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:

Global rank	Week rank	Month rank	IOCs
34	44	↑36	10063

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.

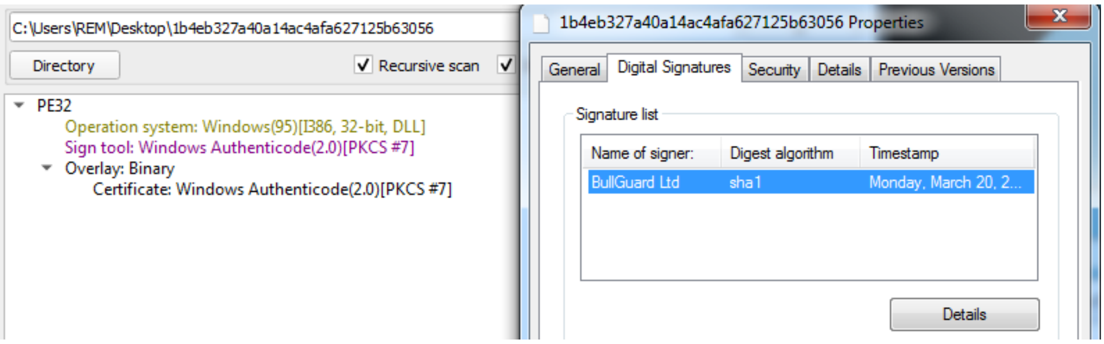
In this article, we will provide detailed analysis and techniques that Zloader uses, including:

- 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.

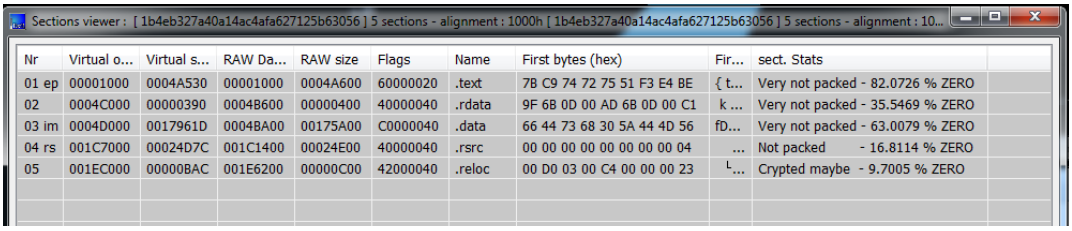
The analyzed sample used in the article: 034f61d86de99210eb32a2dca27a3ad883f54750c46cdec4fcc53050b2f716eb

2. Unpacking Zloader Core Dll

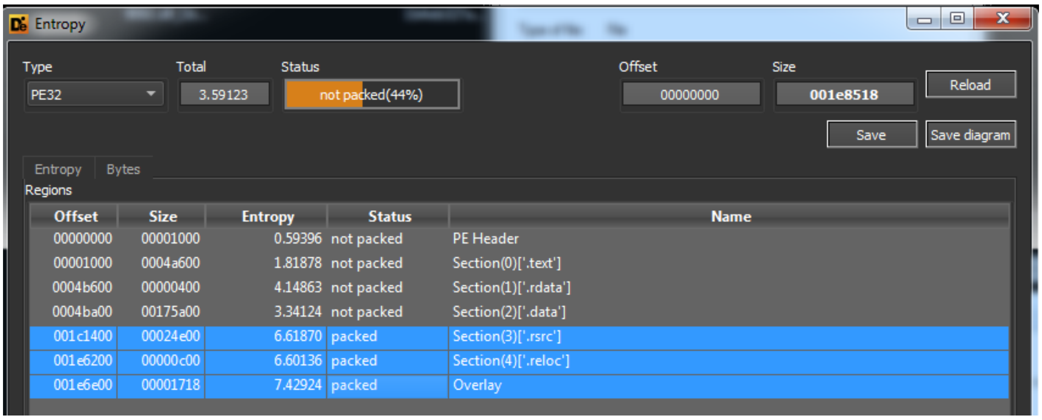
First, check the sample with **Nauz File Detector**:



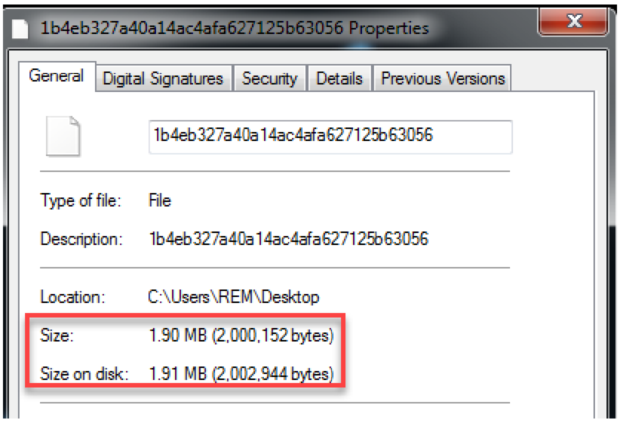
By collecting and combining information about sections from **ExeInfo**, entropy in **DiE** as well as the size of the DLL file, we can confirm that this DLL is packed:



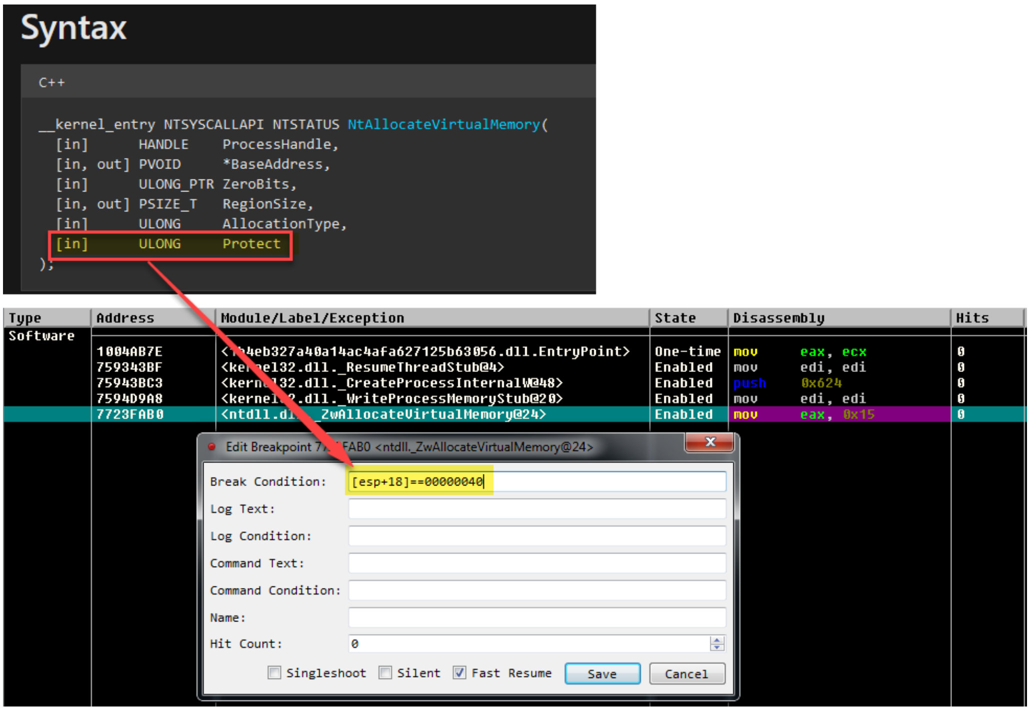
Nr	Virtual o...	Virtual s...	RAW Da...	RAW size	Flags	Name	First bytes (hex)	Fir...	sect. Stats
01 ep	00001000	0004A530	00001000	0004A600	60000020	.text	7B C9 74 72 75 51 F3 E4 BE	{ t...	Very not packed - 82.0726 % ZERO
02	0004C000	00000390	0004B600	00000400	40000040	.rdata	9F 6B 0D 00 AD 6B 0D 00 C1	k ...	Very not packed - 35.5469 % ZERO
03 im	0004D000	0017961D	0004BA00	00175A00	C0000040	.data	66 44 73 68 30 5A 44 4D 56	fD...	Very not packed - 63.0079 % ZERO
04 rs	001C7000	00024D7C	001C1400	00024E00	40000040	.rsrc	00 00 00 00 00 00 00 04	...	Not packed - 16.8114 % ZERO
05	001EC000	00000BAC	001E6200	00000C00	42000040	.reloc	00 D0 03 00 C4 00 00 00 23	└...	Crypted maybe - 9.7005 % ZERO



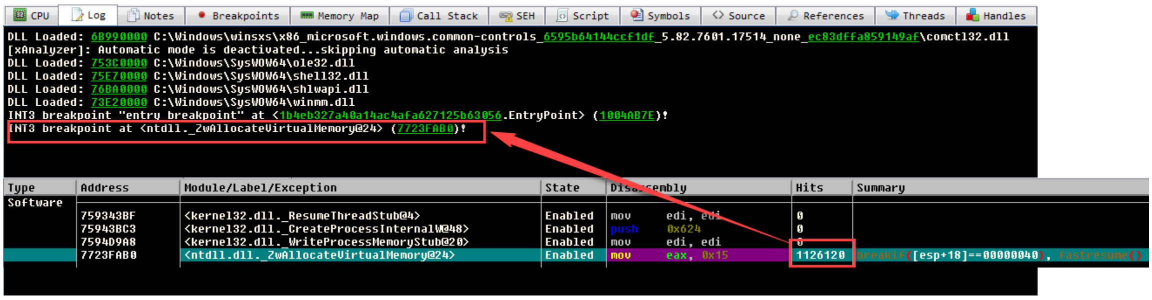
Offset	Size	Entropy	Status	Name
00000000	00001000	0.59396	not packed	PE Header
00001000	0004a600	1.81878	not packed	Section(0)['.text']
0004b600	00000400	4.14863	not packed	Section(1)['.rdata']
0004ba00	00175a00	3.34124	not packed	Section(2)['.data']
001c1400	00024e00	6.61870	packed	Section(3)['.rsrc']
001e6200	00000c00	6.60136	packed	Section(4)['.reloc']
001e6e00	00001718	7.42924	packed	Overlay



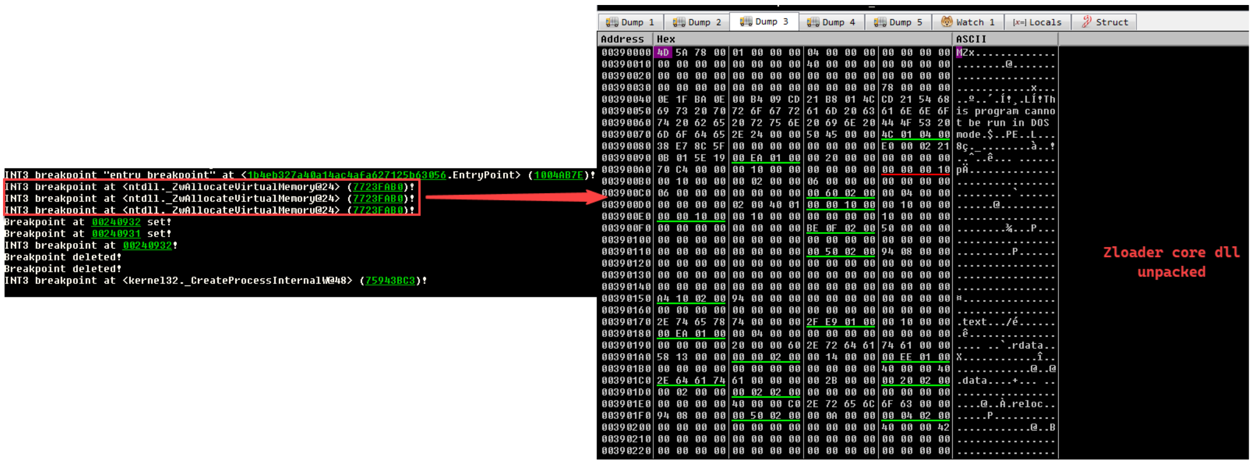
For unpacking, use **x64dbg** to load Dll file, set a `bp NtAllocateVirtualMemory`. Then, modify the breakpoint's condition as follows:



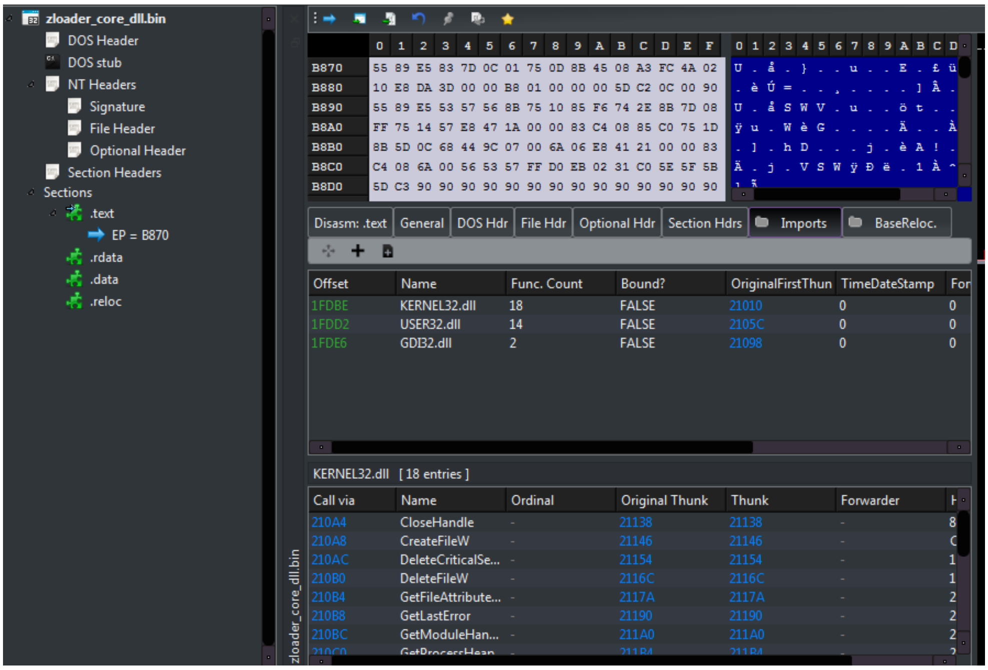
Execute with **F9** and wait until the breakpoint is hit (*after about 1126120 hits*):



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(0x843A5CA6) && sub_10006180(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.

```
.text:1000EDF7 384    add     esp, 4
.text:1000EDFA 380    lea     eax, [ebp+decString]
.text:1000EE00 380    push    eax                     ; decString
.text:1000EE01 384    push    offset word_100204F0    ; encString
.text:1000EE06 388    call    f_zl_decrypt_wstring
.text:1000EE06
.text:1000EE0B 388    add     esp, 8
.text:1000EE0E 380    push    esi
.text:1000EE0F 384    push    eax
.text:1000EE10 388    push    80000001h
.text:1000EE15 38C    call    f_zl_retrieve_type_anc
.text:1000EE15
.text:1000EE1A 38C    add     esp, 0Ch
.text:1000EE1D 380    test    al, al
.text:1000EE1F 380    jnz     short loc_1000EE69
.text:1000EE1F

7 calls, 0 strings

calls:
-             020 call f_zl_return_0x2D5_if_arg1_equal_arg2_else_0x0
-             01C call f_zl_xor_arg_with_0xF6233B5A
-             020 call f_zl_add_arg1_with_arg2
-             01C call f_zl_xor_with_0x3B5A
-             020 call f_zl_and_arg1_with_arg2
-             020 call f_zl_sub_arg1_from_ar2
-             020 call f_zl_sub_arg1_from_ar2
```

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-2ft0j000x`":

```
// xor_key = "PgtrIPF-2ftOj00Ox"
dec_char = *g_PgtrIPF2ftOj00Ox;
LOWORD(dec_char) = *encString ^ dec_char;
*decString = dec_char;                                   // 1st, dec_char = 0x50 (P)
                                                         // decString[0] = encString[0] ^ dec_char
if ( !(_WORD)dec_char )
{
  return ptr_decString;
}
i = 0;
while ( 1 )
{
  val_0x20 = f_zl_sub_arg1_from_ar2(0, 0xFFE0);
  if ( (unsigned __int16)f_zl_sub_arg1_from_ar2(0, val_0x20 - dec_char) ≥ 0x5Fu )
  {
    if ( (unsigned __int16)dec_char > 0xDu )
    {
      break;
    }
    v11 = 0x2600;
    if ( !_bittest(&v11, dec_char) )
    {
      break;
    }
  }
  val_0x917C8E60 = f_zl_xor_arg_with_0xF6233B5A(0x675FB53A);
  // i++
  i = f_zl_add_arg1_with_arg2(i + 0x6E8371A1, val_0x917C8E60);
  enc_char = ptr_encString[i];
  xor_key_val = g_PgtrIPF2ftOj00Ox[i % 0x11];
  // ~xor_key_val & 0x476C
  tmp1 = ~xor_key_val & f_zl_xor_with_0x3B5A(0x7C36);
  // xor_key_val & 0xB893
  tmp2 = f_zl_and_arg1_with_arg2(xor_key_val, 0xB893);
  ptr_decString = decString;
  // dec_char = xor_key_val ^ 0x476C
  dec_char = tmp1 | tmp2;                                dec_char = enc_char ^ xor_key_val
  ptr_encString = encString;
  // finally:
  // dec_char = (enc_char ^ xor_key_val)
  LOWORD(dec_char) = enc_char ^ dec_char ^ 0x476C;
  decString[i] = dec_char;
  if ( !(_WORD)dec_char )
  {
    return ptr_decString;
  }
}
return ptr_encString;
}
```

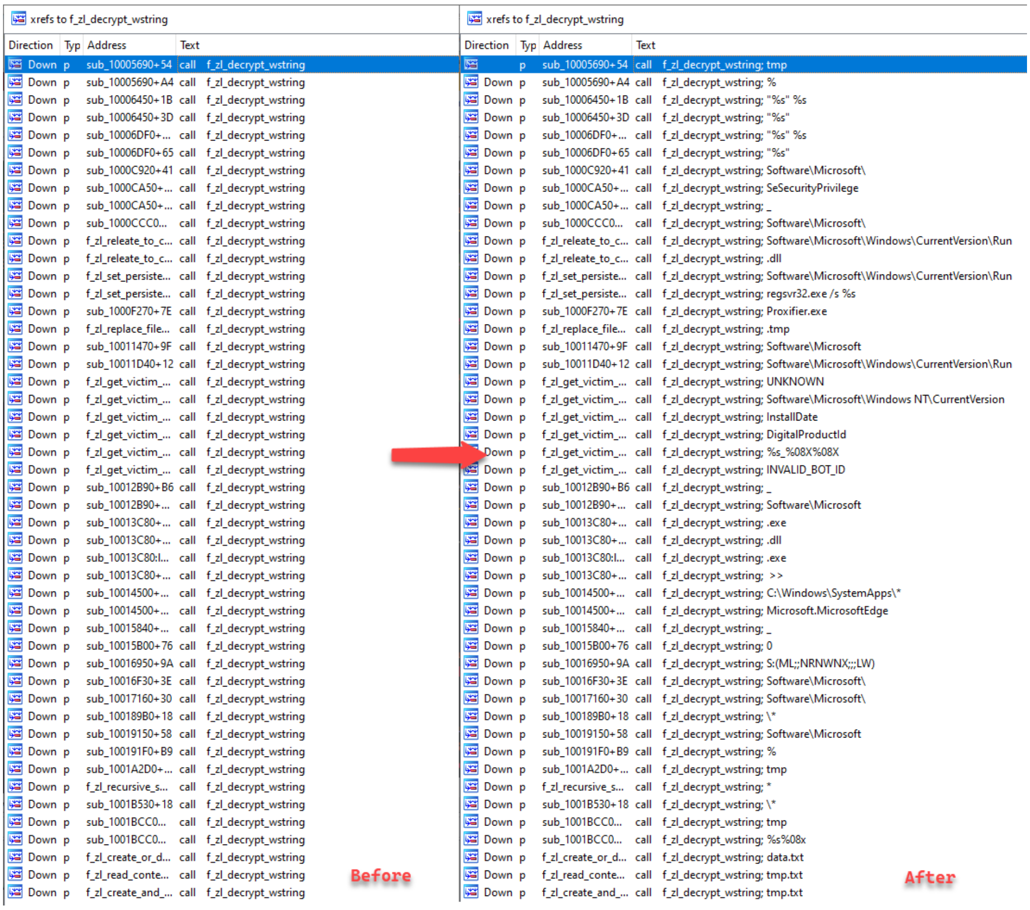
Based on the above pseudocode, the python code that performs decryption as follows:

```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:

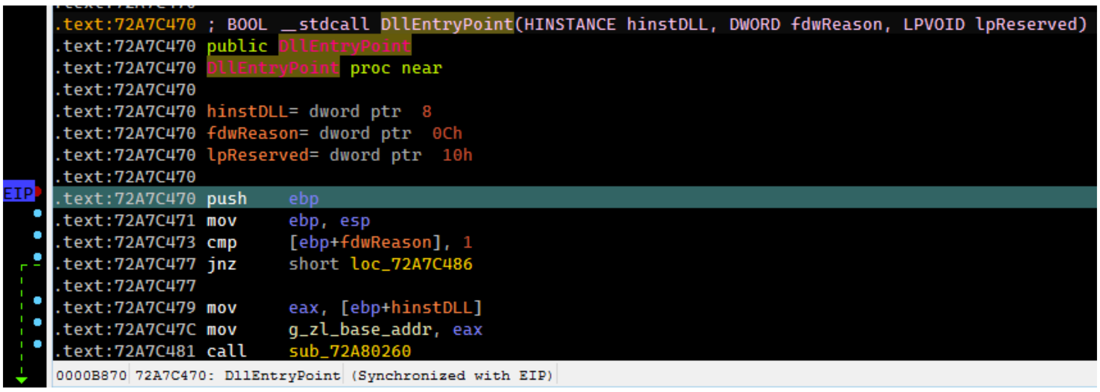


4.2. Use IDA AppCall

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);`

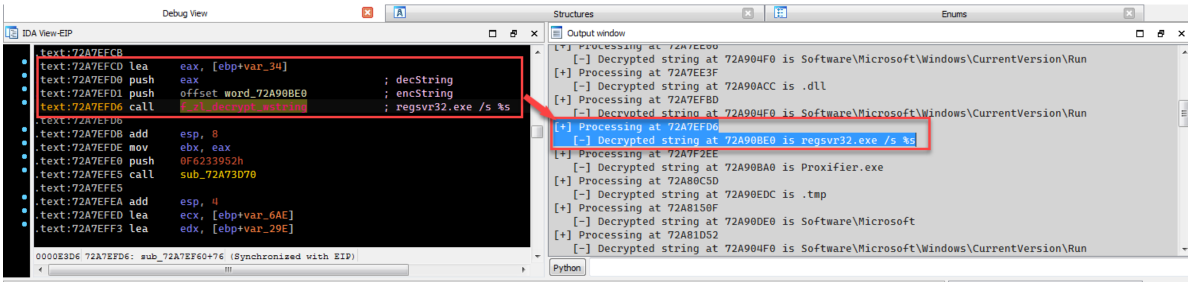
Note again that in order to use AppCall, the program must be debugged. As shown below, IDA is stopping at the breakpoint set at `DllEntryPoint`:



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

```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('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.

```
.text:1001081F 098          lea     eax, [ebp+var_94]
.text:10010825 098          push    eax                      ; decString
.text:10010826 09C          push    offset byte_10020CC0     ; encString
.text:1001082B 0A0          call    f_zl_decrypt_string
.text:1001082B
.text:10010830 0A0          add     esp, 8
.text:10010833 098          mov     esi, edi
.text:10010835 098          mov     [ebp+var_34], eax
.text:10010838 098          neg     esi
.text:1001083A 098          push    1
.text:1001083C 09C          push    0
.text:1001083E 0A0          call    f_zl_sub_arg1_from_
.text:1001083E
.text:10010843 0A0          add     esp, 8
.text:10010846 098          push    eax
```

```
7 calls, 0 strings

calls:
 -          020 call f_zl_xor
 -          01C call f_zl_xor_0x5A
 -          020 call f_zl_and
 -          020 call f_zl_xor
 -          020 call f_zl_or
 -          020 call f_zl_or
 -          020 call f_zl_return_0x0_if_arg1_not_equal_arg2
```

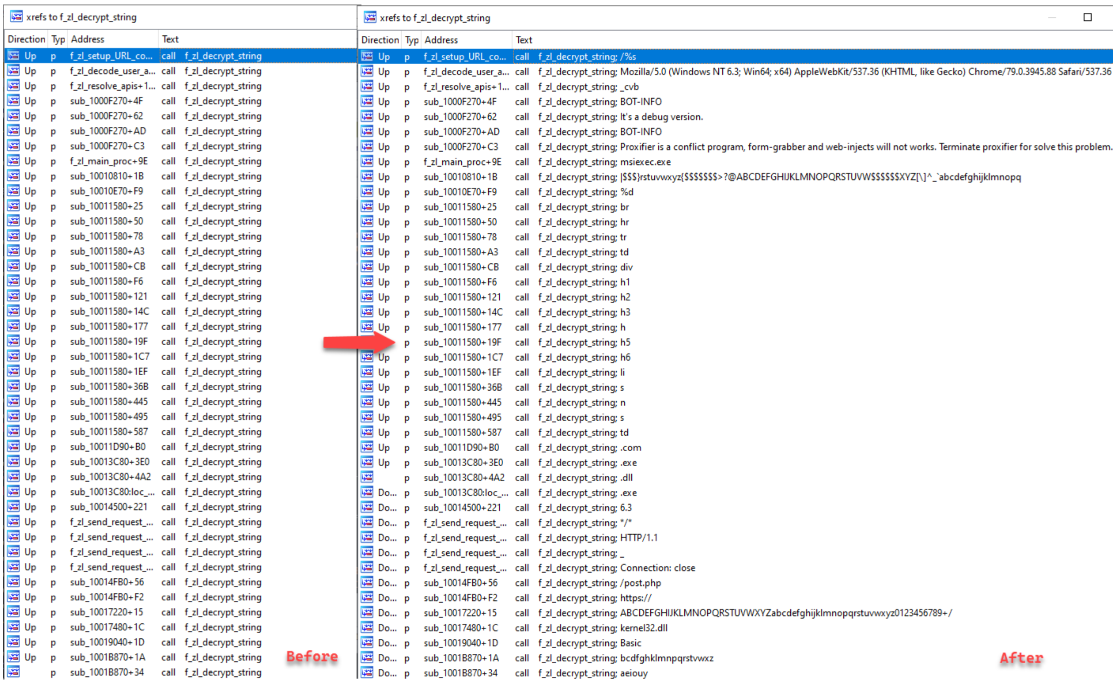
Similar to the above `f_zl_decrypt_wstring` function, the pseudocode of the `f_zl_decrypt_string` function looks quite messy, but it still uses an xor loop to decrypt with the decryption key still "`PgtrIPF-2ftOj000x`":

```
enc_char = *encString;
v3 = ~*encString;
// xor_key = "PgtrIPF-2ftOj000x"
xor_key_val_0x50 = *g_PgtrIPF2ftOj000x;
val_0xAF = f_zl_xor(*g_PgtrIPF2ftOj000x, 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_PgtrIPF2ftOj000x[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-2ftOj0O0x'

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 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



xrefs to f_zl_decrypt_string

Direction	Typ	Address	Text
Up	p	f_zl_setup_URL_co...	call f_zl_decrypt_string
Up	p	f_zl_decode_user_a...	call f_zl_decrypt_string
Up	p	f_zl_resolve_apis+1...	call f_zl_decrypt_string
Up	p	sub_1000F270+4F	call f_zl_decrypt_string
Up	p	sub_1000F270+62	call f_zl_decrypt_string
Up	p	sub_1000F270+AD	call f_zl_decrypt_string
Up	p	sub_1000F270+C3	call f_zl_decrypt_string
Up	p	f_zl_main_proc+9E	call f_zl_decrypt_string
Up	p	sub_10010810+1B	call f_zl_decrypt_string
Up	p	sub_10010E70+F9	call f_zl_decrypt_string
Up	p	sub_10011580+25	call f_zl_decrypt_string
Up	p	sub_10011580+50	call f_zl_decrypt_string
Up	p	sub_10011580+78	call f_zl_decrypt_string
Up	p	sub_10011580+A3	call f_zl_decrypt_string
Up	p	sub_10011580+CB	call f_zl_decrypt_string
Up	p	sub_10011580+F6	call f_zl_decrypt_string
Up	p	sub_10011580+121	call f_zl_decrypt_string
Up	p	sub_10011580+14C	call f_zl_decrypt_string
Up	p	sub_10011580+177	call f_zl_decrypt_string
Up	p	sub_10011580+19F	call f_zl_decrypt_string
Up	p	sub_10011580+1C7	call f_zl_decrypt_string
Up	p	sub_10011580+1EF	call f_zl_decrypt_string
Up	p	sub_10011580+36B	call f_zl_decrypt_string
Up	p	sub_10011580+445	call f_zl_decrypt_string
Up	p	sub_10011580+495	call f_zl_decrypt_string
Up	p	sub_10011580+587	call f_zl_decrypt_string
Up	p	sub_10011D90+B0	call f_zl_decrypt_string
Up	p	sub_10013C80+3E0	call f_zl_decrypt_string
Up	p	sub_10013C80+4A2	call f_zl_decrypt_string
Up	p	sub_10013C80:loc_...	call f_zl_decrypt_string
Up	p	sub_10014500+221	call f_zl_decrypt_string
Up	p	f_zl_send_request_...	call f_zl_decrypt_string
Up	p	f_zl_send_request_...	call f_zl_decrypt_string
Up	p	f_zl_send_request_...	call f_zl_decrypt_string
Up	p	f_zl_send_request_...	call f_zl_decrypt_string
Up	p	sub_10014FB0+56	call f_zl_decrypt_string
Up	p	sub_10014FB0+F2	call f_zl_decrypt_string
Up	p	sub_10017220+15	call f_zl_decrypt_string
Up	p	sub_10017480+1C	call f_zl_decrypt_string
Up	p	sub_10019040+1D	call f_zl_decrypt_string
Up	p	sub_1001B870+1A	call f_zl_decrypt_string
	p	sub_1001B870+34	call f_zl_decrypt_string

Before

xrefs to f_zl_decrypt_string

Direction	Typ	Address	Text
Up	p	f_zl_setup_URL_co...	call f_zl_decrypt_string; /%s
Up	p	f_zl_decode_user_a...	call f_zl_decrypt_string; Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36
Up	p	f_zl_resolve_apis+1...	call f_zl_decrypt_string; _cvb
Up	p	sub_1000F270+4F	call f_zl_decrypt_string; BOT-INFO
Up	p	sub_1000F270+62	call f_zl_decrypt_string; It's a debug version.
Up	p	sub_1000F270+AD	call f_zl_decrypt_string; BOT-INFO
Up	p	sub_1000F270+C3	call f_zl_decrypt_string; Proxifier is a conflict program, form-grabber and web-injects will not works. Terminate proxifier for solve this problem.
Up	p	f_zl_main_proc+9E	call f_zl_decrypt_string; msiexec.exe
Up	p	sub_10010810+1B	call f_zl_decrypt_string; |$$$}rstuvwxyz{$$$$$$$>?@ABCDEFGHIJKLMNOPQRSTUVW$$$$$$XYZ[\]^_`abcdefghijklmnopq
Up	p	sub_10010E70+F9	call f_zl_decrypt_string; %d
Up	p	sub_10011580+25	call f_zl_decrypt_string; br
Up	p	sub_10011580+50	call f_zl_decrypt_string; hr
Up	p	sub_10011580+78	call f_zl_decrypt_string; tr
Up	p	sub_10011580+A3	call f_zl_decrypt_string; td
Up	p	sub_10011580+CB	call f_zl_decrypt_string; div
Up	p	sub_10011580+F6	call f_zl_decrypt_string; h1
Up	p	sub_10011580+121	call f_zl_decrypt_string; h2
Up	p	sub_10011580+14C	call f_zl_decrypt_string; h3
Up	p	sub_10011580+177	call f_zl_decrypt_string; h
	p	sub_10011580+19F	call f_zl_decrypt_string; h5
Up	p	sub_10011580+1C7	call f_zl_decrypt_string; h6
Up	p	sub_10011580+1EF	call f_zl_decrypt_string; li
Up	p	sub_10011580+36B	call f_zl_decrypt_string; s
Up	p	sub_10011580+445	call f_zl_decrypt_string; n
Up	p	sub_10011580+495	call f_zl_decrypt_string; s
Up	p	sub_10011580+587	call f_zl_decrypt_string; td
Up	p	sub_10011D90+B0	call f_zl_decrypt_string; .com
Up	p	sub_10013C80+3E0	call f_zl_decrypt_string; .exe
	p	sub_10013C80+4A2	call f_zl_decrypt_string; .dll
Do...	p	sub_10013C80:loc_...	call f_zl_decrypt_string; .exe
Do...	p	sub_10014500+221	call f_zl_decrypt_string; 6.3
Do...	p	f_zl_send_request_...	call f_zl_decrypt_string; */*
Do...	p	f_zl_send_request_...	call f_zl_decrypt_string; HTTP/1.1
Do...	p	f_zl_send_request_...	call f_zl_decrypt_string; _
Do...	p	f_zl_send_request_...	call f_zl_decrypt_string; Connection: close
Do...	p	sub_10014FB0+56	call f_zl_decrypt_string; /post.php
Do...	p	sub_10014FB0+F2	call f_zl_decrypt_string; https://
Do...	p	sub_10017220+15	call f_zl_decrypt_string; ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
Do...	p	sub_10017480+1C	call f_zl_decrypt_string; kernel32.dll
Do...	p	sub_10019040+1D	call f_zl_decrypt_string; Basic
Do...	p	sub_1001B870+1A	call f_zl_decrypt_string; bcdfghklmnpqrstvwxz
Do...	p	sub_1001B870+34	call f_zl_decrypt_string; aeiouy

After

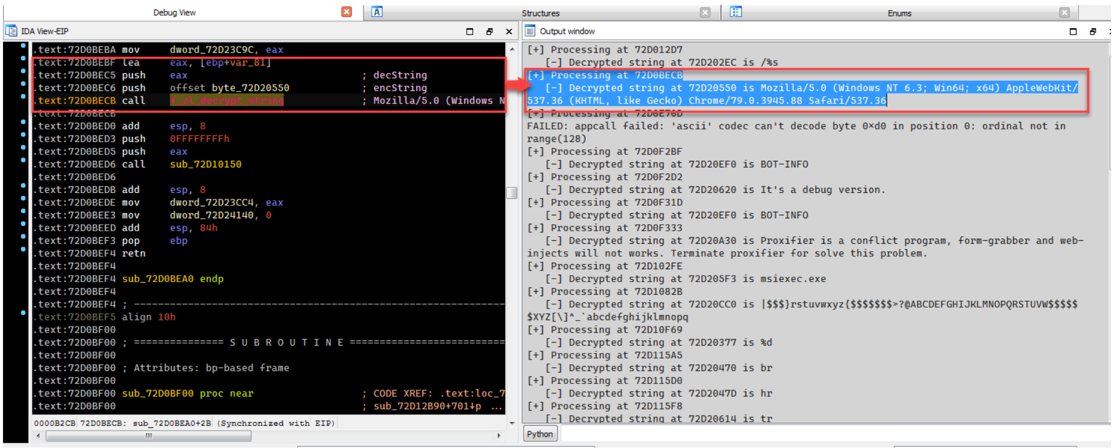
5.2. Use IDA AppCall

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:

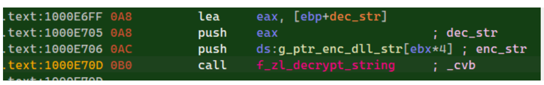
```python
import idc, idaapi, idautils

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)
```

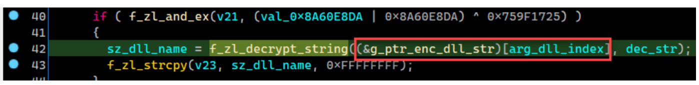
Result after running the script:



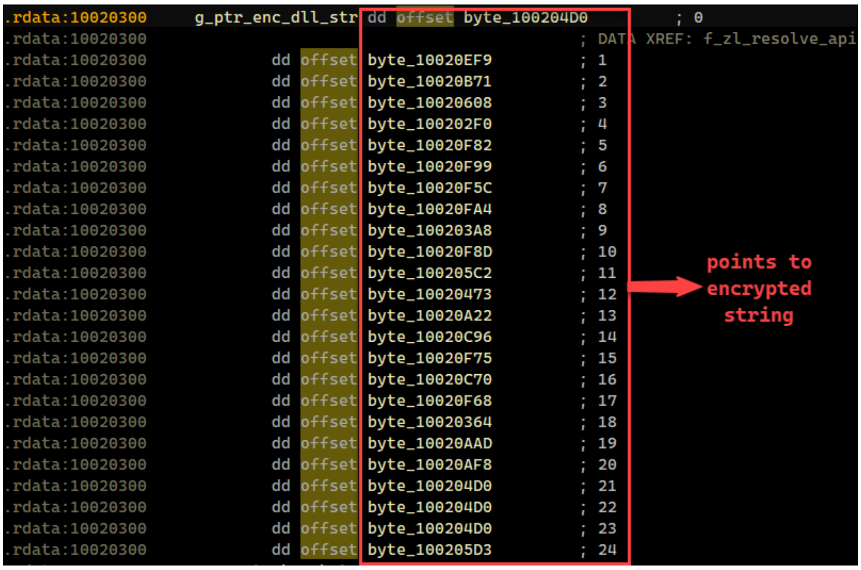
6. List of Dlls used by Zloader

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:



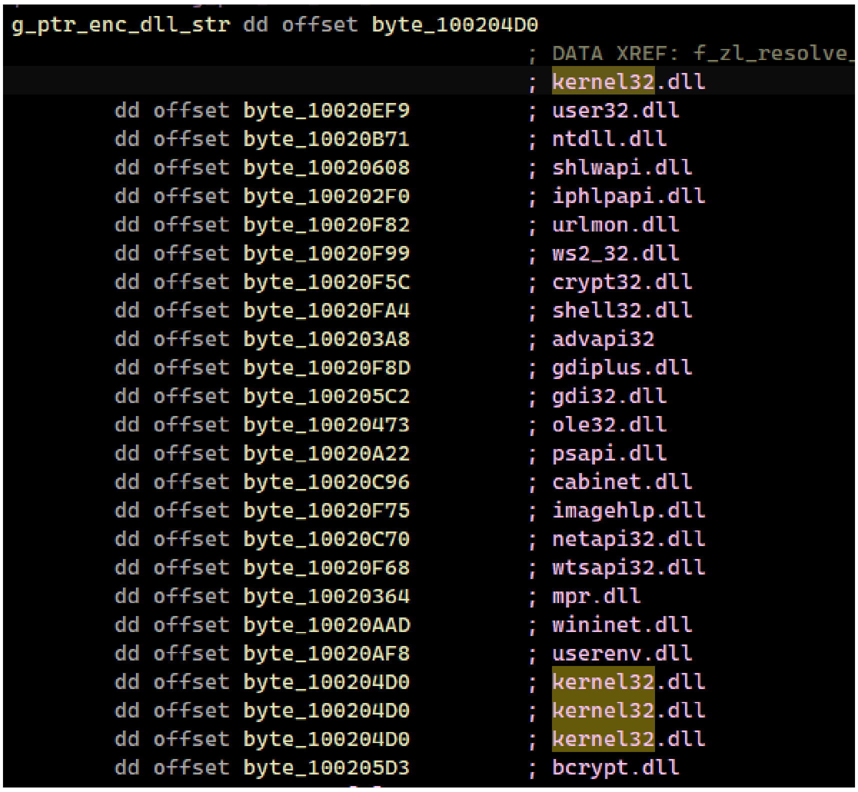


Going to the `g_ptr_enc_dll_str` array (*renamed above*) will see a list of addresses as shown below:



```
.rdata:10020300      g_ptr_enc_dll_str dd offset byte_100204D0      ; 0
.rdata:10020300                                                     ; DATA XREF: f_zl_resolve_api
.rdata:10020300               dd offset byte_10020EF9               ; 1
.rdata:10020300               dd offset byte_10020B71               ; 2
.rdata:10020300               dd offset byte_10020608               ; 3
.rdata:10020300               dd offset byte_100202F0               ; 4
.rdata:10020300               dd offset byte_10020F82               ; 5
.rdata:10020300               dd offset byte_10020F99               ; 6
.rdata:10020300               dd offset byte_10020F5C               ; 7
.rdata:10020300               dd offset byte_10020FA4               ; 8
.rdata:10020300               dd offset byte_100203A8               ; 9
.rdata:10020300               dd offset byte_10020F8D               ; 10
.rdata:10020300               dd offset byte_100205C2               ; 11
.rdata:10020300               dd offset byte_10020473               ; 12
.rdata:10020300               dd offset byte_10020A22               ; 13
.rdata:10020300               dd offset byte_10020C96               ; 14
.rdata:10020300               dd offset byte_10020F75               ; 15
.rdata:10020300               dd offset byte_10020C70               ; 16
.rdata:10020300               dd offset byte_10020F68               ; 17
.rdata:10020300               dd offset byte_10020364               ; 18
.rdata:10020300               dd offset byte_10020AAD               ; 19
.rdata:10020300               dd offset byte_10020AF8               ; 20
.rdata:10020300               dd offset byte_100204D0               ; 21
.rdata:10020300               dd offset byte_100204D0               ; 22
.rdata:10020300               dd offset byte_100204D0               ; 23
.rdata:10020300               dd offset byte_100205D3               ; 24
```


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



```
g_ptr_enc_dll_str dd offset byte_100204D0
                                               ; DATA XREF: f_zl_resolve_
                                               ; kernel32.dll
          dd offset byte_10020EF9              ; user32.dll
          dd offset byte_10020B71              ; ntdll.dll
          dd offset byte_10020608              ; shlwapi.dll
          dd offset byte_100202F0              ; iphlpapi.dll
          dd offset byte_10020F82              ; urlmon.dll
          dd offset byte_10020F99              ; ws2_32.dll
          dd offset byte_10020F5C              ; crypt32.dll
          dd offset byte_10020FA4              ; shell32.dll
          dd offset byte_100203A8              ; advapi32
          dd offset byte_10020F8D              ; gdiplus.dll
          dd offset byte_100205C2              ; gdi32.dll
          dd offset byte_10020473              ; ole32.dll
          dd offset byte_10020A22              ; psapi.dll
          dd offset byte_10020C96              ; cabinet.dll
          dd offset byte_10020F75              ; imagehlp.dll
          dd offset byte_10020C70              ; netapi32.dll
          dd offset byte_10020F68              ; wtsapi32.dll
          dd offset byte_10020364              ; mpr.dll
          dd offset byte_10020AAD              ; wininet.dll
          dd offset byte_10020AF8              ; userenv.dll
          dd offset byte_100204D0              ; kernel32.dll
          dd offset byte_100204D0              ; kernel32.dll
          dd offset byte_100204D0              ; kernel32.dll
          dd offset byte_100205D3              ; bcrypt.dll
```

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.

```
.text:1001029E
.text:100102A4 57C          push    0FDA8B77h               ; pre_api_hash
.text:100102A9 580          push    0                       ; arg_dll_index
.text:100102AB 584          call    f_zl_resolve_api_func_ex    retrieve api address
.text:100102AB
.text:100102B0 584          add     esp, 8
.text:100102B3 57C          lea     esi, [ebp+var_578]
.text:100102B9 57C          push    104h                    ; nSize
.text:100102BE 580          push    esi                     ; lpFilename
.text:100102BF 584          push    g_zl_base_addr          ; hModule
.text:100102C5 588          call    eax                         call api function
```

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                                                          convert API name to lowercase
    {
      *(&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;                                                      compare calculated hash to pre_hash
  }
  ++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);
```

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_0xF6233B5A(0x7472BBA7);
      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_0xF6233B5A(0xE9AAFDBB) - 0x1F89C6E0;
    ++inString;
  }
  while ( i ≠ strLen );
  return calced_hash;
}
```

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

```python
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
```

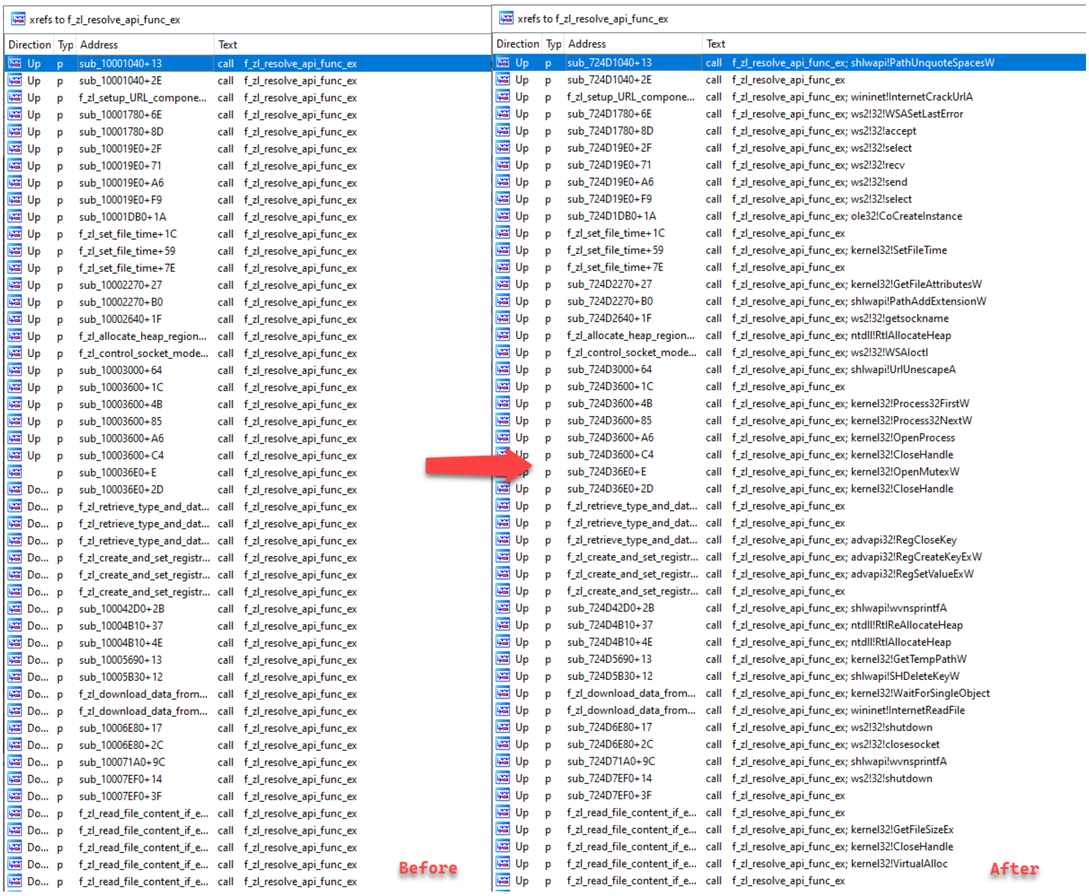
With all the above analysis results, it is possible to write an IDAPython script to recover all the APIs that Zloader uses. However, to avoid having to dig into Zloader's hashing algorithm for each analysis, here I will use AppCall to do this task. The python code that uses AppCall is as follows:

```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)
```

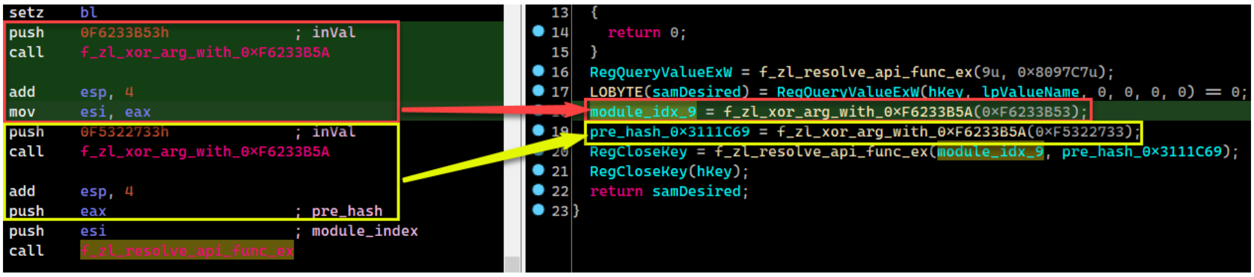
Note, Zloader has many areas of code that call to the **f_zl_resolve_api_func_ex** function, but there will be areas of code that do not have any reference to it and that area has not been defined as a complete function. Therefore, to be able to run the above script, it is necessary to create functions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

```
.text:724E029E
.text:724E02A4 57C    push  0FDA8B77h                 ; pre_api_hash
.text:724E02A9 580    push  0                         ; arg_dll_index
.text:724E02AB 584    call  f_zl_resolve_api_func_ex  ; kernel32!GetModuleFileNameW
.text:724E02AB
.text:724E02B0 584    add   esp, 8
.text:724E02B3 57C    lea   esi, [ebp+var_578]
.text:724E02B9 57C    push  104h                      ; nSize
.text:724E02BE 580    push  esi                       ; lpFilename
.text:724E02BF 584    push  g_zl_base_addr            ; hModule
.text:724E02C5 588    call  eax                       ; kernel32!GetModuleFileNameW
.text:724E02C5
```

```
[+] Processing at 724E02AB
    [-] Module index: 00000000
    [-] Precalculated hash: 0FDA8B77
    [-] Resolved API: kernel32_GetModuleFileNameW
[+] Processing at 724E031F
    [-] Module index: 00000000
    [-] Precalculated hash: 01E16041
    [-] Resolved API: kernel32_CreateProcessA
[+] Processing at 724E0363
[+] Processing at 724E0485
    [-] Module index: 00000000
    [-] Precalculated hash: 0A48B0F9
```



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) )
{
```

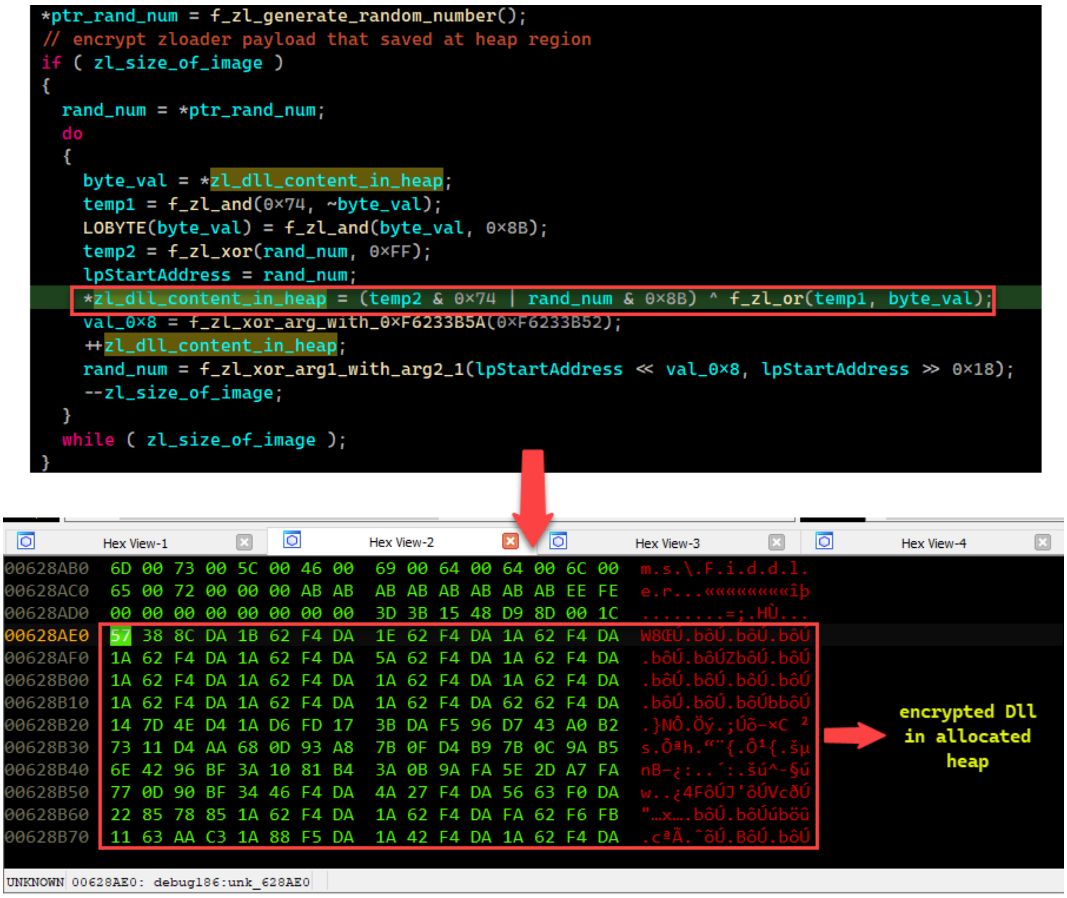
♦ Get `SizeOfImage` value of Zloader Dll being loaded by `rundll32.exe/regsvr32.exe`. Use the `VirtualAllocEx` API function to allocate new memory inside the `msiexec.exe` process:

```
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 )
```

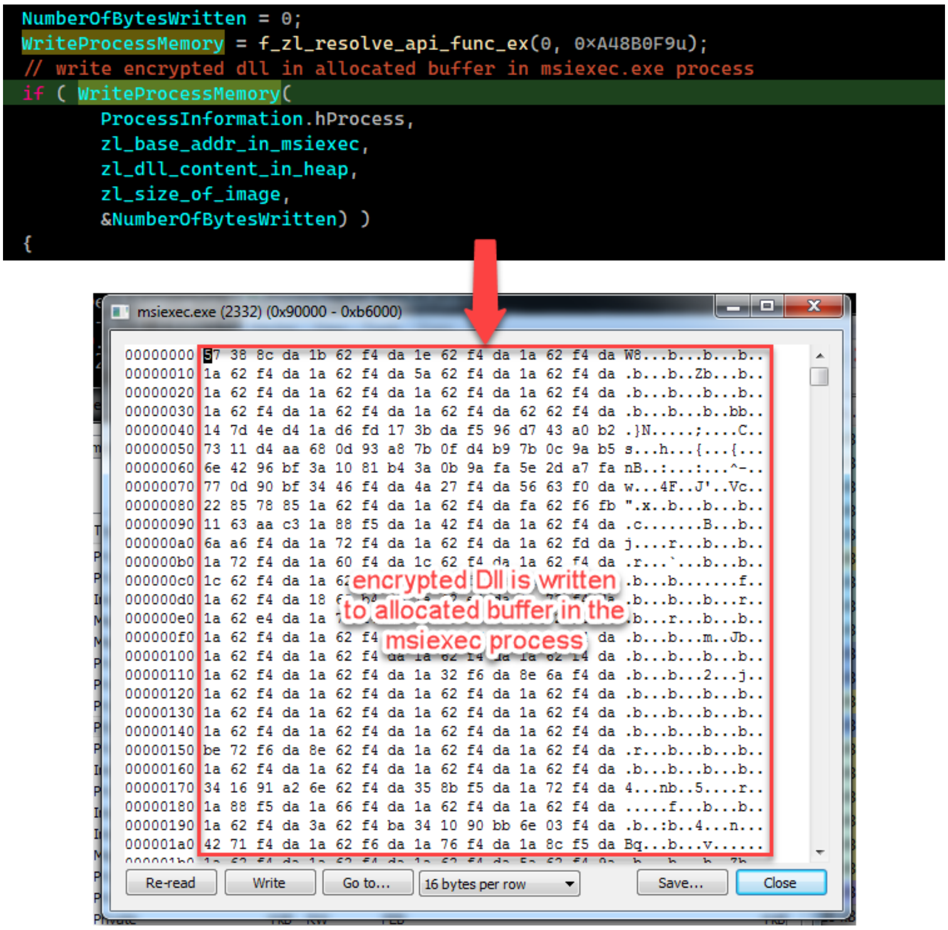
♦ Allocate heap memory, copy the entire contents of the Dll into this heap:

```
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);
```

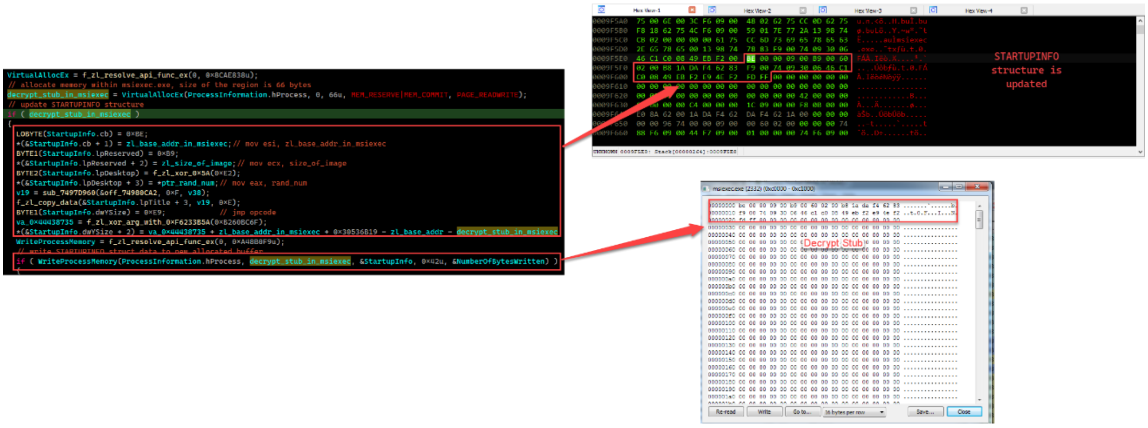
♦ Generate a random number and use it to encrypt the entire payload stored in the heap:



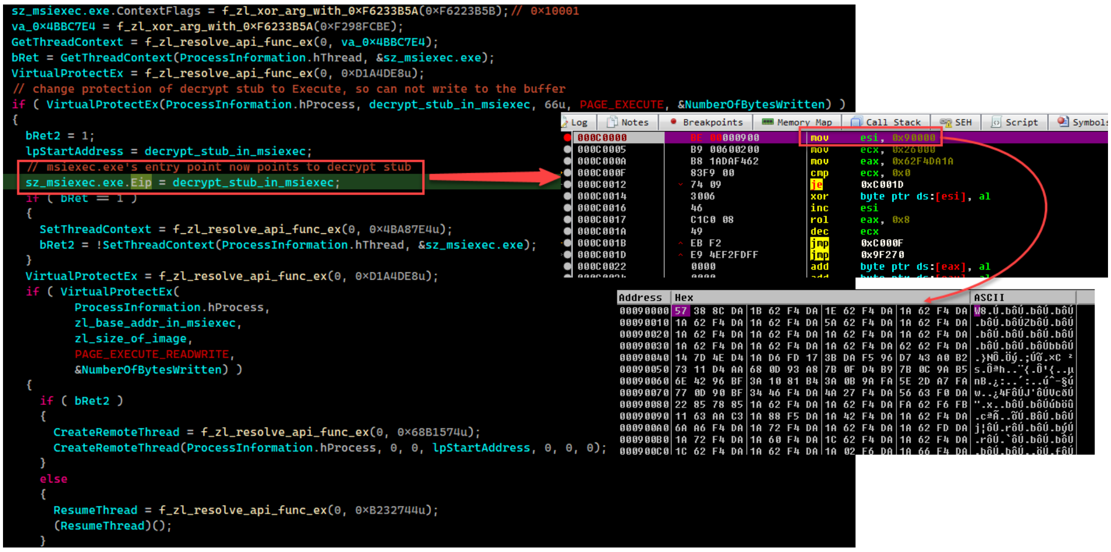
♦ Use the `WriteProcessMemory` API function to write the entire encrypted payload from the heap to the previously allocated memory in the `msiexec.exe` process:



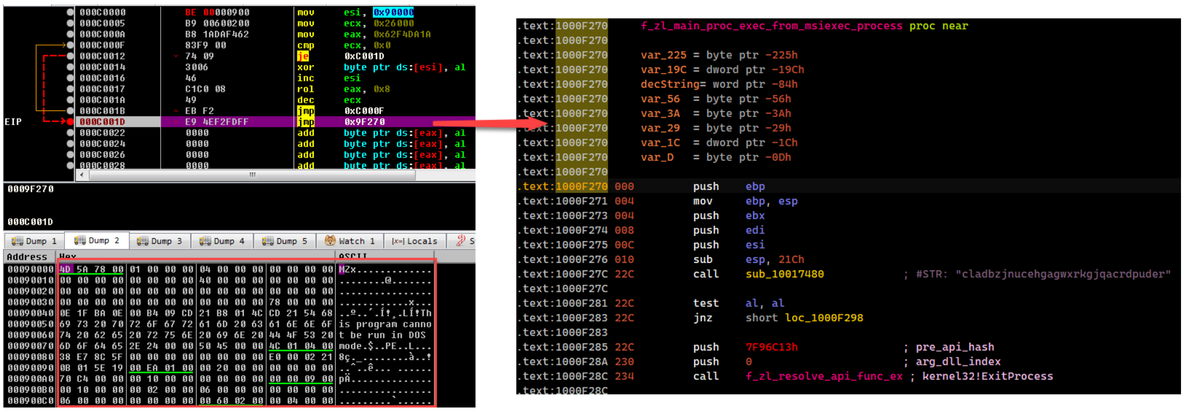
♦ Continue to use the `VirtualAllocEx` API function to allocate a second memory region has size of region are 66 bytes in the `msiexec.exe` process. This memory region will be used to decrypt the entire encrypted Dll above. Update the `STARTUPINFO` structure created by the `CreateProcessA` function before, 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:

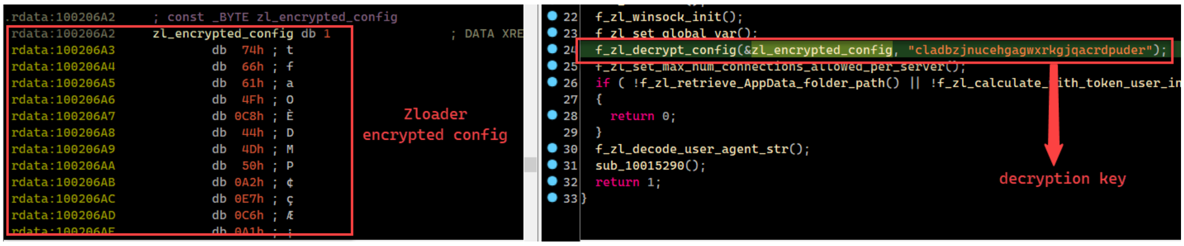


♦ After decrypting the entire Zloader Dll, it will jump to the RVA address of `0xF270` (File offset: `0xE670`) to execute the main tasks of the malware:



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:

```c
int __cdecl f_zl_decrypt_config(_BYTE *zl_enc_cfg, _BYTE *rc4_key)
{
  _BYTE *enc_data; // esi
  int val_0x36F; // eax
  char zl_enc_c2_cfg[4]; // [esp+0h] [ebp-37Ch]
  int v6; // [esp+2D9h] [ebp-A3h]
  int v7; // [esp+2DDh] [ebp-9Fh]
  int v8; // [esp+2E1h] [ebp-9Bh]

  enc_data = f_zl_allocate_heap(0x36F);
  f_zl_return_arg_value_1(enc_data);
  g_zl_enc_c2_cfg = enc_data;
  val_0x36F = f_zl_xor_arg_with_0xF6233B5A(0xF6233835);
  f_zl_copy_data(enc_data, zl_enc_cfg, val_0x36F);
  f_zl_strcpy(g_rc4_key, rc4_key, 0xFFFFFFFF);
  f_zl_return_arg_value_1(zl_enc_c2_cfg);
  f_zl_decrypt_cfg(zl_enc_c2_cfg);
  dword_10022F74 = v6;
```

```c
int __thiscall f_zl_decrypt_cfg(char *zl_enc_c2_cfg)
{
  unsigned __int16 rc4_key_len; // ax

  f_zl_copy_data(zl_enc_c2_cfg, g_zl_enc_c2_cfg, 0x36F);
  rc4_key_len = f_zl_strlen(g_rc4_key);
  return f_zl_RC4_decrypt(g_rc4_key, rc4_key_len, zl_enc_c2_cfg, 0x36Fu);
}
```

```c
int __cdecl f_zl_RC4_decrypt(_BYTE *rc4_key, unsigned int rc4_key_len, _BYTE *zl_enc_cfg, unsigned int enc_cfg_size)
{
  unsigned __int8 s_box[272]; // [esp+0h] [ebp-110h]

  f_zl_rc4_KSA(rc4_key, rc4_key_len, s_box);
  return f_zl_rc4_PRGA(zl_enc_cfg, enc_cfg_size, s_box);
}
```

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

```python
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:

```
Output window
[*] Target address found at 0xa74ddL
[+] Bot name: 9092us
[+] Bot ID: 9092us
[+] Zloader C2 address:
        https://asdfghdsajkl.com/gate.php
        https://lkjhgfgsdshja.com/gate.php
        https://kjdhsasghjds.com/gate.php
        https://kdjwhqejqwij.com/gate.php
        https://iasudjghnasd.com/gate.php
        https://daksjuggdhwa.com/gate.php
        https://dkisuaggdjhna.com/gate.php
        https://eiqwuggejqw.com/gate.php
        https://dquggwjhdmq.com/gate.php
        https://djshggadasj.com/gate.php
[+] Embedded RC4 key: 03d5ae30a0bd934a23b6a7f0756aa504
```

10. Collect and save configuration in Registry

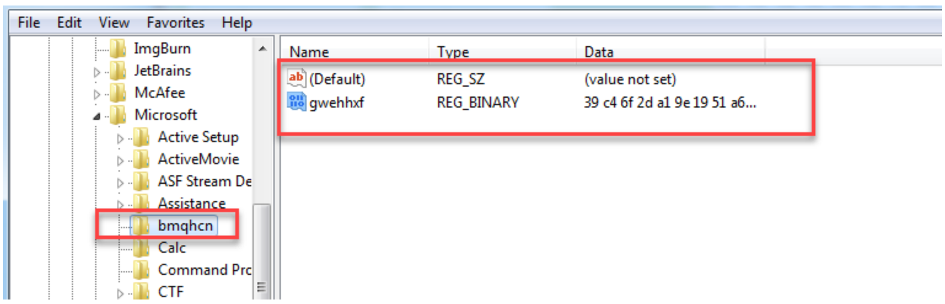
When first executed, Zloader will collect information about the victim including `volume_GUID`, `Computer_Name`, `Windows version`, `Install Date`, create random folders at `%APPDATA%`, generate a random registry key at `HKEY_CURRENT_USER\Software\Microsoft`, then encrypt all relevant information and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

```
if ( !f_zl_gen_random_reg_key_and_retrieve_val() )
{
  f_zl_wchar_strcpy(wsz_zl_dll_path, g_wsz_zl_dll_path);
  bRet = f_zl_collect_victim_info_create_random_folders_and_store_info_in_registry(wsz_zl_dll_path, 1);
  f_zl_free_heap_ex(wsz_zl_dll_path);
  v18 = 1;
  if ( bRet )
  {
    goto LABEL_13;
  }
  ExitProcess = f_zl_resolve_api_func_e(0, 0x7F96C13u);
  ExitProcess(0);
}
```

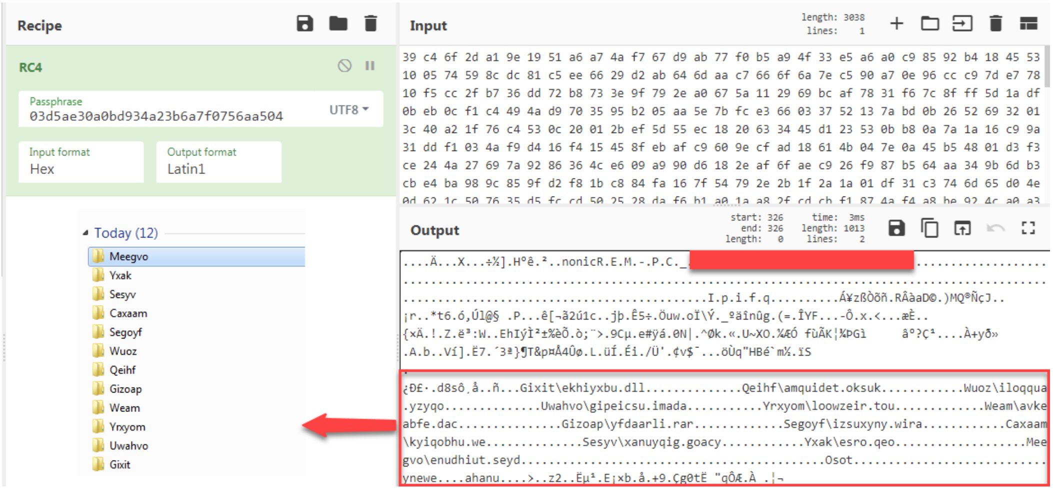
```
zl_victim_ctx = f_zl_allocate_heap_region(0x3D0);
f_zl_retrieve_original_CLSID_of_root_drive(&zl_victim_ctx->pclsid);
f_zl_get_victim_system_info(zl_victim_ctx->ComputerName_VersionInfo_InstallDate);
f_zl_gen_random_wstring(2, &zl_victim_ctx->rand_reg_key, 4u, 8u);// ex:Ipifq
f_zl_rc4_KSA_for_embedded_key(&zl_victim_ctx->rc4_sbox_embedded_key);
  // create 12 random folders
  f_zl_create_rand_directory(&v92, wszAppDataPath, sz_dll, 0);
  f_zl_ctor_struct(&v79);
  f_zl_create_rand_directory(&v79, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v80);
  f_zl_create_rand_directory(&v80, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v81);
  f_zl_create_rand_directory(&v81, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v82);
  f_zl_create_rand_directory(&v82, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v83);
  f_zl_create_rand_directory(&v83, wszAppDataPath, 0, 1);
  f_zl_ctor_struct(&v84);
  f_zl_create_rand_directory(&v84, wszAppDataPath, 0, 1);
  f_zl_ctor_struct(&v85);
  f_zl_create_rand_directory(&v85, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v86);
  f_zl_create_rand_directory(&v86, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v87);
  f_zl_create_rand_directory(&v87, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v88);
  f_zl_create_rand_directory(&v88, wszAppDataPath, 0, 0);
  f_zl_ctor_struct(&v89);
  f_zl_create_rand_directory(&v89, wszAppDataPath, 0, 0);
```

```
  if ( !f_zl_encrypt_data_create_random_registry_and_set_registry_value(zl_victim_ctx) )
  {
LABEL_24:
    bRet = 0;
  }
```

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:

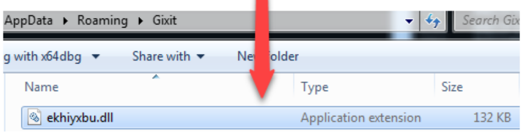


11. Persistence technique

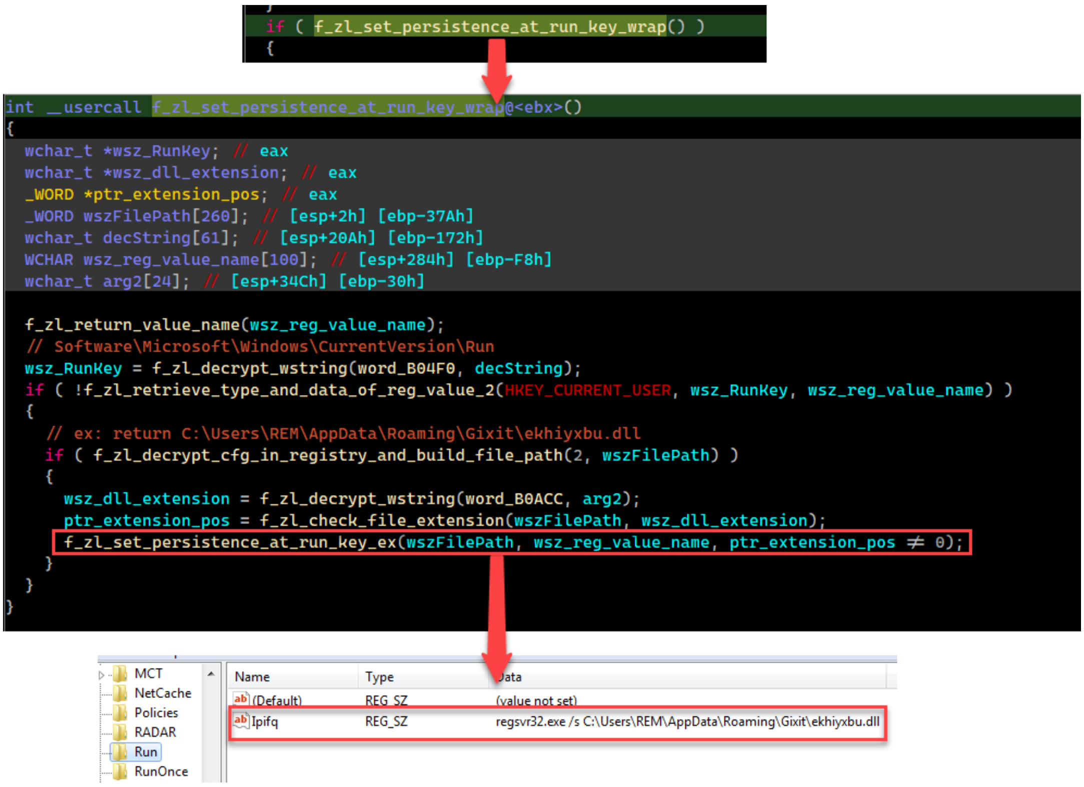
Zloader reads the entire contents of the core Dll from disk into the memory region, then writes to a random dll in a directory created above at %APPDATA%:

```
// read payload content from disk and copy to another buffer
if ( f_zl_read_file_content_from_disk_if_exist(zl_dll_path, &payload_info, 2u) )
{
  f_zl_copy_data_ex(&zl_cloned_payload, payload_info.payload_content, payload_info.payload_content + payload_info.payload_size);
  f_zl_release_payload_info(&payload_info);
}
```

```
// create random dll that stored core dll's content
payload_size = f_zl_return_buf_size(&zl_cloned_payload);
ptr_zl_cloned_payload = f_zl_return_buf(&zl_cloned_payload);
// ex: C:\Users\REM\AppData\Roaming\Gixit\ekhiyxbu.dll
wsz_random_dll_path = f_zl_return_struc_value(&ptr_zl_random_dll_path);
f_zl_create_file(wsz_random_dll_path, wsz_random_dll_path, ptr_zl_cloned_payload, payload_size);
```



Create persistence key at HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run:



12. References

♦ Can You Trust a File's Digital Signature? New Zloader Campaign exploits Microsoft's Signature Verification putting users at risk

♦ Shining a light on "Silent Night" Zloader/Zbot

♦ The DGA of Zloader

♦ 2020-09-11 - ZLOADER (SILENT NIGHT) INFECTION FROM MYRESUME.XLS

♦ Hide and Seek | New Zloader Infection Chain Comes With Improved Stealth and Evasion Mechanism

♦ Zloader Installs Remote Access Backdoors and Delivers Cobalt Strike

Tran Trung Kien (aka m4n0w4r)

Malware Analysis Expert

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