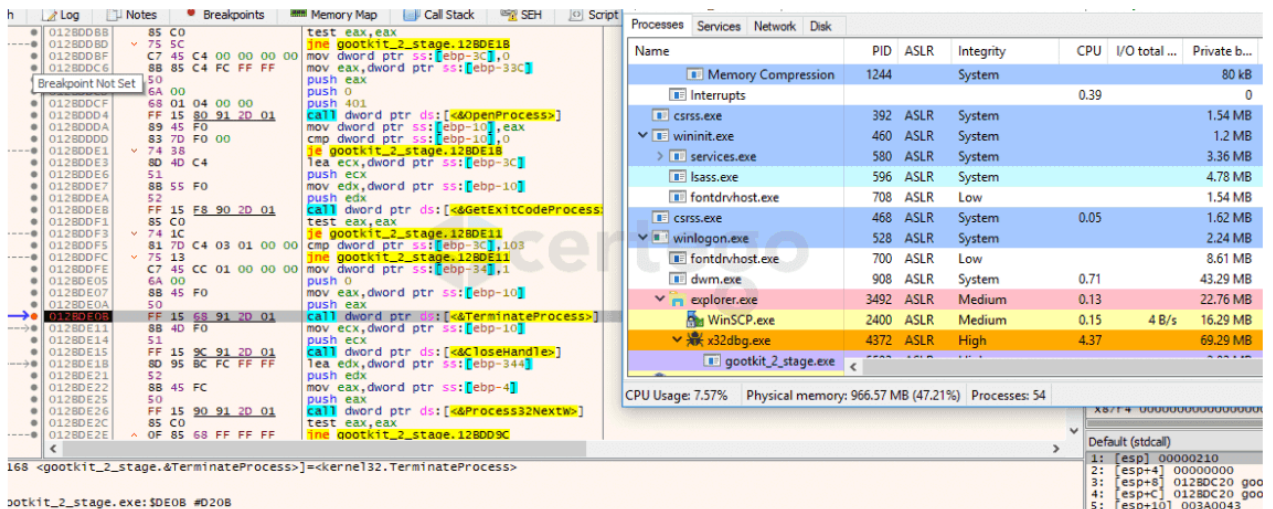
Pretty strange thing. We haven't found the malware to actively use these arguments yet. However, stepping through code we discovered that:

1 - the "--reinstall" command led the execution to some curious code. First, the malware used "CreateToolHelp32Snapshot" to retrieve a list of the current running processes.

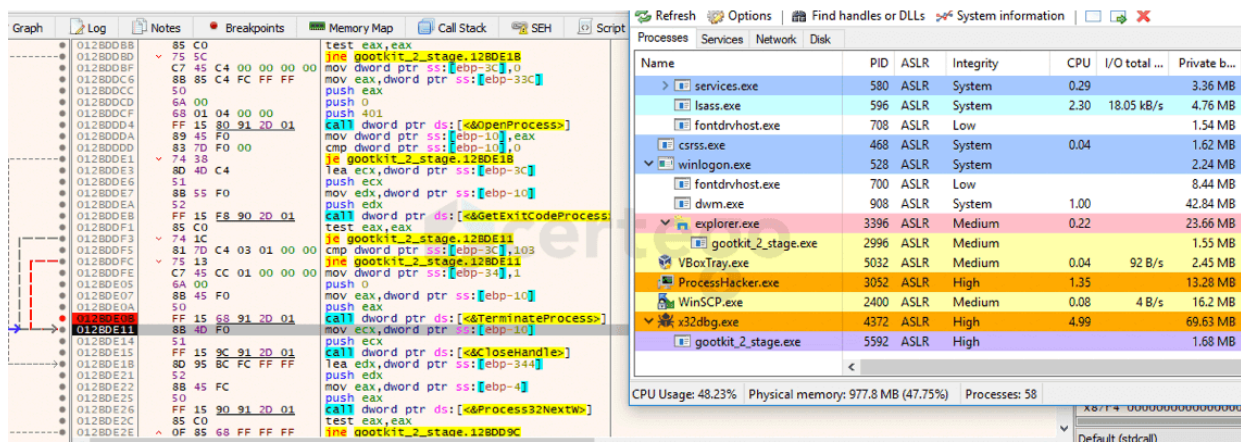
Then, it iterated through the retrieved list via "Process32FirstW" and "Process32NextW" with the aim to get a handle to the active "explorer.exe" instance.



At this point it killed "explorer.exe". The following image shows the process list before the "TerminateProcess" command.



After having executed that command, we found that a new instance of the malware spawned as a child of "explorer.exe".



What happened? We performed some tests and it seems that “explorer.exe” was killed and then automatically restarted by “winlogon.exe”. Therefore “explorer.exe” accessed the keys involved in the persistence mechanism previously explained:

```

0:55: explorer.exe 3676 RegOpenKey HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs
0:55: explorer.exe 3676 RegQueryV... HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs\Path1
0:55: explorer.exe 3676 RegQueryV... HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs\Section1
    
```

2 - the “--service” command did not change the flow of execution with the exception of creating a new environment variable called “USERNAME_REQUIRED” and set it to “TRUE”.

Eventually we found that the final stage checks if the aforementioned variable exists.

```

push eax
call dword ptr ds:[&SetEnvironmentVariable] eax:"USERNAME_REQUIRED"
lea ecx,dword ptr ss:[ebp-154] variable di ambiente per --se
call gootkit_2_stage.A32910 [ebp-154]:"TRUE"
lea ecx,dword ptr ss:[ebp-158]
call gootkit_2_stage.A32990 [ebp-158]:"USERNAME_REQUIRED"
    
```

3 - the “-test” command just terminate the process. Indeed it’s a test.

9. Conclusions

We explored some of the functionalities of one of the most widespread Infostealers of these days, revealing new and old tricks that is using to remain undetected as much time as possible.

Certego is actively monitoring every day threats to improve our detection and response methods, continuously increasing the effectiveness of the incident response workflow.

PS: Let us know if you liked this story and feel free to tell us how we can improve it!

Hash

```

GLS_Notifica.js
5ed739855d05d9601ee65e51bf4fec20d9f600e49ed29b7a13d018de7c5d23bc
gootkit 1st stage
e32d72c4ad2b023bf27ee8a79bf82c891c188c9bd7a200bfc987f41397bd61df
    
```

gootkit 2nd stage

0ad2e03b734b6675759526b357788f56594ac900eeb5bd37c67b52241305a10a

gootkit DLL module

Athor:

Matteo Lodi, Threat Intelligence Lead Engineer

Source: <https://www.certego.net/en/news/malware-tes-gootkit/>