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

blog.vincss.net/re026-a-deep-dive-into-zloader-the-silent-night/

25/04/2022

1. Overview

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.

Zloader has all the standard functionality of a trojan such as being able to fetch information from browsers, stealing cookies and passwords, capturing screenshots, etc. and for making analysis difficult, it applies advanced techniques, including code obfuscation and string encryption, masking Windows APIs call. Recently, CheckPoint expert published an analysis of a Zloader distribution campaign whereby the infection exploited Microsoft's digital signature checking process. In addition, Zloader has also recently partnered with different ransomware gangs are Ryuk and Egregor. This can indicate that the actors behind this malware are still looking for different ways to upgrade it to bypass the defenses. Here is the ranking of Zloader according to the rating from the AnyRun site:

Source: https://any.run/malware-trends/zloader

Most recently, multiple telecommunication providers and cybersecurity firms worldwide partnered with Microsoft's security researchers throughout the investigative effort, including ESET, Black Lotus Labs, Palo Alto Networks' Unit 42, and Avast. They took legal and technical steps to disrupt the ZLoader botnet, seizing control of 65 domains that were used to control and communicate with the infected hosts.

In this article, we will provide detailed analysis and techniques that Zloader uses, including:

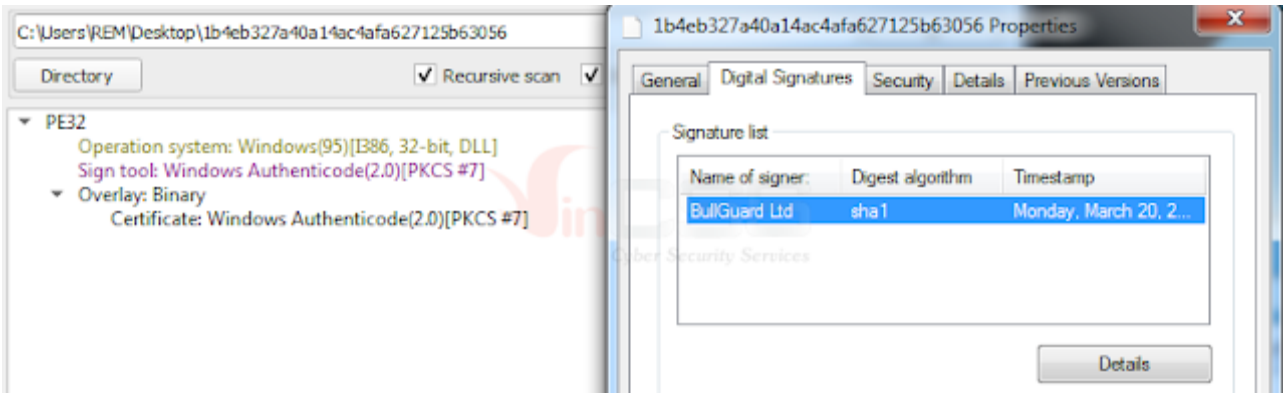
- How to unpack to dump Zloader Core Dll.
- The technique that Zloader makes difficult as well as time consuming in the analysis process.

- Decrypt strings used by Zloader by using both IDAPython and AppCall methods.
- Apply AppCall to recover the Windows API calls.
- Process Injection technique that Zloader uses to inject into the **msiexec.exe** process.
- Decrypt configuration information related to C2s addresses.
- How Zloader collects and saves information in the Registry.
- The Persistence technique.

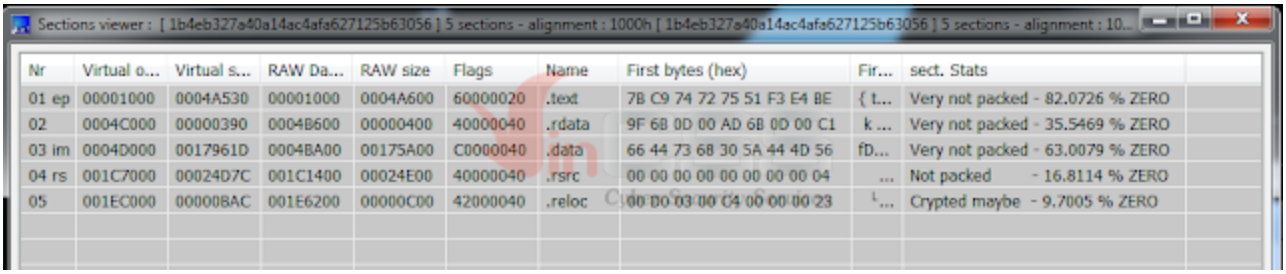
The analyzed sample used in the article:
034f61d86de99210eb32a2dca27a3ad883f54750c46cdec4fcc53050b2f716eb

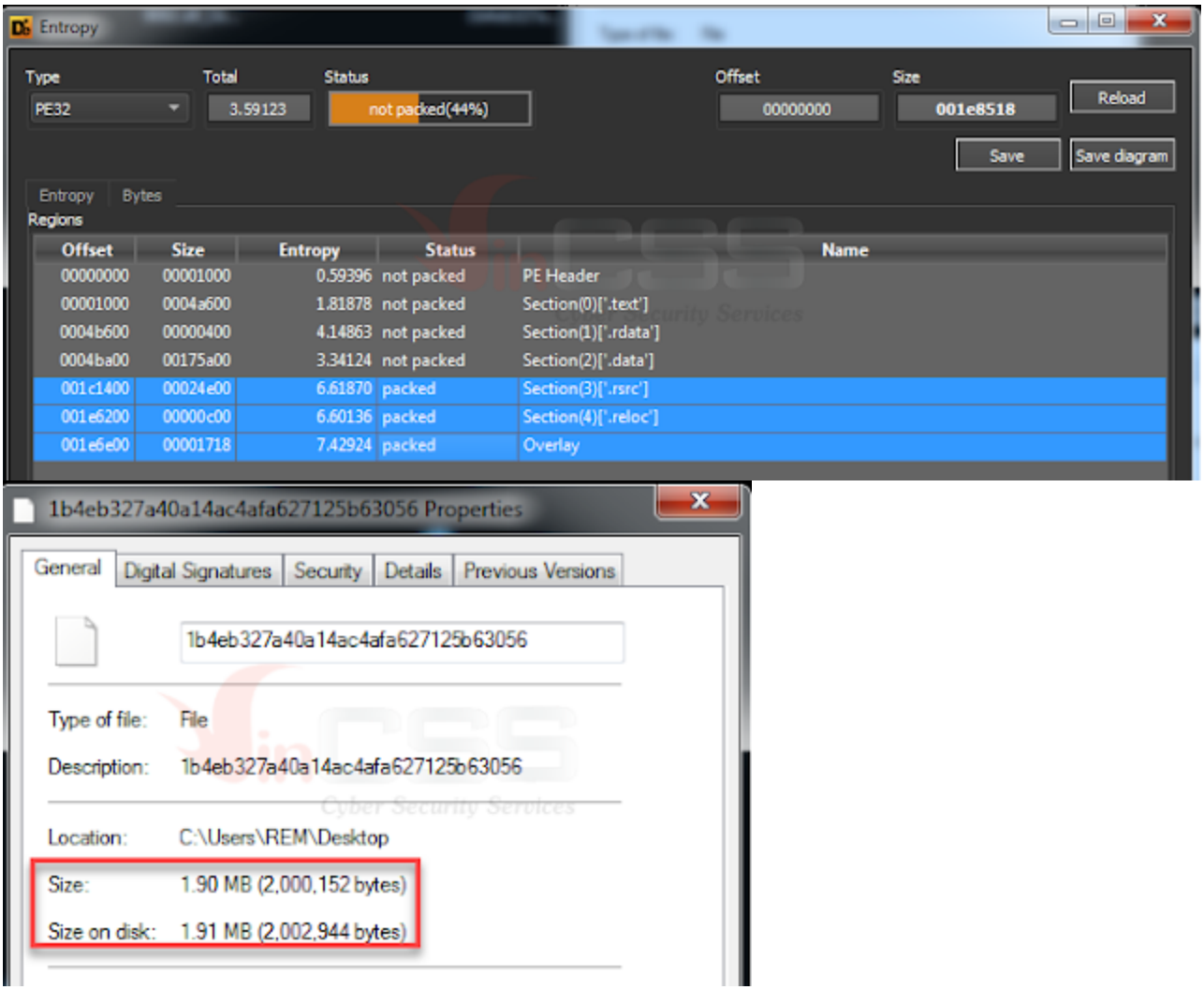
2. Unpacking Zloader Core Dll

First, check the sample with **Nauz File Detector**:

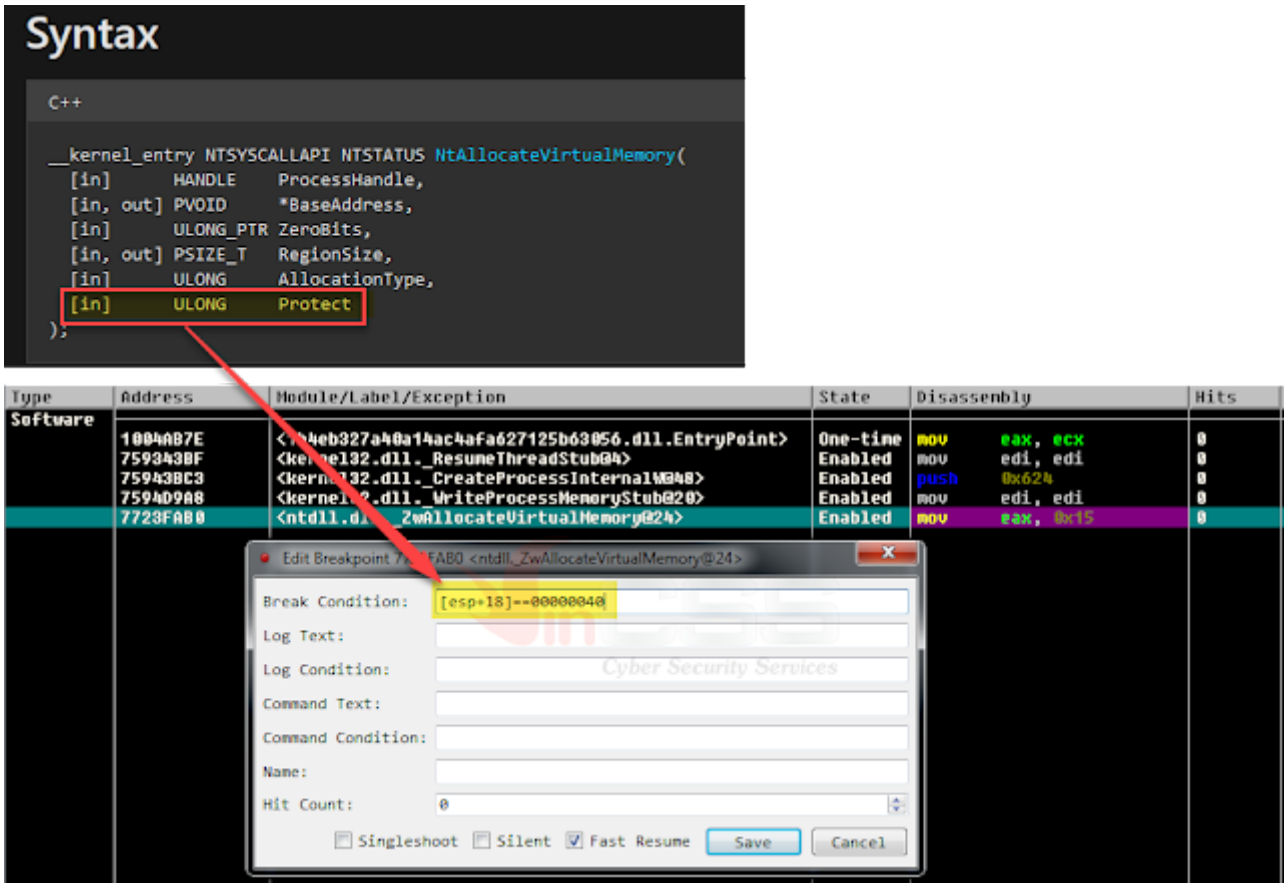


By collecting and combining information about sections from **ExeInfo**, entropy in **DiE** as well as the size of the DLL file, we can confirm that this DLL is packed:

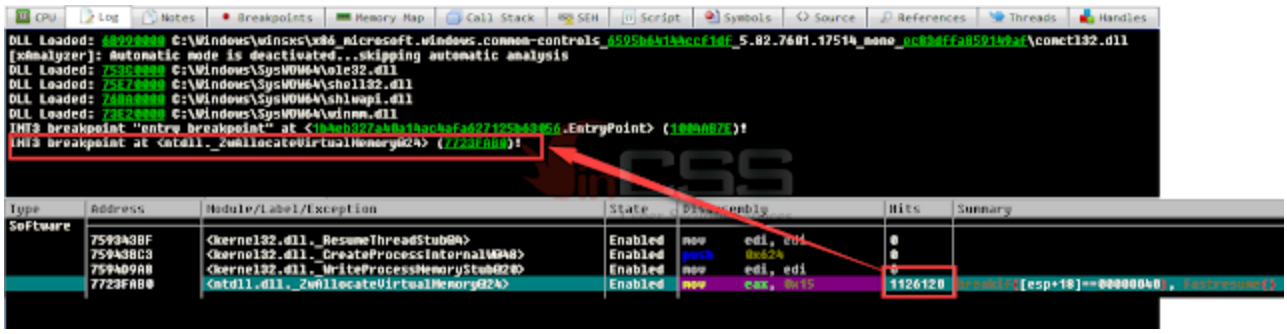




For unpacking, use **x64dbg** to load Dll file, set a **bp NtAllocateVirtualMemory**. Then, modify the breakpoint's condition as follows:

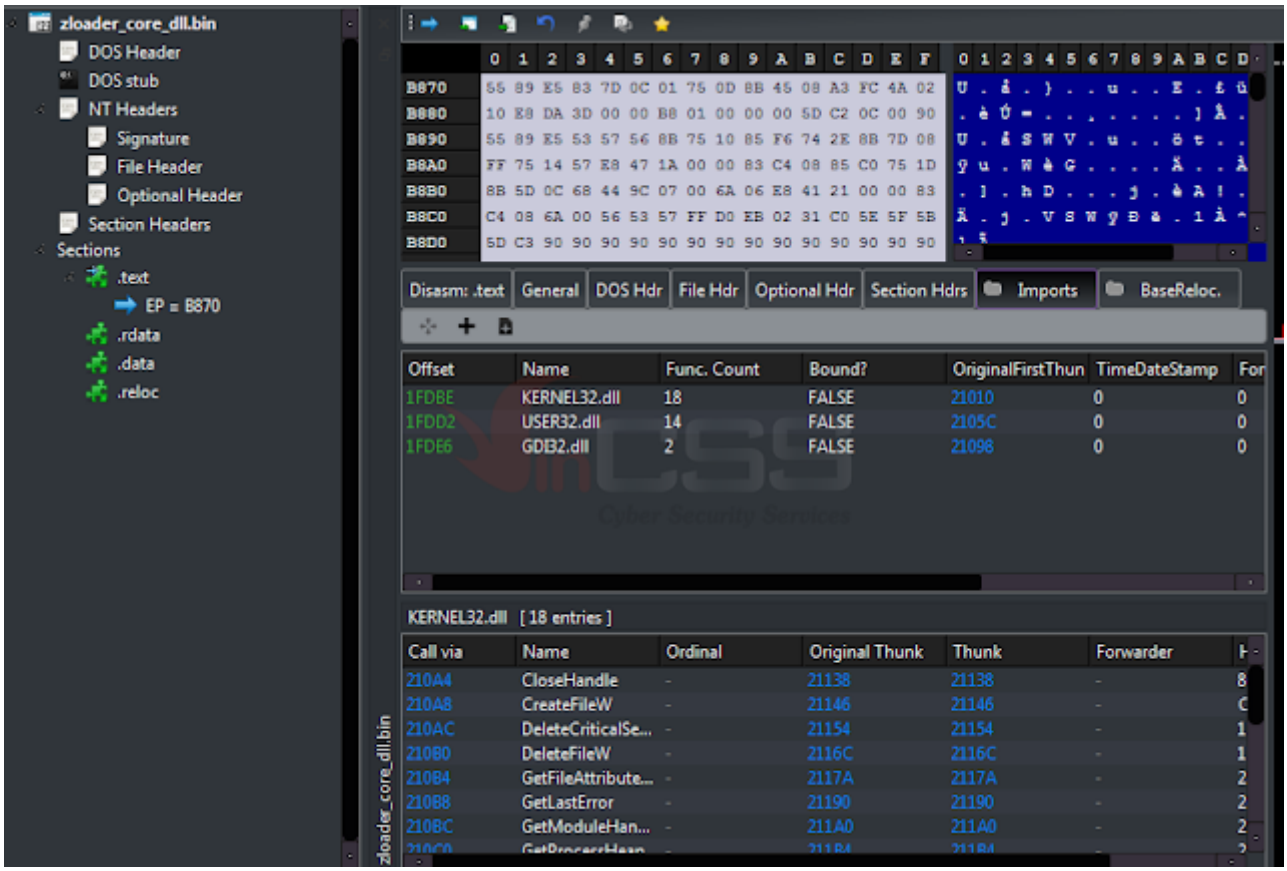


Execute with **F9** and wait until the breakpoint is hit (*after about 1126120 hits*):



Following the allocated memory regions, after the 3rd hit, the core Dll of Zloader will be unpacked:

Dump this Dll to disk, the file has MD5: ***9b5589fcd123a3533584a62956f2231b*.**



3. Anti-analysis

To consume time of the analyst, Zloader uses meaningless functions, or rewrites functions that look very complicated but only to perform simple tasks such as **AND, OR, XOR, ADD, SUB**, etc.

For example, a function that does a meaningless task, however it can cause a delay in execution in a sandbox environment:

```
int __stdcall f_zl_return_weird_value()
{
  signed int tmp1; // edi
  signed int tmp2; // esi
  int tmp3; // esi
  int tmp4; // edi

  tmp1 = ((g_0C80441ADh + 0x2D) ^ (g_0C80441ADh + 0x2D) & (((g_0C80441ADh << 0x18) + 0xB000000) >> 0x18));
  tmp2 = tmp1 * (g_0C80441ADh + 0x12D);
  InsertMenuItemW(
    ((g_0C80441ADh + 0x12D) & (((g_0C80441ADh << 0x18) + 0xB000000) >> 0x18)),
    tmp1 * (g_0C80441ADh + 0x12D),
    (g_0C80441ADh + 0x12D) & (((g_0C80441ADh << 0x18) + 0xB000000) >> 0x18),
    ((g_0C80441ADh + 0x12D) & (((g_0C80441ADh << 0x18) + 0xB000000) >> 0x18)));
  tmp3 = (tmp2 + 0x38B) * tmp2 - (tmp2 + 0x38B);
  tmp4 = (((tmp3 + tmp1) << 0x18) - 0x6E000000) >> 0x18;
  RegisterClassExW(tmp4);
  return (tmp4 & ((0x239 * tmp4 - 0x3A9F6) ^ 0x251 | (((((0x239 * tmp4 - 0x3A9F6) ^ 0x251 | tmp3) << 0x18) + 0x46000000) >> 0x18)))
       + 0x239 * tmp4
       - 0x3A9F6;
}
```

Functions that perform **AND, OR** operations:

```
char __cdecl f_zl_and(char num1, char num2)
{
  int v2; // ebx
  int v3; // edi
  int v4; // edi
  const WCHAR *v5; // ebx
  signed int v7; // [esp+0h] [ebp-14h]
  HDC hdc; // [esp+4h] [ebp-10h]

  hdc = (num1 & num2);
  v2 = (num2 + ((num1 + (num1 & num2)) | num1 & num2) * (num1 + (num1 & num2)));
  v3 = num1 ^ v2;
  v7 = g_0C80441ADh;
  if ( v7 == f_zl_xor_arg_with_0xF6233B5A(0xB43A5CA6) && sub_10006100(num1, hdc) & 1 )
  {
    v4 = v3 - v2;
    v5 = (v4 + num1);
    v3 = v5 * v4;
    sub_10003B60(v5, hdc, v3);
    LOWORD(v3) = (v5 + v3) ^ (hdc ^ num1);
  }
  g_0C80441ADh = (0x176
                * (num1 + 0xCA * (v3 & 8) * v3 - 0x1B1)
                * (f_zl_xor_arg_with_0xF6233B5A(0xF6233AD1) + num1 + 0xCA * (v3 & 8) * v3 - 0x1B1)
                * 0xCA
                * (v3 & 8)
                * v3
                * num1);
  return num1 & num2;
}
```

```
char __cdecl f_zl_or(char num1, char num2)
{
  char tmp; // si
  char result; // al

  tmp = (0xC * (num2 + (num1 ^ (num1 * (num2 + 0x46) + ((num1 * (num2 + 0x46)) ^ 0x9E))))) & num1 & (0xC
                                                                                                    * (num2
                                                                                                     + (num1 ^ (num1 * (num2 + 0x46)
                                                                                                              + ((num1 * (num2 + 0x46)) ^ 0x9E))))
                                                                                                    - num2);
  result = num2 | num1;
  g_0C80441ADh = ((num2 | num1) ^ (tmp + (num2 ^ ((0xBD * tmp + 0x51) * tmp * (tmp ^ (0xBD * tmp + 0x51))))));
  return result;
}
```

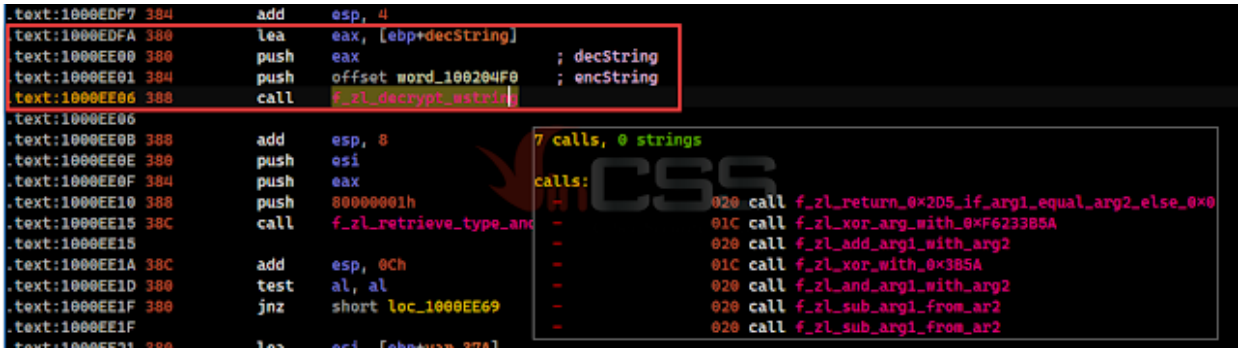
4. Decrypt wide string

4.1. Use IDAPython

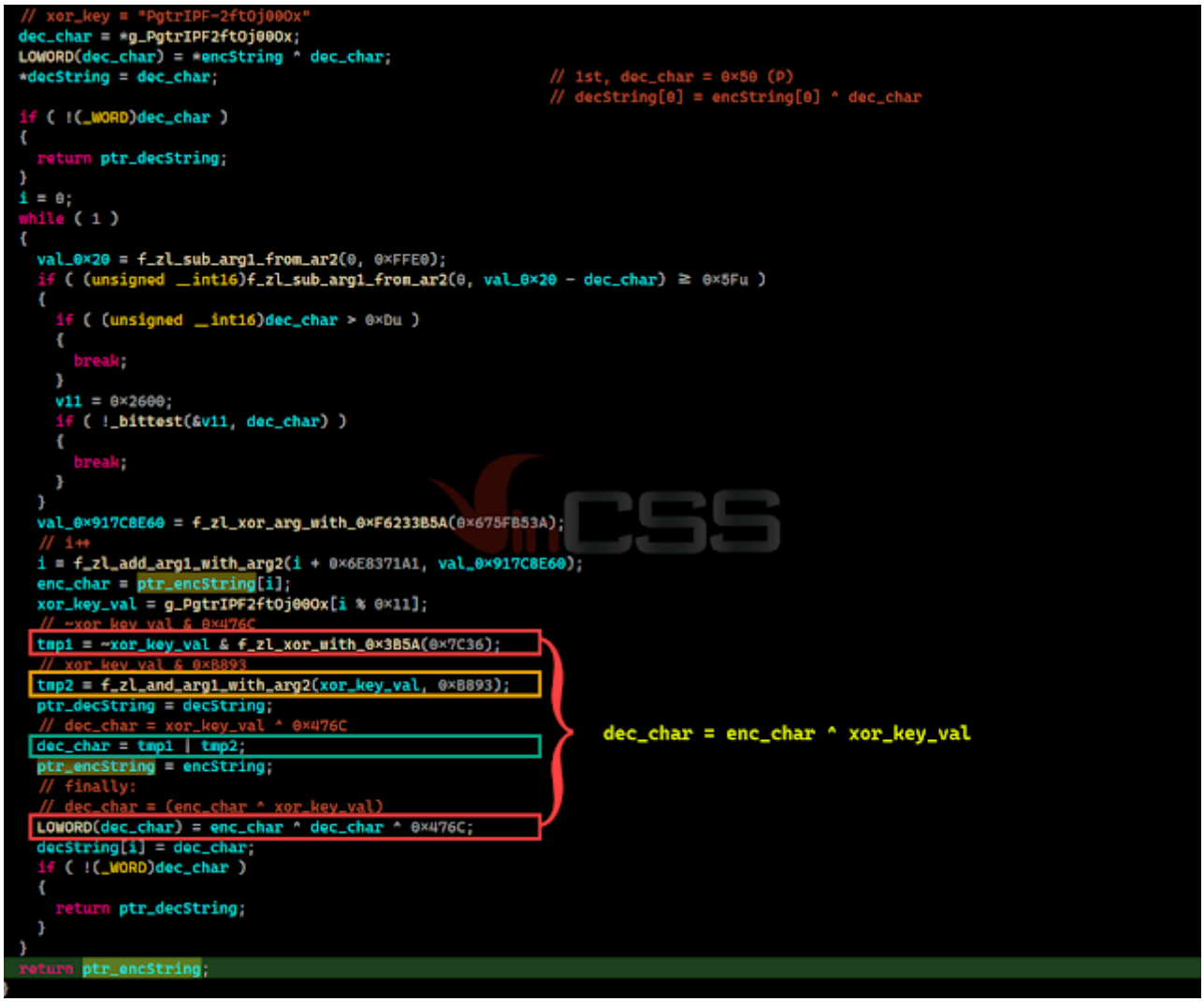
All strings that the core DLL uses are encrypted. The wide string decoder function will take two parameters as input:

- **First parameter:** the address containing the encrypted string.

- **Second parameter:** the address where the string is stored after decoding.



The pseudocode at the **f_zl_decrypt_wstring** decryption function looks confusing, but if we look closely, the function performs a simple xor loop with the decryption key is "**PgtrIPF-2ftOj00Ox**":



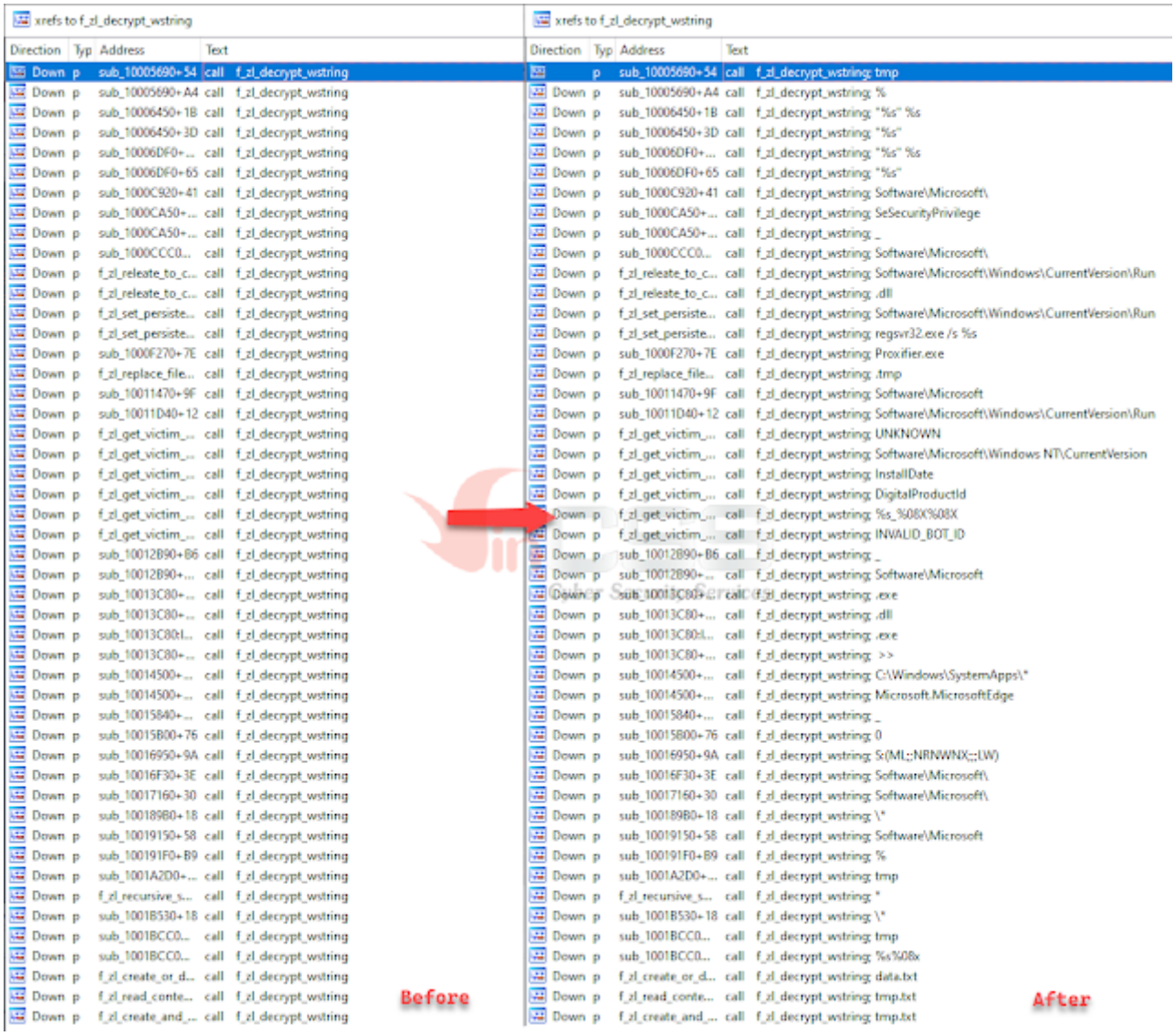
Based on the above pseudocode, the python code that performs decryption as follows:

```python
def decrypt(enc_str):
    """
    decrypt string
    """
    dec_str = ''
    i = 0

    for c in enc_str:
        dec_str += chr(ord(c) ^ ord(xor_key[i % 0x11]))
        i += 1

    return dec_str.rstrip('\x00')
```

With the help of IDAPython, we can automate the whole process of string decoding and add annotations at the decryption functions in IDA for further analysis. The entire python code is as follows:

```python
import idautils, idc, idaapi, ida_search, ida_bytes, ida_auto

xor_key = 'PgtrIPF-2ftOj00Ox'

def read_enc_string(addr):
    """
    read encrypted byte from specified address
    """
    enc_str = ''
    data = idc.get_bytes(addr, idc.get_item_size(addr))
    for i in range(0, len(data), 2):
        enc_str += data[i]

    return enc_str

def decrypt(enc_str):
    """
    decrypt string
    """
    dec_str = ''
    i = 0

    for c in enc_str:
        dec_str += chr(ord(c) ^ ord(xor_key[i % 0x11]))
        i += 1

    return dec_str.rstrip('\x00')

def decrypt_string(func_addr):
    """
    get encrypted string and decrypt it
    """
    args_1 = idaapi.get_arg_addrs(func_addr)[0]
    enc_data_addr = idc.get_operand_value(args_1, 0)
    enc_str = read_enc_string(enc_data_addr)

    return decrypt(enc_str)

def main():
    seg_mapping = {idc.get_segm_name(x): (idc.get_segm_start(x), idc.get_segm_end(x)) for x in idautils.Segments()}
    start = seg_mapping['.text'][0]
    end = seg_mapping['.text'][1]
    pattern = "B9 F1 F0 F0 F0 66 89 45 ?? 89 F8 F7 E1 89 F9 C1 EA 04 89 D0 C1 E0 04 01 D0 29 C1" #mov ecx, 0xf0f0f0f1
    addr = ida_search.find_binary(start, end, pattern, 16, idc.SEARCH_DOWN)
    func_addr = idaapi.get_func(addr).start_ea
    print('[*] Target function found at {}'.format(hex(func_addr)))

    for xref in idautils.XrefsTo(func_addr):
        xref_addr = xref.frm
        if ida_bytes.is_code(ida_bytes.get_full_flags(xref_addr)):
            dec_str = decrypt_string(xref_addr)
            print('    [+] Decrypted string: {} at {}'.format(dec_str, hex(xref_addr)))
            idc.set_cmt(xref_addr, dec_str, 0)

if __name__ == '__main__':
    ida_auto.auto_wait()
    main()
```

The results before and after the script execution will make the analysis easier:

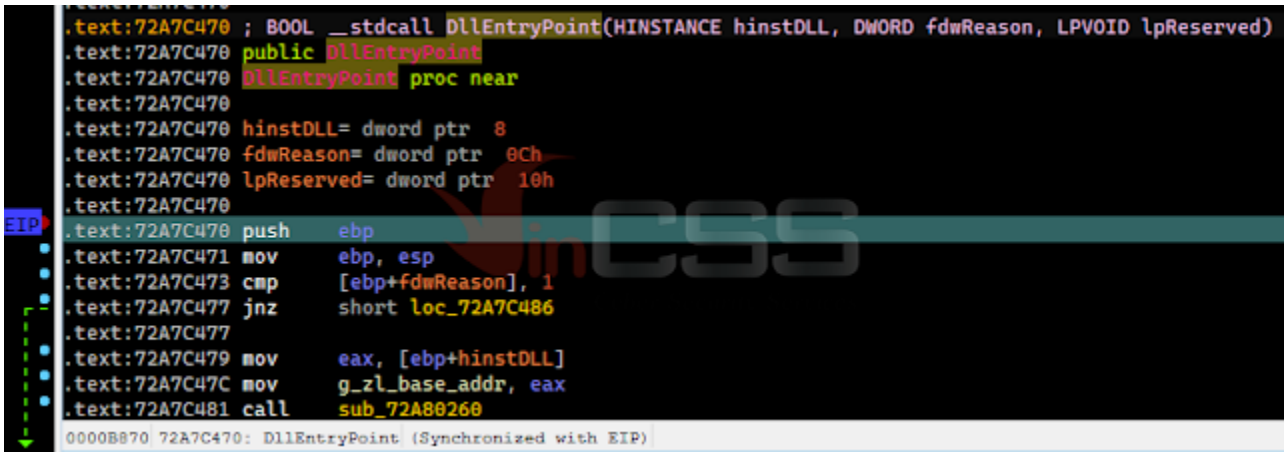


4.2. Use IDA AppCall

If you don't have time to dig into the decryption implementation of the function, or when the algorithm is too complex, we can use IDA's useful feature known as AppCall, to help decrypt the data. Basically, Appcall is a mechanism used to call functions inside the debugged program from the IDA debugger. Before applying AppCall, the first thing is to given a function with a correct prototype. For example, the function **f_zl_decrypt_wstring** has the following protoype:

wchar_t *__cdecl f_zl_decrypt_wstring(wchar_t *encString, wchar_t *decString);

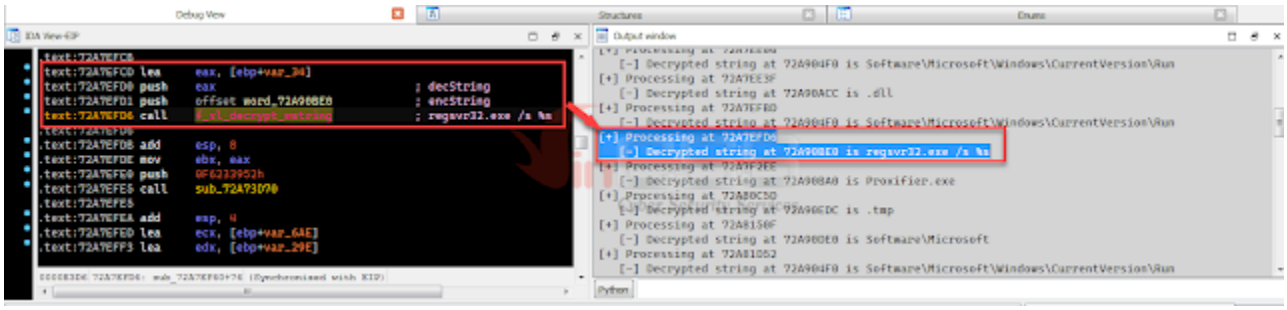
Note again that in order to use AppCall, the program must be debugged. As shown below, IDA is stopping at the breakpoint set at **DllEntryPoint**:



Then execute the below python script to decode and add comments related to decoded strings at the functions:

```
import idc, idaapi, idautils

def decrypt_n_comment(func, func_name):
    """
    Decryption of Zloader string
    """
    for xref in idautils.XrefsTo(idc.get_name_ea_simple(func_name)):
        # init retrieve arguments
        print("[+] Processing at {:08X}".format(xref.frm))
        string_ea = search_inst(xref.frm, "push")
        string_op = idc.get_operand_value(string_ea, 0)

        buf = idaapi.Appcall.buffer("\x00" * 128)

        # Call Zloader's func
        try:
            res = func(string_op, buf)
            if type(res.decode('utf-16')) == str:
                print("   [-] Decrypted string at {:08X} is {}".format(string_op, res.decode('utf-16')))
        except Exception as e:
            print("FAILED: appcall failed: {}".format(e))
            continue

        # Add comments
        try:
            idc.set_cmt(xref.frm, res.decode('utf-16'), idc.SN_NOWARN)
        except:
            print("FAILED: to add comment")
            continue

def search_inst(ea, inst):
    """
    Return the address of wanted instruction
    """
    while True:
        if idc.print_insn_mnem(ea) == inst:
            return ea
        ea = idc.prev_head(ea)

# Initialization
FUNC_NAME = "f_zl_decrypt_wstring"
PROTO = "wchar_t *__cdecl {:s}(wchar_t *encString, wchar_t *decString);".format(FUNC_NAME)

# Execution
decrypt_function = idaapi.Appcall.proto(FUNC_NAME, PROTO)
decrypt_n_comment(decrypt_function, FUNC_NAME)
```

The final result should be similar to the image below:

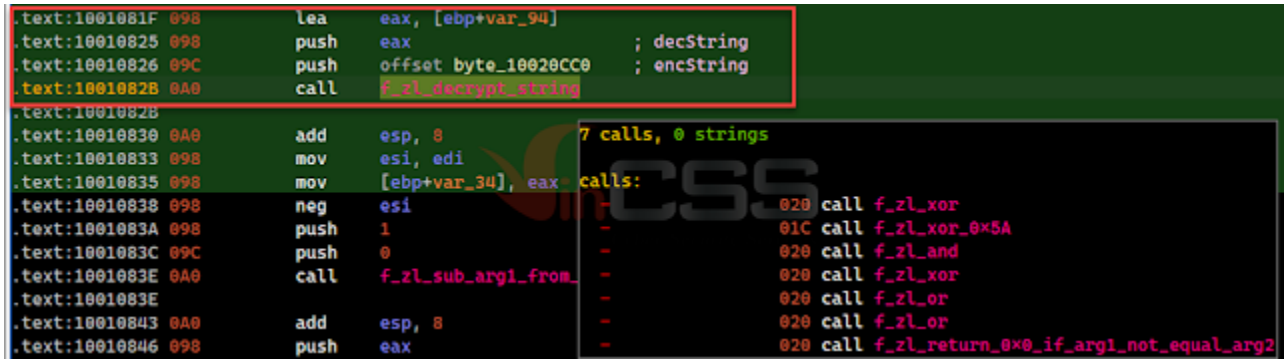


5. Decrypt ansi string

5.1. Use IDAPython

Besides the function to decode wide strings, Zloader also uses the function to decode ansi strings. This function also accepts two arguments:

- **First parameter:** the address containing the encrypted string.
- **Second parameter:** the address where the string is stored after decoding.



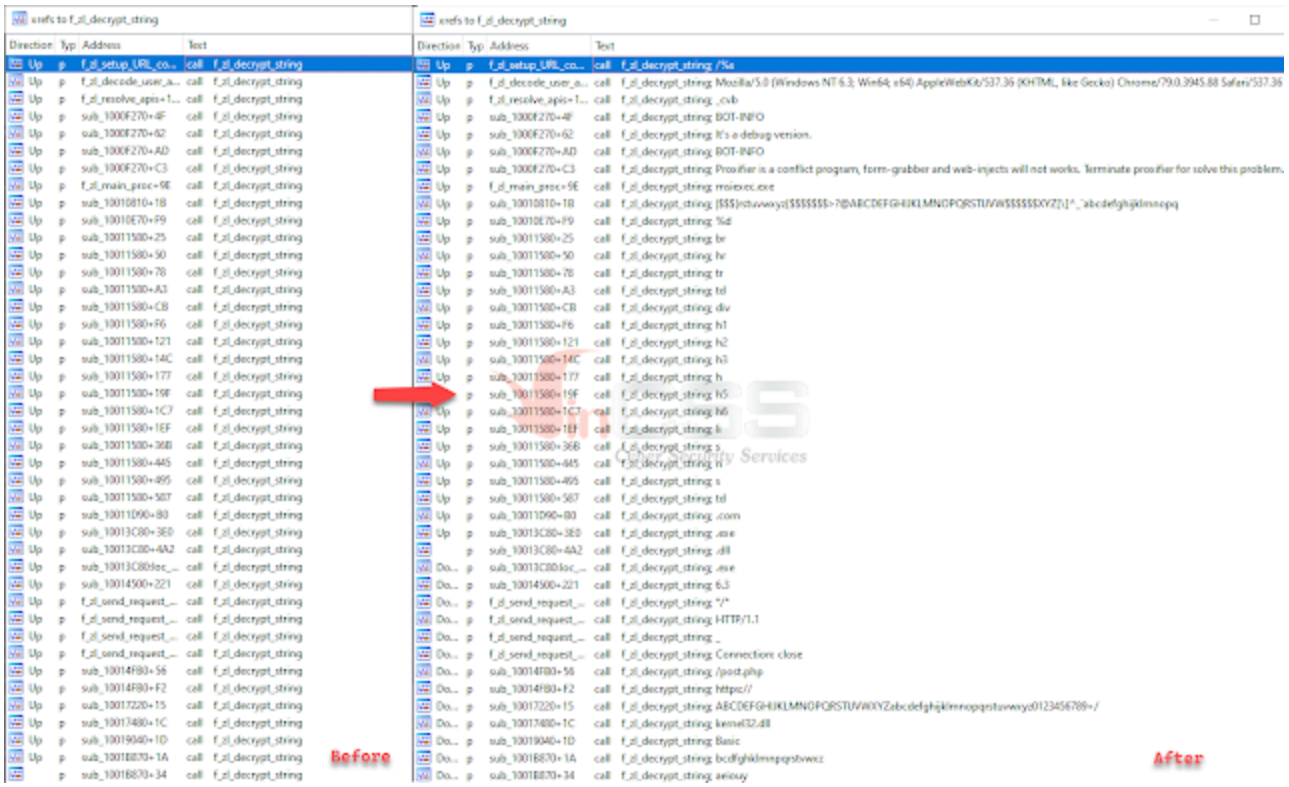
Similar to the above **f_zl_decrypt_wstring** function, the pseudocode of the **f_zl_decrypt_string** function looks quite messy, but it still uses an xor loop to decrypt with the decryption key still "**PgtrIPF-2ftOj00Ox**":

```
enc_char = *encString;
v3 = ~*encString;
// xor_key = "PgtrIPF-2ftOj00Ox"
xor_key_val_0x50 = *g_PgtrIPF2ftOj00Ox;
val_0xAF = f_zl_xor(*g_PgtrIPF2ftOj00Ox, 0xFF);
val_0x59 = f_zl_xor_0x5A(3);
val_9 = f_zl_and(val_0x59, val_0xAF);
val_0xA6 = f_zl_xor(0x59, 0xFF);
v8 = enc_char & val_0xA6;
val_9_ = f_zl_or(val_9, xor_key_val_0x50 & val_0xA6);
// dec_char = val_9 ^ (~enc_char[0] & 0x59 | enc_char[0] & val_0xA6) = enc_char[0] ^ xor_key[0]
dec_char = val_9_ ^ f_zl_or(v3 & 0x59, v8);
*decString = dec_char;
if ( dec_char )
{
  i = 1;
  while ( 1 )
  {
    v11 = f_zl_return_0x0_if_arg1_not_equal_arg2(dec_char, 0x7F);
    if ( dec_char < 0x20 || v11 & 1 )
    {
      if ( (unsigned __int8)dec_char > 0xDu )
      {
        break;
      }
      v12 = 0x2600;
      if ( !_bittest(&v12, (unsigned __int8)dec_char) )
      {
        break;
      }
    }
    // dec_char = encString[i] ^ xor_key[i % 0x11]
    dec_char = encString[i] ^ g_PgtrIPF2ftOj00Ox[0xFFFFFFEF * (i / 0x11) + i];
    ptr_encString = decString;
    decString[i++] = dec_char;
    if ( !dec_char )
    {
      return ptr_encString;
    }
  }
  ptr_encString = encString;
}
else
{
  ptr_encString = decString;
}
return ptr_encString;
}
```

Here is the full python code to automate the whole process of decoding strings and adding comments at functions:

```python
import idautils, idc, idaapi, ida_search, ida_bytes, ida_auto

xor_key = 'PgtrIPF-2ftOj08Ox'

def read_enc_string(addr):
    """
    read encrypted byte from specified address
    """
    enc_str = idc.get_bytes(addr, idc.get_item_size(addr))

    return enc_str

def decrypt(enc_str):
    """
    decrypt string
    """
    dec_str = ''
    i = 0

    for c in enc_str:
        dec_str += chr(ord(c) ^ ord(xor_key[i % 0x11]))
        i += 1

    return dec_str.rstrip('\x00')

def decrypt_string(func_addr):
    """
    get encrypted string and decrypt it
    """
    args_1 = idaapi.get_arg_addrs(func_addr)[0]
    enc_data_addr = idc.get_operand_value(args_1, 0)
    enc_str = read_enc_string(enc_data_addr)

    return decrypt(enc_str)

def main():
    seg_mapping = {idc.get_segm_name(x): (idc.get_segm_start(x), idc.get_segm_end(x)) for x in idautils.Segments()}
    start = seg_mapping['.text'][0]
    end = seg_mapping['.text'][1]
    pattern = "B9 F1 F0 F0 F0 F7 E1 0F B6 C3 89 D6 6A ??" #mov ecx, 0xf0f0f0f1
    addr = ida_search.find_binary(start, end, pattern, 16, idc.SEARCH_DOWN)
    func_addr = idaapi.get_func(addr).start_ea
    print('[*] Target function found at {}'.format(hex(func_addr)))

    for xref in idautils.XrefsTo(func_addr):
        xref_addr = xref.frm
        if ida_bytes.is_code(ida_bytes.get_full_flags(xref_addr)):
            dec_str = decrypt_string(xref_addr)
            print('    [+] Decrypted string: {} at {}'.format(dec_str, hex(xref_addr)))
            idc.set_cmt(xref_addr, dec_str, 0)

if __name__ == '__main__':
    ida_auto.auto_wait()
    main()
```

The results before and after the script execution



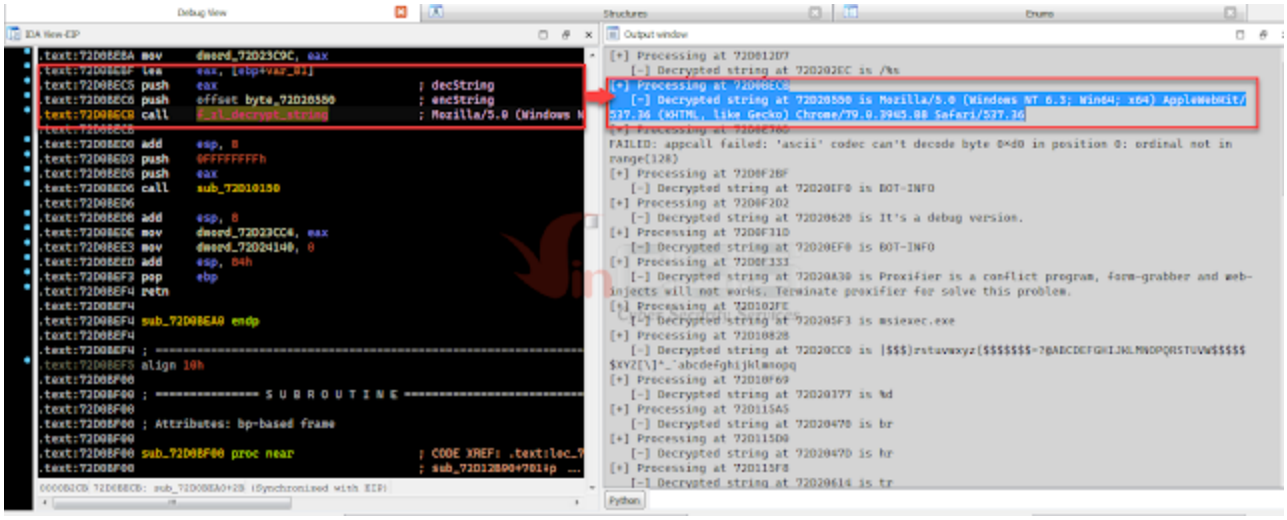
5.2. Use IDA AppCall

To use AppCall, same as above, need to define correctly the prototype for the **f_zl_decrypt_string** function as follows: **char *__cdecl f_zl_decrypt_string(char *encString, char *decString);**

Slightly modified the script used for decoding the wide strings above:

```python
import idc, idaapi, idautils

def decrypt_n_comment(func, func_name):
    """
    Decryption of Zloader string
    """
    for xref in idautils.XrefsTo(idc.get_name_ea_simple(func_name)):
        # init retrieve arguments
        print("[+] Processing at {:08X}".format(xref.frm))
        string_ea = search_inst(xref.frm, "push")
        string_op = idc.get_operand_value(string_ea, 0)

        buf = idaapi.Appcall.buffer("\x00" * 128)

        # Call Zloader's func
        try:
            res = func(string_op, buf)
            if type(res.decode('ascii')) == str:
                print("   [-] Decrypted string at {:08X} is {}".format(string_op, res.decode('ascii')))
        except Exception as e:
            print("FAILED: appcall failed: {}".format(e))
            continue

        # Add comments
        try:
            idc.set_cmt(xref.frm, res.decode('ascii'), idc.SN_NOWARN)
        except:
            print("FAILED: to add comment")
            continue

def search_inst(ea, inst):
    """
    Return the address of wanted instruction
    """
    while True:
        if idc.print_insn_mnem(ea) == inst:
            return ea
        ea = idc.prev_head(ea)

# Initialization ---------------------------------------
FUNC_NAME = "f_zl_decrypt_string"
PROTO = "char *__cdecl {:s}(char *encString, char *decString);".format(FUNC_NAME)

# Execution --------------------------------------------
decrypt_function = idaapi.Appcall.proto(FUNC_NAME, PROTO)
decrypt_n_comment(decrypt_function, FUNC_NAME)
```

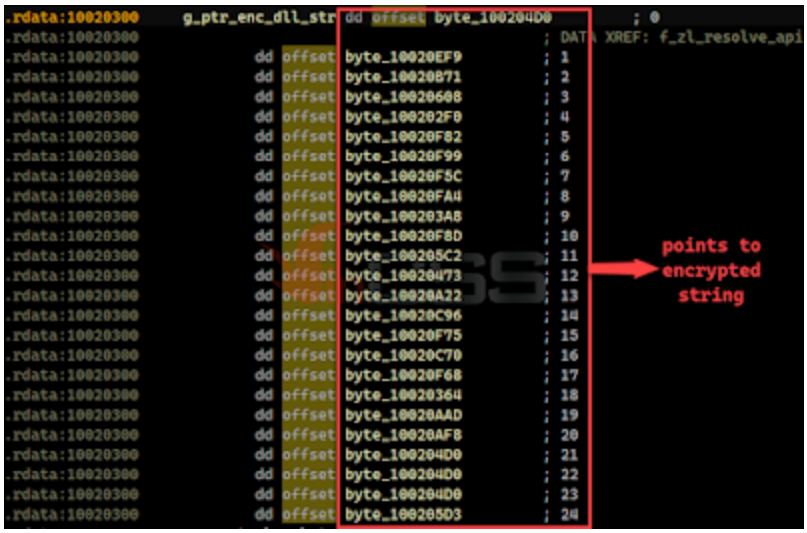
Result after running the script:



6. List of Dlls used by Zloader

In the list of strings decrypted by the **f_zl_decrypt_string** function above, there is a string after the decryption that is quite meaningless. Going to this address, after diving into it I noticed that the first parameter passed to the function is an array containing the addresses of the encrypted strings. Based on the corresponding **index** value of the array will access the address containing the corresponding encrypted string:

Going to the **g_ptr_enc_dll_str** array (*renamed above*) will see a list of addresses as shown below:



Modify the script to decode the specific Dll strings, the results obtained when executing the script are as follows:

To summarize, we have a list of **indexes** corresponding to the DLLs that Zloader can use to retrieve the addresses of APIs:

Index	Dll Name
0	kernel32.dll
1	user32.dll
2	ntdll.dll
3	shlwapi.dll
4	iphlpapi.dll
5	urlmon.dll
6	ws2_32.dll
7	crypt32.dll
8	shell32.dll
9	advapi32.dll
10	gdiplus.dll
11	gdi32.dll
12	ole32.dll

13	psapi.dll
14	cabinet.dll
15	imagehlp.dll
16	netapi32.dll
17	wtsapi32.dll
18	mpr.dll
19	wininet.dll
20	userenv.dll
21	bcrypt.dll

7. Dynamic APIs resolve

Similar to other advanced malware… Zloader will also get the address of API function(s) through searching by pre-computed hash value based on API function name.

As shown in the above figure, the **f_zl_resolve_api_func_ex** function takes two parameters:

> (1): The first parameter is **dll_index**. Based on this parameter, the function will decode the name of the corresponding Dll, then call the **LoadLibraryA** function to get the base address of this Dll.

```
{
  // decrypt Dll name based on Dll index
  sz_dll_name = f_zl_decrypt_string((&g_ptr_enc_dll_str)[arg_dll_index], dec_str);
  f_zl_strcpy(lpLibFileName, sz_dll_name, 0xFFFFFFFF);
}
```

```
else
{
  hModule = LoadLibraryA(lpLibFileName);
  if ( !hModule )
  {
    goto LABEL_18;
  }
}
```

(2): The second parameter is **pre_api_hash**. This parameter is the pre-computed hash of the API function name. The function **f_zl_resolve_api_func_ex** will call **f_zl_resolve_api_func** to retrieve the corresponding API address:

```
retrieve_api_addr:
        api_addr = f_zl_resolve_api_func(hModule, pre_api_hash);
        if ( api_addr )
```

The pseudocode at the **f_zl_resolve_api_func** function as follows:

```
export_dir_va = (dll_base_addr + export_dir_rva);
export_dir_size = pOptionalHeaders->DataDirectory[0].Size;
AddressOfNameOrdinals_sub_0x317D5B64 = dll_base_addr + export_dir_va->AddressOfNameOrdinals - 0x317D5B64;
val_0xCE82A49C = f_zl_xor_arg_with_0xF6233B5A(0x38A19FC6);
pOrdinalsTbl = f_zl_sub_arg1_from_arg2(AddressOfNameOrdinals_sub_0x317D5B64, val_0xCE82A49C);
pFuncNameAddr = (dll_base_addr + f_zl_add_arg1_with_arg2(export_dir_va->AddressOfNames, 0x10601647) - 0x10601647);
i = 0;
while ( 1 )
{
  func_name_rva = *pFuncNameAddr;
  val_0x64 = f_zl_xor_arg_with_0xF6233B5A(0xF6233B3E);
  f_zl_memset_ex(&sz_api_name, val_0x64);
  // get first char of api name
  c = *(dll_base_addr + func_name_rva);
  // convert api name to lowercase and store in buffer
  if ( !(f_zl_return_0x0_if_arg1_not_equal_arg2(c, 0) & 1) )
  {
    ptr_api_name = dll_base_addr + func_name_rva;
    j = 0;
    do
    {
      *(&sz_api_name + j) = f_zl_lower_case(c);
      val_0xFFFFFFFF = f_zl_sub_arg1_from_arg2(0, 1);
      j -= val_0xFFFFFFFF;
      f_zl_add_arg1_with_arg2(j, 1);
      c = ptr_api_name[j];
    }
    while ( c );
  }
  if ( f_zl_calc_hash_ex(&sz_api_name, 0xFFFFFFFF) == pre_api_hash )
  {
    break;
  }
  ++i;
  ++pFuncNameAddr;
  ++pOrdinalsTbl;
  if ( i >= export_dir_va->NumberOfNames )
  {
    return 0;
  }
}
api_addr = (f_zl_add_arg1_with_arg2(*(&dll_base_addr[*pOrdinalsTbl] + export_dir_va->AddressOfFunctions) + 0x74C029BC, dll_base_addr) - 0x74C029BC);
```

convert API name to lowercase

compare calculated hash to pre_hash

The entire pseudocode of the function that performs the hash calculation by the API function name is as follows:

```
int __fastcall f_zl_calc_hash(char *inString, int strLen)
{
  int calced_hash; // edi MAPDST
  unsigned int i; // edx MAPDST
  int v6; // ebx
  int val_0x825180FD; // eax
  int val_0x7DAE7F02; // eax

  calced_hash = 0;
  if ( !inString || strLen <= 0 )
  {
    return calced_hash;
  }
  i = 0;
  calced_hash = 0;
  do
  {
    // calced_hash = (calced_hash << 0x4) + ord(c)
    calced_hash = 16 * calced_hash + *inString;
    if ( calced_hash & 0xF0000000 )
    {
      v6 = calced_hash & f_zl_xor_arg1_with_arg2(calced_hash, 0xF0000000);
      val_0x825180FD = f_zl_xor_arg_with_0xF623385A(0x7472B8A7);
      val_0x7DAE7F02 = f_zl_xor_arg1_with_arg2(val_0x825180FD, 0xFFFFFFFF);// ~0x7DAE7F02 = 0x825180FD
      calced_hash = (((calced_hash & 0xF0000000) >> 0x18) ^ 0x825180FD | val_0x7DAE7F02 & ((calced_hash & 0xF0000000) >> 0x18)) ^ (~v6 & 0x825180FD | val_0x7DAE7F02 & v6);
    }
    // i = i + 1
    i = i + f_zl_xor_arg_with_0xF623385A(0xE9AAFD8B) - 0x1F89C6E8;
    ++inString;
  }
  while ( i < strLen );
  return calced_hash;
}
```

Based on the above pseudocode, re-implement using Python code as follows:

```
def calc_api_hash(api_name):
    func_name = api_name.lower()
    mask = 0xf0000000
    calced_hash = 0
    for c in func_name:
        calced_hash = (calced_hash << 0x4) + ord(c)
        if calced_hash & mask:
            calced_hash = (((calced_hash & mask) >> 0x18) ^ 0x825180FD | ~0x825180FD & ((calced_hash & mask) >> 0x18)) ^ (calced_hash
            ^ calced_hash & mask ^ 0x825180FD)
    return calced_hash & 0xffffffff
```

Results when using the above function to find API functions corresponding to hash values hash **0xFDA8B77**, **0xB1C1FE3**, **0x8ADF2D1**:

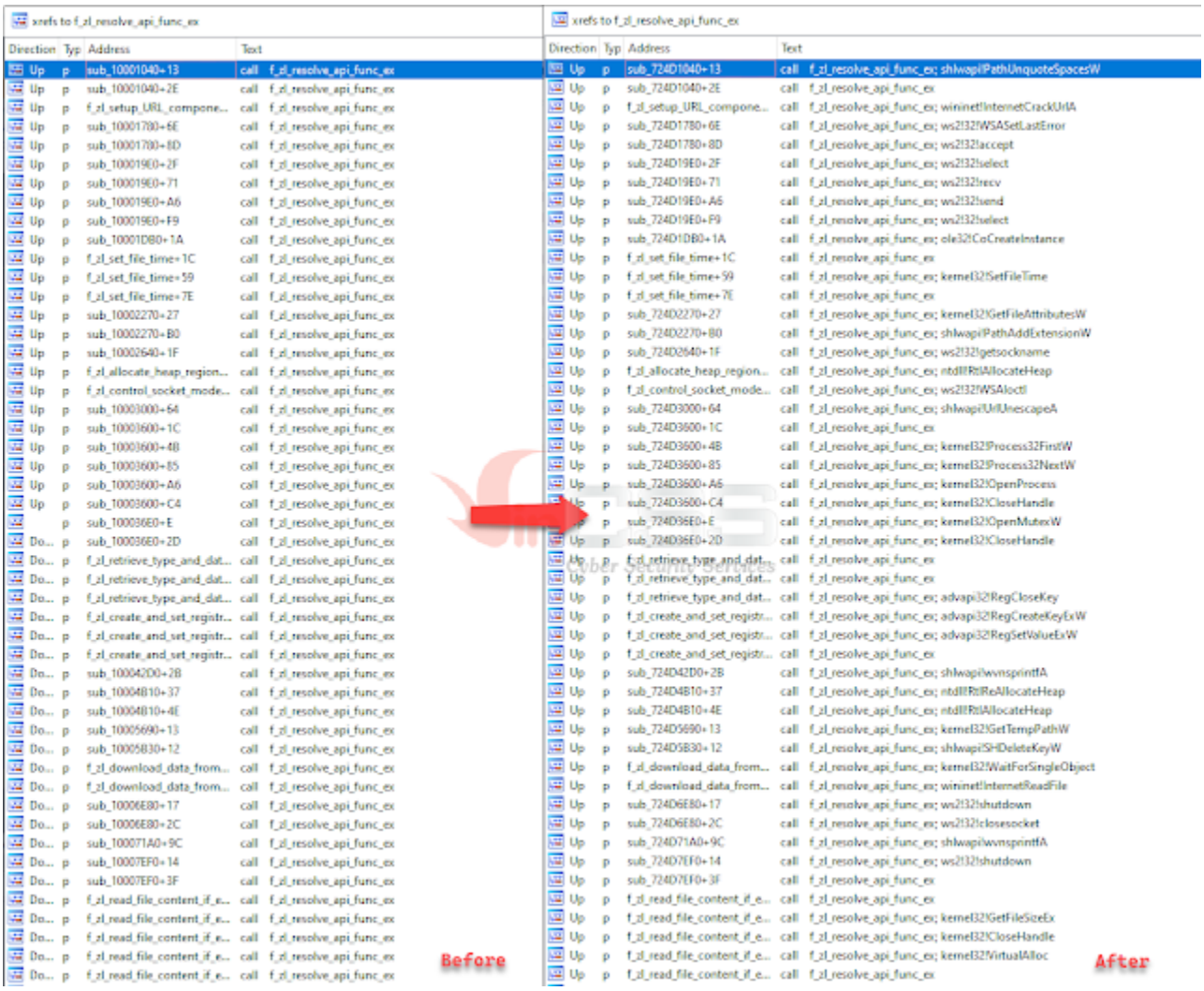
```
v1 = f_zl_resolve_api_func_ex(0, 0xFDA8B77u);
(v1)(g_zl_base_addr, v36, MAX_PATH);
::GetProcAddress = f_zl_resolve_api_func(dll_base_addr, 0xB1C1FE3);
LoadLibraryA = f_zl_resolve_api_func(dll_base_addr, 0x8ADF2D1);
```

```
~@@)  python .\zloader_brute_api_funcs.py
API hash: 0xFDA8B77 --> API found: GetModuleFileNameW
API hash: 0xB1C1FE3 --> API found: GetProcAddress
API hash: 0x8ADF2D1 --> API found: LoadLibraryA
```

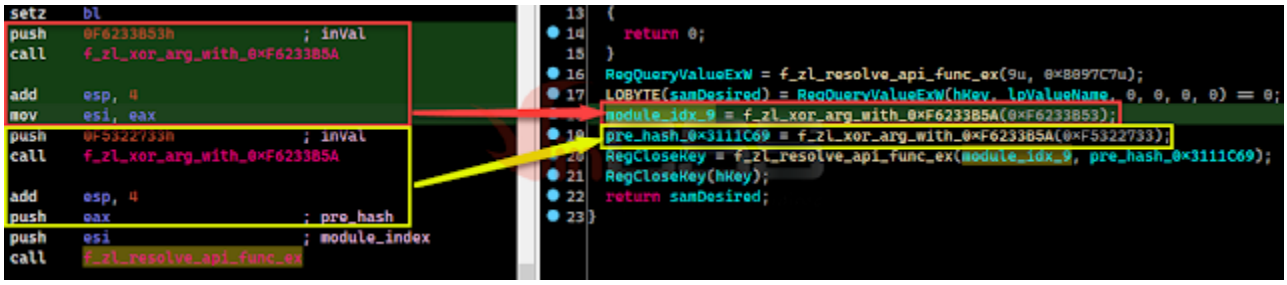
With all the above analysis results, it is possible to write an IDAPython script to recover all the APIs that Zloader uses. However, to avoid having to dig into Zloader's hashing algorithm for each analysis, here I will use AppCall to do this task. The python code that uses AppCall is as follows:

```python
import idc, idaapi, idautils

def resolve_n_comment(func, func_name):
    """
    Resolve API
    """
    for xref in idautils.XrefsTo(idc.get_name_ea_simple(func_name), 0):
        # init retrieve arguments
        xref_addr = xref.frm
        print("[+] Processing at {:08X}".format(xref_addr))
        arg1_ea = idaapi.get_arg_addrs(xref_addr)[0]
        module_index = idc.get_operand_value(arg1_ea, 0)
        arg2_ea = idaapi.get_arg_addrs(xref_addr)[1]
        pre_api_hash = idc.get_operand_value(arg2_ea, 0)

        if module_index < 0 or pre_api_hash <= 4:
            continue

        # Call Zloader's resolve api func
        try:
            print ("    [-] Module index: {:08X}".format(module_index))
            print ("    [-] Precalculated hash: {:08X}".format(pre_api_hash))
            addr = func(module_index, pre_api_hash)
        except Exception as e:
            print("FAILED: appcall failed: {}".format(e))
            continue

        try:
            # Get exported api_name of all loaded modules (cover all segments)
            api_name = idaapi.get_debug_names(idaapi.cvar.inf.minEA, idaapi.cvar.inf.maxEA)
            print ("    [-] Resolved API: {}".format(api_name[addr]))
            # Add comments
            idc.set_cmt(xref_addr, "{:}".format(api_name[addr].replace("_", "!")),0)
            set_cmt_api_call(xref_addr, "{:}".format(api_name[addr].replace("_", "!")))
        except:
            print("FAILED: to get exported name and add comment")
            continue

def set_cmt_api_call(addr, api_name):
    """
    Set comment api name at call eax
    """
    curr_addr = addr
    address_plus_50 = addr + 50
    while curr_addr <= address_plus_50:
        curr_addr = idc.next_head(curr_addr)
        if idc.print_insn_mnem(curr_addr) == "call" and 'eax' in idc.print_operand(curr_addr, 0):
            idc.set_cmt(curr_addr, api_name, idaapi.SN_NOWARN)

# Initialization ---------------------------------------
FUNC_NAME = "f_zl_resolve_api_func_ex"
PROTO = "int __cdecl {:s}(unsigned int arg_dll_index, unsigned int pre_api_hash);".format(FUNC_NAME)

# Execution --------------------------------------------
resolve_function = idaapi.Appcall.proto(FUNC_NAME, PROTO)
resolve_n_comment(resolve_function, FUNC_NAME)
```

Note, Zloader has many areas of code that call to the **f_zl_resolve_api_func_ex** function, but there will be areas of code that do not have any reference to it and that area has not been defined as a complete function. Therefore, to be able to run the above script, it is necessary to create functions for those first. The final result after executing the script will be as follows:



However, as shown in the figure there are still places where the API function can't be recovered, that's because Zloader has performed the previous calculation of the **dll_index** and **pre_api_hash** values and saved them in the register. After that, call the **f_zl_resolve_api_func_ex** function:



8. Process Injection Technique

Zloader, when executed, will inject Core Dll into the **msiexec.exe** process. The whole process is as follows:

Use the **CreateProcessA** API function to create the **msiexec.exe** process in the **SUSPENDED** state.

```
// msiexec.exe
sz_msiexec = f_zl_decrypt_string(asc_749805F3, v38);
f_zl_strcpy(sz_msiexec.exe, sz_msiexec, 0xFFFFFFFF);
// msiexec.exe process is created in a suspended state
CreateProcessA = f_zl_resolve_api_func_ex(0, 0x1E16041u);
if ( CreateProcessA(0, sz_msiexec.exe, 0, 0, 0, CREATE_SUSPENDED, 0, 0, &StartupInfo, &ProcessInformation) )
{
```

Get **SizeOfImage** value of Zloader Dll being loaded by **rundll32.exe/regsvr32.exe**. Use the **VirtualAllocEx** API function to allocate new memory inside the **msiexec.exe** process:

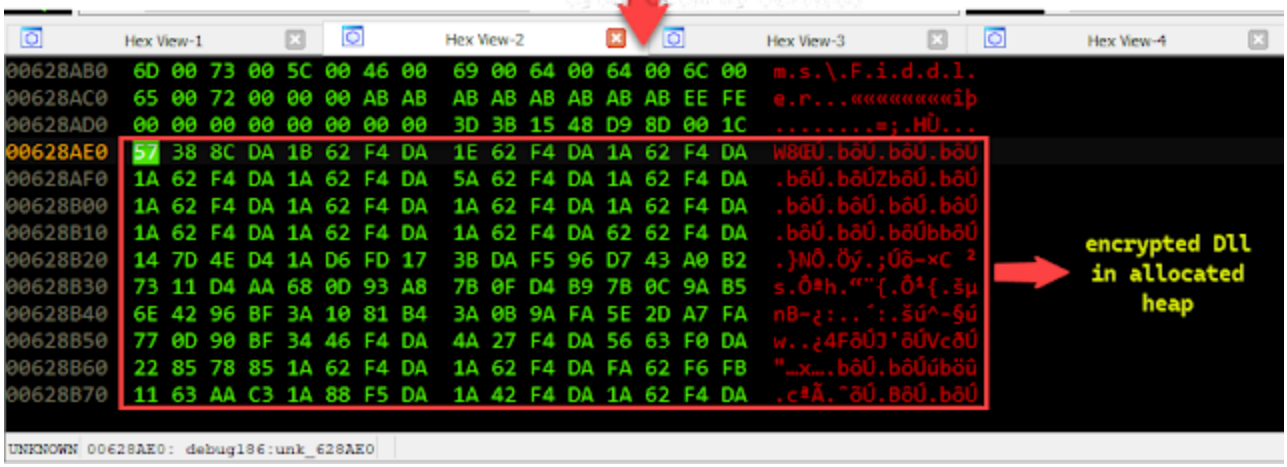
```
zl_size_of_image = f_zl_retrieve_size_of_image(zl_base_addr);
val_0x8CAE838 = f_zl_xor_arg_with_0xF6233B5A(0xFEE9D362);
VirtualAllocEx = f_zl_resolve_api_func_ex(0, val_0x8CAE838);
// allocate region within msiexec.exe with size of region is Zloader's SizeOfImage
zl_payload_buf_in_msiexec = VirtualAllocEx(ProcessInformation.hProcess, 0, zl_size_of_image, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
if ( zl_payload_buf_in_msiexec )
```

Allocate heap memory, copy the entire contents of the Dll into this heap:

```
if ( zl_payload_buf_in_msiexec )
{
  g_zl_payload_buf_in_msiexec = zl_payload_buf_in_msiexec;
  zl_base_addr_in_msiexec = zl_payload_buf_in_msiexec;
  f_zl_wchar_strcpy(sz_msiexec.exe, wsz_zl_dll_path);
  // store zloader dll path into global var
  f_zl_wstrcpy_ex(sz_msiexec.exe);
  f_zl_free_heap_ex(sz_msiexec.exe);
  // copy zloader content to new allocated heap region
  zl_dll_content_in_heap = f_zl_memcpy_ex(zl_base_addr, zl_size_of_image);
  f_zl_update_image_base(zl_dll_content_in_heap, zl_base_addr);
  f_zl_perform_base_relocation(zl_dll_content_in_heap, zl_base_addr_in_msiexec);
```

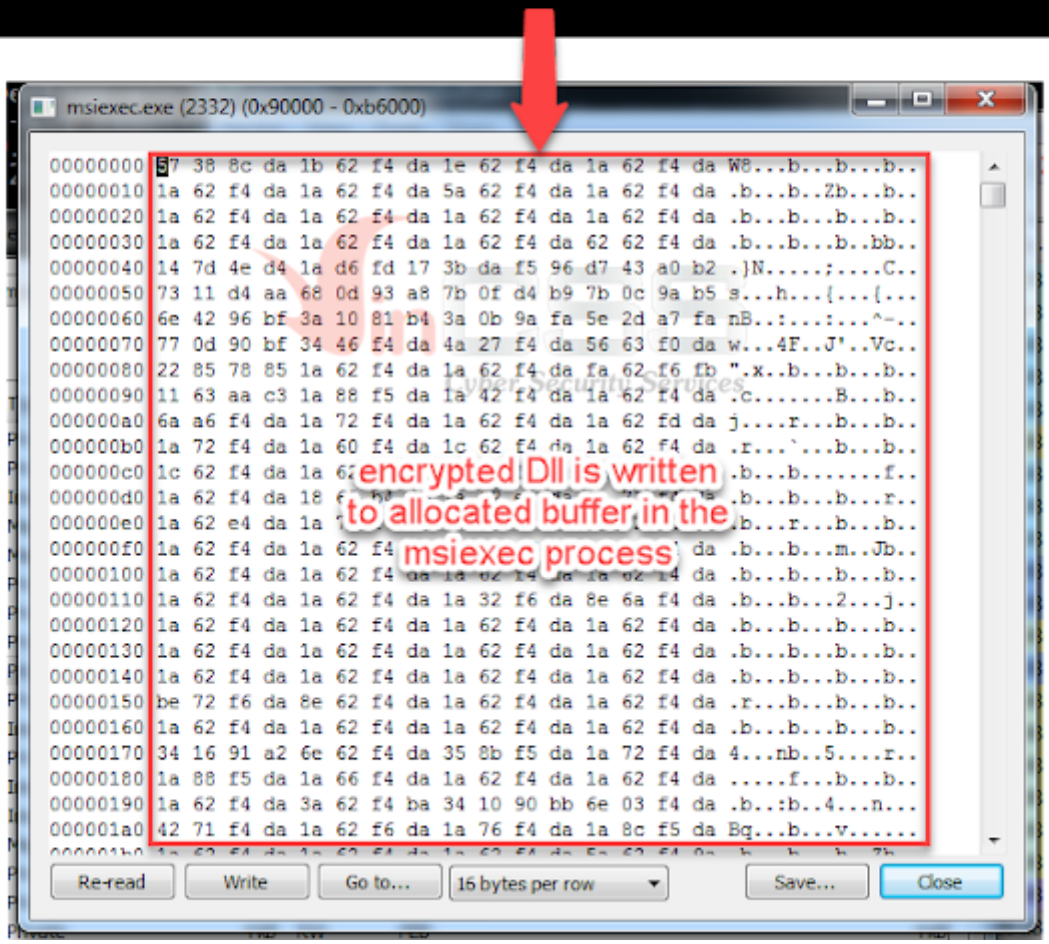
Generate a random number and use it to encrypt the entire payload stored in the heap:

```
*ptr_rand_num = f_zl_generate_random_number();
// encrypt zloader payload that saved at heap region
if ( zl_size_of_image )
{
  rand_num = *ptr_rand_num;
  do
  {
    byte_val = *zl_dll_content_in_heap;
    temp1 = f_zl_and(0x74, ~byte_val);
    LOBYTE(byte_val) = f_zl_and(byte_val, 0x8B);
    temp2 = f_zl_xor(rand_num, 0xFF);
    lpStartAddress = rand_num;
    *zl_dll_content_in_heap = (temp2 & 0x74 | rand_num & 0x8B) ^ f_zl_or(temp1, byte_val);
    val_0x8 = f_zl_xor_arg_with_0xF6233B5A(0xF6233B52);
    ++zl_dll_content_in_heap;
    rand_num = f_zl_xor_arg1_with_arg2_1(lpStartAddress << val_0x8, lpStartAddress >> 0x18);
    --zl_size_of_image;
  }
  while ( zl_size_of_image );
}
```

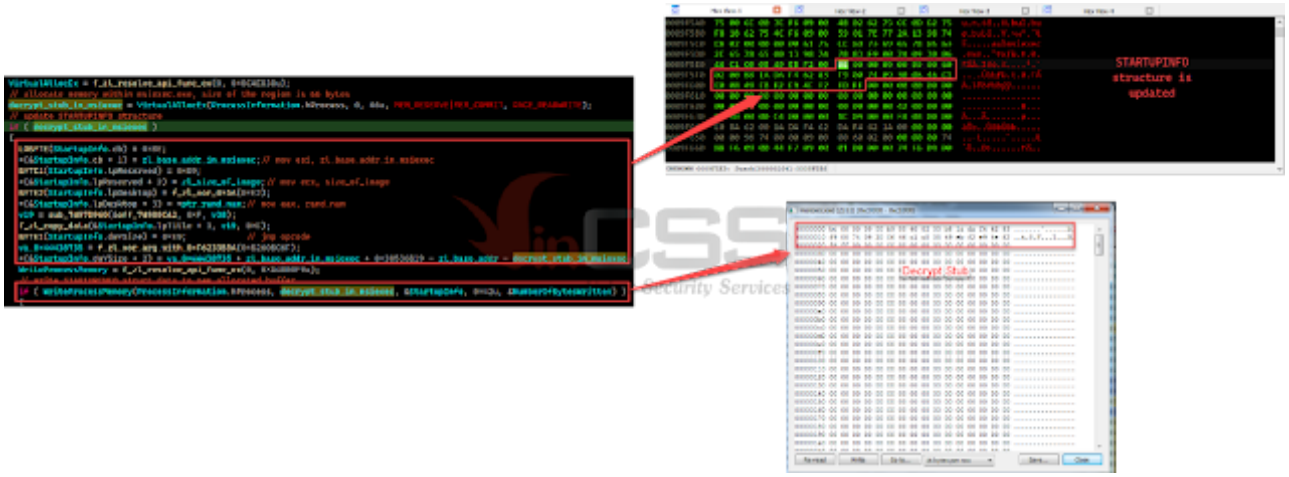


Use the **WriteProcessMemory** API function to write the entire encrypted payload from the heap to the previously allocated memory in the **msiexec.exe** process:

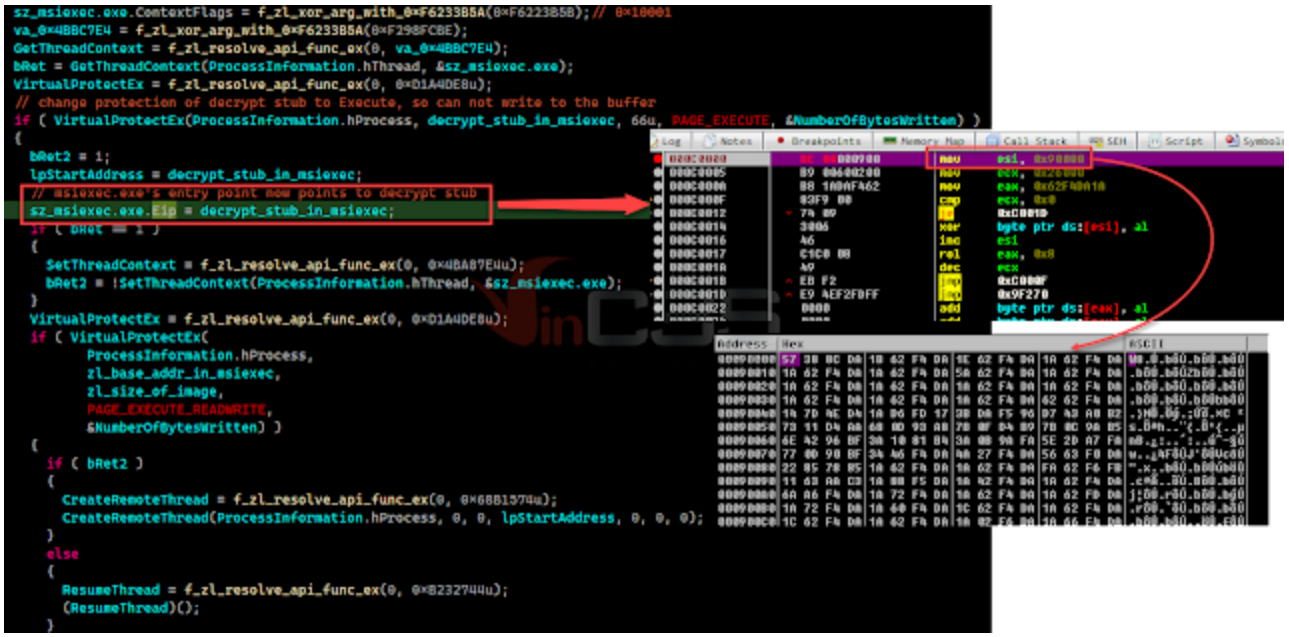
```
NumberOfBytesWritten = 0;
WriteProcessMemory = f_zl_resolve_api_func_ex(0, 0×A48B0F9u);
// write encrypted dll in allocated buffer in msiexec.exe process
if ( WriteProcessMemory(
       ProcessInformation.hProcess,
       zl_base_addr_in_msiexec,
       zl_dll_content_in_heap,
       zl_size_of_image,
       &NumberOfBytesWritten) )
{
```



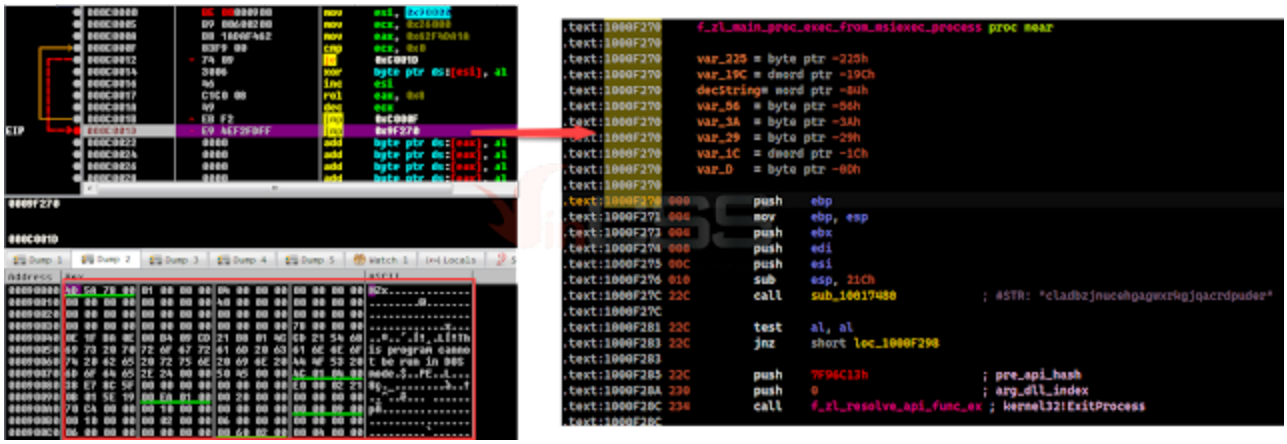
Continue to use the **VirtualAllocEx** API function to allocate a second memory region has size of region are 66 bytes in the **msiexec.exe** process. This memory region will be used to decrypt the entire encrypted Dll above. Update the **STARTUPINFO** structure created by the **CreateProcessA** function before, the data here are the assembly code that will be used to decrypt the encrypted Dll. Then, call the **WriteProcessMemory** function to write the updated contents of **STARTUPINFO** to the newly created memory region.



Finally, use the **GetThreadContext**, **SetThreadContext**, **ResumeThread** or **CreateRemoteThread** API functions to execute the **msiexec.exe** process. At this point, the entry point executed at **msiexec.exe** will be the memory region that containing the code to perform the decrypting mission:

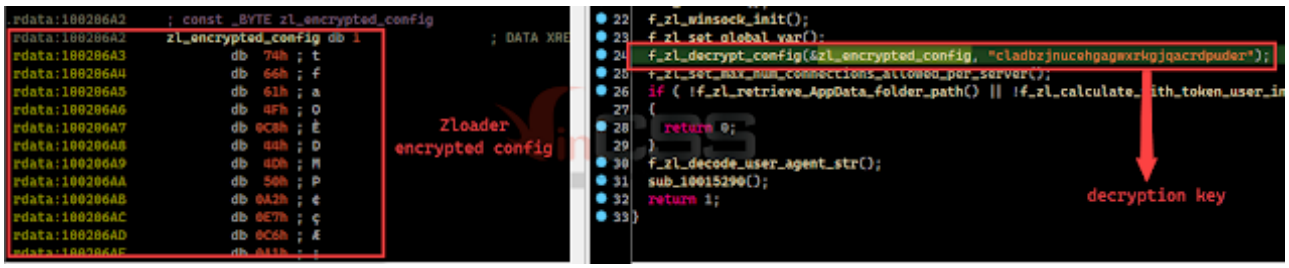


After decrypting the entire Zloader Dll, it will jump to the RVA address of **0xF270** (File offset: **0xE670**) to execute the main tasks of the malware:

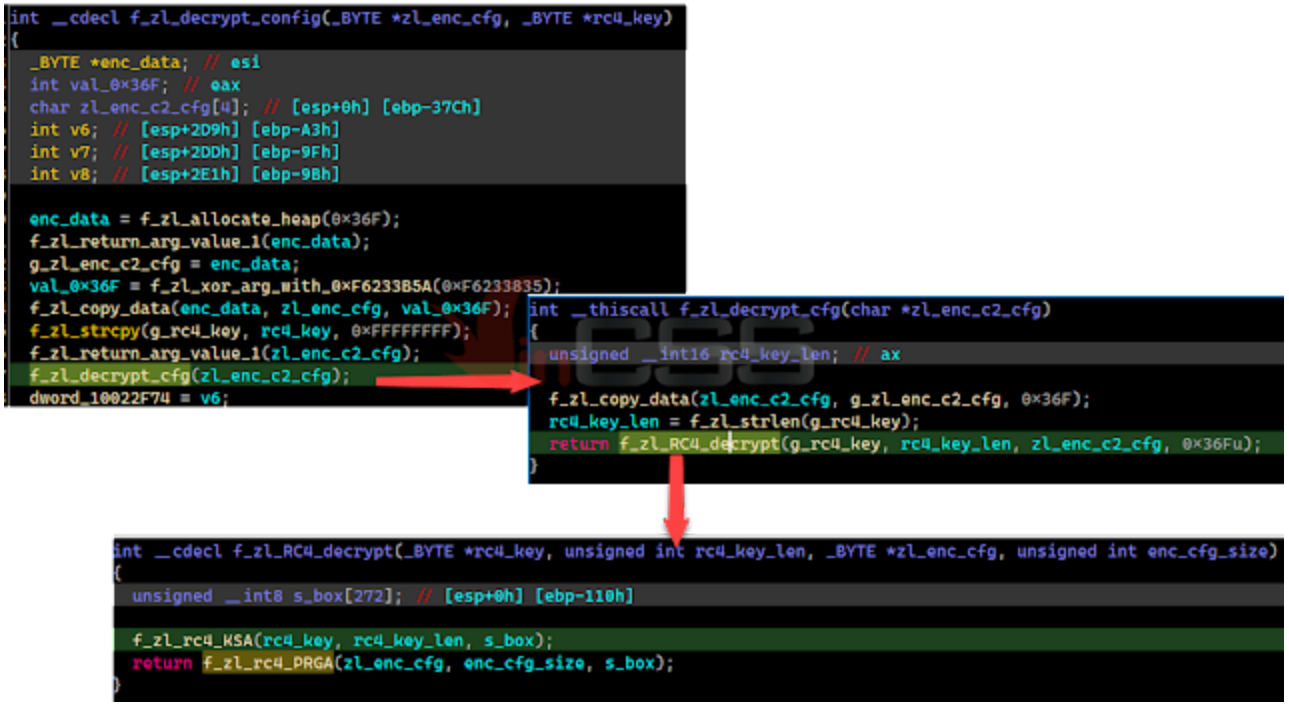


9. Decrypt Zloader config

The configuration info of the Zloader has been encrypted and stored in the **.rdata** section. The decrypt function takes two parameters are the encrypted configuration data and the key used to decrypt:



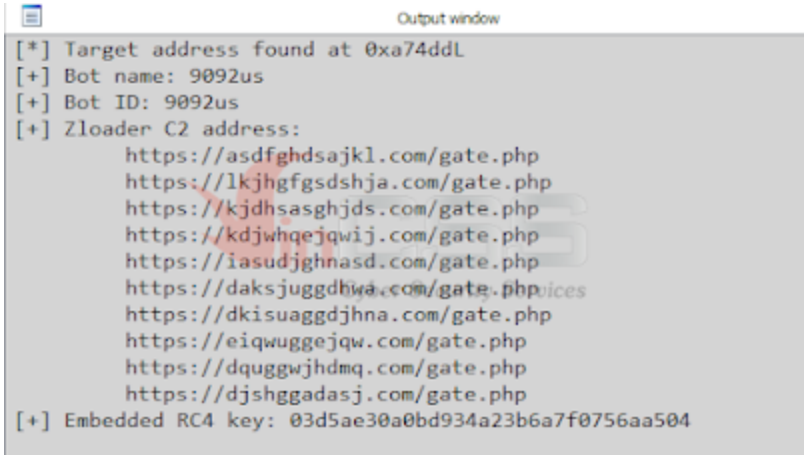
Inside the function **f_zl_decrypt_config** will use the RC4 algorithm to decrypt the data:



With the analyzed results, we can use IDAPython code below to perform the decoding:

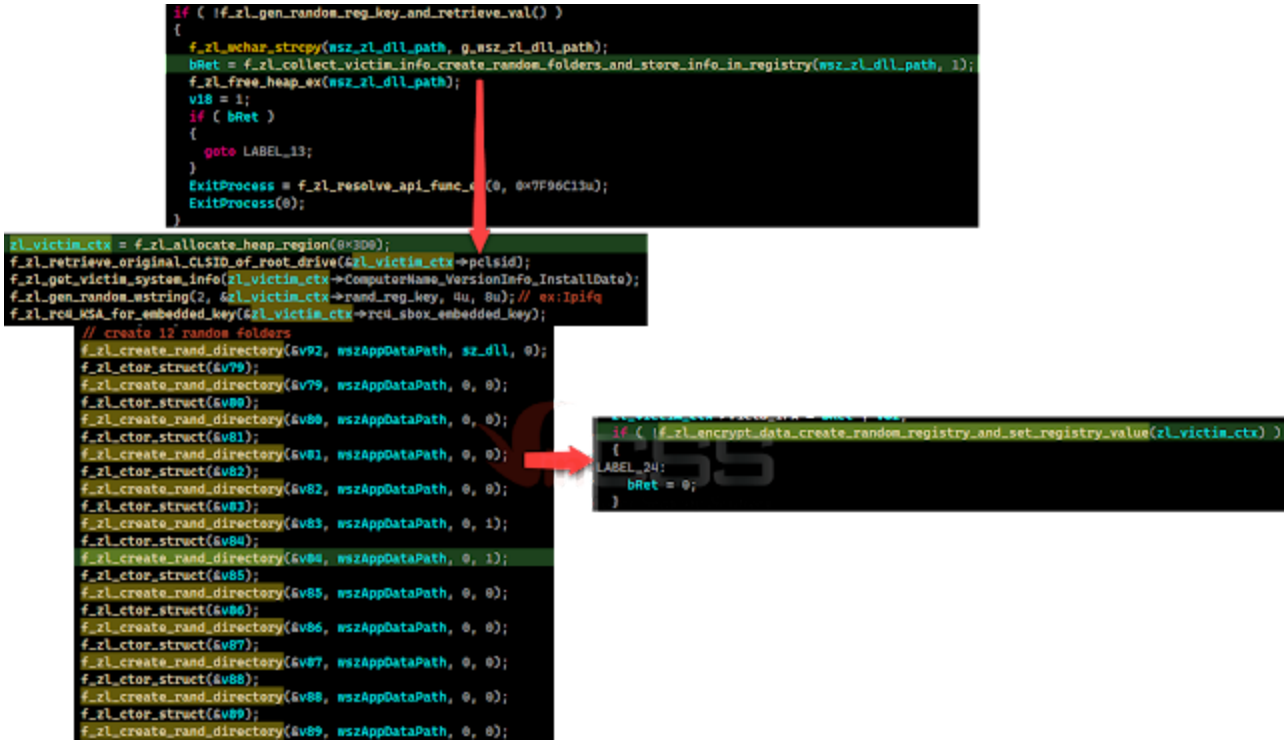
```
import idautils, idc, ida_search

def rc4crypt(data, key):
    """
    Simple rc4 algo. Ref: https://gist.github.com/OALabs/1b07f7ef90e19e77745cad4101af78e9
    """
    x = 0
    box = range(256)
    for i in range(256):
        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i]
    x = 0
    y = 0
    out = []
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))

    return ''.join(out)

def read_all_bytes(addr):
    """
    read encrypted byte from specified address
    """
    enc_cfg = idc.get_bytes(addr, idc.next_head(addr) - addr)

    return enc_cfg

def main():
    seg_mapping = {idc.get_segm_name(x): (idc.get_segm_start(x), idc.get_segm_end(x)) for x in idautils.Segments()}
    start = seg_mapping['.text'][0]
    end = seg_mapping['.text'][1]
    pattern = "68 ?? ?? ?? ?? 68 ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 C4 08 E8 ?? ?? ?? ??"
    addr = ida_search.find_binary(start, end, pattern, 16, idc.SEARCH_DOWN)
    print('[*] Target address found at {}'.format(hex(addr)))

    rc4_key_op = idc.get_operand_value(addr, 0)
    rc4_key = idc.get_bytes(rc4_key_op, idc.get_item_size(rc4_key_op)).rstrip('\x00')

    enc_cfg_op = idc.get_operand_value(idc.next_head(addr),0)
    enc_cfg = read_all_bytes(enc_cfg_op)

    dec_cfg = rc4crypt(enc_cfg, rc4_key)
    cfg_items = filter(None, dec_cfg.split(b"\x00\x00"))
    print ('[+] Bot name: {}'.format(cfg_items[1].lstrip(b"\x00")))
    print ('[+] Bot ID: {}'.format(cfg_items[2].lstrip(b"\x00")))
    print ('[+] Zloader C2 address:')
    for item in cfg_items:
        item = item.lstrip(b"\x00")
        if 'http' in item:
            print ('\t'+ item)
        elif 16 < len(item) <42:
            print ('[+] Embedded RC4 key: {}'.format(item))

if __name__ == '__main__':
    main()
```

Result after executing the script:

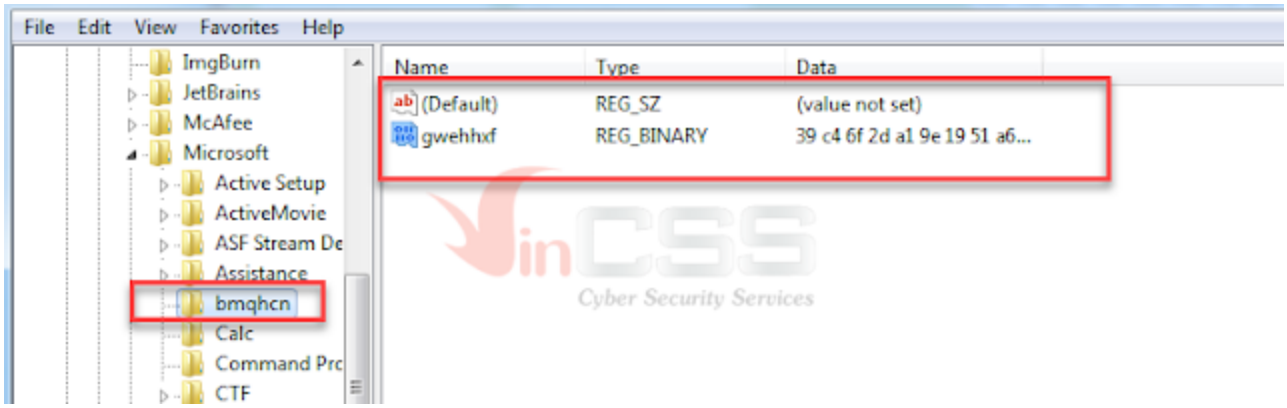


10. Collect and save configuration in Registry

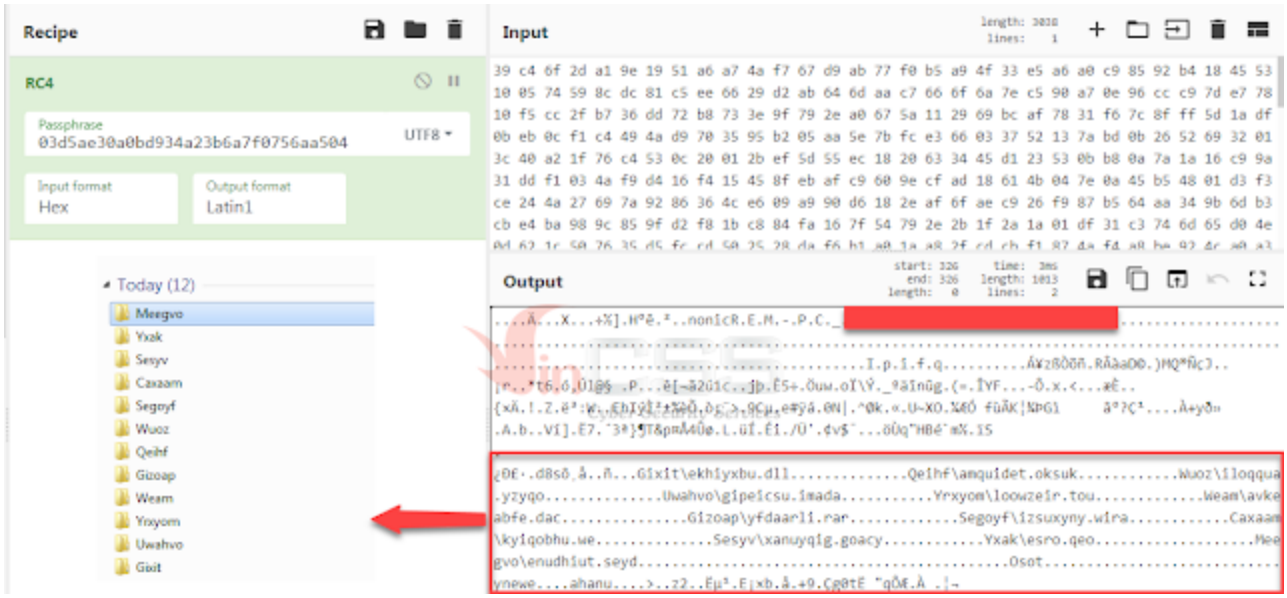
When first executed, Zloader will collect information about the victim including **volume_GUID**, **Computer_Name**, **Windows version**, **Install Date**, create random folders at **%APPDATA%**, generate a random registry key at **HKEY_CURRENT_USERSoftwareMicrosoft**, then encrypt all relevant information and save it in the created registry:



The information stored in the registry is similar to the following:

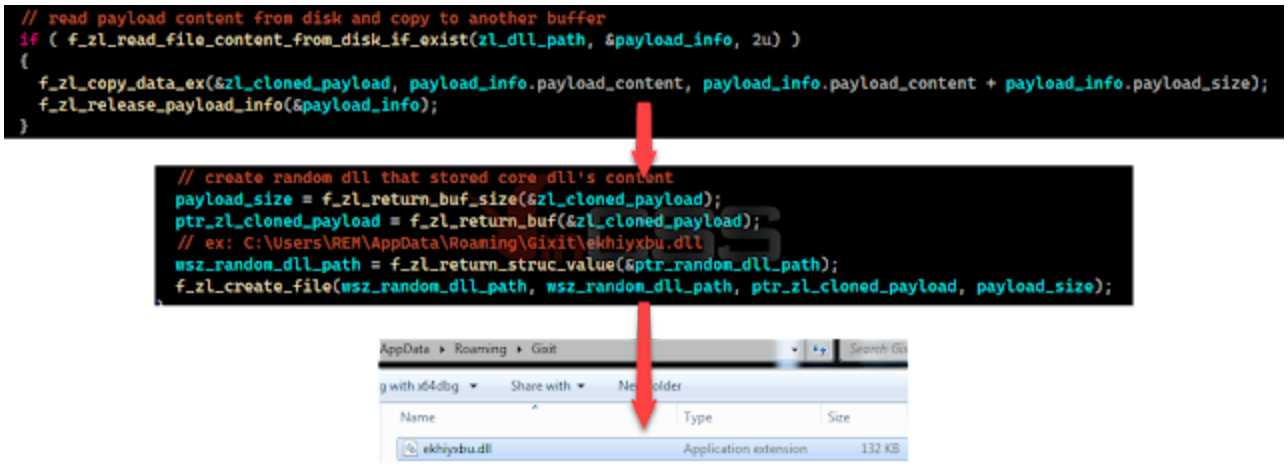


To decrypt the data stored in the above Registry, use the decoded embedded RC4 key above. With the support of **CyberChef**, we can easily decrypt data as follows below:

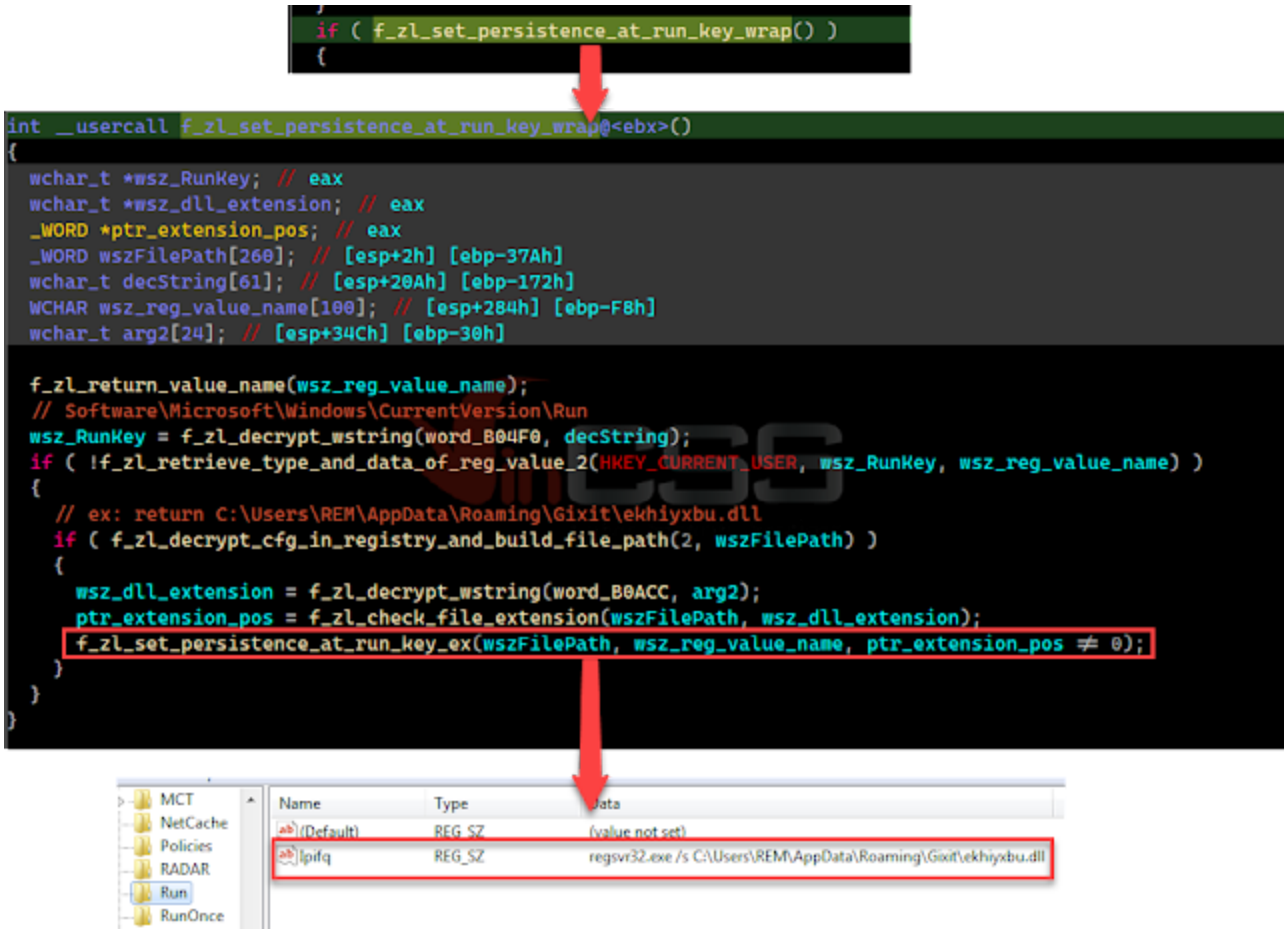


11. Persistence technique

Zloader reads the entire contents of the core Dll from disk into the memory region, then writes to a random dll in a directory created above at **%APPDATA%**:



Create persistence key at **HKEY_CURRENT_USERSoftwareMicrosoftWindowsCurrentVersionRun**:



12. References

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.

09/11/2021

[EX008] The exploit chain allows to take control of Zalo user accounts

While using the Zalo application, one of the popular chat applications in Vietnam today (According to statistics from Wikipedia, since May 2018, Zalo has reached 100 million users), the Threat Hunting team from VinCSS LLC discovered some security vulnerabilities that allow the attacker to form an exploit chain to take control of Zalo accounts.

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 […]

10/08/2021

[EX007] How playing CS: GO helped you bypass security products

Many of us love to play games, and as offensive security engineers, we also want to learn about how game studios are dealing with cheaters. We have observed that cheaters have used vulnerable graphic drivers to bypass anti-cheat mechanisms from several gaming cheating forums. In some cases, the cheaters tried to install vulnerable driver versions onto their computers, then exploited the vulnerability to read and write the game process's memory with the kernel privileges.

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.