FickerStealer: A New Rust Player in the Market

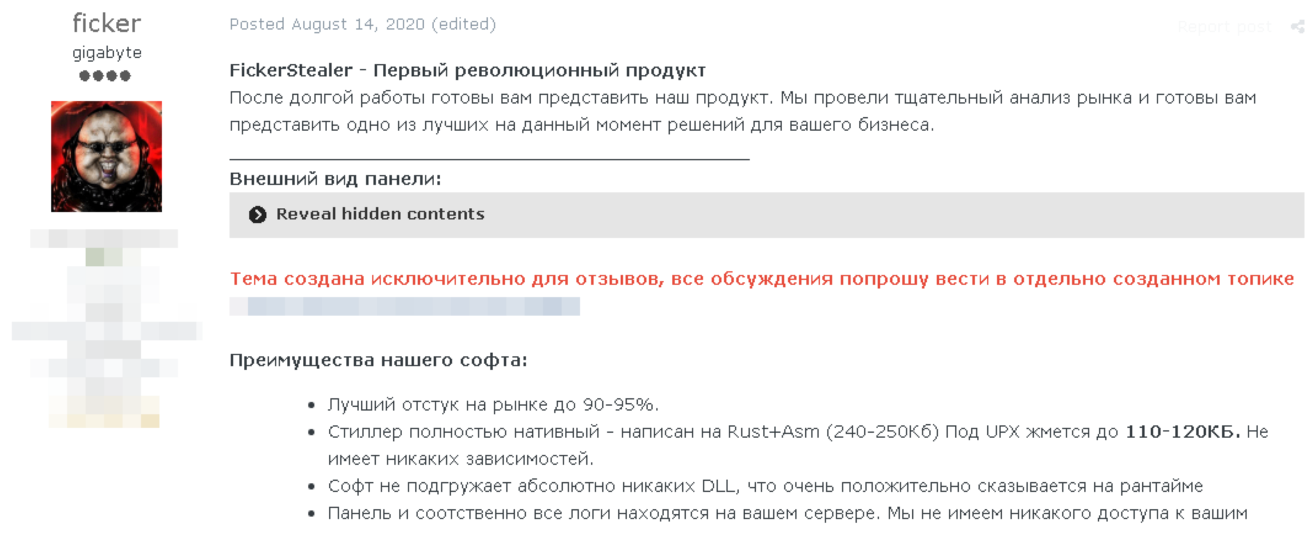
cyberark.com/resources/threat-research-blog/fickerstealer-a-new-rust-player-in-the-market

July 19, 2021

This blog introduces a new information stealer, written in Rust and interestingly named **FickerStealer**. In this blog post, we provide an in-depth analysis of this new threat and its obfuscation techniques, including an IDAPython deobfuscator script.

Background

FickerStealer is an exceptional information stealer, first seen in the wild in **August 2020**. This info-stealer was sold on underground forums as a MaaS (Malware-as-a-service) by a seller named @ficker (@ficker_sup on Telegram). The average price of an information stealer is $200, and it varies depending on their additional capabilities and the subscription time for the service. The price for FickerStealer cost ranges between $90 up to $900, and it only depends on the subscription time to the service.


ficker
gigabyte

Posted August 14, 2020 (edited)

FickerStealer - Первый революционный продукт
После долгой работы готовы вам представить наш продукт. Мы провели тщательный анализ рынка и готовы вам представить одно из лучших на данный момент решений для вашего бизнеса.

Внешний вид панели:

Reveal hidden contents

Тема создана исключительно для отзывов, все обсуждения попрошу вести в отдельно созданном топике

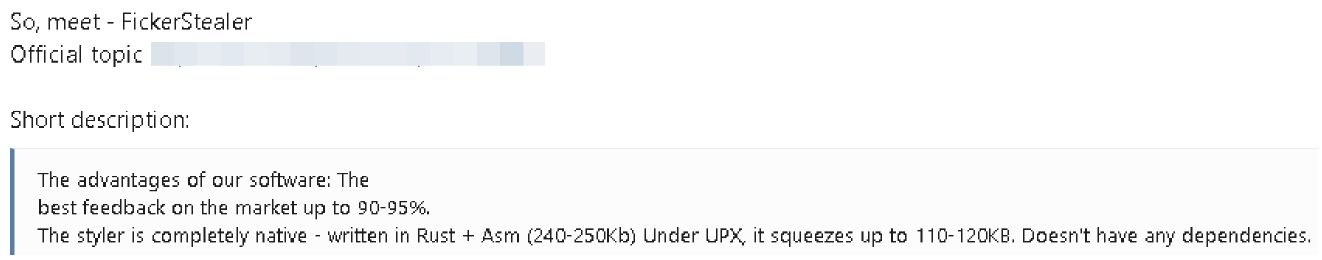
Преимущества нашего софта:

- Лучший отстук на рынке до 90-95%.
- Стиллер полностью нативный - написан на Rust+Asm (240-250Кб) Под UPX жмется до **110-120КБ.** Не имеет никаких зависимостей.
- Софт не подгружает абсолютно никаких DLL, что очень положительно сказывается на рантайме
- Панель и соотственно все логи находятся на вашем сервере. Мы не имеем никакого доступа к вашим

Figure 1: FickerStealer post on a known Russian forum

Once a buyer purchases the Ficker service, an “on-prem” package will be provided, including the C2 server setup (called *Panel*) and the stealer executable (called *Build*). After the purchase, the buyer is required to specify what the C2 server’s address will be, so the malware author (the seller) will configure the malware (*build*) to communicate with the attacker’s C2 server.
One of the more surprising things about FickerStealer is that it’s written in Rust and Assembly. Rust language isn’t a natural choice for malware developers. While it is very fast and memory-efficient and runs on embedded devices, developers might take a longer time to learn Rust because of its complexities. The advantages of this programming language are not utilized to their fullest when building malware, except that it’s difficult to reverse engineer it. Because of that, there are few known malware families written in Rust, so it was a bit surprising to find a new Rust infostealer.

FickerStealer has quickly gained popularity among cybercriminals on underground forums because of its attractive price and interesting and different methodology as an information stealer (we will dive into that later). The malware received many good reviews from customers. For example: ***“I use it for targeted attacks, in a couple of months the product showed itself at its best, I recommend.”*** It even received an excellent review from some known underground forum moderators — the dream of every malware author.


So, meet - FickerStealer
Official topic

Short description:

> The advantages of our software: The
> best feedback on the market up to 90-95%.
> The styler is completely native - written in Rust + Asm (240-250Kb) Under UPX, it squeezes up to 110-120KB. Doesn't have any dependencies.

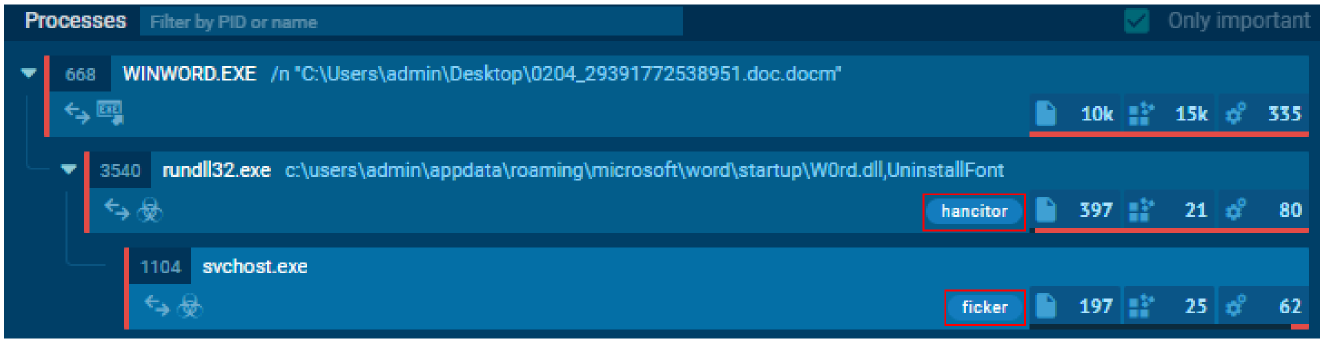
Figure 2: Part of the forum moderator review (translated from Russian)

The malware is used to steal sensitive information, including **login credentials**, **credit card information**, **cryptocurrency wallets and browser information** from applications such as WinSCP, Discord, Google Chrome, Electrum, etc. It does all that by implementing a different approach than other stealers (we'll cover it later). Additionally, FickerStealer can function as a **File Grabber** and collect additional files from the compromised machine, and it can act as a **Downloader** to download and execute several second-stage malware.

Delivery Methods – Malware Distribution

We noticed that FickerStealer is primarily delivered by two different methods: **Exploit kit – RigEK** and **Malspam – Hancitor**.

- **RigEK** –The victims enter a malicious/compromised website. The malicious website acts as a *gate* that can perform some checks of the user characteristics, like host machine, GeoIP, user-agent, etc. In case this type of user is targeted, the user is redirected to the RigEK landing page.
  The landing page scans and profiles the victim's browser and checks if it is vulnerable to any of the exploit kit vulnerabilities (most of them are 1-day vulnerabilities). In case the browser is vulnerable, the landing page executes the exploit to load the malware payload.
- **Hancitor** – Malicious spam campaigns have proven to be an efficient infection vector. The malware is delivered via targeted phishing emails with malicious macro-based documents attached. The macro drops the Hancitor loader, which then injects the final payload using process hollowing. It's worth mentioning that we found many samples of FickerStealer that Hancitor delivers. If you are interested in more information about Hancitor, you should check Binary Defense's blog.



```
Processes   Filter by PID or name                                   Only important
668  WINWORD.EXE  /n "C:\Users\admin\Desktop\0204_29391772538951.doc.docm"
                                                        10k   15k   335
  3540  rundll32.exe  c:\users\admin\appdata\roaming\microsoft\word\startup\W0rd.dll,UninstallFont
                                             hancitor   397   21    80
    1104  svchost.exe
                                             ficker     197   25    62
```

Figure 3: Hancitor loading Ficker payload in a sandbox environment

Technical Overview

As we mentioned earlier, Ficker is written in Rust, making the analysis complex and challenging because it's compiled differently than C/C++ language. On top of that, the Ficker executable is also heavily obfuscated.

FikcerStealer’s main objective, like all credential theft malware, is to gather as much sensitive information as possible. Before we dive into analyzing FickerStealer‘s features, the following are the main diffrencess between Ficker and other stealers:

- Ficker **doesn’t have dependencies**. It doesn’t need to download/load other DLLs like *dll*, *nss3.dll* on runtime to fulfill its stealing capabilities.
- The **communication channel with the C2 server is encrypted/obfuscated** from the client-side, unlike most stealers that rely on HTTP protocol and send the data as plain text.
- Ficker **sends the stolen data after each successful stealing operation** and by doing that, it doesn’t write any logs/files to disk. Other stealers write the stolen data to some temp folder, compress it and send it as a zip file to the C2 server. In addition, Ficker sends the sensitive Mozilla browser data to the C2 server and **decrypts it on the server-side**.
- The Downloader feature can download and **execute several PE files, including executing malicious executables and loading DLLs**.

Ficker tries to make its malware more reliable and stealthier than other stealers.

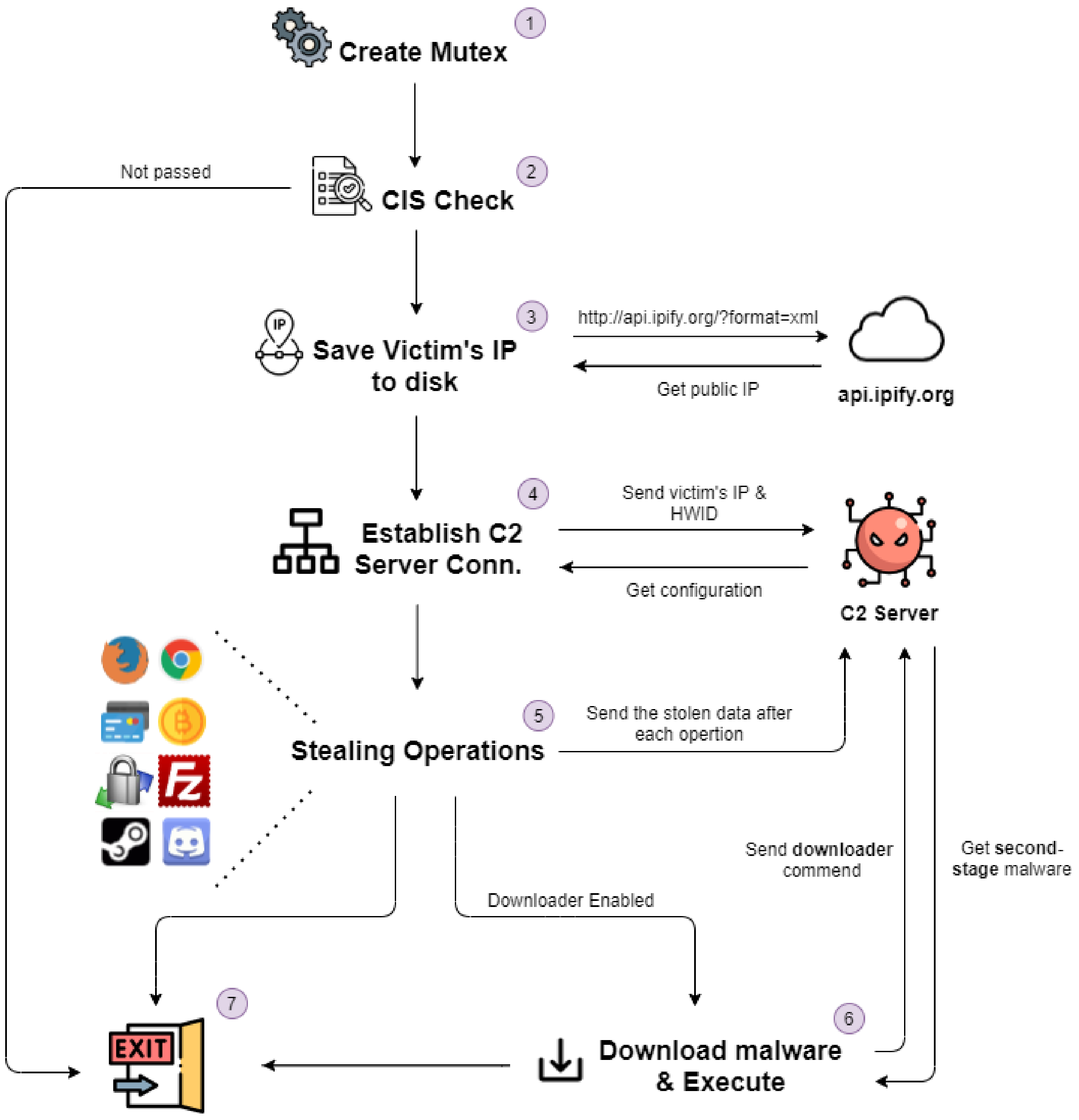


Figure 4: Malware flow

In-depth Analysis

As far as we know, the first Rust malware was spotted in the wild in 2016. So far, we haven't heard about many Rust malware families or any trend of writing malware in Rust among malware developers as we have seen lately with Golang. Most of the Rust malware we stumbled upon, like Telebots backdoor, **contains Rust debug information**, which makes our life easier while analyzing those Rust samples. Unfortunataley, in our case, the debug information of our FickerStealer sample is stripped out, making the analysis more difficult.

As you can probably guess, we began by opening it in IDA, and we noticed that there are no symbols for Rust functions (or any developer functions). In addition, we could easily tell that the sample was heavily obfuscated: all the strings are encrypted; it uses many dynamic calls; and most of the functions are just wrappers on wrappers with minor modifications.

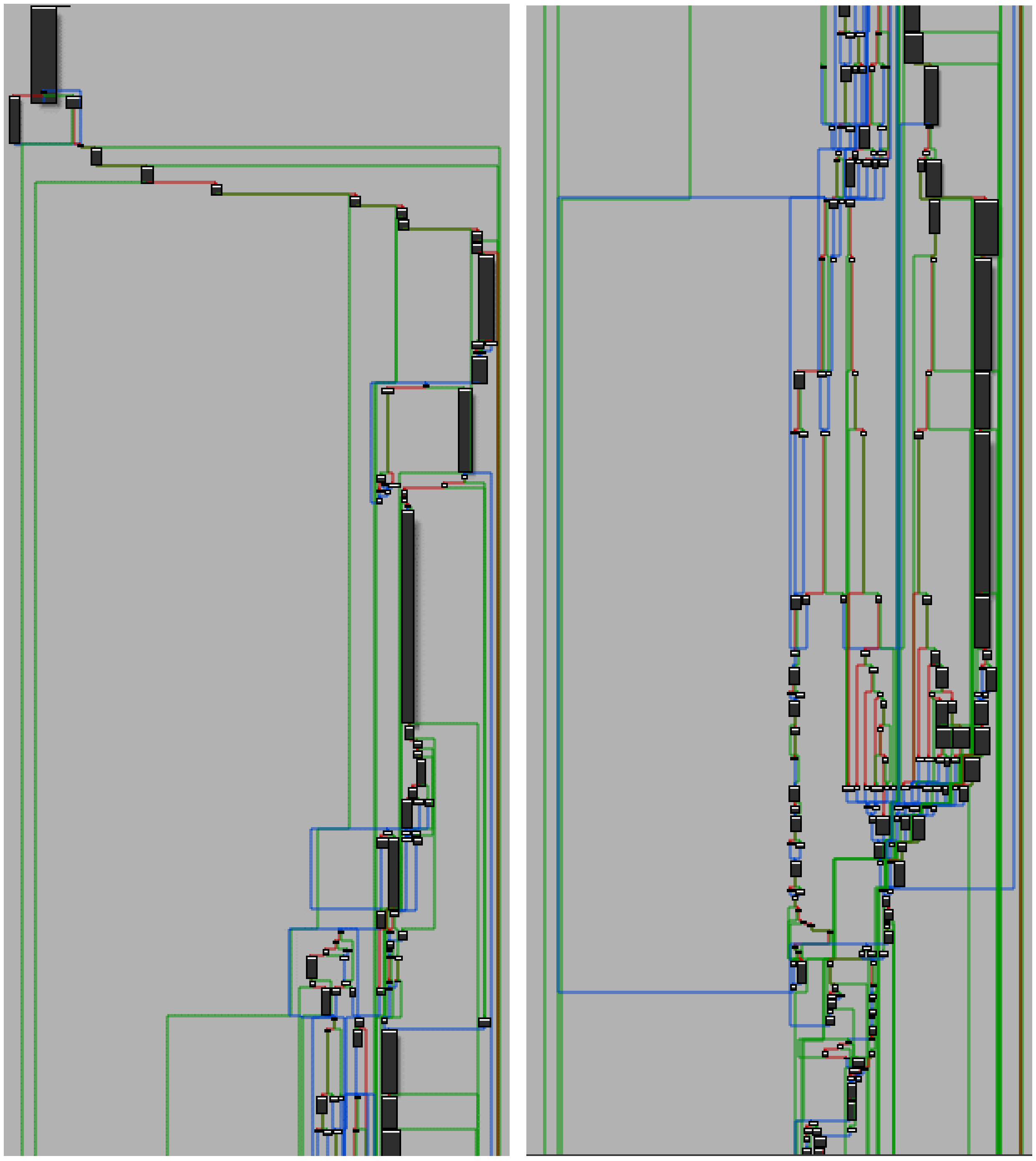


Figure 5: Part of Ficker's WinMain function

Let's see how we approached the task of dissecting the malware.

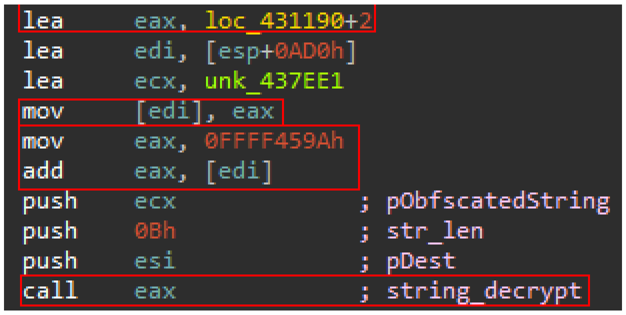
Our mindset was to first “map” the malware by finding its strings, as that would reveal the malware’s functionality and allow us to make educated guesses about what a snippet of code or function does.

Strings Obfuscation

The problem is that all Ficker’s strings are obfuscated, including API function names, C2 server address, mutex name, sensitive and targeted file names, etc. As mentioned, revealing those strings can shed some light on the malware capabilities and IoCs.

While analyzing Ficker’s string decryption process, we found that Ficker doesn’t decrypt all the strings at once like Oski Stealer does. It decrypts each string at a different location on runtime just before it uses it. Therefore, we had to **find the string decryption function** and **find every call to it**.

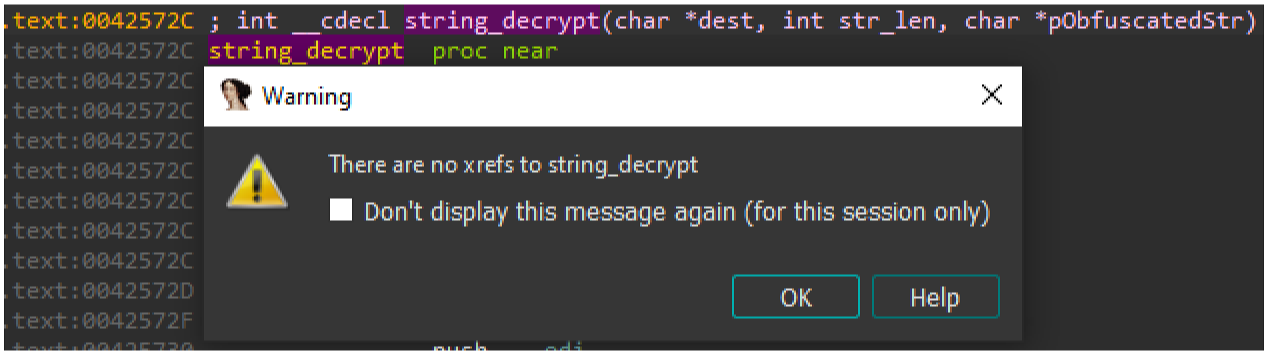
Also, while going over the malware, we noticed many dynamic calls, so we started debugging it. Ficker tries to hide its call to the string decryption function (string_decrypt) by obfuscating the function address. It calculates the function’s address by using different instructions and different offsets. In figure 6 (highlighted with red boxes), you can see a simple dynamic call to string_decrypt using short and straightforward obfuscation (which is not the case in most calls to string_decrypt).



```
lea     eax, loc_431190+2
lea     edi, [esp+0AD0h]
lea     ecx, unk_437EE1
mov     [edi], eax
mov     eax, 0FFFF459Ah
add     eax, [edi]
push    ecx                     ; pObfscatedString
push    0Bh                     ; str_len
push    esi                     ; pDest
call    eax                     ; string_decrypt
```

Figure 6: Obfuscated call to string_decrypt function

In our case, the string decryption function is at **0x42572C** (0x431192 + 0xFFFF459A). Our next step is finding where the function is called; however, when we checked the *xrefs* for this function, we saw that it had no references.



```
.text:0042572C ; int __cdecl string_decrypt(char *dest, int str_len, char *pObfuscatedStr)
.text:0042572C string_decrypt  proc near
```

Warning

There are no xrefs to string_decrypt

☐ Don't display this message again (for this session only)

Figure 7: string_dercypt function has no references

We believe there are two reasons for this:

1. The malware calls this function dynamically in an obfuscated way, as we saw earlier, or
2. It has other string decryption routines (which is not likely.)

Before digging into this issue, let's focus on the string_decrypt function.

String Decryption Function

The string decryption function receives three arguments: a **destination pointer to write the decrypted string**, the **string length** and the **pointer to the obfuscated string**.

The obfuscated string has a pretty simple structure. The first four bytes are the **key** for the decryption algorithm, and the rest of the bytes (as the size of the string length argument) are the **encrypted string**.

After delving deeper into the function, we noticed a few things:

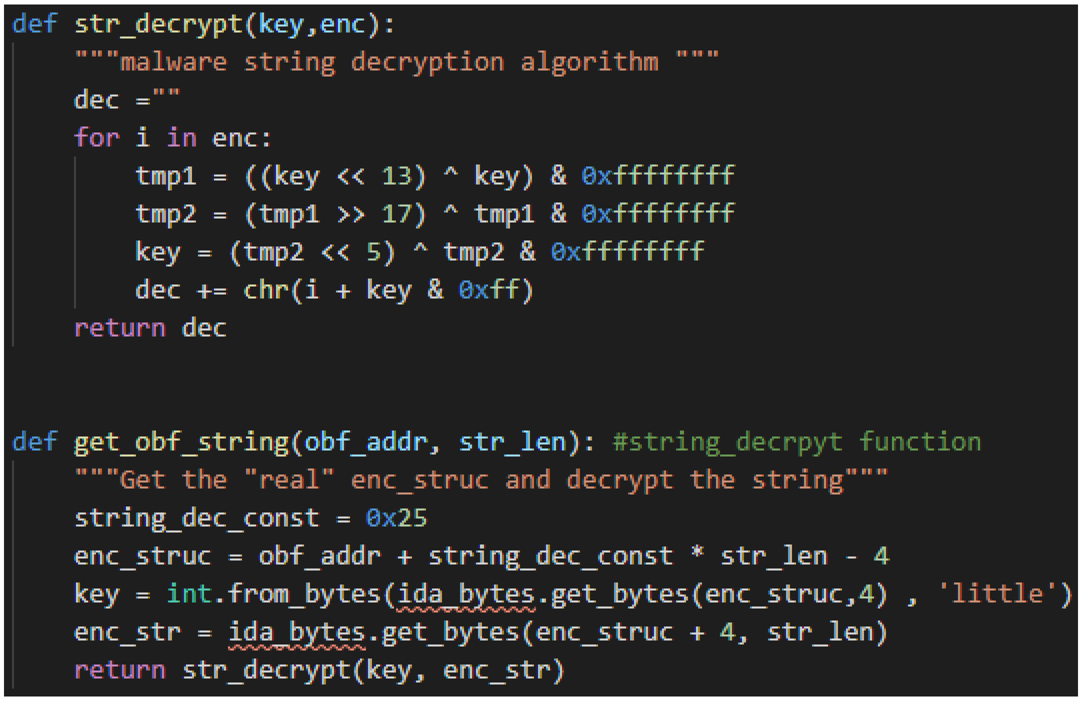
1. The pointer for the obfuscated string is also obfuscated; the pointer is not the one for the "real" structure (that we described before).
2. The function first calculates the "real" address and retrieves the encrypted string and the decryption key.
3. After getting the "real" structure, the function implements the decryption algorithm.

The function uses a hardcoded constant to calculate the address of the "real" string structure. As much as we know — and at the time of writing this blog post — this constant remains the same in all Ficker's samples. It's **0x25**.

The address of the structure is calculated in this way:
enc_struc = obf_addr + string_dec_const * str_len – 4.
For example, the function gets the address 0x436FF8 (*obf_addr*) and the length of the decrypted string which is 0x06 (*str_len*). Eventually, we'll get the address 0x437494 that holds the "real" string structure for Pidgin.

After getting the "real" address, the function decrypts the string using a simple decryption algorithm.

We wrote an IDAPython script that simulates the string_decrypt function, including the decryption algorithm.



```
def str_decrypt(key,enc):
    """malware string decryption algorithm """
    dec =""
    for i in enc:
        tmp1 = ((key << 13) ^ key) & 0xffffffff
        tmp2 = (tmp1 >> 17) ^ tmp1 & 0xffffffff
        key = (tmp2 << 5) ^ tmp2 & 0xffffffff
        dec += chr(i + key & 0xff)
    return dec


def get_obf_string(obf_addr, str_len): #string_decrpyt function
    """Get the "real" enc_struc and decrypt the string"""
    string_dec_const = 0x25
    enc_struc = obf_addr + string_dec_const * str_len - 4
    key = int.from_bytes(ida_bytes.get_bytes(enc_struc,4) , 'little')
    enc_str = ida_bytes.get_bytes(enc_struc + 4, str_len)
    return str_decrypt(key, enc_str)
```

Figure 8: IDAPython script for string_decrypt function

After writing a script that can decrypt any string, we just need to find where it’s called and what the parameters are for each call (on every location).

It sounds like a great mission for an IDAPython script! If you remember, we noted that it is more likely that string_decrypt will be called on runtime.

Let’s break our mission into tasks:

1. Find all dynamic calls in the binary — find all call instructions with some register (e.g., call eax).
2. Calculate the address in the register (can be obfuscated) and compare the calculated address to string_decrypt function’s address.
3. Get the function arguments’ values (string length, pointer to obfuscated string structure), which can also be obfuscated.
4. Use our IDAPython str_decrypt function to decrypt the string (figure 8).

The interesting part is how we can deal with Ficker’s obfuscation. Ficker mostly uses obfuscation techniques like calculating the data, using useless instructions and pushing-popping data.

To deal with that, we wrote two IDAPython functions get_value, get_ref.

The primary function (get_value) is a recursive function, which gets the instruction’s address as an argument and returns the value used on this instruction, e.g., the address of push ebx will return the value ebx holds.

But a question comes to mind: How can we find it? We use the get_ref function that gets this instruction's address and the operand index as arguments and returns the first reference to the given operand.

The get_value function first identifies the instruction (call, mov, lea, add instruction, etc.'). Therefore, we can tell that it knows which operand may contain the data and checks this operand's type if it is mem_addr or imm_value. These types of operand indicate if it's a piece of data that's stored as part of the instruction itself instead of being in a memory location or a register. In case the operand holds any value (mem_addr/imm_value), the function will return this value. Otherwise, the function gets a reference to this operand by calling get_ref and then it calls recursively to itself with the new instruction's address. The recursive function breaks when it finds mem_addr/imm_value.

Let's see how it works with an example. The goal here is to find the address for the call on 0x415C6A:

```
.text:00415C51                  lea     eax, loc_4342CA+2
.text:00415C57                  lea     ecx, unk_437BEF
.text:00415C5D                  mov     [edi], eax
.text:00415C5F                  mov     eax, 0FFFF1460h
.text:00415C64                  add     eax, [edi]
.text:00415C66                  push    ecx
.text:00415C67                  push    21h ; '!'
.text:00415C69                  push    esi
.text:00415C6A                  call    eax
```

Code Snippet 1: Disassembly snippet from Ficker's binary

We call our recursive function with the address 0x415C6A to find the address of eax. The function checks the type of the relevant operand (in this case, the first one because it's a call instruction). The operand is a register, which is not a mem_addr/imm_value. Therefore, the function searches for a reference to this operand (eax). The function calls itself with the new address of the reference, 0x415C64.

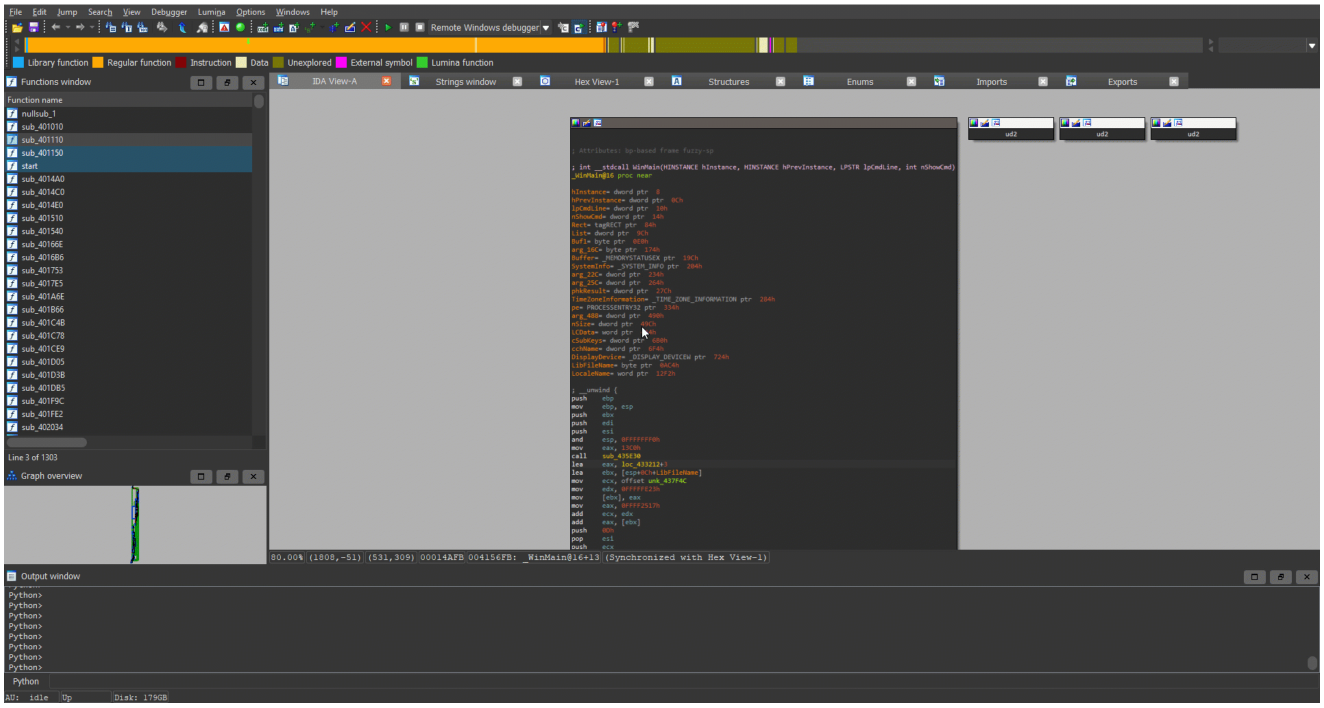
And again, it checks the type of both operands because it add instruction and searches a reference for both operands (eax and edi). After the function finds those registers' values recursively, it finally will return the sum of them.

Ficker Deobfuscator: An IDAPython script

You can find our IDAPython deobfuscator in our Malware Research repository — FickerStealer.

We gathered all those parts together and added more functionalities into one IDAPython script. The script defeats Ficker's obfuscation and decrypts all the strings along the binary that Ficker uses.

1. The script finds the decryption function address by searching (using pattern-based) the string's decryption algorithm.
2. It obtains the addresses of all dynamic calls in the binary, and for each one, deobfuscate the address on the call instruction and compares it to the decryption function address.
3. It gets the arguments (by stack) of the string decryption call; the string decryption function gets three arguments (destination pointer, string length and a pointer to the obfuscated string.)
4. It gets the data that the arguments should hold.
5. Deobfuscate the string using the function get_obf_string, which gets the pointer for the obfuscated string and the string length.
6. This function returns the decrypted string.



```
Function name
nullsub_1
sub_401010
sub_401110
sub_401150
start
sub_4014A0
sub_4014C0
sub_4014E0
sub_401510
sub_401540
sub_40166E
sub_4016B6
sub_401753
sub_4017E5
sub_401A6E
sub_401B66
sub_401C4B
sub_401C78
sub_401CE9
sub_401D05
sub_401D3B
sub_401DB5
sub_401F9C
sub_401FE2
sub_402034
Line 3 of 1303
```

```
; Attributes: bp-based frame fuzzy-sp

; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
_WinMain@16 proc near

hInstance= dword ptr  8
hPrevInstance= dword ptr  0Ch
lpCmdLine= dword ptr  10h
nShowCmd= dword ptr  14h
Rect= tagRECT ptr  84h
List= dword ptr  9Ch
Buf1= byte ptr  0E0h
arg_16C= byte ptr  174h
Buffer= _MEMORYSTATUSEX ptr  19Ch
SystemInfo= _SYSTEM_INFO ptr  204h
arg_22C= dword ptr  234h
arg_25C= dword ptr  264h
phkResult= dword ptr  27Ch
TimeZoneInformation= _TIME_ZONE_INFORMATION ptr  284h
pe= PROCESSENTRY32 ptr  334h
arg_488= dword ptr  490h
nSize= dword ptr  49Ch
LCData= word ptr  [illegible]4h
cSubKeys= dword ptr  6B0h
cchName= dword ptr  6F4h
DisplayDevice= _DISPLAY_DEVICEW ptr  724h
LibFileName= byte ptr  0AC4h
LocaleName= word ptr  12F2h

; __unwind {
push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi
and     esp, 0FFFFFFF0h
mov     eax, 13C0h
call    sub_435E30
lea     eax, loc_433212+3
lea     ebx, [esp+0Ch+LibFileName]
mov     ecx, offset unk_437F4C
mov     edx, 0FFFFFE23h
mov     [ebx], eax
mov     eax, 0FFFF2517h
add     ecx, edx
add     eax, [ebx]
push    0Dh
pop     esi
push    ecx
```

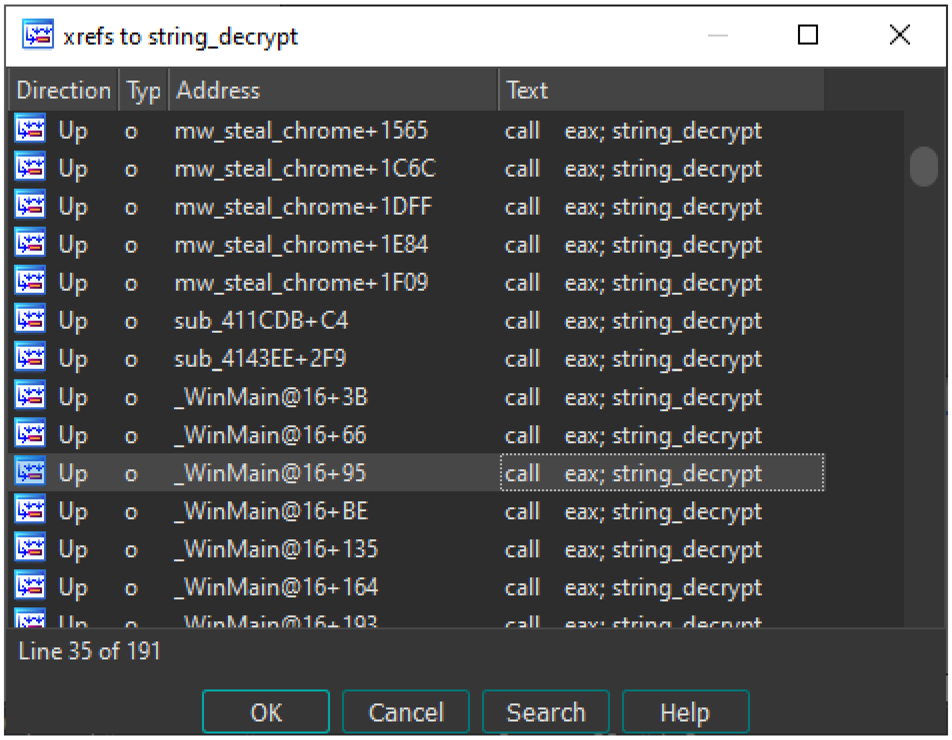
```
80.00% (1808,-51) (531,309) 00014AFB 004156FB: _WinMain@16+13 (Synchronized with Hex View-1)
```

```
Python>
Python>
Python>
Python>
Python>
Python>
Python>
Python>
Python
AU:  idle   Up      Disk: 179GB
```

Figure 9: FickerStealer's deobfuscator usage

The script also sets a reference to string_decrypt function, adds a comment with the decrypted string and dumps all the data to a JSON file.

And now, having collected the fruits of our labor, our deobfuscator script was able to identify 191 calls to the string decryption function (string_decrypt) and to decrypt 182 strings in this sample.


xrefs to string_decrypt

Direction	Typ	Address	Text
Up	o	mw_steal_chrome+1565	call eax; string_decrypt
Up	o	mw_steal_chrome+1C6C	call eax; string_decrypt
Up	o	mw_steal_chrome+1DFF	call eax; string_decrypt
Up	o	mw_steal_chrome+1E84	call eax; string_decrypt
Up	o	mw_steal_chrome+1F09	call eax; string_decrypt
Up	o	sub_411CDB+C4	call eax; string_decrypt
Up	o	sub_4143EE+2F9	call eax; string_decrypt
Up	o	_WinMain@16+3B	call eax; string_decrypt
Up	o	_WinMain@16+66	call eax; string_decrypt
Up	o	_WinMain@16+95	call eax; string_decrypt
Up	o	_WinMain@16+BE	call eax; string_decrypt
Up	o	_WinMain@16+135	call eax; string_decrypt
Up	o	_WinMain@16+164	call eax; string_decrypt
Up	o	_WinMain@16+193	call eax; string_decrypt

Line 35 of 191

Figure 10: References for string_decrypt function

Finally, after we got all Ficker's strings, we have an idea about what most functions and code parts do, so we can now move forward.

C2 server communication

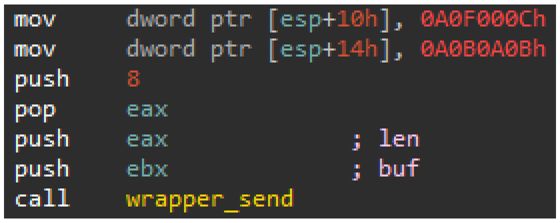
Establishing Connection

Ficker holds the address and port for the C2 server as an encrypted string, IP:PORT, which can be easily identified by running the IDAPython script. The string can be an IP address or domain name, and if the C2 address is a domain name, the malware tries to resolve it until it succeeds.

The malware communicates with the C2 server over TCP. It creates a TCP socket to the C2 server by using the IP address and the port. After the connection is established, Ficker calls to recv function twice. First, it receives two bytes that indicate the data size it needs to allocate. It then allocates this memory and receives the data from the C2 server.

The received data is the Grabber rules configuration of the C2 server and contains a list of Grabber rules configured by the attacker. We will focus on this section on the Grabber part.

After getting the grabber rules sent as plaintext, Ficker sends some hardcoded value 0x0C000F0A0B0A0B0A (not encrypted) to the C2. We assume that this hardcoded value indicates that the connection was established successfully and the configuration has been received.



```
mov     dword ptr [esp+10h], 0A0F000Ch
mov     dword ptr [esp+14h], 0A0B0A0Bh
push    8
pop     eax
push    eax                 ; len
push    ebx                 ; buf
call    wrapper_send
```

Figure 11: Send the hardcoded value

The last part of this connection establishing method is sending the address of the C2 server encrypted/obfuscated using a function we named send_enc_data.

Sending Data to C2

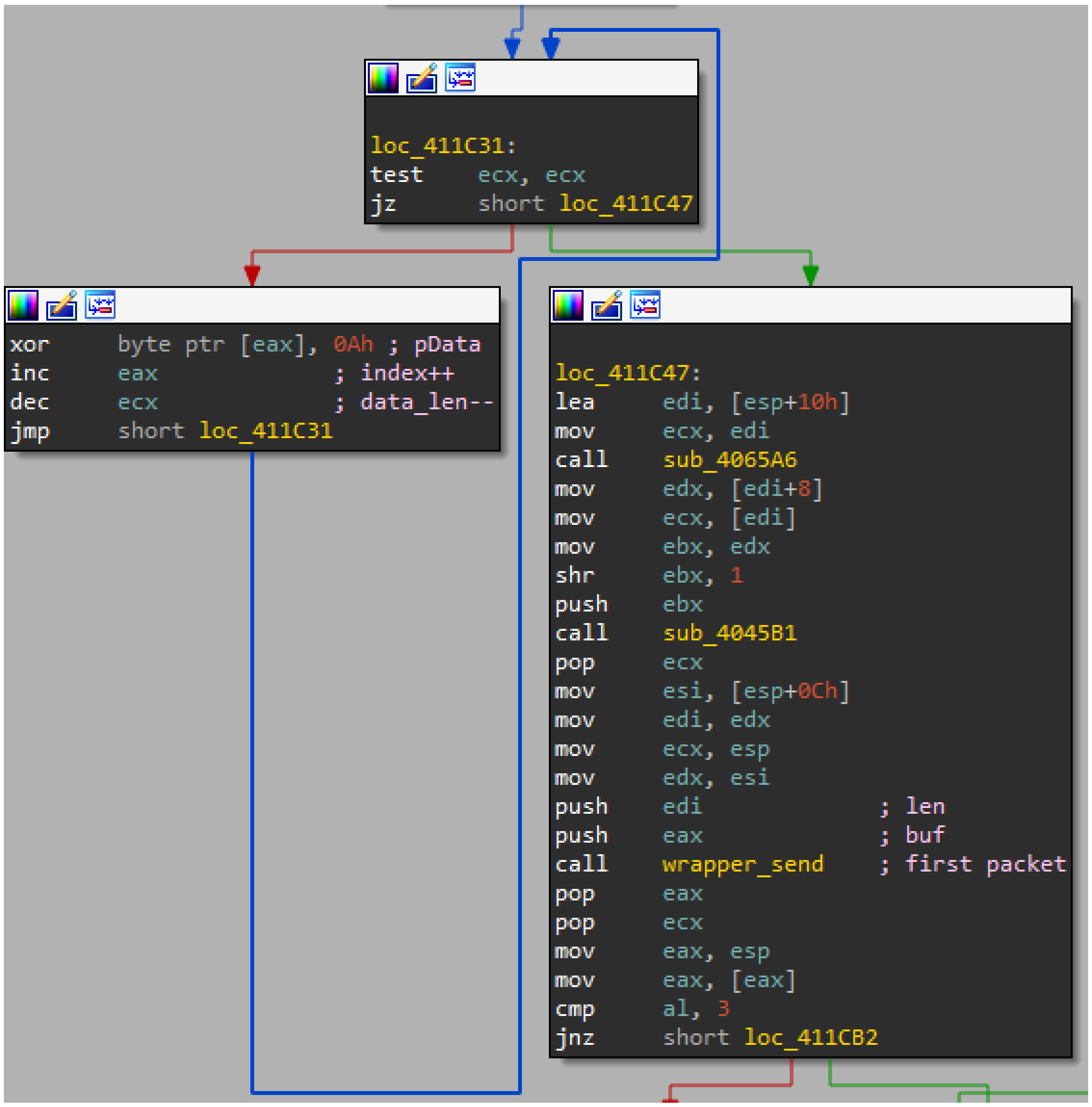
FickerStealer has a singular method for sending the data to the C2 server, and it does it in an encrypted/obfuscated way. Using this method makes the network analysis of this malware a little bit more difficult and stealthier than sending the data as plaintext like most of the stealers.

The function send_enc_data gets two arguments: a **pointer to the data** and an **index flag**. The index flag indicates the data type used by the C2 server. This function first “encrypts” the data using simple XOR rotation with one byte XOR key: 0x0A. So far, we noticed the XOR byte is the same in all Ficker’s samples.

After encrypting the data and before sending it, Ficker creates a packet for sending the data:

4 bytes: the size of the encrypted data,
1 byte: the index flag
X bytes: the encrypted data

Finally, Ficker sends this data to the C2 server in **two separate requests**.



```
loc_411C31:
test    ecx, ecx
jz      short loc_411C47
```

```
xor     byte ptr [eax], 0Ah ; pData
inc     eax                 ; index++
dec     ecx                 ; data_len--
jmp     short loc_411C31
```

```
loc_411C47:
lea     edi, [esp+10h]
mov     ecx, edi
call    sub_4065A6
mov     edx, [edi+8]
mov     ecx, [edi]
mov     ebx, edx
shr     ebx, 1
push    ebx
call    sub_4045B1
pop     ecx
mov     esi, [esp+0Ch]
mov     edi, edx
mov     ecx, esp
mov     edx, esi
push    edi                 ; len
push    eax                 ; buf
call    wrapper_send        ; first packet
pop     eax
pop     ecx
mov     eax, esp
mov     eax, [eax]
cmp     al, 3
jnz     short loc_411CB2
```

Figure 12: The simple XOR rotation and sending the first packet

As we mentioned earlier, the index flag indicates the data type sent to the C2 server. We check the xrefs of this function, and we found out it is called from 13 different locations and gets 12 different values of the index flag argument, from 0x00 to 0x0B.

By setting a conditional breakpoint on the entry of this function, we could dump the plaintext data and the index flag on every call. By correlating these parameters, we figured out what each index flag value represents.

Index Flag value	**Data type**
0x00	Chromium-based browsers
0x01	Mozilla-based browsers
0x02	FTP clients

Index Flag value	Data type
0x03	Grabber files
0x04	Screenshots
0x05	System information
0x06	Cryptocurrency wallets
0x07	Hardware ID
0x08	Others (Desktop clients)
0x09	C2 address and port
0x0A	poppush – Downloader command
0x0B	Victim public IP

Table 1: Mapped index flag values

Stealing Capabilities

Most applications store confidential information in various files/registry keys, and most of the information stealers have a hardcoded list of known/default locations where the sensitive file is located for each targeted application. Ficker's methodology of stealing sensitive information from files is quite different from other stealers.

Ficker doesn't have this typical path list for each application. It recursively searches a unique file that belongs to the targeted application from a starting location (path). This recursive search is implemented by a function we named find_file. The function gets a path (starting location) and the sensitive file's name to search, which can also be a regex expression. The function returns a list of all the locations where the file was found under the starting location. This method allows Ficker to find the application folder containing the sensitive files without knowing exactly where they are stored (the application may be installed on a different location) — and without having all those suspicious strings.

Mozilla-based applications

First, Ficker calls to find_file with the full path of %appdata% and the file name cookies.sqlite — a file used by Mozilla-based application that stores web cookies. By doing that, it gets a list of all profile folders for every installed Mozilla-based application. After it finds every sensitive folder, it moves to the next step: **stealing the data**. Ficker steals the user's **saved login credentials, cookies,** and **autocomplete history**. All those sensitive files are under the locations Ficker found at the first step.

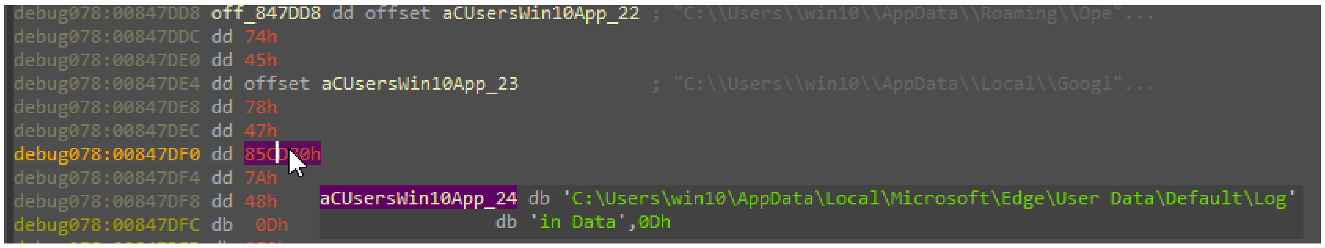
The main difference for stealing Mozilla-based applications' information is the way in which Ficker steals the user's saved credentials. Most of the stealers parse the login.json file and decrypt the confidential information by getting the decryption key from other known database files (key4.db, cert9.db, or key3.db, cert8.db for older versions). Instead of parsing the credentials file and dealing with the decryption part, Ficker sends those files to the C2 server, and the server does all the parsing and decrypting work of these credentials. Our guess here is that Ficker does this to run without having any dependencies like *nss3.dll* and to appear less suspicious.

In the case of data that doesn't require decryption operations, such as the autocomplete and cookies data, Ficker parses it on the victim's side and then sends the data to the C2 server.

Chromium-based browsers

Ficker employs a similar methodology to steal Chromium-based data like it used to steal from Mozilla-based applications.

To begin with, Ficker decrypts five strings that will be starting locations: %appdata%, %localappdata%, %userprofile%\Desktop, %userprofile%\Documents, %USERPROFILE%\Local Settings\Application Data. It also decrypts the string Login Data, the DB file name used by Chromium-based browsers that stores saved credentials. By searching this file under those starting locations, Ficker gets all Chromium-based browsers users' profile folders.



```
debug078:00847DD8 off_847DD8 dd offset aCUsersWin10App_22 ; "C:\\Users\\win10\\AppData\\Roaming\\Ope"...
debug078:00847DDC dd 74h
debug078:00847DE0 dd 45h
debug078:00847DE4 dd offset aCUsersWin10App_23           ; "C:\\Users\\win10\\AppData\\Local\\Googl"...
debug078:00847DE8 dd 78h
debug078:00847DEC dd 47h
debug078:00847DF0 dd 85CD70h
debug078:00847DF4 dd 7Ah
debug078:00847DF8 dd 48h    aCUsersWin10App_24 db 'C:\Users\win10\AppData\Local\Microsoft\Edge\User Data\Default\Log'
debug078:00847DFC db  0Dh                      db 'in Data',0Dh
```

Figure 13: The folder found by find_file

For each user's profile folder, it extracts the user's master key from the Local State file by parsing this JSON file and getting the value of encrypted_key. This value is *Base64* encoded and encrypted using *DPAPI*; hence, Ficker decodes this value and decrypts it using CryptUnprotectData. This key is used to decrypt confidential information that Chromium-based browsers save in their SQLite database files.

For each existing profile in the browser, Ficker targets the Login Data, Cookies, and Web Data SQLite DB files in order to steal the user's **saved login credentials, cookies, and autocomplete history**.

One of the exciting things about SQLite database files is that Ficker **parses those files and extracts the sensitive data independently**. As we mentioned earlier, Ficker doesn't have any dependencies like most of the stealers, which download or drop some dlls like *sqlite3.dll* that make their job easier. It is common to see information stealers using *sqlite3.dll*'s exported functions to extract sensitive data using SQL queries. For instance, getting the user's saved credentials from Login Data using this query: SELECT origin_url, username_value, password_value FROM logins.

However, Ficker doesn't use any SQL query to extract the sensitive data from the DB files. It goes over the SQLite3 database file format and extracts the sensitive file by parsing the DB file.

Cryptocurrency wallets

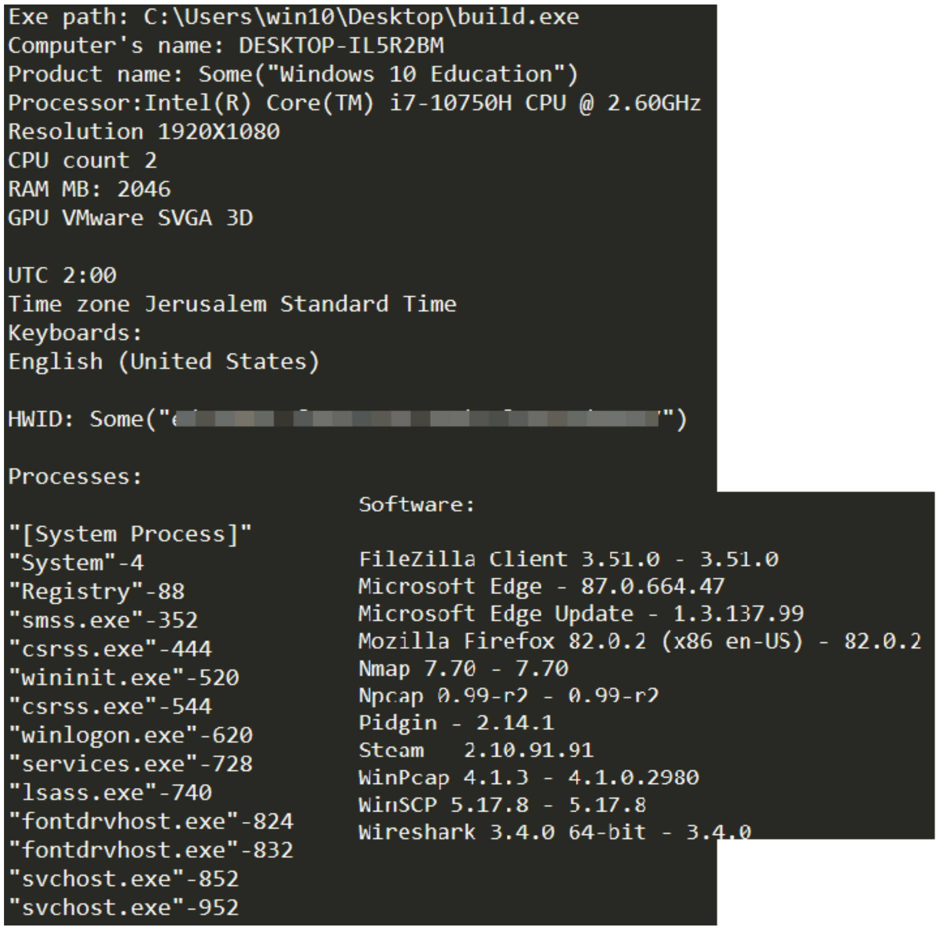
Like most information stealers, Ficker also steals crypto wallets. It doesn't target every crypto wallet application, but it does target the more popular ones. For every wallet application, Ficker decrypts the application folder location and the sensitive file name (e.g., *.wallet). After getting the location of those files using find_file, Ficker reads those files into memory and sends all the data to the C2 server.

The targeted applications are Bitcoin, Jaxx, Exodus, Atomic, Electrum, Zcash, Bytecoin, Ethreum, Monero, Litecoin, and Dash.

System information

Ficker gathers information and takes a screenshot of the compromised machine. When it finishes collecting this information, it gathers all the data together and sends it to the C2 server.

Moreover, Ficker collects basic information about the machine like the location of the running executable, OS version, hardware and graphic information, system time zone and language. In addition, it lists all the running process and their PIDs, as well as the installed applications on the victim's machine.



```
Exe path: C:\Users\win10\Desktop\build.exe
Computer's name: DESKTOP-IL5R2BM
Product name: Some("Windows 10 Education")
Processor:Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Resolution 1920X1080
CPU count 2
RAM MB: 2046
GPU VMware SVGA 3D

UTC 2:00
Time zone Jerusalem Standard Time
Keyboards:
English (United States)

HWID: Some("[redacted]")

Processes:

"[System Process]"
"System"-4
"Registry"-88
"smss.exe"-352
"csrss.exe"-444
"wininit.exe"-520
"csrss.exe"-544
"winlogon.exe"-620
"services.exe"-728
"lsass.exe"-740
"fontdrvhost.exe"-824
"fontdrvhost.exe"-832
"svchost.exe"-852
"svchost.exe"-952
```

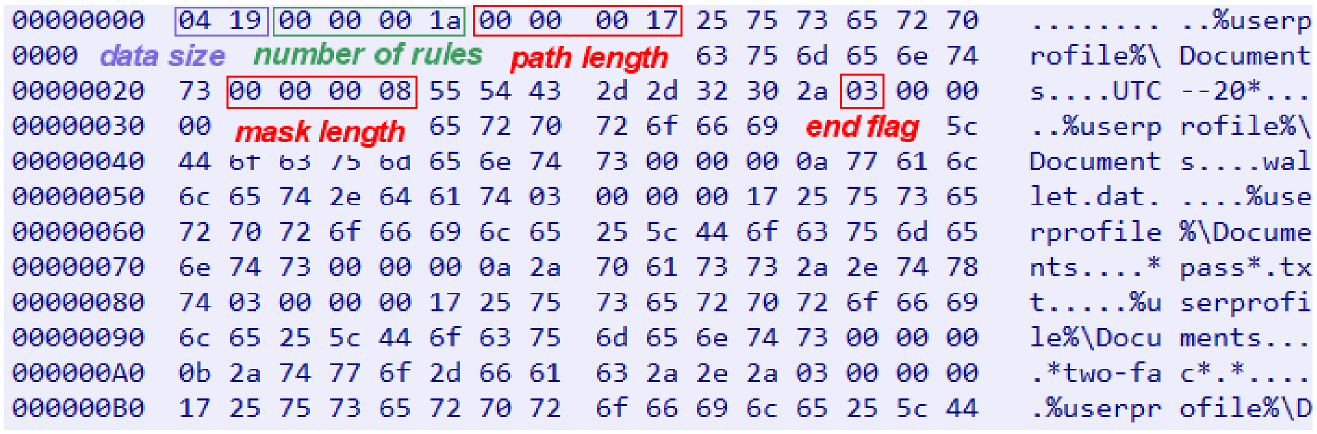
```
Software:

FileZilla Client 3.51.0 - 3.51.0
Microsoft Edge - 87.0.664.47
Microsoft Edge Update - 1.3.137.99
Mozilla Firefox 82.0.2 (x86 en-US) - 82.0.2
Nmap 7.70 - 7.70
Npcap 0.99-r2 - 0.99-r2
Pidgin - 2.14.1
Steam   2.10.91.91
WinPcap 4.1.3 - 4.1.0.2980
WinSCP 5.17.8 - 5.17.8
Wireshark 3.4.0 64-bit - 3.4.0
```

Figure 14: Collected system information

Grabber

FickerStealer has a grabber feature that allows the attacker to collect any files from the victim’s machine and send those to the C2 server. This feature is configurable, and the attacker can enable or disable it from the C2 panel. When the malware establishes its connection to the C2 server (as we covered earlier), it gets the Grabber’s configuration from the C2 server.

The received data contains a list of Grabber rules. Each rule has the start location and its mask; the mask can be a regex or file name, so the malware searches for files by the mask under the start location recursively.



```
00000000  04 19 00 00 00 1a 00 00  00 17 25 75 73 65 72 70  ........ ..%userp
0000                                63 75 6d 65 6e 74  rofile%\ Document
00000020  73 00 00 00 08 55 54 43  2d 2d 32 30 2a 03 00 00  s....UTC --20*...
00000030  00                65 72 70  72 6f 66 69       5c  ..%userp rofile%\
00000040  44 6f 63 75 6d 65 6e 74  73 00 00 00 0a 77 61 6c  Document s....wal
00000050  6c 65 74 2e 64 61 74 03  00 00 00 17 25 75 73 65  let.dat. ....%use
00000060  72 70 72 6f 66 69 6c 65  25 5c 44 6f 63 75 6d 65  rprofile %\Docume
00000070  6e 74 73 00 00 00 0a 2a  70 61 73 73 2a 2e 74 78  nts....* pass*.tx
00000080  74 03 00 00 00 17 25 75  73 65 72 70 72 6f 66 69  t.....%u serprofi
00000090  6c 65 25 5c 44 6f 63 75  6d 65 6e 74 73 00 00 00  le%\Docu ments...
000000A0  0b 2a 74 77 6f 2d 66 61  63 2a 2e 2a 03 00 00 00  .*two-fa c*.*....
000000B0  17 25 75 73 65 72 70 72  6f 66 69 6c 65 25 5c 44  .%userpr ofile%\D
```

Figure 15: The received Grabber's configuration

The structure of the Grabber configuration looks like Code Snippet 2:

```
struct rule{
        int path_len;
        char* startFolder; #size of path_len
        int mask_len;
        char* mask; #size of mask_len
        char end_flag;
};

struct grabberRules{
        int rules_num;
        rule* rule_list; #size of rules_num
};
```

Code Snippet 2: Grabber configuration structure

We wrote a simple python script that parses this data by this structure. We noticed that attackers mainly focus on searching other cryptocurrency wallets in different locations, password manager confidential information and password files. For instance, path:%userprofile%\Documents, mask:*.kdbx targets KDBX files, which are used by KeePass and usually refer to the KeePass Password Database. You can check our IoCs page in our Malware Research Repository to find more examples of Grabber's rules.

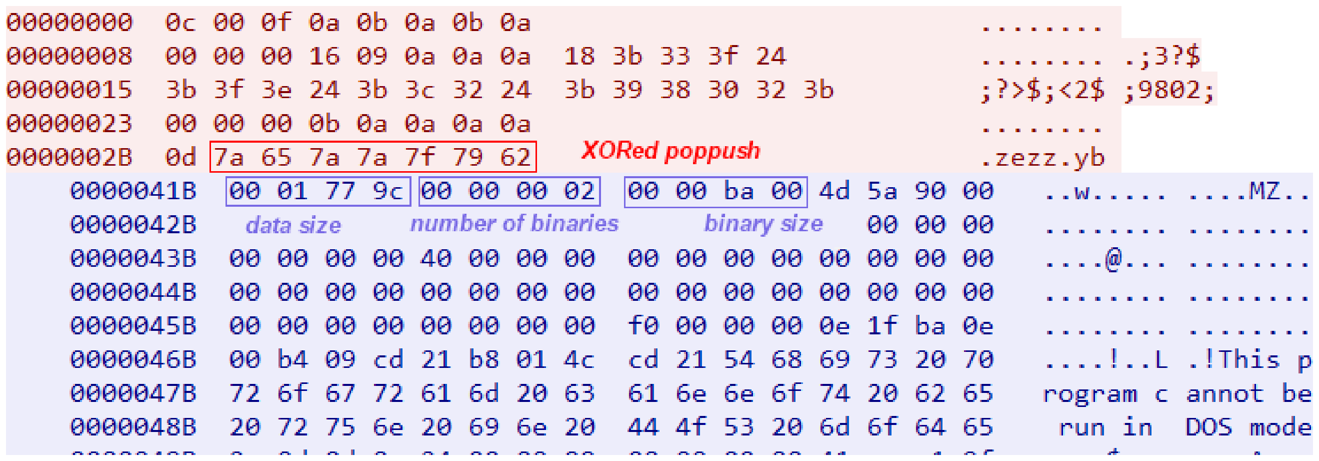
Downloader

After Ficker steals as much data as possible, it moves for its last feature, which is the Loader. Ficker can download and execute several second-stage malware that can be done by running executables or loading DLLs.

The attacker can enable/disable the downloader feature from the C2 panel. The panel of the malware also allows setting checks prior to downloading the second stage malware — checking some of the victim's characteristics like GeoIP, stolen credentials type, etc.

Before receiving the second stage malware from the C2 server, Ficker sends a command to the C2 server- poppush.

The malware first decrypts the string poppush and then sends it to the C2 server by using send_enc_data function. We assume that this string/magic commands the C2 server to return the second-stage malware data.



```
00000000  0c 00 0f 0a 0b 0a 0b 0a                          ........
00000008  00 00 00 16 09 0a 0a 0a  18 3b 33 3f 24          ........ .;3?$
00000015  3b 3f 3e 24 3b 3c 32 24  3b 39 38 30 32 3b       ;?>$;<2$ ;9802;
00000023  00 00 00 0b 0a 0a 0a 0a                          ........
0000002B  0d 7a 65 7a 7a 7f 79 62                          .zezz.yb
    0000041B  00 01 77 9c 00 00 00 02  00 00 ba 00 4d 5a 90 00   ..w..... ....MZ..
    0000042B                                       00 00 00   ........ ........
    0000043B  00 00 00 00 40 00 00 00  00 00 00 00 00 00 00 00   ....@... ........
    0000044B  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
    0000045B  00 00 00 00 00 00 00 00  f0 00 00 00 0e 1f ba 0e   ........ ........
    0000046B  00 b4 09 cd 21 b8 01 4c  cd 21 54 68 69 73 20 70   ....!..L .!This p
    0000047B  72 6f 67 72 61 6d 20 63  61 6e 6e 6f 74 20 62 65   rogram c annot be
    0000048B  20 72 75 6e 20 69 6e 20  44 4f 53 20 6d 6f 64 65    run in  DOS mode
```

Figure 16: Downloader communication

As we can see in figure 16, the malware sends the XORred poppush string, and it receives a data structure from the C2 server.

The first four bytes are the data size, so Ficker first receives only four bytes and then allocates a new memory buffer for the Downloader data. The next four bytes indicate the number of binaries that received from the C2 server. In the communication with the C2 server above, we can see it's configured to download two binaries. The next four bytes indicate the first binary size, which is right after these four bytes.

After receiving the second-stage malware, Ficker generates a random name for each binary and gets the Temp folder of the current user. Ficker decrypts the binary's extension string, which can be .exe or .dll depending on the binary type. Then it concatenates all those three strings into a valid file path – %temp%+random+decrypted_exetension, e.g., *C:\Users\win10\AppData\Local\Temp\1606398000688.exe*. After that, Ficker writes the binary data to this path.

The last part of this feature is executing the second stage malware. Ficker uses CreateProcess to run an executable, and it uses LoadLibraryA to load a DLL.

Conclusion

We foresee an expanding number of opportunities for FickerStealer among cybercriminals because of its efficiency and differentiation from other stealers, such as its stealing methodology and being written in Rust.

Ficker has all the capabilities that cybercriminals look for in information stealers. Moreover, Ficker uses different obfuscation techniques and because it’s written in Rust, it makes the analysis of the malware more difficult — leading us to one of our main conclusions. Our first strategy to “map” the malware by finding its strings paid off because it saved us time analyzing the sample. By doing that, we could eventually understand any part of the malware quite quickly and focus only on the features that interest us without digging into the malware.

Credential-theft malware will always be a popular resource used by adversaries. Even though information stealers might not be the most sophisticated malware, they are still effective enough to cause significant damage to organizations and individuals.

To defend against these threats, organizations can use an endpoint protection solution like CyberArk Endpoint Privilege Manager, combine it with fundamental security awareness, deploy MFA and use of strong and unique passwords to further mitigate risk.

Appendix A: Targeted Applications

Targeted Applications
All Mozilla-based browsers and Thunderbird
All Chromium-based browsers
Cryptocurrency wallets application: Bitcoin, Jaxx, Exodus, Atomic, Electrum, Zcash, Bytecoin, Ethreum, Monero, Litecoin, and Dash.
Windows Vault
FTP clients: WinSCP, FileZilla
Dicscord, Pidgin
Steam

Appendix B: IoCs & YARA Rule

Hashes
a4113ccb55e06e783b6cb213647614f039aa7dbb454baa338459ccf37897ebd6
25fe7dd7a49dac5706ac0772f8baf415e7b554e68d904bc2e026ac2cb4848527
6029558794981d135cf41756c3e2de0cb4b08f1533a7dcd945b2ac9ff02535bf
ed635f60d1cc542377ea9f0b0723f19fe998a8eb6319373a1a3177066d5d4816

382055f0fca8a6172846236a2a9031ce9103359673b4b13e1c4c04bc1861d941
94e60de577c84625da69f785ffe7e24c889bfa6923dc7b017c21e8a313e4e8e1
25fe7dd7a49dac5706ac0772f8baf415e7b554e68d904bc2e026ac2cb4848527
5a904972e7ce7ef0b9484daa3eee9860ece27845692ca5750d2b80b24acd6a8f
6029558794981d135cf41756c3e2de0cb4b08f1533a7dcd945b2ac9ff02535bf
b00121c90392716403386a4d407015430e121eced0603af7dc0c8a996a61cb5f
e5ac51cbaab11a34c180417118933758ceae64d5fbbc00ebf210e6664963082c
4abf05cd0f538e4223732861771175268 7faddfe314afe5adf07a24155305df5

C2 servers

45.141.84[.]139:80
195.154.168[.]132:81
sweyblidian[.]com:80
mamkindomen[.]info:80
93.115.22[.]72:80
95.217.5[.]249:80
139.59.66[.]32:81

Yara Rule

FickerStealer Yara Rule