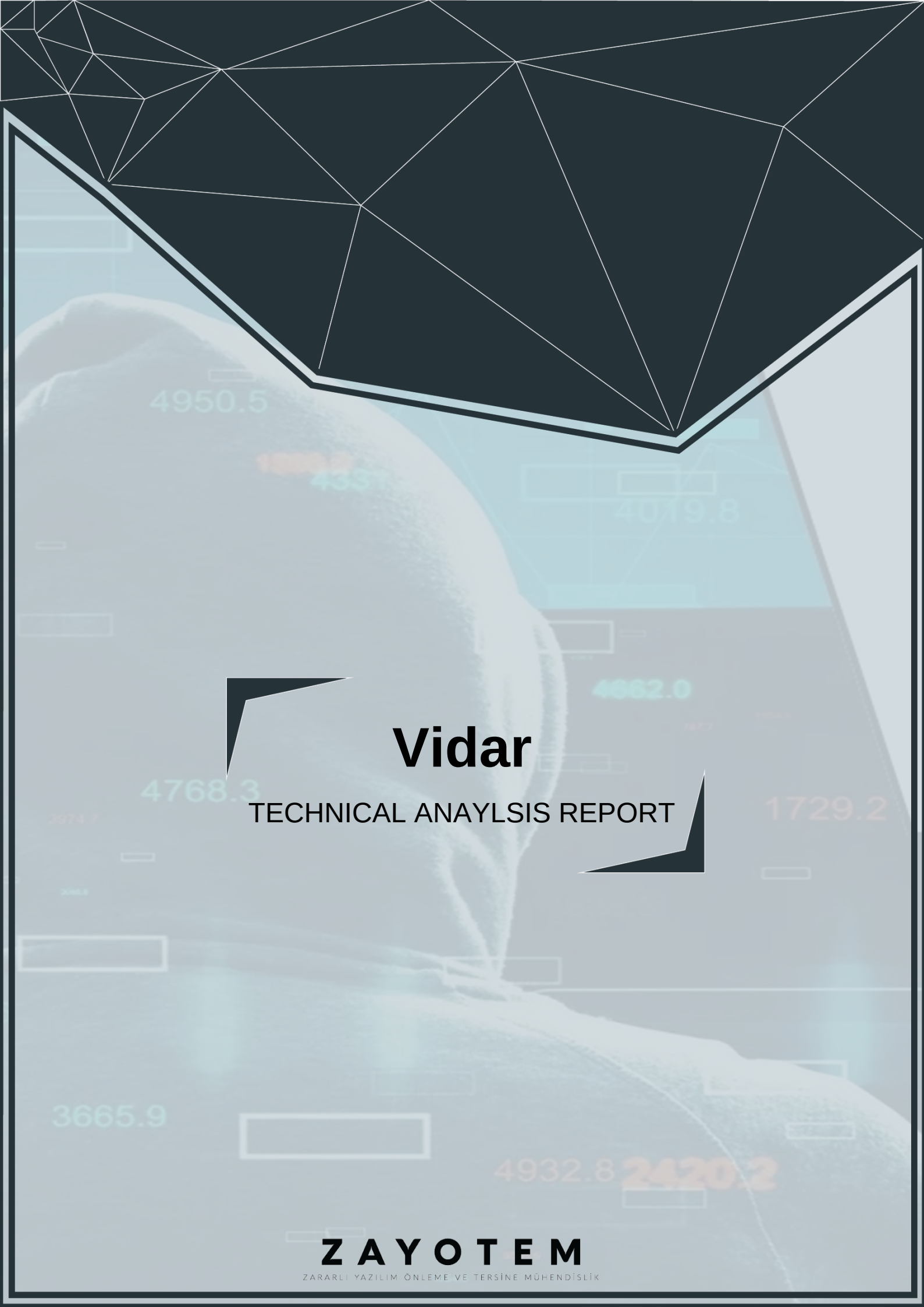


Vidar

TECHNICAL ANAYLSIS REPORT

ZAYOTEM

ZARARLI YAZILIM ÖNLEME VE TERSİNE MÜHENDİSLİK

Contents

Overview

Vidar is an information theft software first discovered in late 2018. It targets and runs on the Windows operating system. It collects various sensitive data from browsers and digital wallets. Vidar can also be used as a downloader to download ransomware.

Vidar malware is spread via email attachments or ISO files. It is distributed as imitations of popular and legitimate software, such as Adobe Photoshop and Microsoft Teams, disguised in fake installers. Vidar is also delivered to target systems through attacks using the fallout exploit kit or phishing emails. These methods are used to mislead users into downloading and running the malicious ISO file.

Vidar performs information theft and often uses social media accounts as part of its command and control server (**C2**). The IP address of Vidar's **C2** infrastructure is often embedded in user profiles on popular platforms such as steam or telegram in order to hide the attackers' trail. This method allows the malware to establish command and control connections through these platforms. Vidar can access these profiles, communicate with the specified IP address and download configuration files, instructions and other malware.

This malware may infect computers that are infected;

- Collect system information,
- Stealing browser data,
- It intercepts file data,
- Makes SQL queries,
- Steals application data,
- In addition, it downloads malicious software,
- Takes a screenshot.

File.exe Analysis

Adı	File.exe
MD5	834EA699F82AA32660CB329A96986165
SHA256	ac5be0e12802839366243997af6620e86ae4540a9bd888e1ac140323400095c1
Dosya Türü	PE32 / EXE

Static Analysis

File Type	Portable Executable 32
File Info	Microsoft Visual C++ 8
File Size	422.00 KB (432128 bytes)
PE Size	422.00 KB (432128 bytes)

Figure 1- Obtaining file information

The malware is a 32-bit executable and is written in Microsoft Visual C++ 8. The software is 422.00 KB in size.

```
DWORD sub_40647B()
{
  char *v0; // esi
  HANDLE Thread; // eax

  v0 = (char *)VirtualAlloc(0, 0x4ACu, 0x1000u, 0x40u);
  sub_404D0D(&unk_468040, 1196);
  memmove(v0, &unk_468040, 0x4ACu);
  sub_404CF3();
  Thread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)(v0 + 392), &unk_436040, 0, 0);
  return WaitForSingleObject(Thread, 0xFFFFFFFF);
}
```

Figure 2- Formation and invocation of the area separation thread

As a result of the static analysis, a memory space is allocated with the **VirtualAlloc** API. Then, a thread is executed in this allocated memory space using the **CreateThread** API. The **WaitForSingleObject** API is used to wait for the created thread object to finish its work.

.

Dynamic Analysis

```
00063CCB    8BC1            mov eax,ecx                                  bellek üzer
00063CCD    884C0C 5C       mov byte ptr ss:[esp+ecx+5C],cl
00063CD1    99              cdq
00063CD2    F7BC24 70020000 idiv dword ptr ss:[esp+270]
00063CD9    8A0432          mov al,byte ptr ds:[edx+esi]
00063CDC    88840C 5C010000 mov byte ptr ss:[esp+ecx+15C],al
00063CE3    41              inc ecx
00063CE4    3BCB            cmp ecx,ebx
00063CE6  ^ 7C E3           jl ac5be0e12802839366243997af6620e86ae4!     döngü sonu
00063CE8    33F6            xor esi,esi
```

Figure 3- Memory decrypt function

Decryption is performed for complex strings in a 100 byte area within the loop.

```
00F93D97    59              pop ecx
00F93D98    894424 18       mov dword ptr ss:[esp+18],eax
00F93D9C    68 98B3FB00     push ac5be0e12802839366243997af6620e86a
00F93DA1    8D4C24 30       lea ecx,dword ptr ss:[esp+30]
00F93DA5    897424 2C       mov dword ptr ss:[esp+2C],esi
00F93DA9    E8 84F6FFFF     call ac5be0e12802839366243997af6620e86a
00F93DAE    8D4424 28       lea eax,dword ptr ss:[esp+28]
00F93DB2    50              push eax
00F93DB3    8D4424 24       lea eax,dword ptr ss:[esp+24]
00F93DB7    50              push eax
00F93DB8    8D4C24 18       lea ecx,dword ptr ss:[esp+18]
00F93DBC    E8 F6D9FFFF     call ac5be0e12802839366243997af6620e86a
00F93DC1    8D4C24 2C       lea ecx,dword ptr ss:[esp+2C]
00F93DC5    E8 150D0000     call ac5be0e12802839366243997af6620e86a
00F93DCA    46              inc esi
00F93DCB    83FE 0A         cmp esi,A
00F93DCE  ^ 7C CC           jl ac5be0e12802839366243997af6620e86ae4
00F93DD0    8B4424 18       mov eax,dword ptr ss:[esp+18]
00F93DD4    8D4C24 44       lea ecx,dword ptr ss:[esp+44]
00F93DD8    8A4404 5C       mov al,byte ptr ss:[esp+eax+5C]
00F93DDC    30042F          xor byte ptr ds:[edi+ebp],al
00F93DDF    E8 210D0000     call ac5be0e12802839366243997af6620e86a
00F93DE4    8B5424 1C       mov edx,dword ptr ss:[esp+1C]
00F93DE8    47              inc edi
00F93DE9    3BBC24 68020000 cmp edi,dword ptr ss:[esp+268]
00F93DF0  ^ 0F8C 3DFFFFFF  jl ac5be0e12802839366243997af6620e86ae4
00F93DF6    8D4424 10       lea eax,dword ptr ss:[esp+10]
```

Figure 4- Copy of section parts

The .data section containing the malicious codes that it will inject into the thread it will create is copied to the .data section of the running process.

```
012364B3    55              push ebp
012364B4    55              push ebp
012364B5    68 40602601     push ac5be0e12802839366243997af6620e86ae
012364BA    8D86 88010000   lea eax,dword ptr ds:[esi+188]
012364C0    50              push eax
012364C1    55              push ebp
012364C2    55              push ebp
012364C3    FF15 0CB02501   call dword ptr ds:[<&CreateThread>]
012364C9    6A FF           push FFFFFFFF
012364CB    50              push eax
012364CC    FF15 08B02501   call dword ptr ds:[<&WaitForSingleObject
012364D2    5F              pop edi
012364D3    5E              pop esi
012364D4    5D              pop ebp
012364D5    5B              pop ebx
012364D6    C3              ret
```

Figure 5- Creation with CreateThread function

This section shows that the malware creates a thread at **0x012364C3** with the **CreateThread** API. Then this thread is executed. Finally, it waits for the thread to finish its work with the **WaitForSingleObject** API. After this stage, the analysis continues by examining the memory space allocated for the thread.

```
000201C3    89DD            mov ebp,ebx
000201C5    8B34AF          mov esi,dword ptr ds:[edi+ebp*4]
000201C8    01C6            add esi,eax                     esi:"LoadLibraryA"
000201CA    45              inc ebp
000201CB    813E 4C6F6164   cmp dword ptr ds:[esi],64616F4C esi:"LoadLibraryA"
000201D1  ^ 75 F2           jne 201C5
000201D3    817E 08 61727941 cmp dword ptr ds:[esi+8],41797261 esi+8:"aryA"
000201DA  ^ 75 E9           jne 201C5
000201DC    8B7A 24         mov edi,dword ptr ds:[edx+24]
```

Figure 6- Obtaining LoadLibray Function name with API Hashing

It was observed that the **LoadLibraryA** API, which was previously taken as a hexadecimal value, was split into two and stored separately in memory to make it difficult to detect, and was combined and used at runtime.

```
00020209    8B34AF          mov esi,dword ptr ds:[edi+ebp*4]
0002020C    01C6            add esi,eax                     esi:"GetProcAddress"
0002020E    45              inc ebp
0002020F    813E 47657450   cmp dword ptr ds:[esi],50746547 esi:"GetProcAddress"
00020215  ^ 75 F2           jne 20209
00020217    817E 0A 72657373 cmp dword ptr ds:[esi+A],73736572 esi+A:"ress"
0002021E  ^ 75 E9           jne 20209
00020220    8B7A 24         mov edi,dword ptr ds:[edx+24]
```

Figure 7- Obtaining the function name GetProcAddress with API Hashing

It was found that the **GetProcAddress** API, which was previously received in hexadecimal format, was divided into two parts and stored separately in memory in order to make it difficult to detect, and these parts were combined and used during operation.

```
00020241    57              push edi
00020242    56              push esi
00020243    FFD0            call eax
00020245    8985 50010000   mov dword ptr ss:[ebp+150],eax
0002024B    8B75 08         mov esi,dword ptr ss:[ebp+8]
0002024E    8B45 04         mov eax,dword ptr ss:[ebp+4]
00020251    8D7D 26         lea edi,dword ptr ss:[ebp+26]
00020254    57              push edi
00020255    56              push esi
00020256    FFD0            call eax
00020258    8985 54010000   mov dword ptr ss:[ebp+154],eax
0002025E    8B75 08         mov esi,dword ptr ss:[ebp+8]
00020261    8B45 04         mov eax,dword ptr ss:[ebp+4]
00020264    8D7D 33         lea edi,dword ptr ss:[ebp+33]
00020267    57              push edi
00020268    56              push esi
00020269    FFD0            call eax
0002026B    8985 58010000   mov dword ptr ss:[ebp+158],eax
00020271    8B75 08         mov esi,dword ptr ss:[ebp+8]
00020274    8B45 04         mov eax,dword ptr ss:[ebp+4]
00020277    8D7D 44         lea edi,dword ptr ss:[ebp+44]
```

Figure 8- Dynamic API Resolving

In the API Hashing section, it is seen that it obtains the handle addresses of the functions it will use later by using the functions it analyzes and these addresses are written in certain fields.

```
Address   Hex                                              ASCII
012F8040  55 05 00 00 37 13 00 00 00 00 00 00 75 73 65 72  U...7.......user
012F8050  33 32 2E 64 6C 6C 00 43 72 65 61 74 65 50 72 6F  32.dll.CreatePro
012F8060  63 65 73 73 41 00 56 69 72 74 75 61 6C 41 6C 6C  cessA.VirtualAll
012F8070  6F 63 00 47 65 74 54 68 72 65 61 64 43 6F 6E 74  oc.GetThreadCont
012F8080  65 78 74 00 52 65 61 64 50 72 6F 63 65 73 73 4D  ext.ReadProcessM
012F8090  65 6D 6F 72 79 00 56 69 72 74 75 61 6C 41 6C 6C  emory.VirtualAll
012F80A0  6F 63 45 78 00 57 72 69 74 65 50 72 6F 63 65 73  ocEx.WriteProces
012F80B0  73 4D 65 6D 6F 72 79 00 53 65 74 54 68 72 65 61  sMemory.SetThrea
012F80C0  64 43 6F 6E 74 65 78 74 00 52 65 73 75 6D 65 54  dContext.ResumeT
012F80D0  68 72 65 61 64 00 39 05 00 00 BC 04 00 00 00 00  hread.9...¼.....
012F80E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 9- Names of the decrypted API functions

During the calls of the functions received in the handle, the malicious file performs the following operations respectively:

- Firstly, the regasm.exe file is run in suspend mode with the **CreateProcessA** API.
- Then, a 4 byte area is allocated with the **VirtualAlloc** API and a predetermined value is written to this allocated area.
- The context part of the newly opened regasm.exe file is written to this opened area with the **GetThreadContext** API.
- After these steps, the **WriteProcessMemory** API reads the PEB part of regasm.exe and the **VirtualAllocEx** API frees 246,000 bytes of space at address **0x400000** inside regasm.exe.

- Finally, code from the .data section of the malware is injected into the field at address **0x400000** using the **WriteProcessMemory** API

This is accomplished by the **Process Hollowing** technique. This technique is used by malware and is based on the insertion of malicious code when a legitimate application is launched, by replacing the memory space of that process. In this context, the attacker starts a process and then replaces the original code with malicious code in the memory of that process

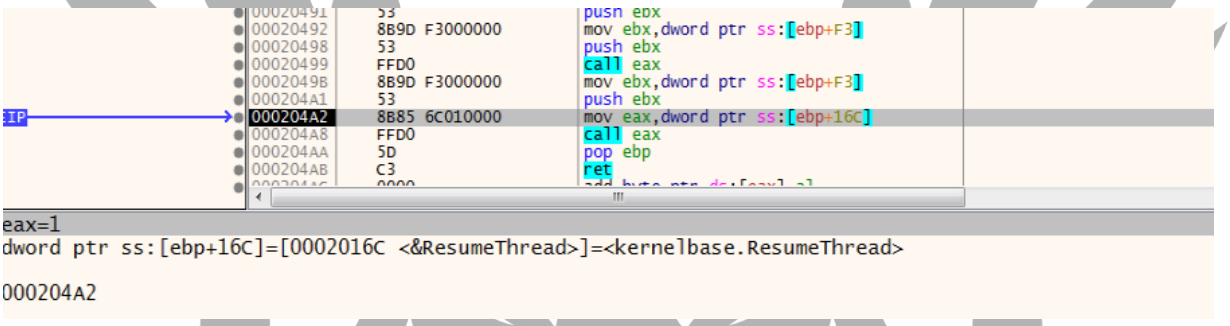


Figure 10- ResumeThread function to keep the Thread part running

In this section, it is seen that the previously suspended thread object continues to run. After these operations, we proceed to the RegAsm.exe analysis section.

RegAsm.exe Analysis

Adı	Regasm.exe
MD5	db8f071d389c007289e2b3ef2112e465
SHA256	4d1b17586f1382c603449966bce52c59c32dd568cf142f1ceaca8f21231e9c3e
Dosya Türü	PE32/EXE

Dynamic Analysis

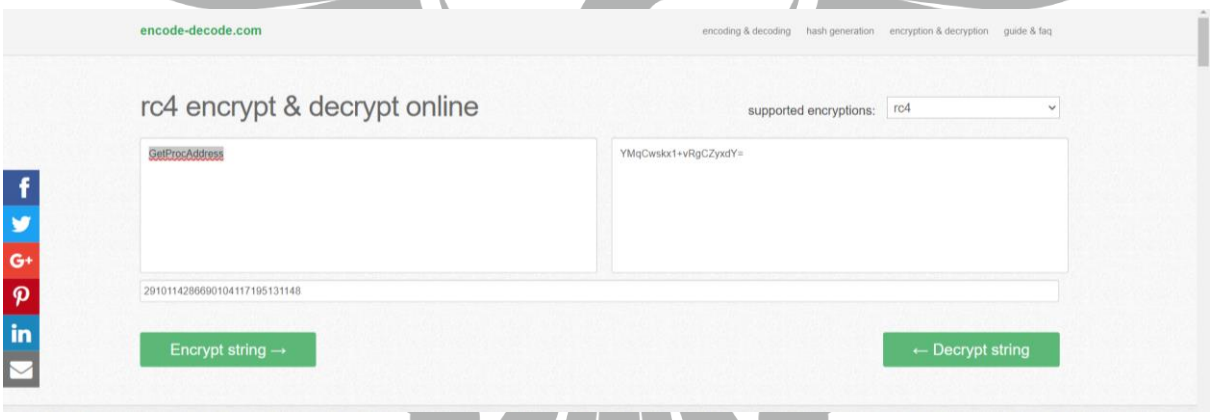


Figure 11- RC4 decrypting encrypted string

The strings encrypted with **RC4** algorithm are decrypted with the key **291011428669010411719513148** in order not to be visible in static analysis.

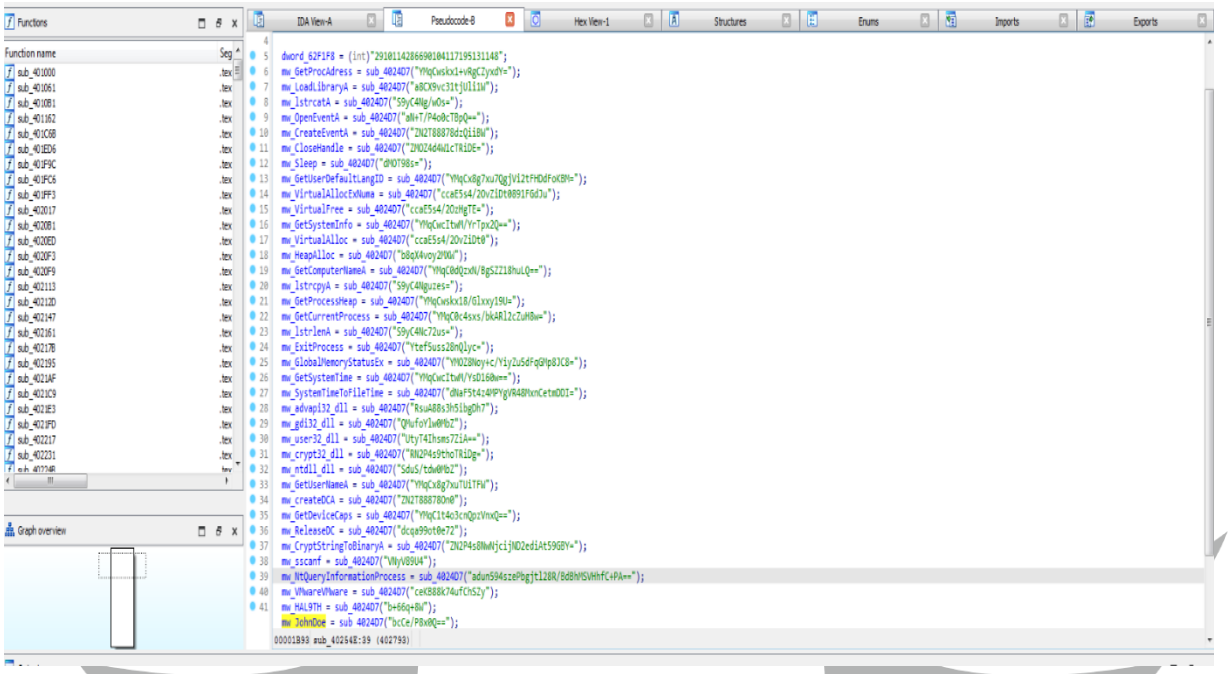


Figure 12- Decypted strings and their equivalents

The values and equivalents of the analysed strings are shown.

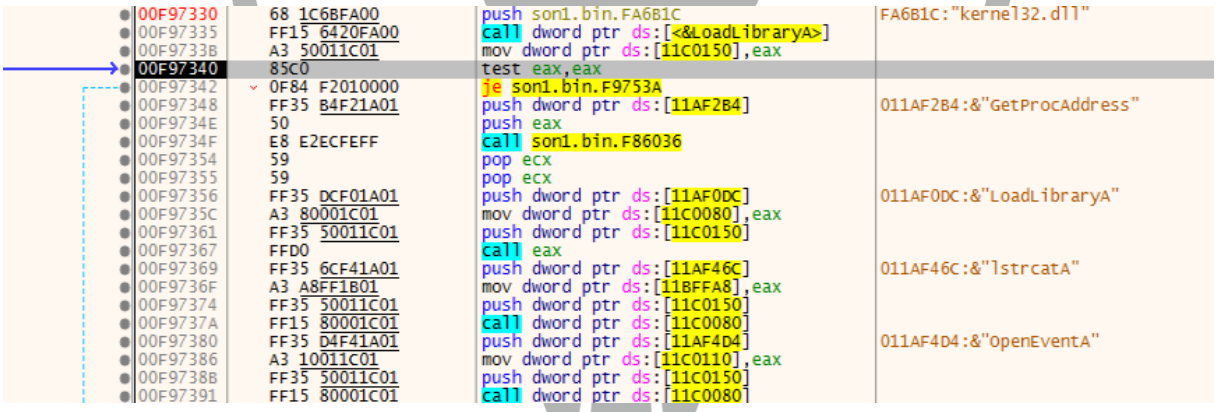


Figure 13- Dynamic API Resolving

The address values of the API names in the parsed strings are obtained with the dynamic API parsing technique by using the **LoadLibraryA** and **GetProcAddress** APIs.

.

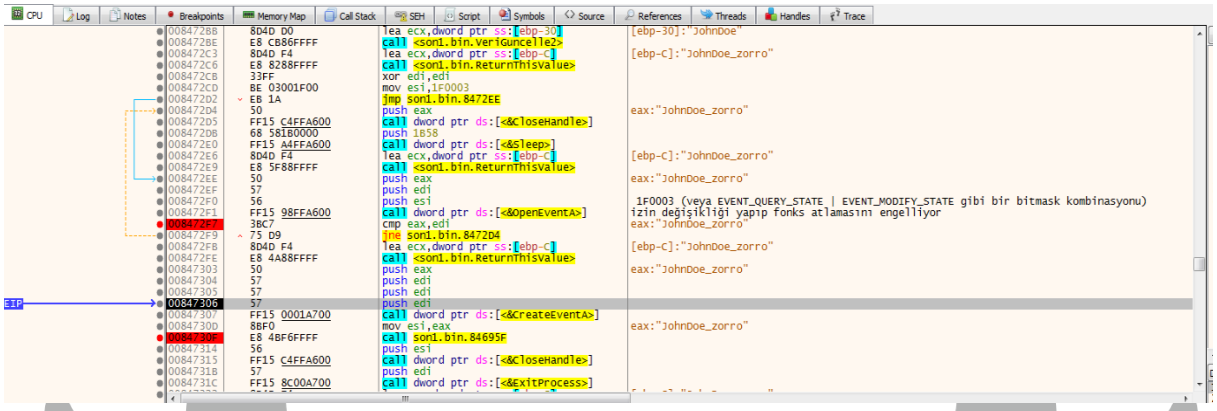


Figure 14- Control of Sandbox

After getting the username of the computer, it is concatenated with the word "**JohnDoe**" with the character "_" in between. The resulting name checks the **OpenEventA** API for the existence of an event with the name "JohnDoe_{username}". If there is an event with this name, it redirects to the **Sleep** API.

```
.text:000527EF
.text:000527EF sub_527EF proc near
.text:000527EF
.text:000527EF pszString= dword ptr -4
.text:000527EF
.text:000527EF push    offset aYmqc19uo3djaij ; "YMqC19Uo3djaijly2NFdDc1mYTVEg9Q="
.text:000527F4 call    sub_524D7
.text:000527F9 mov     nw_GetEnvironmentVariableA, eax
.text:000527FE mov     [esp+4+pszString], offset aYmqc1niy0evbkc ; "YMqC1NIy0evBkCZ+1NB/CcxO"
.text:00052805 call    sub_524D7
.text:0005280A mov     nw_GetFileAttributesA, eax
.text:0005280F mov     [esp+4+pszString], offset aYmoz8noyMxwjw ; "YMOZ8Noy+MXWjw=="
.text:00052816 call    sub_524D7
.text:0005281B mov     nw_GlobalLock, eax
.text:00052820 mov     [esp+4+pszString], offset aB8qx4v0s0c8 ; "b8qX4v0s0c8="
.text:00052827 call    sub_524D7
.text:0005282C mov     nw_HeapFree, eax
.text:00052831 mov     [esp+4+pszString], offset aYmqc1niy0fncnj ; "YMqC1NIy0fncnjE="
.text:00052838 call    sub_524D7
.text:0005283D mov     nw_GetFileSize, eax
.text:00052842 mov     [esp+4+pszString], offset aYmoz8noy58ppgq ; "YMOZ8Noy58PPgQ=="
.text:00052849 call    sub_524D7
.text:0005284E mov     nw_GlobalSize, eax
.text:00052853 mov     [esp+4+pszString], offset aZn2t88874mxaid ; "ZN2T88874MXaiDxy2tU4XuxhYSdbjvq5"
.text:0005285A call    sub_524D7
.text:0005285F mov     nw_CreateToolhelp32Snapshot, eax
.text:00052864 mov     [esp+4+pszString], offset aBtyhCxogprhizd ; "btyh/cxogPrHizdyxdY="
.text:0005286B call    sub_524D7
.text:00052870 mov     nw_IsWow64Process, eax
.text:00052875 mov     [esp+4+pszString], offset aD92z8d4tx5mhqj ; "d92Z8d4tx5mHqjFvwg=="
.text:0005287C call    sub_524D7
.text:00052881 mov     nw_Process32Next, eax
.text:00052886 mov     [esp+4+pszString], offset aYmqc3tq91cbhjt ; "YMqC3tQ91cbhjTly"
.text:0005288D call    sub_524D7
.text:00052892 mov     nw_GetLocalTime, eax
```

Figure 15- String decryption

In the sub_527EF function, the decrypt operation of previously encrypted words is performed.

Figure 16- Connection to Steam profile

It sends a **GET** request to the steam profile address registered in the .rdata section. With the **InternetReadFile** API call, it retrieves the HTML codes of the steam profile in the link.

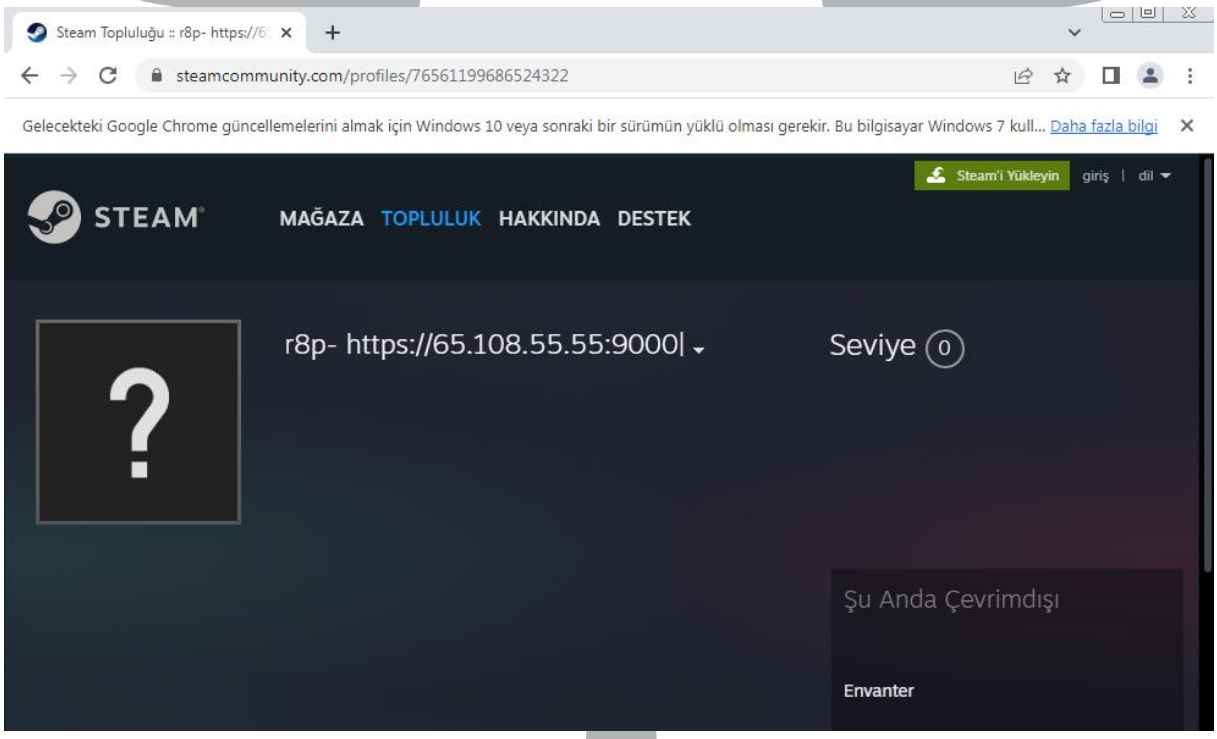


Figure 17- Steam profile

The Steam profile shows a link in the username field. The characters "**r8p-**" and "**|**" are used to locate IP and port addresses in the HTML code. Sending a connection request to **https[:]//65.108.55.55** on port **9000** returns **RST, ACK**.

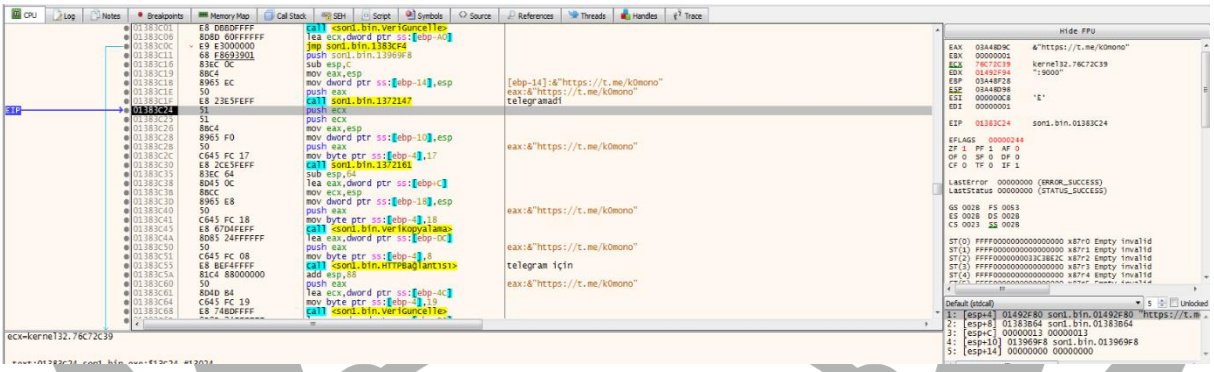


Figure 18- Connection to Telegram

If the server from the Steam profile cannot be accessed, it tries to do the same via telegram. As with the Steam profile, it pulls HTML code using a **GET** request. It looks for an address between the characters "**r8p-**" and "**|**".

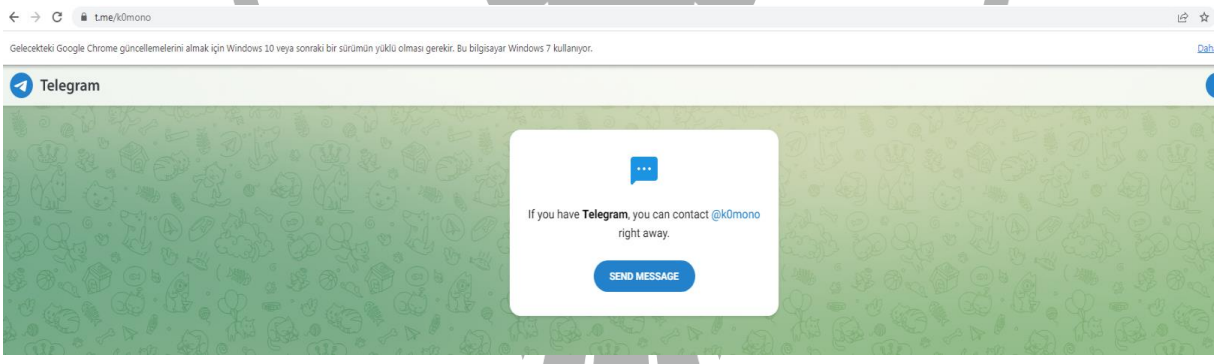
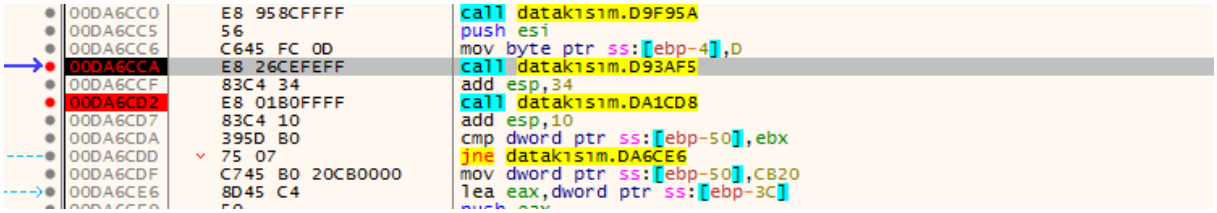


Figure 19- Telegram address

As with Steam, it looks for an address between the characters "**r8p-**" and "**|**". However, when the telegram channel is accessed, there is no indication. Therefore, there is no data retrieval via telegram.



Figre 18- Function that sends a POST request

The function located at **0x00DA6CCA** sends hwid and build_id values in multipart/form-data format. It returns a response to the server's incoming **POST** request. The response is in "*|*|*|*|*" format and is saved if the * fields are filled in the function at address **0x000A6CD2**. Since the**C2** server cannot be reached, the "*" parts cannot be analyzed. It sends 3 more **POST** requests in the same way. First, it sets the "mode" value to 1, 2 in the 2nd **POST** request, and 21 in the 3rd **POST** request and sends it. It saves the values returned by the server in the same way.

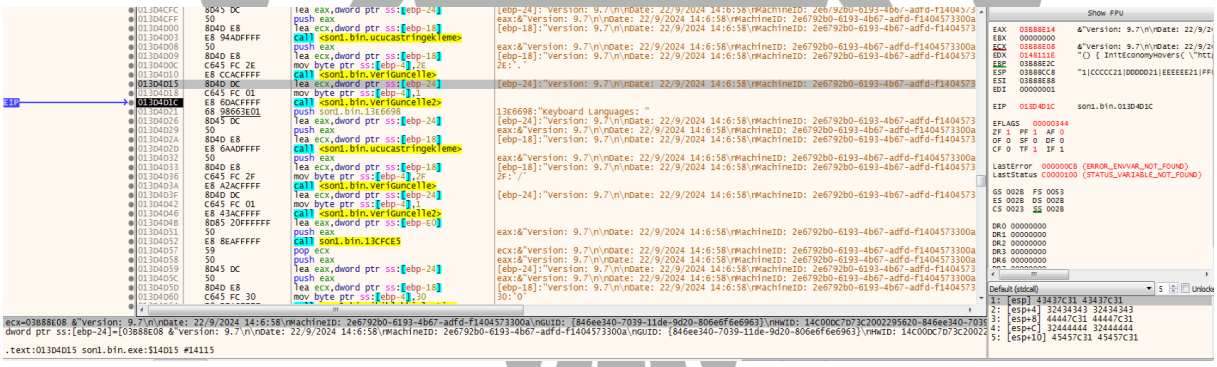


Figure 21- Obtaining some data about the computer

The malware collects the computer's system information. The collected system information is shown in table-1 below.

The data received are as follows:

Version	Work Dir: In memory	Display Resolution	Cores
Date	Windows	Keyboard Languages	Threads
MachineID	Install Date	Local Time	RAM
GUID	AV	TimeZone	VideoCard
HWID	Computer Name	[Hardware]	[Processes]
Path	User Name	Processor	[Software]

Table 1- System Informations

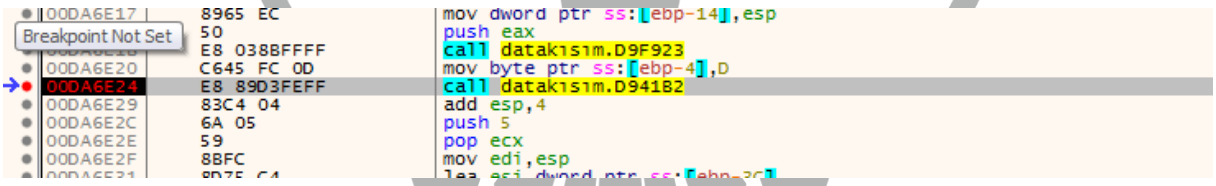


Figure 22- Function that sends GET request to sqlx.dll

The selected function sends a **GET** request to https[:]//65.108.55.55/sqlx.dll. It retrieves the **sqlite3.dll** file with the **InternetReadFile** API call. Then, it writes the code of **sqlite3.dll** in a memory space

.

```
if ( !result )
{
  result = sub_55A53();
  if ( !result )
  {
    result = sub_55B06(a1, a2, a3);
    if ( !result )
    {
      result = sub_55B8A();
      if ( !result )
      {
        result = sub_55C2E();
        if ( !result )
        {
          result = sub_55D69();
          if ( !result )
          {
            if ( (a3 & 1) != 0 || !v5 || (v9 = v7 + v5, ((int (__stdcall *)(int, int, _DWORD))(v7 + v5))(v7, 1, 0)) )
            {
              if ( a4 )
              {
                a4[1] = a3;
                a4[2] = v7;
                a4[3] = v8;
                a4[4] = v9;
                a4[5] = v6;
                a4[6] = 0;
                *a4 = 32;
                a4[7] = 0;
              }
              return 0;
```

Figure 23- Manual dll installation

When all of the functions in the if blocks successfully fulfil their functions, '**sqlite3.dll**' is installed manually on the memory.

```
.text:000560F6 push    nw_sqlite3_open
.text:000560FC push    esi
.text:000560FD call    sub_5601D
.text:00056102 push    nw_sqlite3_prepare_v2
.text:00056108 mov     h_sqlite3_open, eax
.text:0005610D push    esi
.text:0005610E call    sub_5601D
.text:00056113 push    nw_sqlite3_step
.text:00056119 mov     h_sqlite3_prepare_v2, eax
.text:0005611E push    esi
.text:0005611F call    sub_5601D
.text:00056124 push    nw_sqlite3_column_text
.text:0005612A mov     h_sqlite3_step, eax
.text:0005612F push    esi
.text:00056130 call    sub_5601D
.text:00056135 push    nw_sqlite3_finalize
.text:0005613B mov     h_sqlite3_column_text, eax
.text:00056140 push    esi
.text:00056141 call    sub_5601D
.text:00056146 push    nw_sqlite3_close
.text:0005614C mov     h_sqlite3_finalize, eax
.text:00056151 push    esi
.text:00056152 call    sub_5601D
.text:00056157 push    nw_sqlite3_column_bytes
.text:0005615D mov     h_sqlite3_close, eax
.text:00056162 push    esi
.text:00056163 call    sub_5601D
.text:00056168 push    nw_sqlite3_column_blob
.text:0005616E mov     h_sqlite3_column_bytes, eax
.text:00056173 push    esi
.text:00056174 call    sub_5601D
.text:00056179 mov     h_sqlite3_column_blob, eax
.text:0005617E xor     eax, eax
.text:00056180 add     esp, 40h
.text:00056183 inc     eax
.text:00056184 jmp     short loc_56188
```

Figure 24- sqlite3.dll calls

After Sqlite3.dll installation, it receives the addresses of the calls it will use.

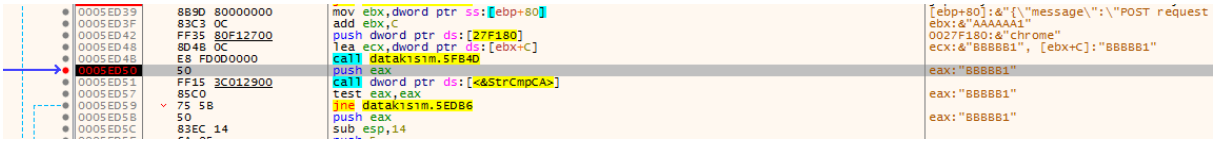


Figure 25- Browser control

The mode-valued **POST** requests mentioned above start to be used here. It is the 3rd part of the 2nd **POST** request with the value “BBBBB1”. The “BBBBB1” value is compared with the words chrome, opera and firefox. After the matching browser is detected, we switch to the function where the information of the matching browser is stolen.

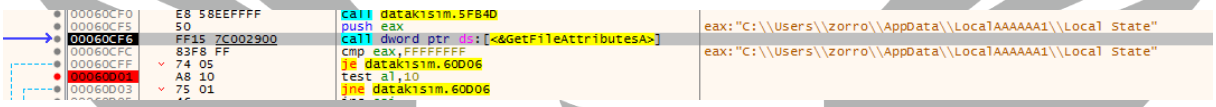


Figure 26- Browser file location

In functions where the information of the scanners is retrieved, the file locations are complemented with the responses of the **POST** requests sent earlier. The value “AAAAAA1” is the 2nd part of the 2nd **POST** request.

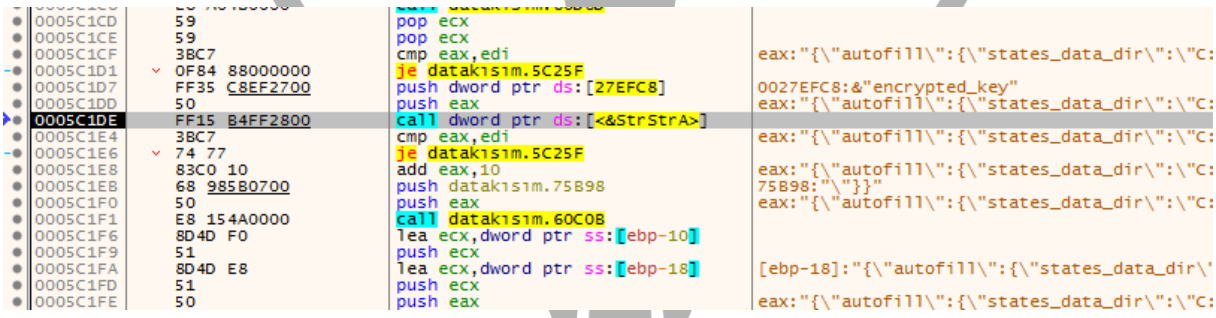


Figure 27- Local State file content

Gets the **encrypted_key** value in the Local State file in Chrome and Opera browsers.

Figure 28- DPAPI security mechanism

*After decoding the **encrypted_key** value, it is checked whether the first value in it is **DPAPI**.*

DPAPI provides secure encryption and decryption of data based on the operating system. This mechanism increases data security by managing encryption keys based on user credentials or system ID. When using **APPB**, it will not be possible to decrypt data from encrypted files because **APPB** does not have an integration at the operating system level.

Figure 29- Logins Sql query

This SQL query selects the **origin_url**, **username_value** and **password_value** columns from the **logins** table in a database. This table is usually where browsers store username and password information. The result of the query lists the URL (website) to which each record belongs and the username and password used on that website.

The select query made:

SELECT origin_url, username_value, password_value FROM logins

Table 2- Retrieves the website login information registered with the sql query.

Figure 30- Cookies Sql query

This SQL query selects some information about cookies from the **cookies** table in a browser's database. **HOST_KEY** specifies the domain name (website) to which the cookie belongs, while **is_httponly and is_secure** determine whether the cookie can be accessed over HTTP or only over HTTPS. **Path** is the path where the cookie is valid, **expires_utc** is the expiration time of the cookie (converted to Unix time), name is the name of the cookie and **encrypted_value** is the encrypted value of the cookie.

The select query made:

SELECT HOST_KEY, is_httponly, path, is_secure, (expires_utc/1000000)-11644480800, name, encrypted_value from cookies

Table 3- Sql query retrieves Cookie information in the browser.

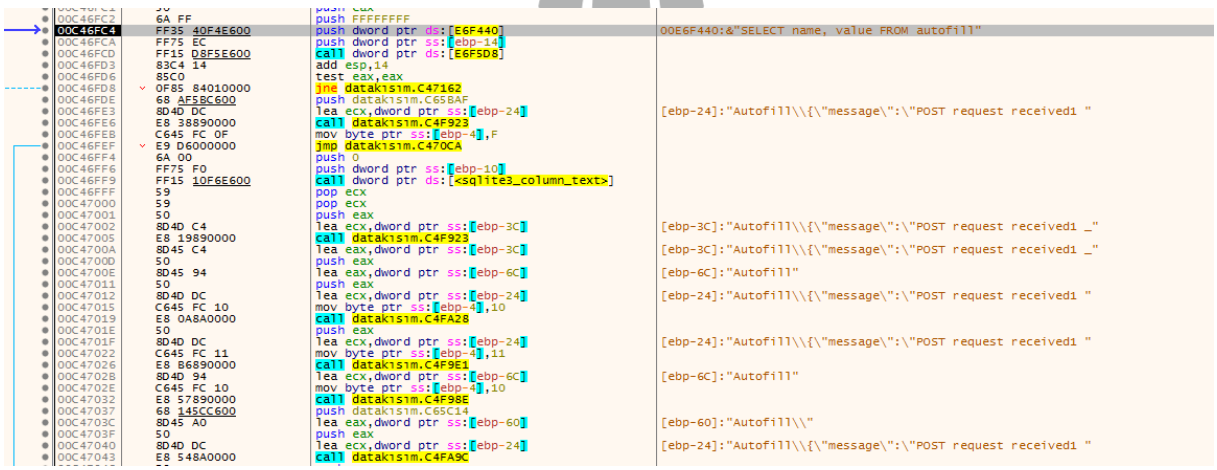


Figure 31- AutofillSql query

This SQL query selects the **name** and **value** columns from the **autofill** table in the browser's database. **Name** specifies the name of the field (e.g. name, address, phone number, etc.) saved for the autofill feature, while **value** contains the saved value corresponding to this field.

The select query made:

SELECT name, value FROM autofill

Table 4- Sql query retrieves the autofill data stored in the browser.

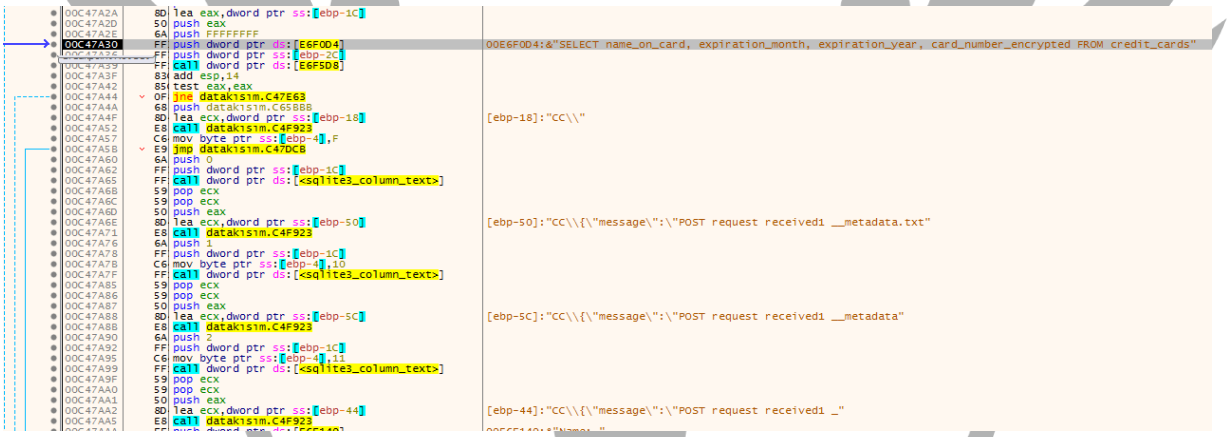


Figure 32- Credit-carts Sql query

This SQL query selects credit card information from the **credit_cards** table in the browser's database. **Name_on_card** contains the cardholder's name; **expiration_month** and **expiration_year** contain the card expiration date; **card_number_encrypted** contains the encrypted card number.

The select query made:

SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted FROM credit_cards

Table 5- Sql query retrieves the credit card information stored in the browser.

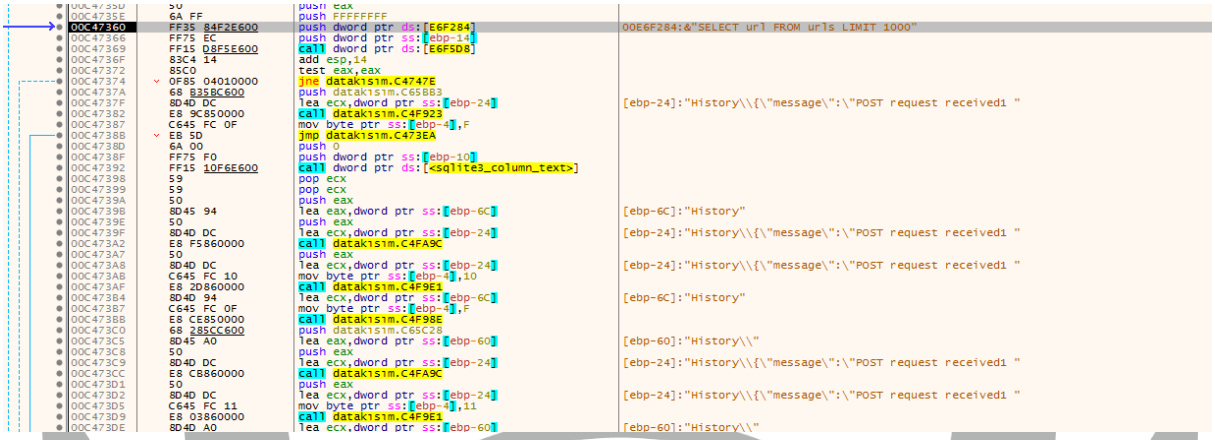


Figure 33- Urls sql query

This SQL query selects the **url** column from the **urls** table in the browser's database and returns up to 1000 **URLs**. This table contains records of the websites visited by the browser.

The select quey made:

SELECT url FROM urls LIMIT 1000

Table 6- Gets the history information of the browser with the sql query.

Figure 34- Downloads query

This SQL query selects the **target_path** and **tab_url** columns from the **downloads** table in the browser's database. **Target_path** contains the local file path (target location) where the downloaded file is saved, **tab_url** contains the URL of the page where the file is downloaded.

The select query made:.

SELECT targer_path, tab_url from downloads

Table 7- Sql query to get the names of the files downloaded from the browser.

The sql queries used are as follows:

SELECT origin_url, username_value, password_value FROM logins
SELECT HOST_KEY, is_httponly, path, is_secure, (expires_utc/1000000)-11644480800, name, encrypted_value from cookies
SELECT name, value FROM autofill
SELECT name_on_card, expiration_month, expiration_year, card_number_encrypted FROM credit_cards
SELECT url FROM urls LIMIT 1000
SELECT targer_path, tab_url from downloads

Table 8- Sql querys

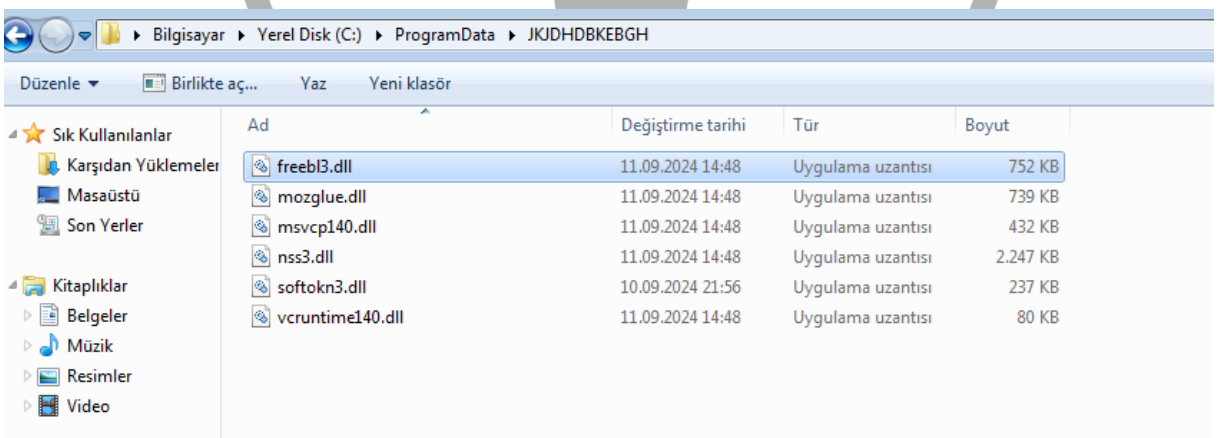


Figure 35- Downloaded dlls for Firefox

The Firefox browser needs some DLL files to encrypt information. Therefore, it fetches **nss3.dll**, **freebl3.dll**, **mozglue.dll**, **msvcp140.dll**, **softokn3.dll** and **vcruntime140.dll** from the server and saves them in a directory with a random name.

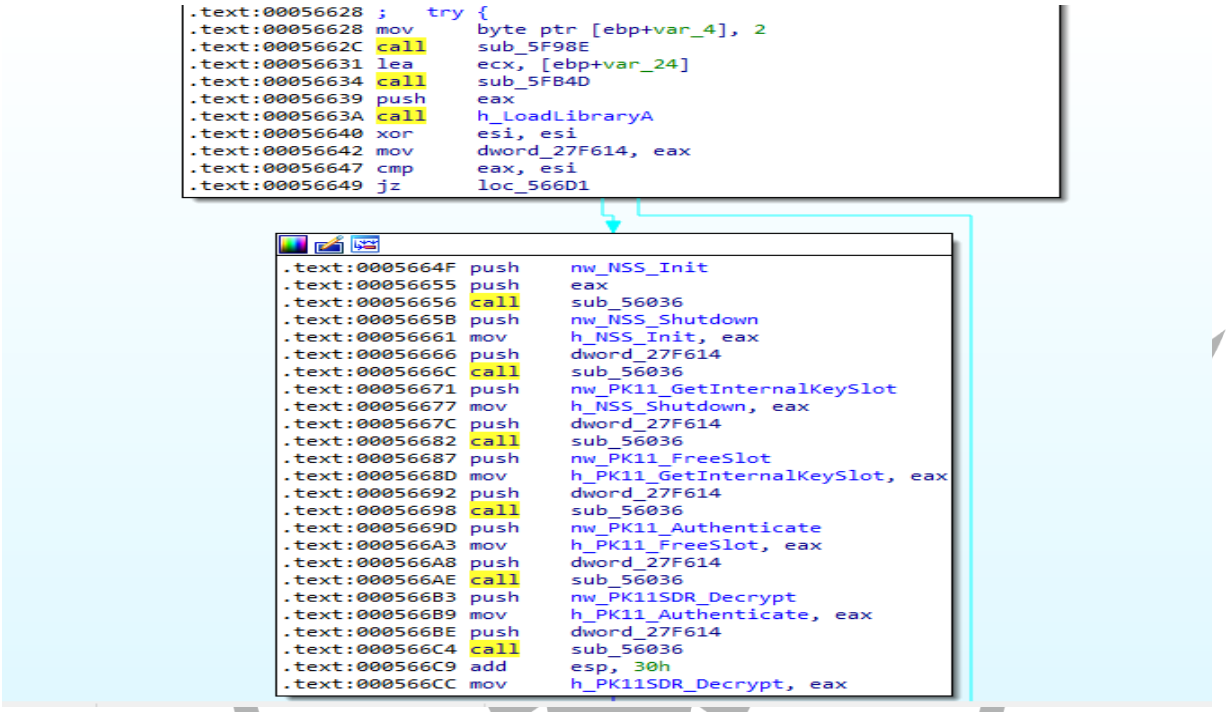


Figure 36- Sql querys

Firefox loads **nss3.dll** with the **LoadLibraryA** API call to decrypt the data. Then it saves the addresses of the calls it will use.

cookies.sqlite, places.sqlite, formhistory.sqlite *files are sent to the server without performing any operation on them.* ***Nss3.dll*** *is used to read the* ***logins.json*** *file*.

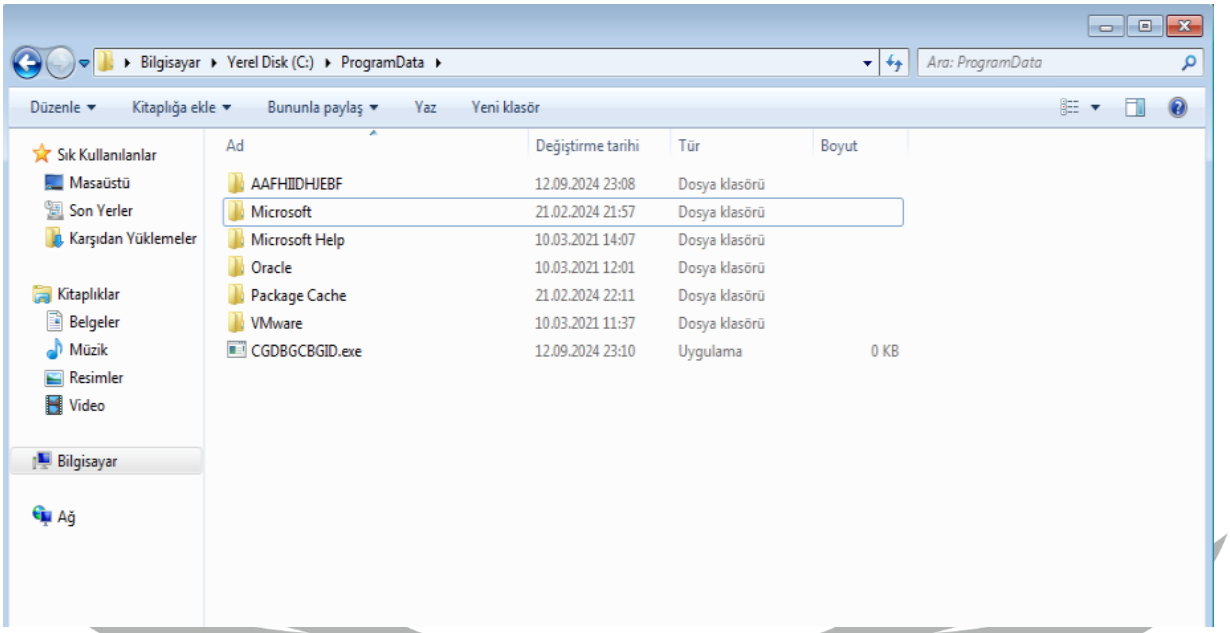


Figure 37- Directory and exe created by the malware

CreateDirectory API creates a directory with a random name in **ProgramData**. **Firefox DLL** files in figure-30 are downloaded into this directory. Also, when a file is to be read, the file is copied into this directory and read operation is performed.

The files read are as follows:

Chrome	**Opera/Opera GX**	**Firefox**
Login Data	Login Data	cookies.sqlite
Cookies	Cookies	formhistory.sqlite
Web Data	Web Data	logins.json
History	History	places.sqlite
	Local extensions	prefs.js
	sync extensions	
	indexedDB	

Table 9- Files read

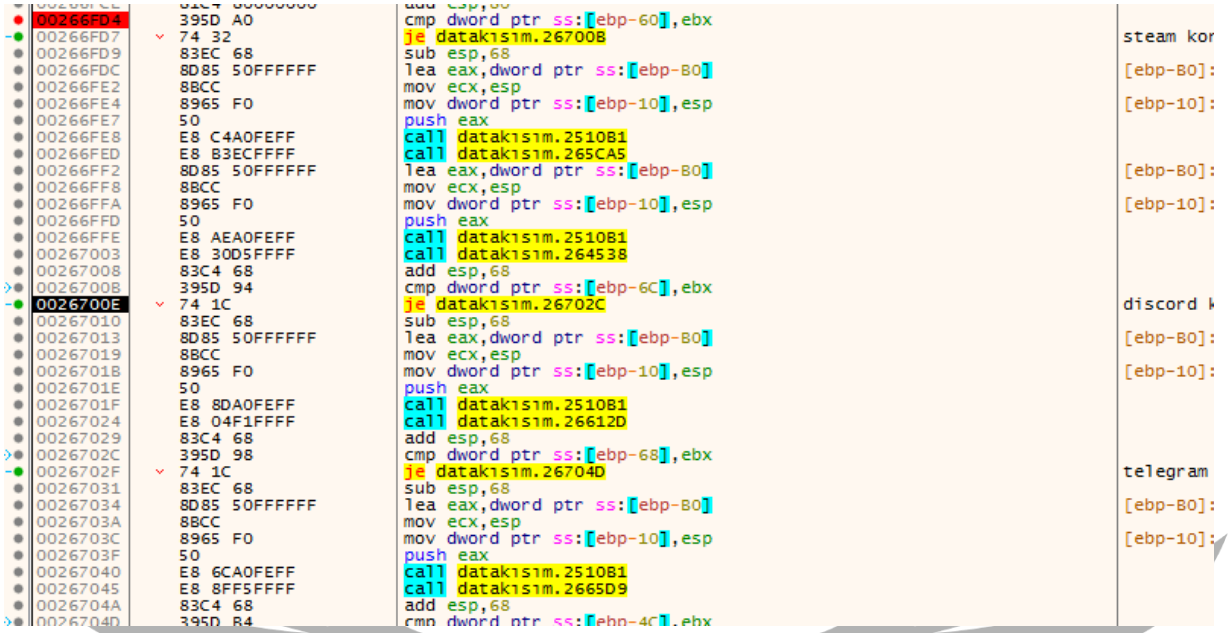


Figure 38- Steam, Discord, Telegram functions

Reads the **leveldb\CURRENT** file in the discord folder from the **AppData/Roaming** folder at**0x00267024** and sends its contents to the server.

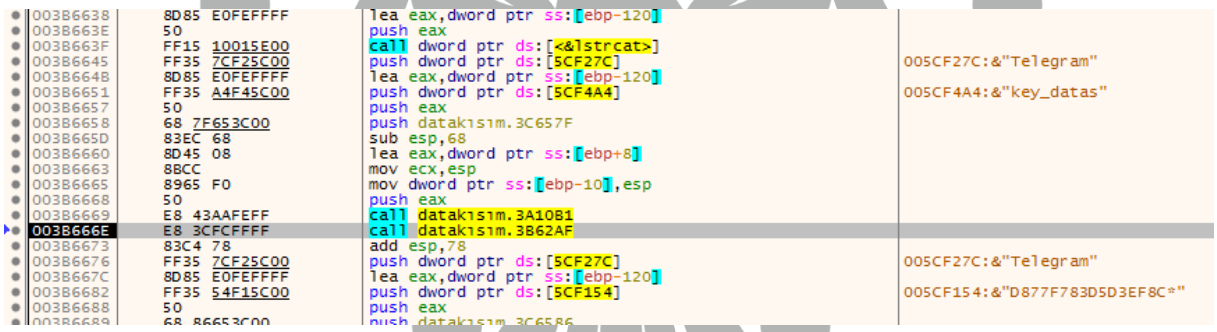


Figure 39- 2665D9 function

Function 2665D9 in Figure-33 is a function of the telegram web application;

- **key_datas,**
- **D877F783DF5D3EF8C*,**
- **map*,**
- **A7FDF864FBC10B77*,**
- **A92DAA6EA5F891F2*,**
- **F8806DD0C461824F*,**

reads the files and sends their contents to the server.

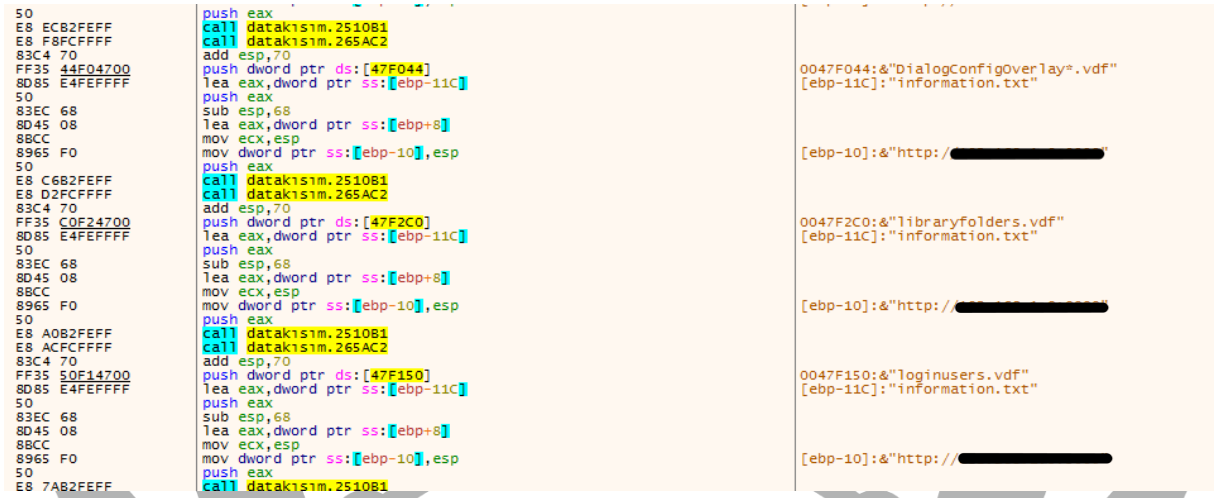


Figure 40- 265CA5 function

It enters the registry and gets the file path of steam. It reads the files starting with **ssfn** with the **ssfn*** query in the file path and sends the contents to the server. Then,

- **config.vdf,**
- **libraryfolders.vdf,**
- **loginusers.vdf,**
- **DialogConfigOverlay.vdf,**
- **DialogConfigOverlay[*].vdf,**

files are read and their contents are sent to the server.

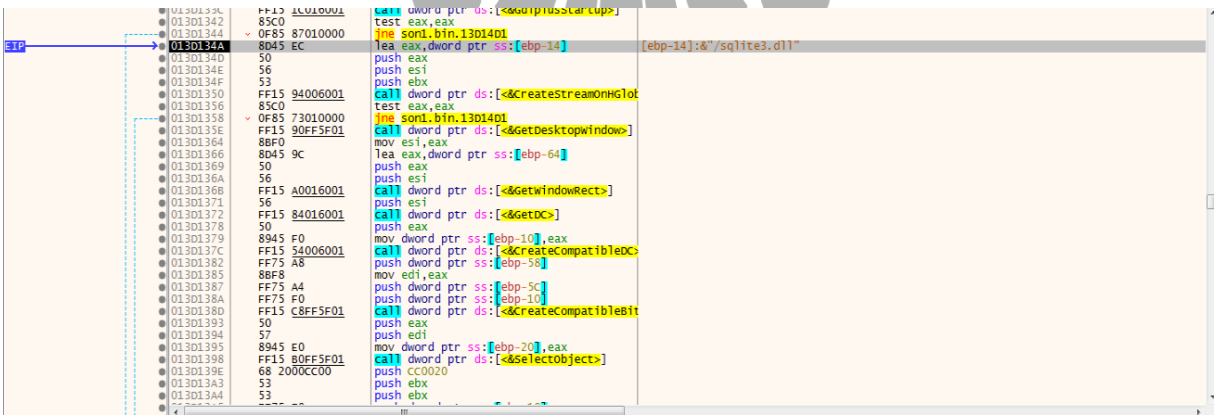


Figure 41- Screen capture function

In the 3B12FD function, **a screenshot** is taken using **GDI+** graphics functions. The screenshot is **encrypted** and sent to the server before being saved to a file

.

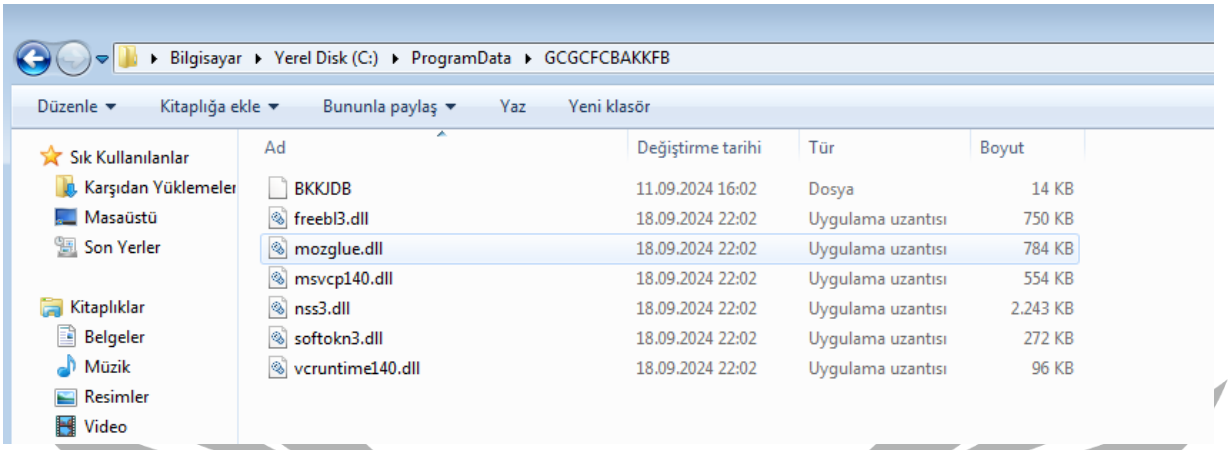


Figure 42- Directory where the files to be read are saved

The malware copies the file containing critical information into a directory with a random name. It performs a read operation on the file. After sending the read file to the **C2** server, the file is deleted.

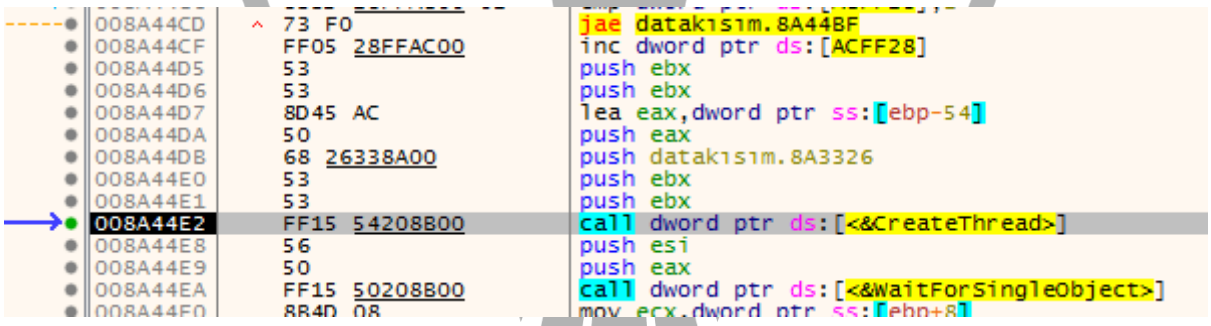


Figure 43- Thread creation

The malware starts a thread at **0x8A3326** to send the collected information to the **C2** server.

```
008A3326    B8 D40C8B00     mov eax,datakısım.8B0CD4
008A332B    E8 784D0000     call <JMP.&_EH_prolog>
008A3330    83EC 2C         sub esp,2C
008A3333    8B45 08         mov eax,dword ptr ss:[ebp+8]
008A3336    53              push ebx
008A3337    56              push esi
008A3338    57              push edi
008A3339    8D48 04         lea ecx,dword ptr ds:[eax+4]
008A333C    8965 F0         mov dword ptr ss:[ebp-10],esp
008A333F    8945 08         mov dword ptr ss:[ebp+8],eax
008A3342    E8 06C8FFFF     call datakısım.89FB4D
008A3347    50              push eax
008A3348    FF15 2000AD00   call dword ptr ds:[<&lstrlenA>]
008A334E    83F8 01         cmp eax,1
008A3351    7D 08           jge datakısım.8A335E
008A3353    FF0D 28FFAC00   dec dword ptr ds:[ACFF28]
008A3359    E9 E3000000     jmp datakısım.8A3441
008A335E    8D4D D4         lea ecx,dword ptr ss:[ebp-2C]
008A3361    E8 B0C5FFFF     call datakısım.89F916
008A3366    8365 FC 00      and dword ptr ss:[ebp-4],0
```

Figure 44- The address that the created thread executes

With the **lstrlenA** API, it checks the presence of the token value, which is one of the return values of previously sent **POST** requests. If the token value is present, it continues. The collected information kept in memory is encrypted and sent to the server in **multipart/form-data** format.

```
011B1250    55              push ebp
011B1251    8BEC            mov ebp,esp
011B1253    83EC 20         sub esp,20
011B1256    8B4D 08         mov ecx,dword ptr ss:[ebp+8]
011B1259    33C0            xor eax,eax
011B125B    8945 E0         mov dword ptr ss:[ebp-20],eax
011B125E    8945 F2         mov dword ptr ss:[ebp-E],eax
011B1261    8945 F6         mov dword ptr ss:[ebp-A],eax
011B1264    894D E8         mov dword ptr ss:[ebp-18],ecx
011B1267    8D45 E0         lea eax,dword ptr ss:[ebp-20]
011B126A    B9 14040000     mov ecx,414
011B126F    50              push eax
011B1270    C745 E4 03000000 mov dword ptr ss:[ebp-1C],3
011B1277    C745 EC 99641C01 mov dword ptr ss:[ebp-14],son1.bin.11C64
011B127E    66:894D F0      mov word ptr ss:[ebp-10],cx
011B1282    C745 FA 9A641C01 mov dword ptr ss:[ebp-6],son1.bin.11C649
011B1289    FF15 04211C01   call dword ptr ds:[<&SHFileOperation>]
011B128F    C9              leave
011B1290    C3              ret
```

Figure 45- The function that deletes the array named random

The **SHFileOperation** call deletes a directory with a random name.

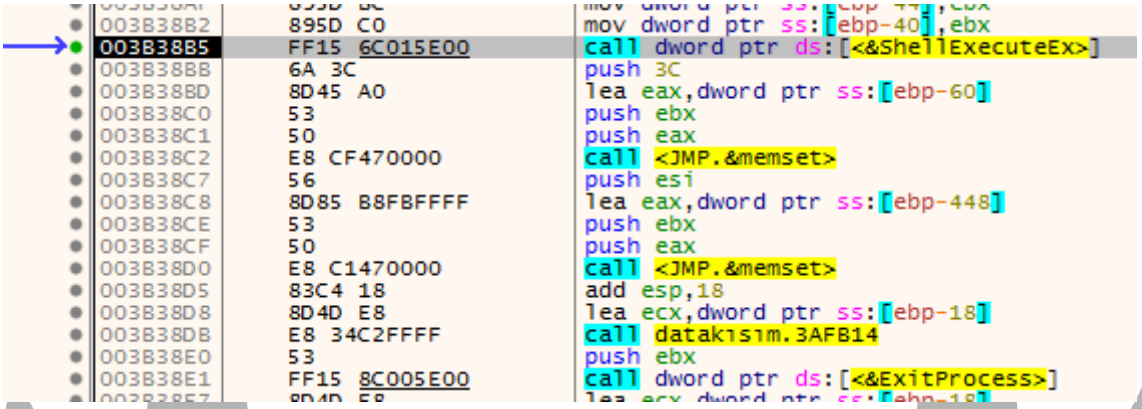


Figure 46- Self-wipe and switch off function

Malware with the **ShellExecuteEx** API

/c timeout /t 10 & del /f /q "C:\Users\Desktop**\dataKisim.exe"&rd/s/q "C:\ProgramData\HCAEBFBKKJDH" & exit**

runs the cmd command and then shuts itself down with the **ExitProcess** API.

Network Analysis

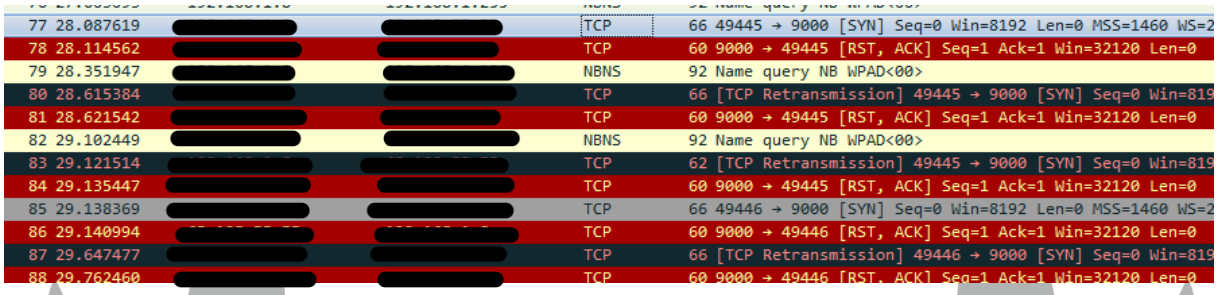


Figure 47- Request to the c2 Server

A **GET** request is sent to the IP address obtained from**https[:]//steamcommunity.com/profiles/76561199686524322**. With the status value returned as a result of this **GET** request, it is checked whether the server is active or not.

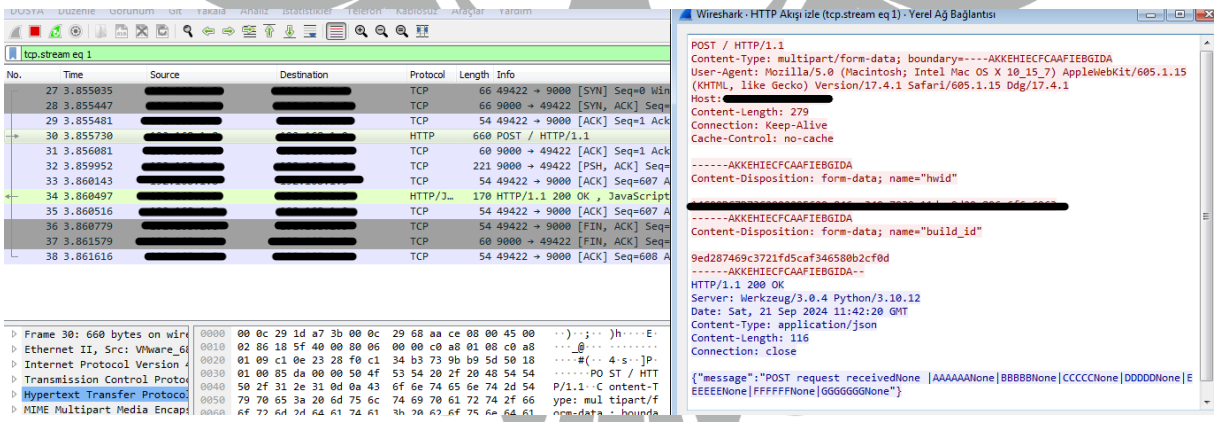


Figure 48- 1. POST request to C2 Server

It sends two parameters, **hwid** and **build_id**, to the **C2** server. The**C2** server returns certain values to incoming **POST** requests. In this **POST** request, **DDDDDNone** is the token value. It will be used as the token value in subsequent **POST** requests and will be checked for the presence of the token value before sending the information collected from the system to the **C2** server

.

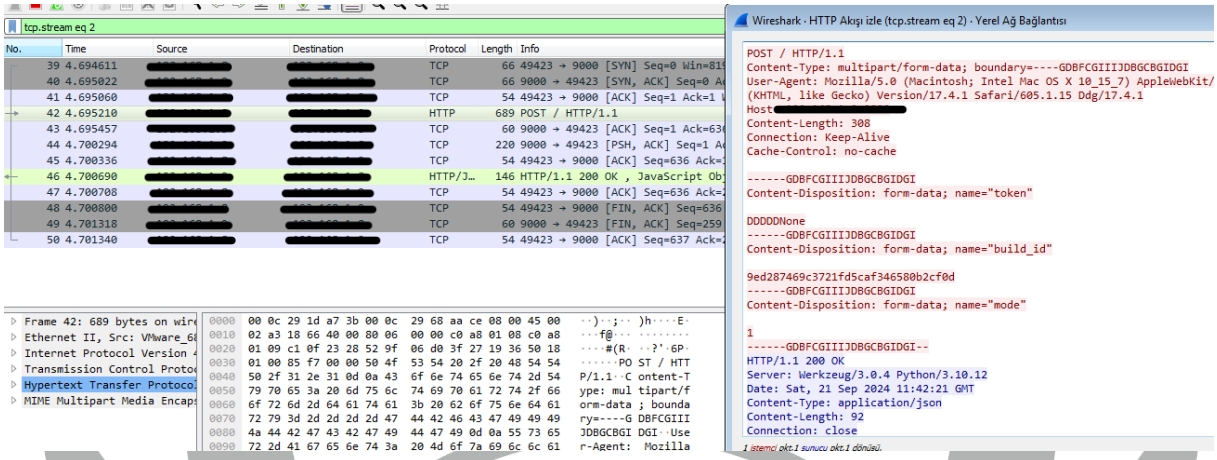


Figure 49- 2. POST request to C2 Server

A total of 4 ***POST*** *requests are sent to the* ***C2*** *server with the "mode" value set to 1, 2, 21 and 5 respectively. The return values of each request are recorded systematically*.

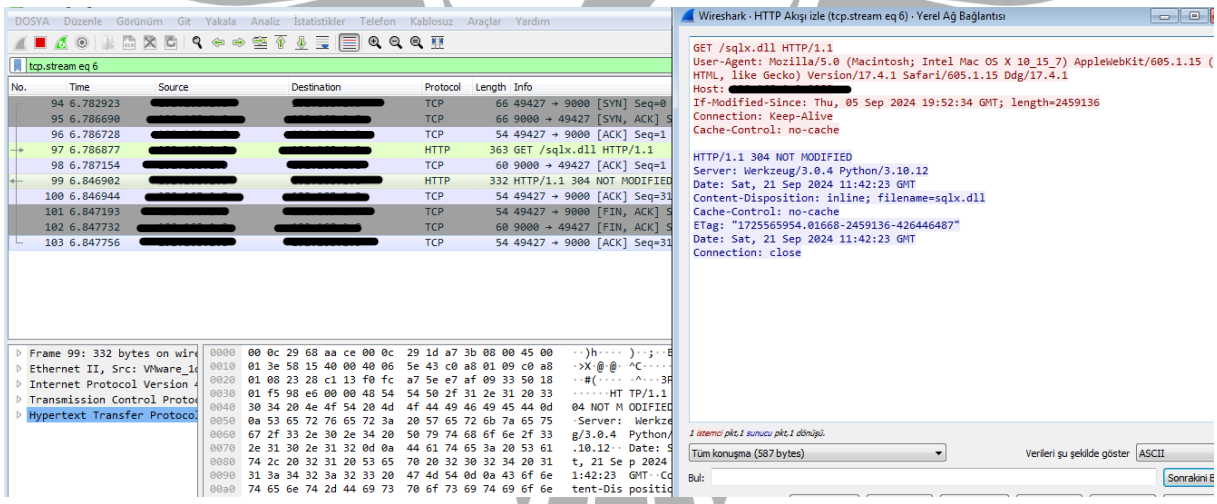


Figure 50- Sqlx.dll received from C2 Server

A **GET** request is sent to the **C2** server for a file named ****sqlx.dll****. This request downloads the necessary files from the server.

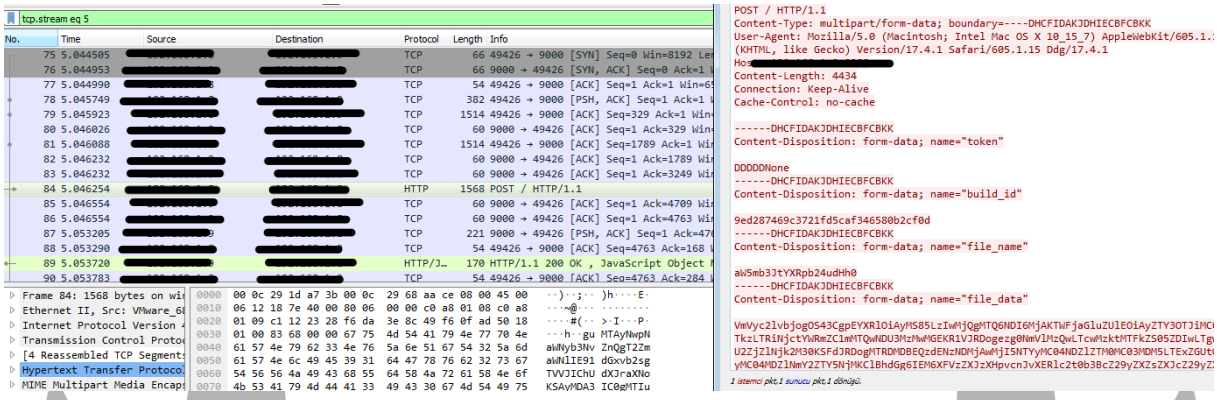


Figure 51- Information collected and sent to the C2 server

At the end of a certain function, the malware encrypts all the information it has collected in **Base64** format and sends it via the **file_data** parameter.

YARA Rule

```
import "pe"

import "math"

rule Zararli {

meta:

Author = "Zayotem Takim 4"

Date = "21.09.2024"

Description =

"ac5be0e12802839366243997af6620e86ae4540a9bd888e1ac140323400095c1.exe detection rule"

strings:

$str1 = "Madino Mino"

$str2 = "r%t^2xt="

$str3 = "ONLY NUMBERS!!!"

$str4 = "Don't TRY TO WRITE WORDS!!!"

$str5 = "root@calculator-unstable:~# "

$str6 = "Ctrl+C - Emergency stop"

$str7 = "Division:"

$str8 = "Subtraction:"

$str9 = "Sum:"

$str10 = "Multiplication:"

$str11 = "Commands\n 1 - Sum\n 2 - Multiplication Y\n 3 - Subtraction\n 4 - Division\n 5 - Help\n 6 - Close\n 7 - Factorial\n Ctrl+C - Emergency stop\n"

$data_section = { 05 91 11 9B AD B2 44 EC 67 BD 28 94 3E 69 57 52 19 48 66 AB C9 80 DE E4 B2 1B CC 91 25 40 AE 23 C7 CE 2B 17 98 AA C1 AB 5D 33 71 40 9E 31 8A B9 }

$shellcode_decrypt_func = { 8A 04 2F 34 73 2C 15 88 04 2F 8B C6 2B C2 C1 F8 02 3B 44 24 28 73 2E 89 5C 24 10 3B F1 74 0B 89 1E 83 C6 04 89 74 24 18 EB 1B 8D 44 24 10 50 56 8D 4C 24 1C E8 D2 CA FF FF 8B 4C 24 1C 8B 74 24 18 8B 54 24 14 8A 04 2F 83 C3 02 2C 57 34 74 04 4E 34 70 2C 65 34 22 2C 73 34 2A 88 04 2F 47 3B 7C 24 28 72 9B }

condition:

math.in_range(math.entropy(pe.sections[2].raw_data_offset, pe.sections[2].raw_data_size),7.8, 8.0) and $str1 and $str7 or $str2 and $str8 or $str3 and $str9 or $str4 and $str10 or $str5 and $str11 and $str6 or $data_section and $shellcode_decrypt_func

}
```

YARA Rule 2

```
import "hash"

import "pe"

import "math"



rule Zararli_regasm {

meta:

    author = "Zayotem Takim 4"

    date = "20.09.2024"

    description = "Detects regasm.exe"

strings:

    $str1 = "sqlx.dll"

    $encrypted_str = "dMOT98s="

    $str2 = "SELECT target_path, tab_url from downloads"

    $str3 = "\\BraveWallet\\Preferences"

    $wallet = "\\Monero\\wallet.keys"

    $browser1 = "Opera"

    $browser2 = "firefox"

    $winit1 = "https://steamcommunity.com/profiles/76561199686524322"

    $winit2 = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.4.1 Safari/605.1.15 Ddg/17.4.1"

    $winit3 = "https://t.me/k0mono"

    $rc4_key = "291011428669010411719513114 8"

    $build_id = "9ed287469c3721fd5caf346580b2cf0d"

condition:

    2 of ($str*) and $encrypted_str and $rc4_key and $build_id and all of ($browser*) and all of ($winit*) and $wallet and filesize<212KB and hash.md5(0,filesize)== "db8f071d389c007289e2b3ef2112e465" and pe.is_pe  and pe.entry_point >= 00017250  and

(math.entropy(0x400, 0x20800) < 6.5 or math.entropy(0x20C00, 0x0B400) < 5.2 or math.entropy(0x2C000, 0x01000) < 3.9 or math.entropy(0x2D000, 0x05000) < 4.7   )

}
```

MITRE ATTACK TABLE

Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Exfliration	Discovery
Command and Scripting Interpreter (**T1059**)	Account Manipulation (**T1098**)	Process Injection (**T1055**)	Deobfuscate/Decode Files or Information (**T1140**)	Credentials in Registry (**T1555**)	Exfliration Over C2 Channel (**T1041**)	Account Discovery (**T1087**)
	Create or Modify System Process (**T1543**)		Indirect Command Execution (**T1202**)	Exploitation for Credential Access (**T1212**)		System Information Discovery (**T1082**)
			Virtualization/Sandbox Evasion (**T1497**)	OS Credential Dumping (**T1003**)		

Solution Suggestions

1. Using antivirus software is one of the most effective methods for detecting and removing malware. Antivirus software can detect malware by scanning the files and websites you download or open on your computer.

2. You can ensure the security of your computer by regularly updating your operating system and other software. Updates help to close various security gaps.

3. When downloading files, take care to download from reliable sources. Files downloaded from unknown or suspicious sources may contain malware.

4. Ensure that users do not accept cookies and website data by changing their browser settings. This prevents data from being stored locally. Browser plugins can be used to automatically delete cookies.

PREPARED BY

Mehmet Yiğit Türk LinkedIn

Mehmet Emin Gündüzlü LinkedIn