1. Giới thiệu

Zloader, một banking trojan còn biết đến với những tên gọi khác như **Terdot** hay **Zbot**. Dòng trojan này được phát hiện lần đầu tiên vào năm 2016, và theo thời gian số lượng phát tán của nó liên tục gia tăng. Code của Zloader được cho là xây dựng dựa trên mã nguồn bị rò rỉ của mã độc **ZeuS** nổi tiếng. Vào năm 2011, khi mã nguồn của ZeuS được công khai thì từ đó tới nay nó được sử dụng trong nhiều mẫu mã độc khác nhau.

Zloader có đầy đủ chức năng tiêu chuẩn của một trojan như có thể lấy thông tin từ các trình duyệt, thu thập cookie và mật khẩu, chụp ảnh màn hình… đồng thời để gây khó khăn cho các nhà phân tích, nó áp dụng các kĩ thuật cao cấp, bao gồm code obfuscation và mã hóa chuỗi, che dấu các hàm APIs. Gần đây, các chuyên gia của CheckPoint đã công bố phân tích về một chiến dịch phát tán Zloader theo đó quá trình lây nhiễm đã khai thác quy trình kiểm tra chữ kí số của Microsoft. Bên cạnh đó, Zloader còn được dùng để phát tán các loại mã độc khác bao gồm cả ransomware như Ryuk và Egregor. Điều này cho thấy, những kẻ đứng sau mã độc này vẫn đang tìm các phương thức khác nhau để nâng cấp nhằm vượt qua các biện pháp phòng vệ. Dưới đây là bảng xếp hạng của Zloader theo đánh giá từ trang AnyRun:

Mới đây nhất, nhiều đơn vị cung cấp dịch vụ viễn thông và công ty an ninh mạng trên toàn thế giới, bao gồm ESET, Black Lotus Labs, Đơn vị 42 của Palo Alto Networks và Avast đã hợp tác với các chuyên gia nghiên cứu bảo mật của Microsoft trong suốt nỗ lực điều tra, đã thực hiện các bước pháp lý và kỹ thuật để phá vỡ mạng botnet ZLoader, chiếm quyền kiểm soát 65 tên miền được sử dụng để kiểm soát và giao tiếp với các máy chủ bị lây nhiễm.

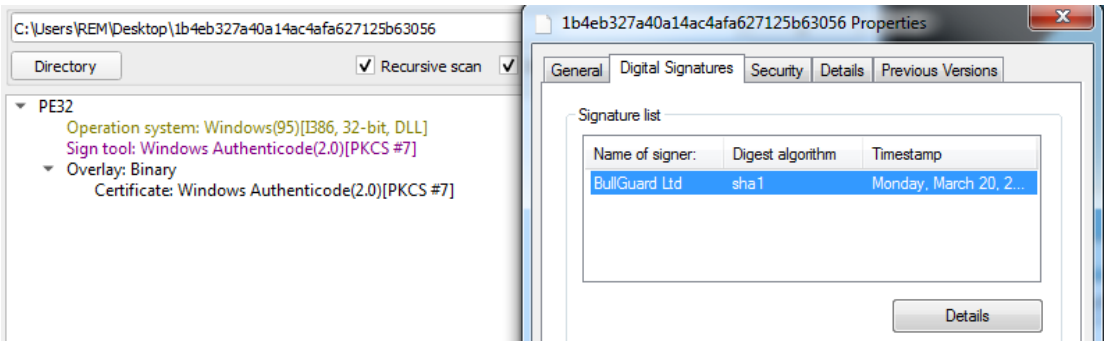
Bài viết này sẽ cung cấp chi tiết quá trình phân tích và kĩ thuật mà Zloader sử dụng, bao gồm:

- Cách thức để unpack để dump Zloader Core Dll.
- Cách thức mà Zloader gây khó khăn cũng như làm mất thời gian trong quá trình phân tích.
- Giải mã các chuỗi được sử dụng bởi Zloader bằng cả hai phương pháp IDAPython và AppCall.
- Áp dụng AppCall để khôi phục lại các hàm APIs mà Zloader sử dụng.
- Kĩ thuật Process Injection mà Zloader áp dụng để inject vào tiến trình `msiexec.exe`.
- Giải mã thông tin cấu hình liên quan tới các địa chỉ C2s.
- Cách thức Zloader thu thập và lưu thông tin tại Registry.
- Kĩ thuật tạo Persistence.

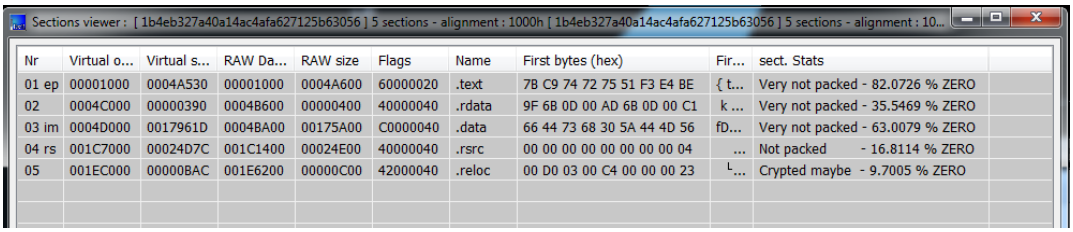
Sample được sử dụng trong bài viết: https://www.virustotal.com/gui/file/034f61d86de99210eb32a2dca27a3ad883f54750c46cdec4fcc53050b2f716eb

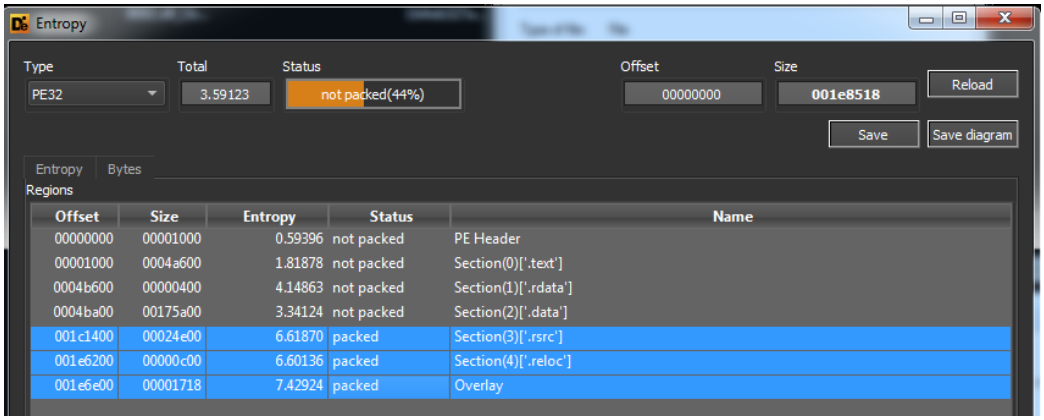
2. Unpacking Zloader Core Dll

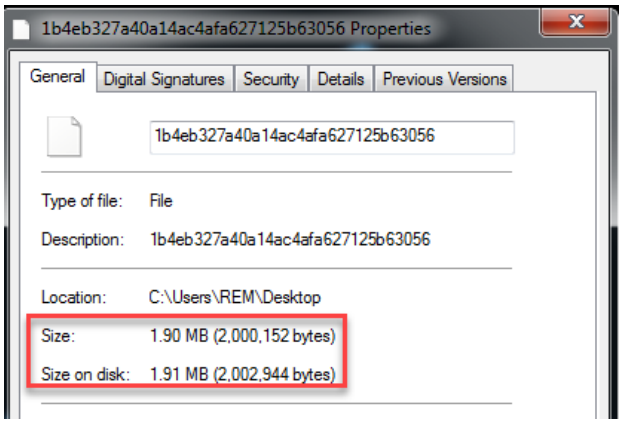
Trước tiên, kiểm tra sample bằng **Nauz File Detector**:



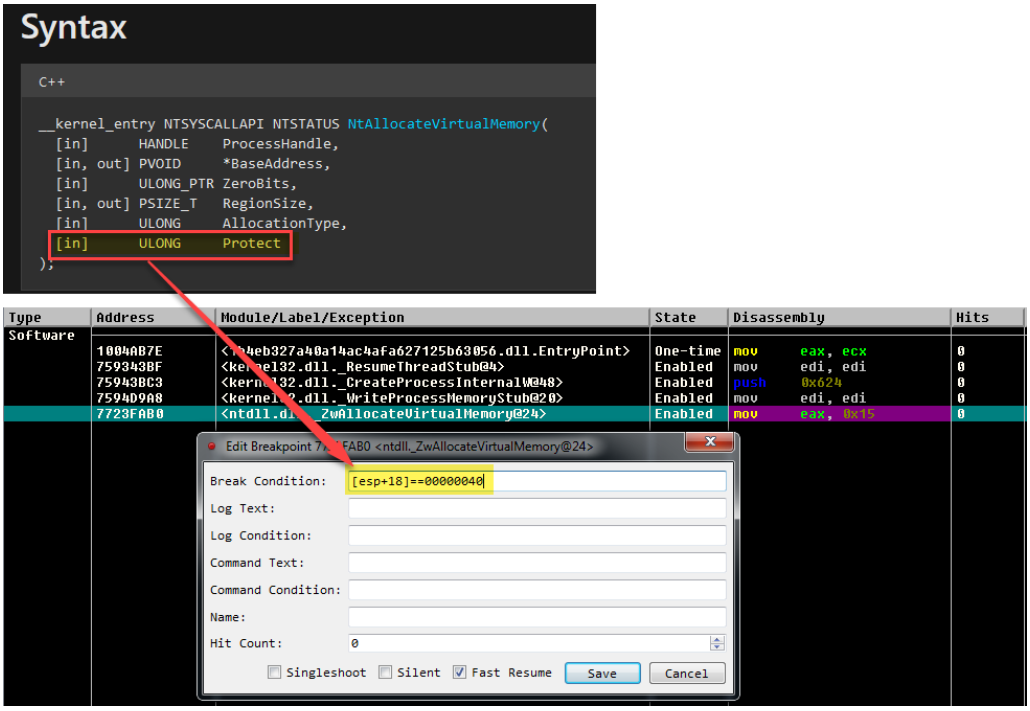
Kết hợp các thông tin về sections từ **ExeInfo**, thông tin Entropy bằng **DiE** cũng như kích thước của file DLL thì có thể khẳng định DLL này bị packed:



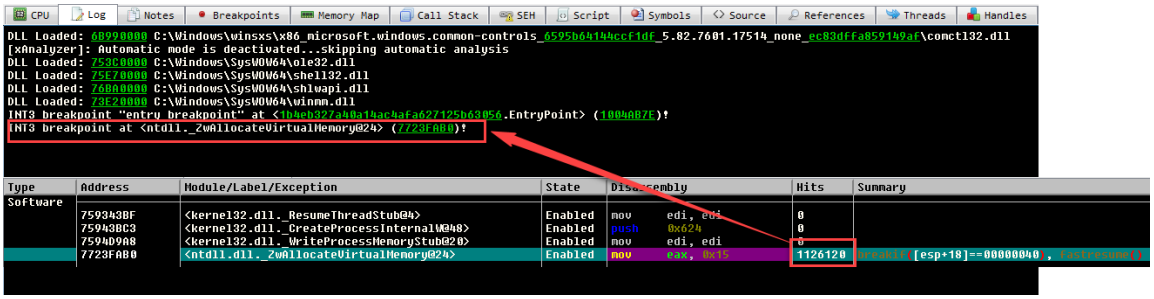




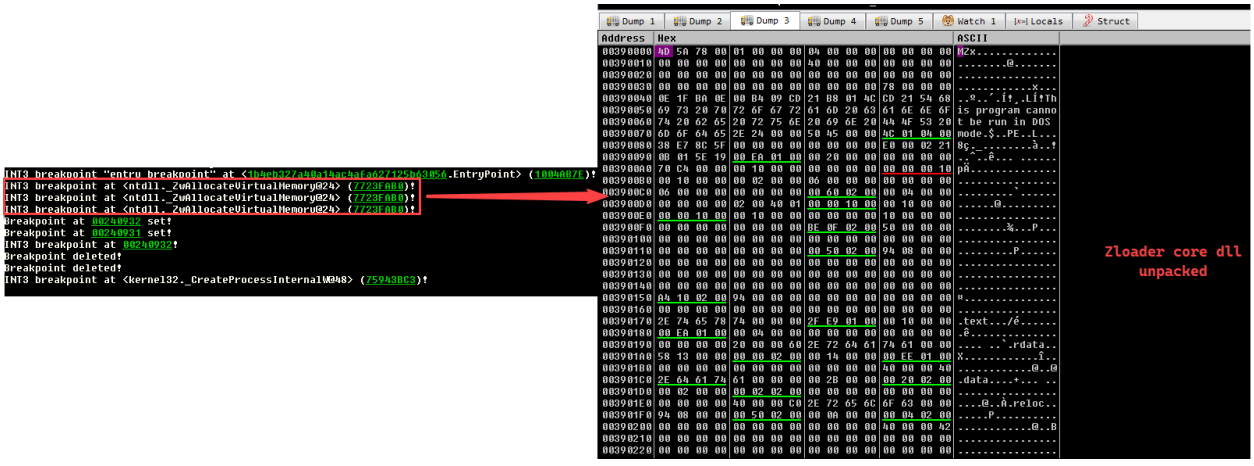
Để unpack, sử dụng **x64dbg** load file Dll, đặt **bp NtAllocateVirtualMemory**. Sau đó, sửa lại điều kiện của breakpoint như sau:



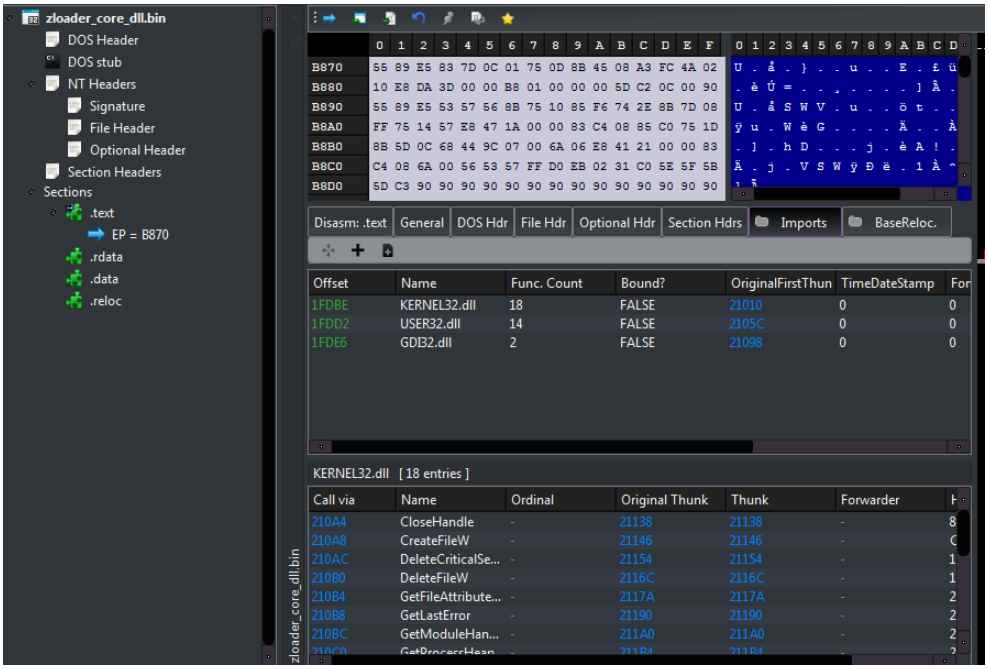
Thực thi bằng **F9** và đợi cho tới khi hit breakpoint thỏa điều kiện đã thiết lập (*sau khoảng 1126120 hits*):



Follow theo các vùng nhớ đã được cấp phát, sau lần hit bp thứ 3, core Dll của Zloader sẽ được unpack:



Dump Dll này ra disk, file có MD5: *9b5589fcd123a3533584a62956f2231b*.



3. Anti-analysis

Để làm tốn thời gian của người phân tích, Zloader sử dụng các hàm vô nghĩa, hoặc viết lại các hàm nhìn rất rắc rối nhưng chỉ để thực hiện các tác vụ đơn giản như AND, OR, XOR, ADD, SUB, …

Ví dụ, một hàm thực hiện tác vụ vô nghĩa:

```
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;
}
```

Ví dụ, hàm thực hiện tác vụ **AND**, **OR**:

```
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_0×F6233B5A(0×843A5CA6) && 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 = (0×176
                * (num1 + 0×CA * (v3 & 8) * v3 - 0×1B1)
                * (f_zl_xor_arg_with_0×F6233B5A(0×F6233AD1) + num1 + 0×CA * (v3 & 8) * v3 - 0×1B1)
                * 0×CA
                * (v3 & 8)
                * v3
                * num1);
  return num1 & num2;
}
```

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

  tmp = (0×C * (num2 + (num1 ^ (num1 * (num2 + 0×46) + ((num1 * (num2 + 0×46)) ^ 0×9E))))) & num1 & (0×C
                                                                                                     * (num2
                                                                                                      + (num1 ^ (num1 * (num2 + 0×46)
                                                                                                              + ((num1 * (num2 + 0×46)) ^ 0×9E))))
                                                                                                     - num2);
  result = num2 | num1;
  g_0C80441ADh = ((num2 | num1) ^ (tmp + (num2 ^ ((0×BD * tmp + 0×51) * tmp * (tmp ^ (0×BD * tmp + 0×51))))));
  return result;
}
```

4. Decrypt wide string

4.1. Sử dụng IDAPython

Các strings mà core Dll sử dụng đều đã bị mã hóa. Hàm giải mã wide string sẽ nhận hai tham số truyền vào:

- **Tham số thứ nhất:** địa chỉ chứa chuỗi bị mã hóa.
- **Tham số thứ hai:** địa chỉ lưu chuỗi sau giải mã.

```
.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_and
.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_0×2D5_if_arg1_equal_arg2_else_0×0
  -               01C call f_zl_xor_arg_with_0×F6233B5A
  -               020 call f_zl_add_arg1_with_arg2
  -               01C call f_zl_xor_with_0×3B5A
  -               020 call f_zl_and_arg1_with_arg2
  -               020 call f_zl_sub_arg1_from_ar2
  -               020 call f_zl_sub_arg1_from_ar2
```

Mã giả tại hàm giải mã `f_zl_decrypt_wstring` nhìn có vẻ khả rối, tuy nhiên nếu phân tích kĩ hàm này thực hiện vòng lặp xor đơn giản với khóa giải mã là "`PgtrIPF-2ftOj00Ox`":

```
// 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;
  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;
}
```

dec_char = enc_char ^ xor_key_val

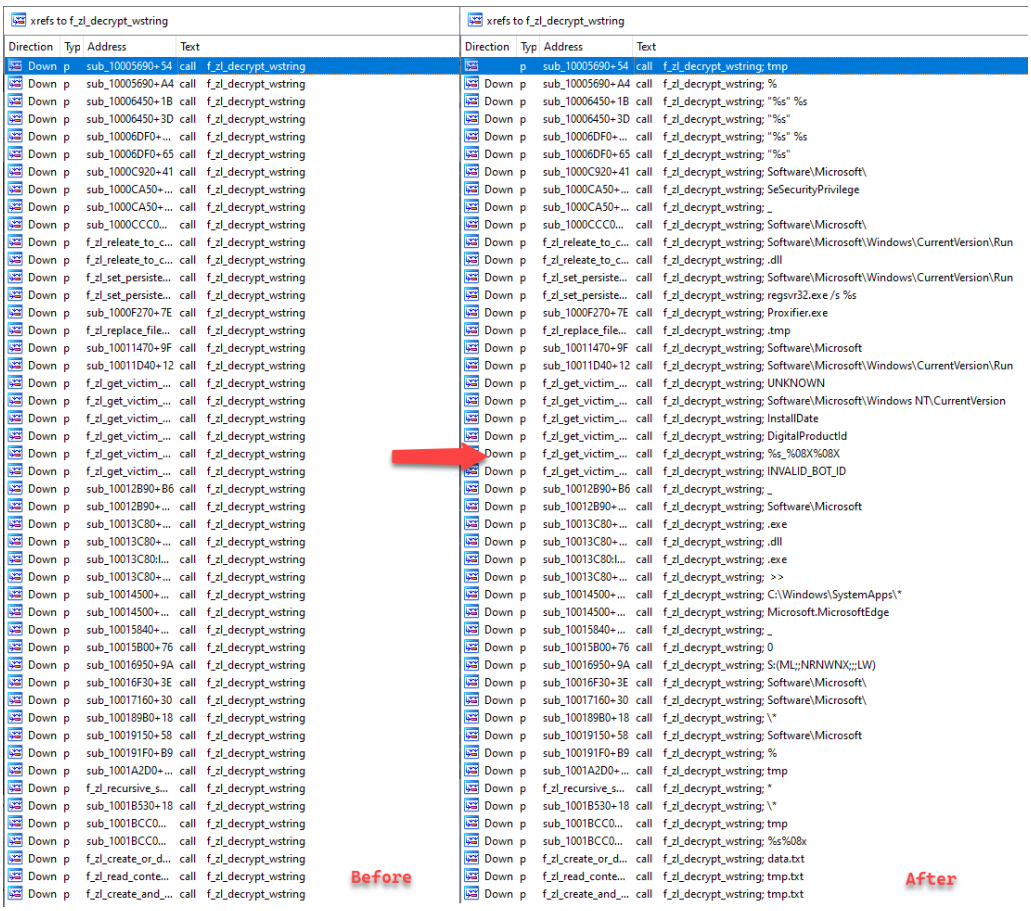
Dựa vào mã giả trên, viết lại đoạn code thực hiện giải mã bằng python như sau:

```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')
```

Với sự hỗ trợ của IDAPython, ta có thể tự động hóa toàn bộ quá trình giải mã string và thêm các chú thích tại các hàm giải mã trong IDA để phục vụ cho việc phân tích. Toàn bộ code python như sau:

```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()
```

Kết quả trước và sau khi thực hiện script sẽ giúp công việc phân tích dễ dàng hơn:

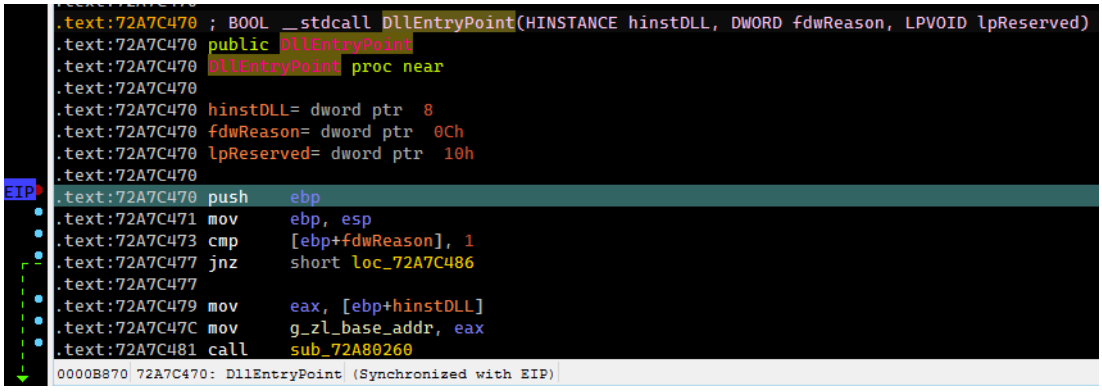


4.2. Sử dụng IDA AppCall

Nếu không có thời gian để đào sâu vào quá trình thực hiện giải mã của hàm hoặc khi thuật toán thực hiện quá phức tạp, ta có thể sử dụng tính năng hữu ích của IDA là AppCall để giúp giải mã dữ liệu. Về cơ bản, Appcall là một cơ chế được sử dụng để thực thi các hàm bên trong chương trình đang được debug bởi trình debugger của IDA. Trước khi áp dụng AppCall, việc đầu tiên là cần định nghĩa chính xác prototype của hàm. Ví dụ, hàm `f_zl_decrypt_wstring` có protoype như sau:

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

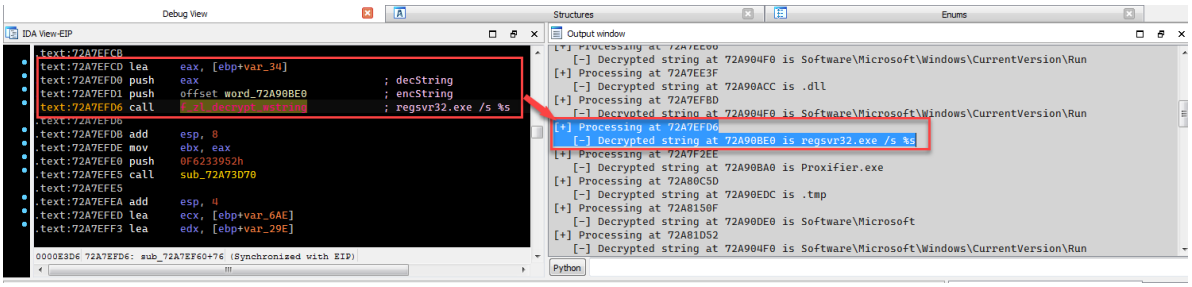
Lưu ý một lần nữa là để sử dụng được AppCall thì phải thực hiện debug. Như hình dưới đây, IDA đang dừng ở breakpoint đã đặt `DllEntryPoint`:



Sau đó, thực thi python script dưới đây để giải mã và thêm chú thích các chuỗi đã giải mã được tại các hàm:

```
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)
```

Kết quả cuối cùng sẽ tương tự như hình dưới đây:



5. Decrypt ansi string

5.1. Sử dụng IDAPython

Bên cạnh hàm giải mã các wide string, Zloader cũng sử dụng hàm giải mã các ansi string. Hàm này cũng nhận hai tham số truyền vào:

- **Tham số thứ nhất:** địa chỉ chứa chuỗi bị mã hóa.
- **Tham số thứ hai:** địa chỉ lưu chuỗi sau giải mã.

```
.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
```

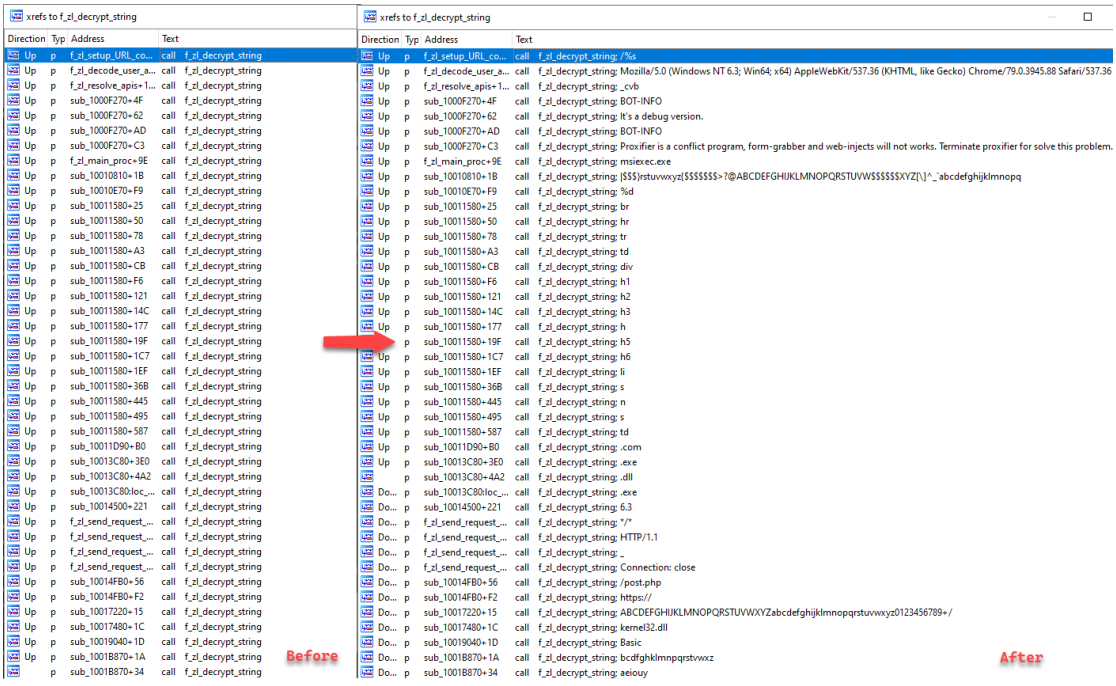
Cũng tương tự như hàm `f_zl_decrypt_wstring`, mã giả của hàm `f_zl_decrypt_string` nhìn cũng khá rối, tuy nhiên nó vẫn sử dụng vòng lặp xor để giải mã với khóa giải mã vẫn là "`PgtrIPF-2ftOj00Ox`":

```
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;
}
```

Dưới đây là toàn bộ code python để tự động hóa toàn bộ quá trình giải mã các string và thêm các chú thích tại các hàm:

```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 = 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()
```

Kết quả trước và sau khi thực hiện script:



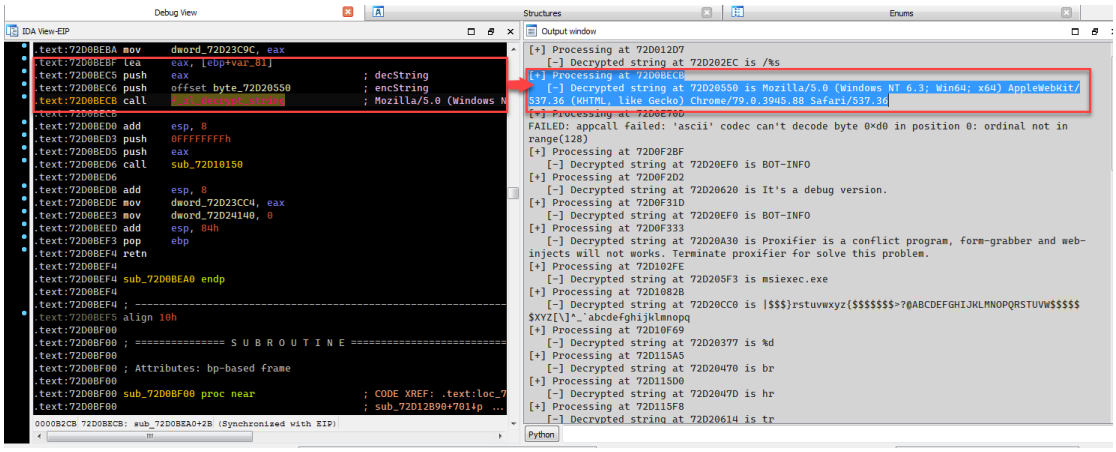
5.2. Sử dụng IDA AppCall

Để sử dụng AppCall, tương tự như trên, cần định nghĩa lại chính xác prototype cho hàm `f_zl_decrypt_string` như sau: `char *__cdecl f_zl_decrypt_string(char *encString, char *decString);`

Sửa lại một chút script đã sử dụng cho việc giải mã các wide string ở trên:

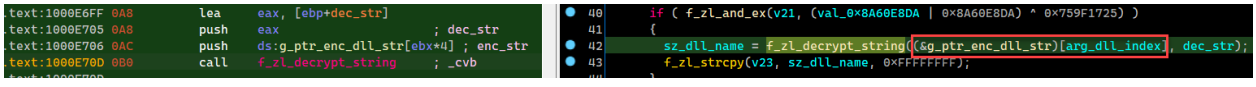
```
import idc, idaapi, idautils

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)
```

Kết quả sau khi chạy script:

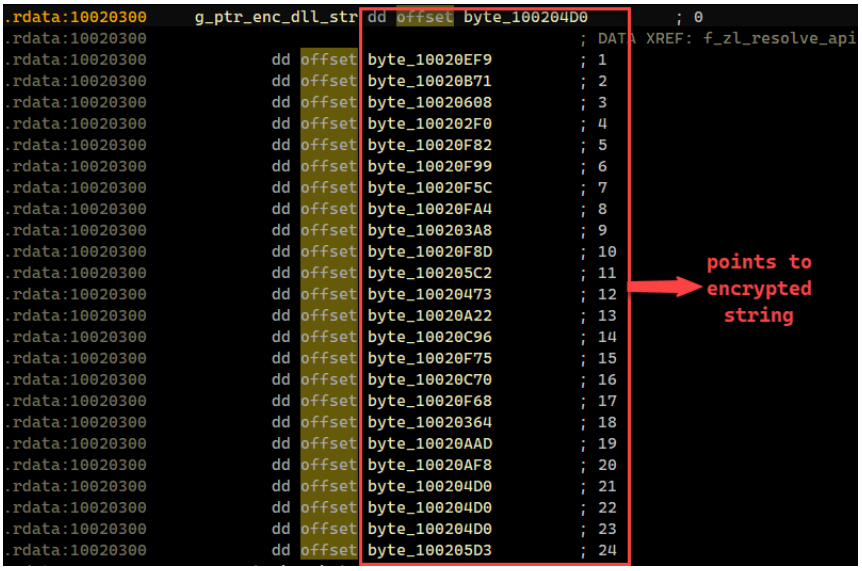


6. Danh sách các Dlls được Zloader sử dụng

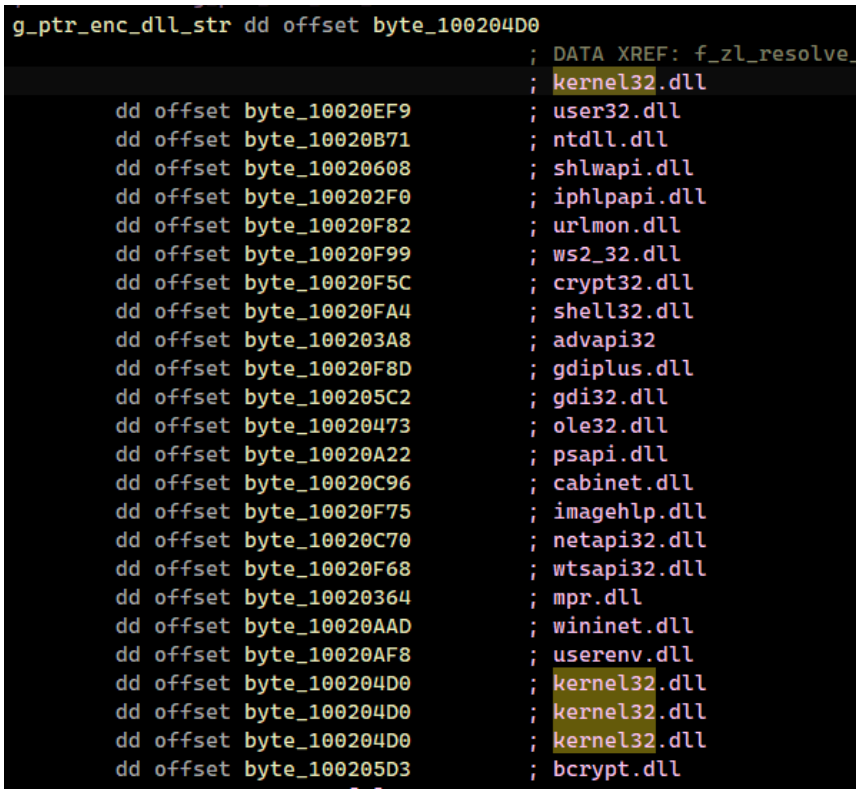
Trong danh sách các string được giải mã bởi hàm `f_zl_decrypt_string` ở trên, có một string sau giải mã khá vô nghĩa. Đi tới địa chỉ này, sau khi phân tích tôi nhận thấy tham số đầu tiên được truyền vào cho hàm là mảng chứa địa chỉ của các chuỗi bị mã hóa. Dựa vào giá trị `index` tương ứng của mảng sẽ truy cập tới địa chỉ chứa chuỗi bị mã hóa tương ứng:



Đi tới mảng `g_ptr_enc_dll_str` (*đã đổi tên ở trên*) sẽ thấy được danh sách các địa chỉ như hình dưới đây:



Sửa lại script để giải mã các riêng cho các chuỗi Dll, kết quả có được khi thực hiện script như sau:



Tổng kết lại, ta có bảng danh sách `index` tương ứng với các Dll mà có thể Zloader sẽ sử dụng để từ đó phục vụ việc lấy địa chỉ của các hàm 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

Tương tự như các dòng mã độc cao cấp khác… Zloader cũng sẽ lấy địa chỉ các hàm API(s) thông qua việc tìm kiếm theo giá trị hash được tính toán trước dựa vào tên hàm API.

```
.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
```

Như trên hình, hàm `f_zl_resolve_api_func_ex` nhận hai tham số truyền vào:

♦ (1): Tham số thứ nhất là `dll_index`. Dựa vào tham số này, hàm sẽ thực hiện giải mã ra tên của của Dll tương ứng, từ đó gọi hàm `LoadLibraryA` để lấy địa chỉ base của Dll này.

```
{
  // 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): Tham số thứ hai là `pre_api_hash`. Tham số này là hash được tính toán trước theo tên hàm API. Hàm `f_zl_resolve_api_func_ex` sẽ gọi tới `f_zl_resolve_api_func` để tìm kiếm địa chỉ API tương ứng:

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

Mã giả tại hàm `f_zl_resolve_api_func` thực hiện tìm kiếm địa chỉ hàm API như sau:


```
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);
```

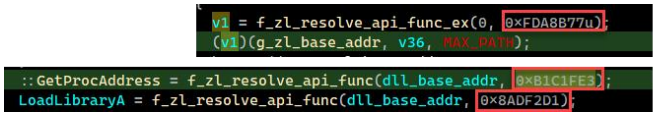
Toàn bộ mã giả của hàm thực hiện tính toán hash theo tên hàm API như sau:

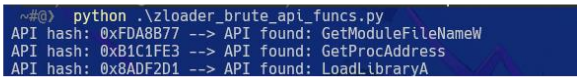
```
int __fastcall f_zl_calc_hash(char *inString, int strLen)
{
  int calced_hash; // edi MAPDST
  unsigned int i; // edx MAPDST
  int v6; // ebx
  int val_0x825180FD; // eax
  int val_0x7DAE7F02; // eax

  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;
}
```

Dựa vào mã giả trên, viết lại bằng Python code dưới đây:

```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
```

Kết quả khi sử dụng hàm trên để tìm các hàm API tương ứng với các giá trị hash `0xFDA8B77`, `0xB1C1FE3`, `0x8ADF2D1`:



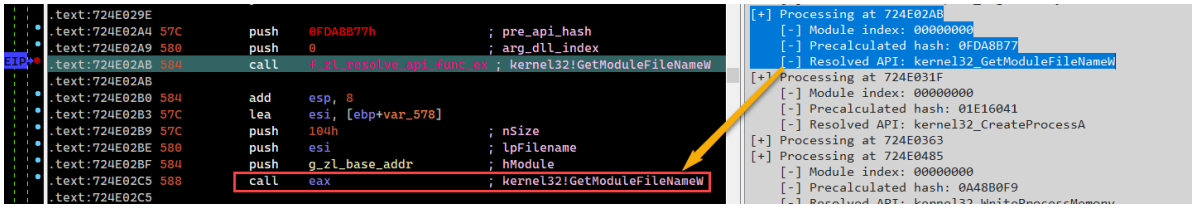


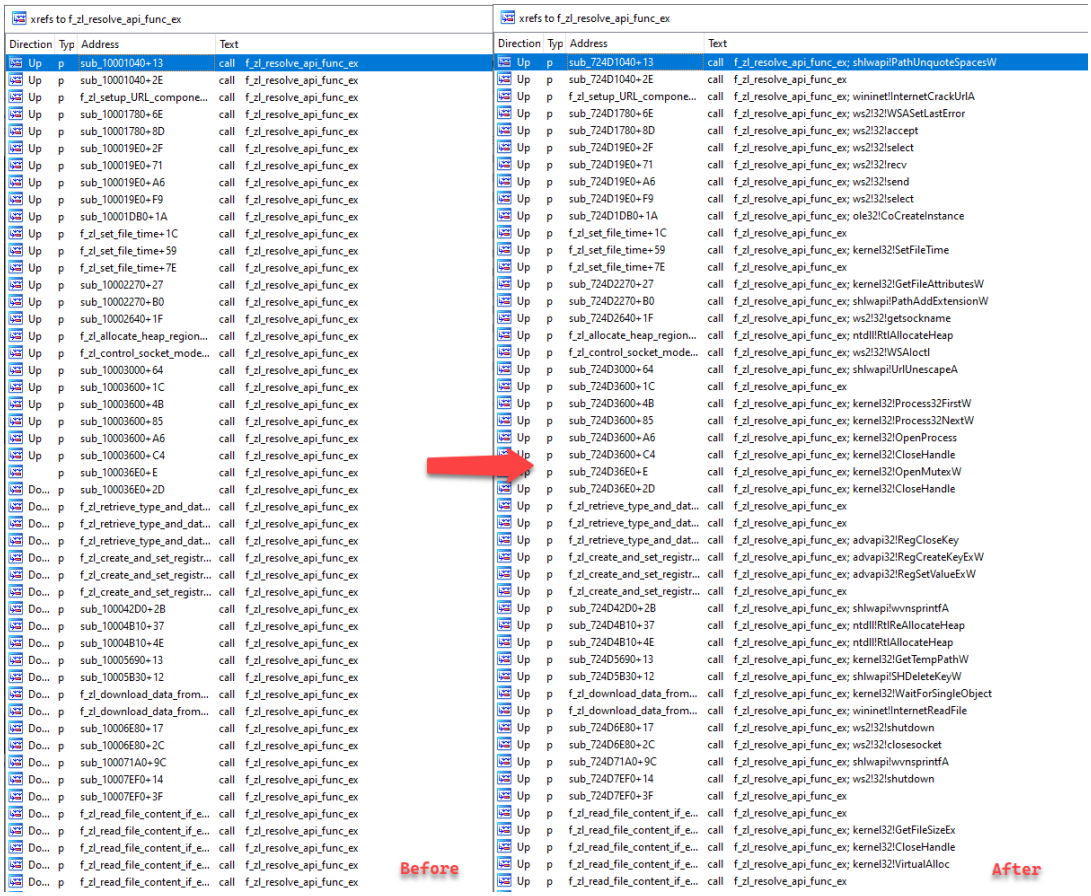
Với những gì đã phân tích ở trên, hoàn toàn có thể viết IDAPython script để khôi phục lại toàn bộ các hàm APIs mà Zloader sử dụng. Tuy nhiên, để tránh việc phải đào sâu vào thuật toán tính toán hash của Zloader cho mỗi lần phân tích, ở đây tôi sẽ sử dụng AppCall thực hiện nhiệm vụ này. Code python sử dụng AppCall như sau:

```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)
```

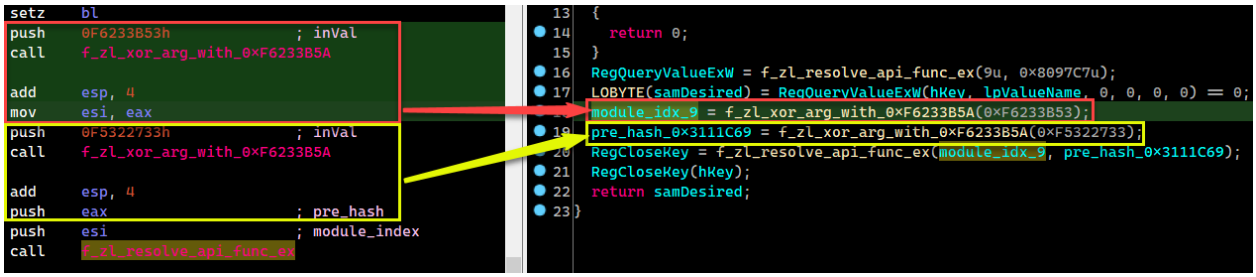
Lưu ý, Zloader có nhiều vùng code gọi tới hàm `f_zl_resolve_api_func_ex`, tuy nhiên sẽ có những vùng code mà không có bất kì tham chiếu nào tới nó cũng như vùng code đó chưa

được tạo thành hàm hoàn chỉnh. Do đó, để có thể chạy được script trên, trước tiên cần phải thực hiện tạo hàm cho những đó. Kết quả sau khi thực thi script sẽ như sau:





Tuy nhiên, như trên hình vẫn còn những vị trí không khôi phục lại được hàm API, đó là bởi vì Zloader đã thực hiện việc tính toán giá trị các giá trị `dll_index` và `pre_api_hash` trước đó và lưu vào thanh ghi. Sau đó mới thực hiện gọi hàm `f_zl_resolve_api_func_ex`:



8. Process Injection

Zloader khi thực thi sẽ tiến hành inject Core Dll vào tiến trình `msiexec.exe`. Toàn bộ quá trình thực hiện như sau:

♦ Sử dụng hàm API `CreateProcessA` để tạo tiến trình `msiexec.exe` ở trạng thái `SUSPENDED`.

```
// 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) )
{
```

♦ Lấy thông tin `SizeOfImage` của Zloader Dll đang load bởi `rundll32.exe/regsvr32.exe`. Sử dụng hàm API `VirtualAllocEx` để cấp phát vùng nhớ mới bên trong tiến trình `msiexec.exe`:

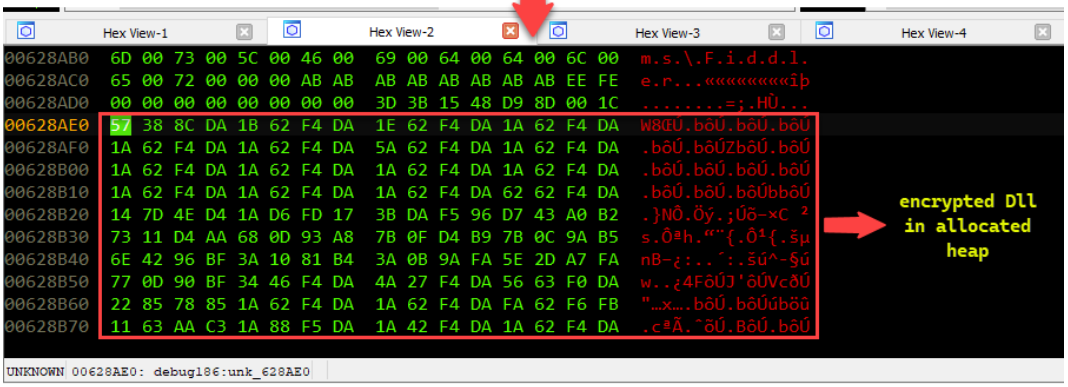
```
zl_size_of_image = f_zl_retrieve_size_of_image(zl_base_addr);
val_0x8CAE838 = f_zl_xor_arg_with_0xF6233B5A(0xFEE9D362);
VirtualAllocEx = f_zl_resolve_api_func_ex(0, val_0x8CAE838);
// allocate region within msiexec.exe with size of region is Zloader's SizeOfImage
zl_payload_buf_in_msiexec = VirtualAllocEx(ProcessInformation.hProcess,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

♦ Cấp phát vùng nhớ heap, thực hiện copy toàn bộ nội dung của Dll vào vùng nhớ heap này:

```
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);
```

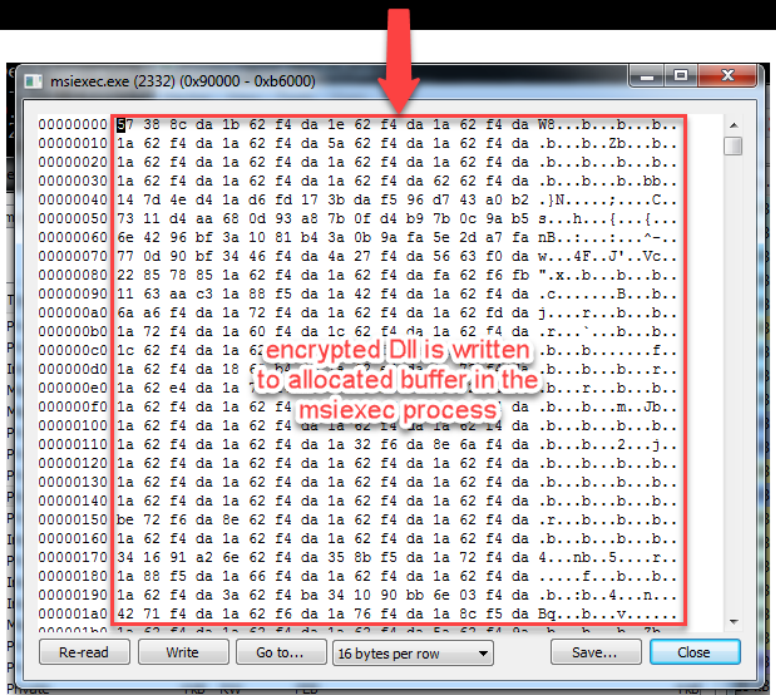
♦ Tạo một số ngẫu nhiên và sử dụng nó vào quá trình mã hóa toàn bộ payload đã lưu tại vùng nhớ heap:

```
*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 );
}
```

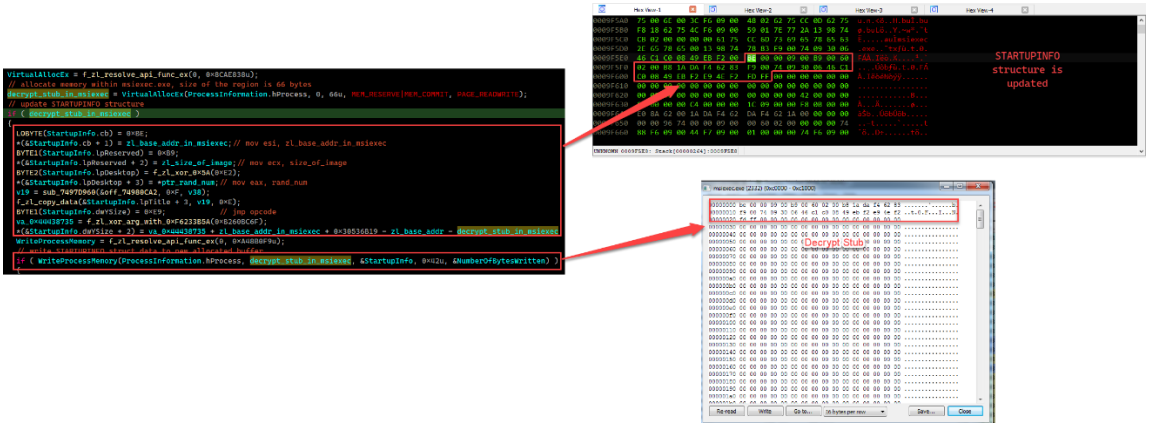


♦ Sử dụng hàm API `WriteProcessMemory` để ghi toàn bộ encrypted payload từ vùng nhớ heap vào vùng nhớ đã cấp phát trước đó trong tiến trình `msiexec.exe`:

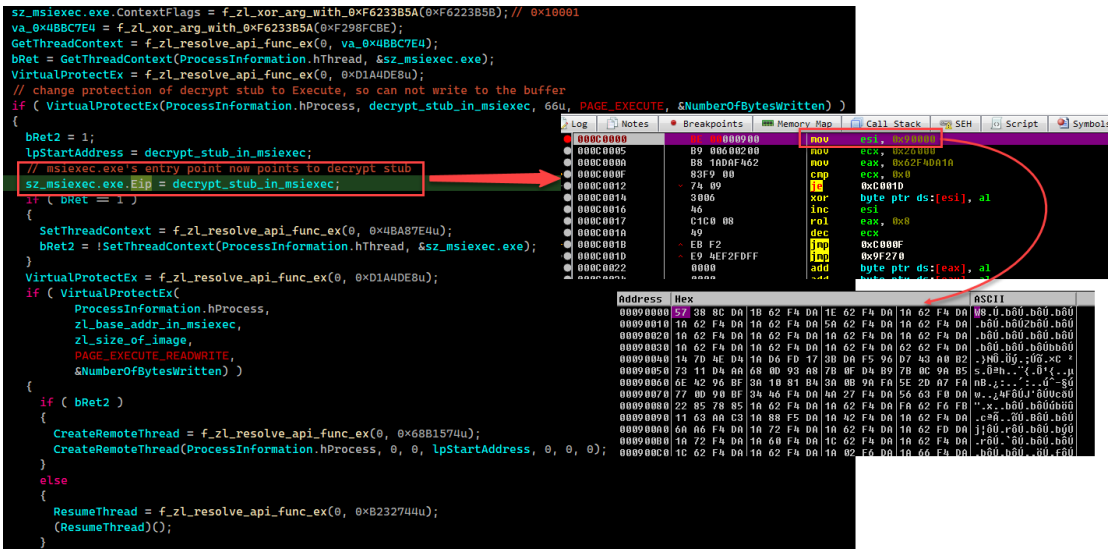
```
NumberOfBytesWritten = 0;
WriteProcessMemory = f_zl_resolve_api_func_ex(0, 0xA48B0F9u);
// write encrypted dll in allocated buffer in msiexec.exe process
if ( WriteProcessMemory(
       ProcessInformation.hProcess,
       zl_base_addr_in_msiexec,
       zl_dll_content_in_heap,
       zl_size_of_image,
       &NumberOfBytesWritten) )
{
```



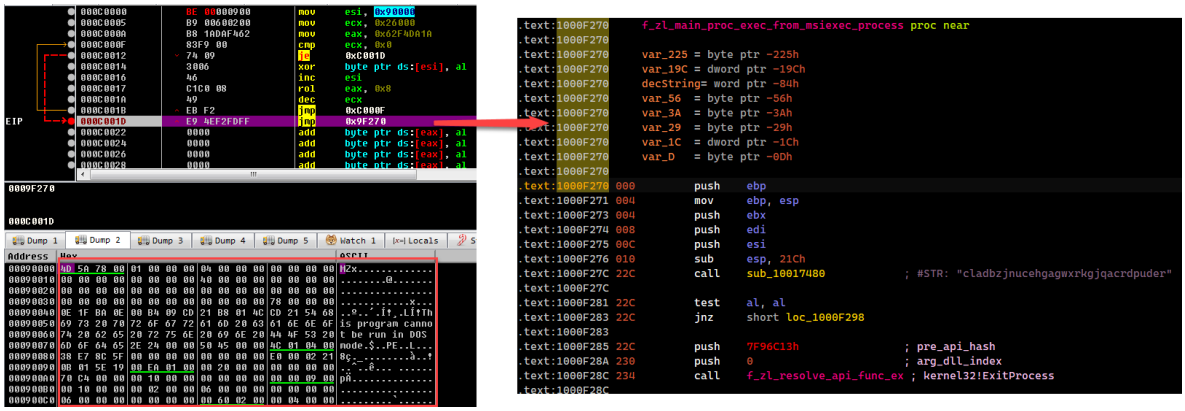
♦ Tiếp tục sử dụng hàm API `VirtualAllocEx` để cấp phát vùng nhớ thứ 2 có kích thước 66 bytes trong tiến trình `msiexec.exe`. Vùng nhớ này sẽ được sử dụng để làm nhiệm vụ giải mã toàn bộ Dll đã mã hóa ở trên. Thực hiện cập nhật lại cấu trúc `STARTUPINFO` đã tạo bởi hàm `CreateProcessA` trước đó, dữ liệu tại đây là những lệnh sẽ được dùng để giải mã encrypted Dll. Sau đó, gọi hàm `WriteProcessMemory` để ghi nội dung đã cập nhật của `STARTUPINFO` vào vùng nhớ vừa tạo.



♦ Cuối cùng, sử dụng các hàm API `GetThreadContext`, `SetThreadContext`, `ResumeThread` hoặc `CreateRemoteThread` để thực thi tiến trình `msiexec.exe`. Lúc này địa chỉ entry point thực thi tại `msiexec.exe` sẽ chính là vùng nhớ chứa đoạn code thực hiện giải mã ở trên:

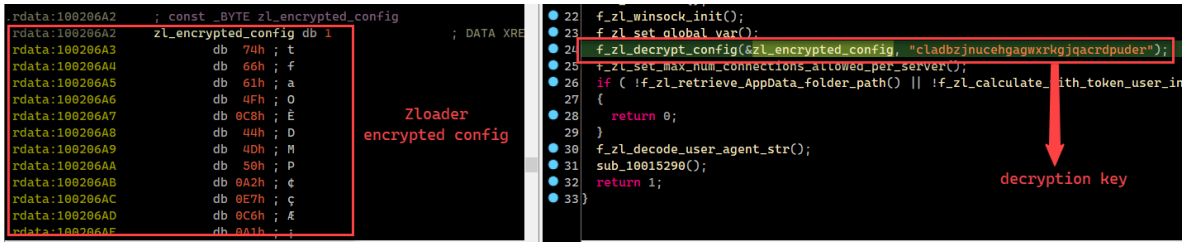


♦ Sau khi thực hiện decrypt lại toàn bộ Zloader Dll, sẽ nhảy tới địa chỉ RVA là `0xF270` (File offset: `0xE670`) để thực thi các tác vụ chính của mã độc:

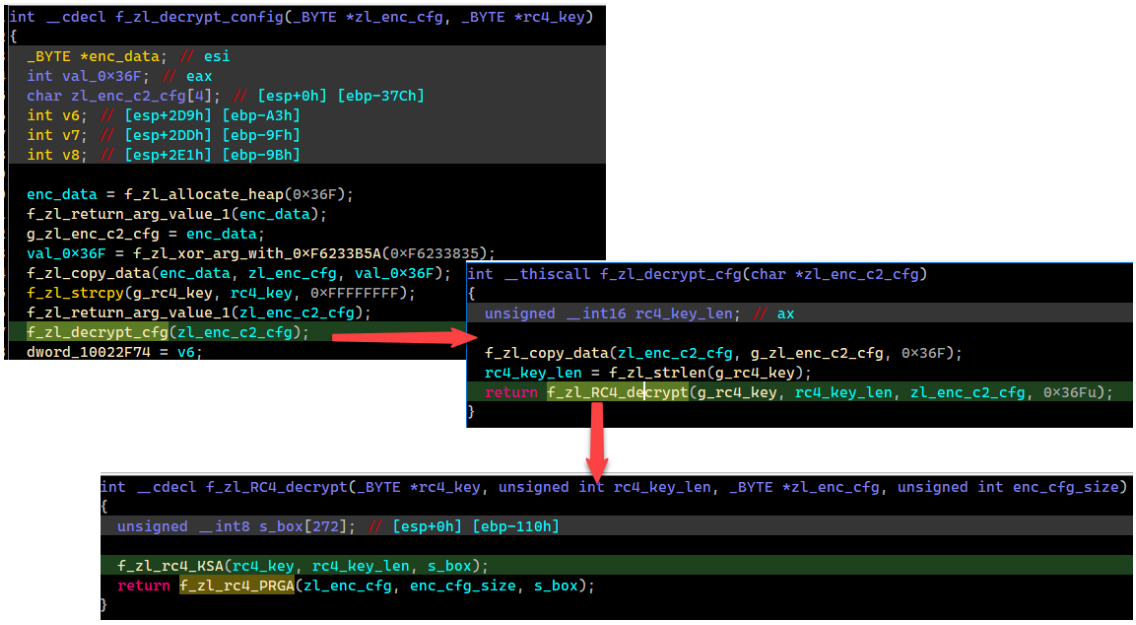


9. Giải mã cấu hình

Thông tin cấu hình của Zloader đã bị mã hóa và lưu tại section `.rdata`. Hàm giải mã nhận hai tham số truyền vào là cấu hình đã mã hóa và key dùng để giải mã:



Bên trong hàm `f_zl_decrypt_config` sẽ sử dụng thuật toán RC4 để giải mã dữ liệu:



Với toàn bộ thông tin đã phân tích được, viết lại bằng đoạn code IDAPython dưới đây để thực hiện giải mã:

```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()
```

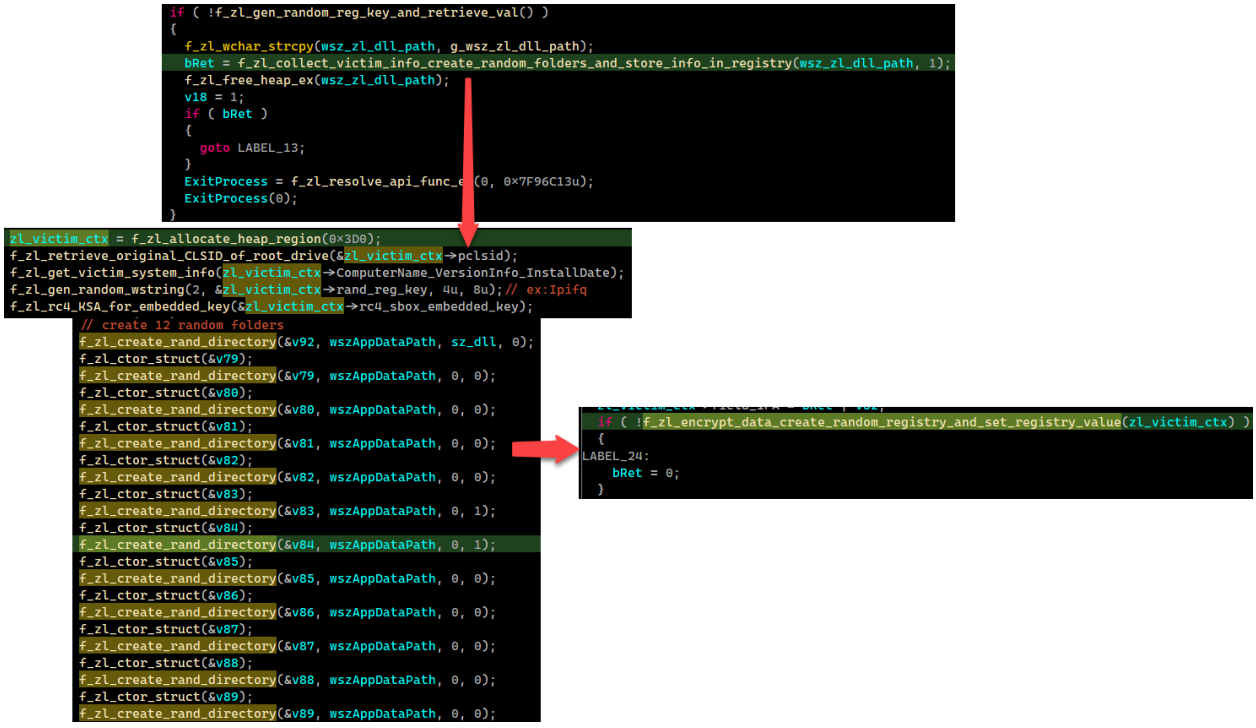
Kết quả thu được sau khi thực hiện script như sau:

Output window

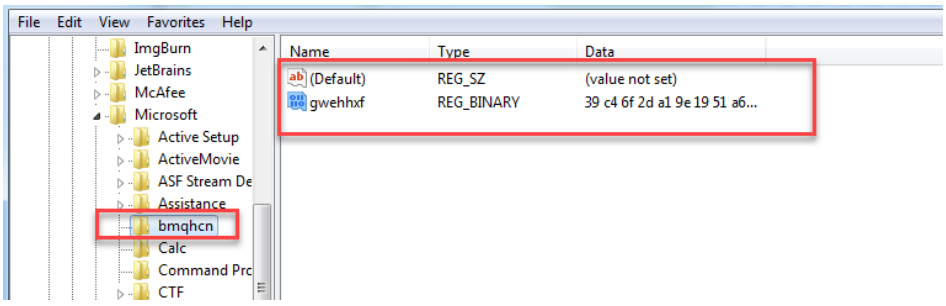
```
[*] Target address found at 0xa74ddL
[+] Bot name: 9092us
[+] Bot ID: 9092us
[+] Zloader C2 address:
        https://asdfghdsajkl.com/gate.php
        https://lkjhgfgsdshja.com/gate.php
        https://kjdhsasghjds.com/gate.php
        https://kdjwhqejqwij.com/gate.php
        https://iasudjghnasd.com/gate.php
        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. Lưu thông tin thu thập và cấu hình tại Registry

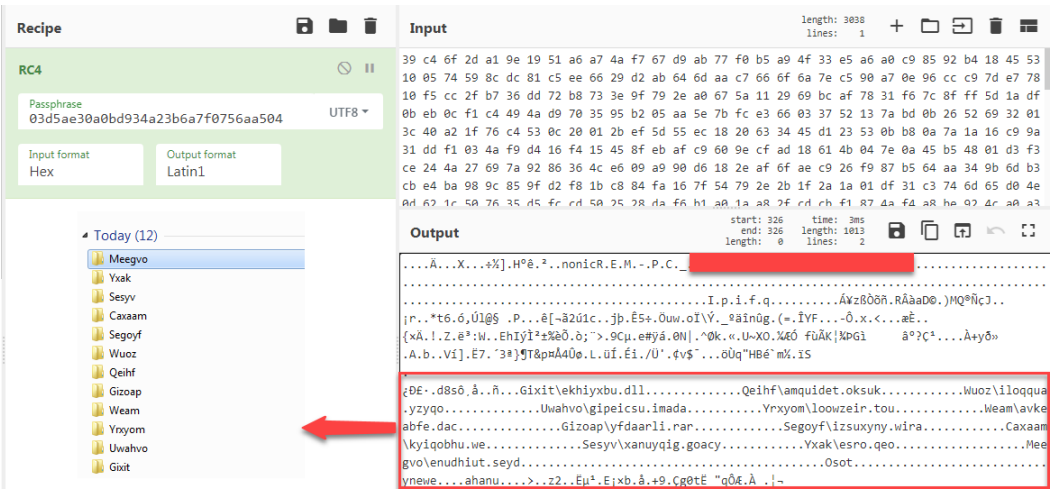
Khi lần đầu thực thi, Zloader sẽ thực hiện thu thập thông tin về victim gồm `volume_GUID`, `Computer_Name`, `Windows version`, `Install Date`, tạo các thư mục ngẫu nhiên tại `%APPDATA%`, tạo registry key ngẫu nhiên tại nhánh `HKEY_CURRENT_USER\Software\Microsoft`, sau đó mã hóa toàn bộ thông tin liên quan rồi lưu vào registry đã tạo:



Thông tin lưu tại Registry tương tự như sau:

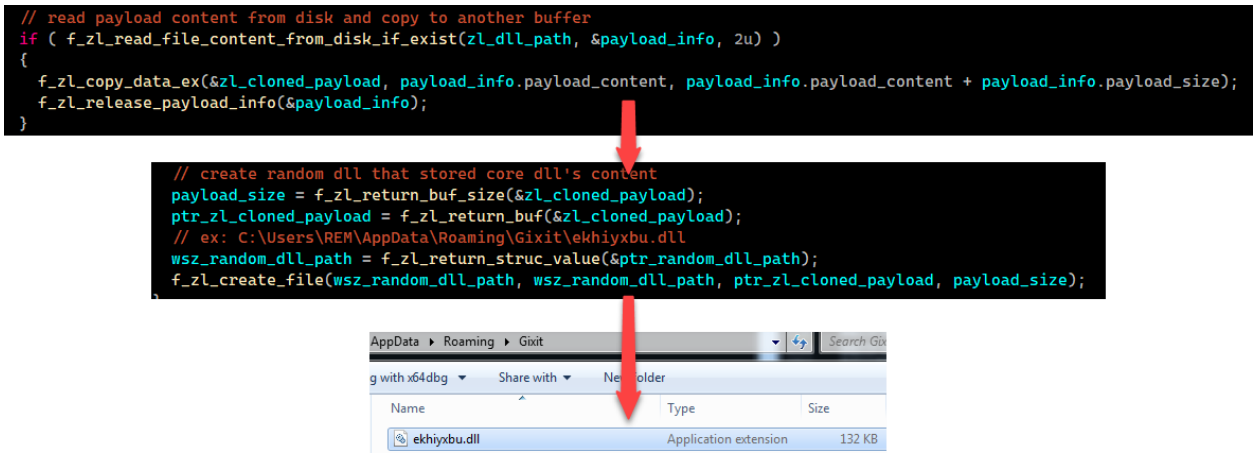


Để giải mã data lưu tại Registry trên, sử dụng embedded RC4 key đã giải mã được ở trên. Với sự hỗ trợ của **CyberChef** ta có thể dễ dàng thực hiện việc giải mã ra dữ liệu tương tư như hình dưới đây:

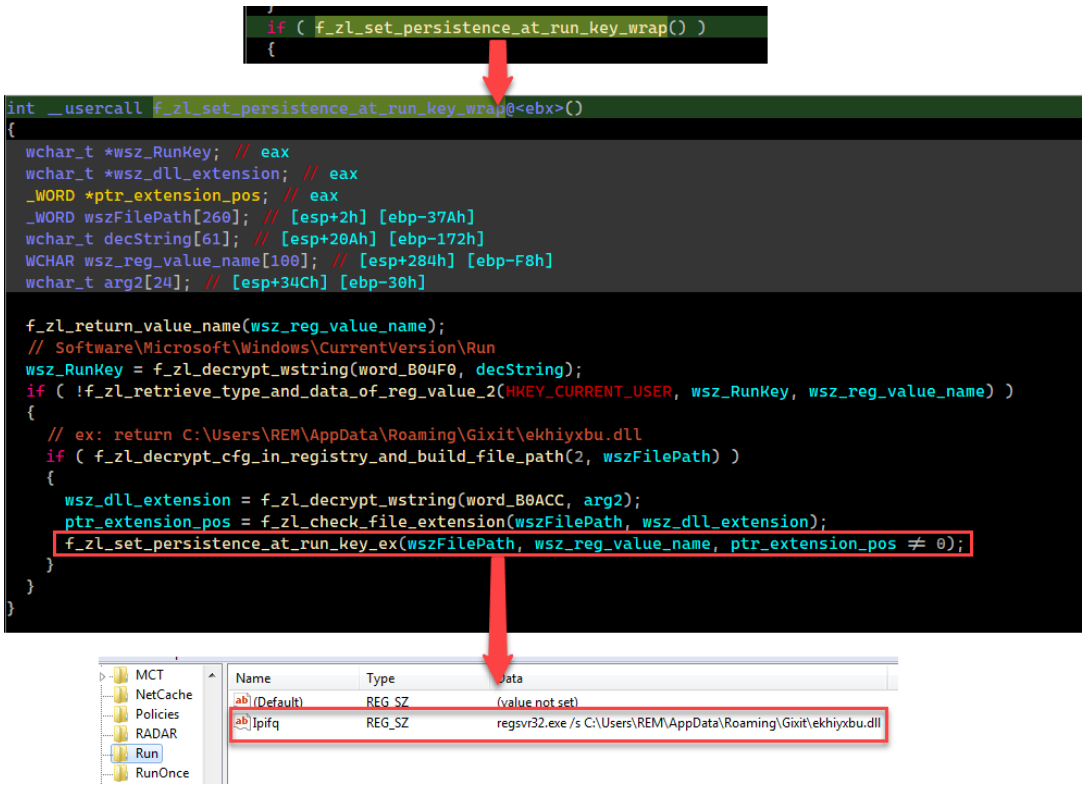


11. Tạo persistence key

Zloader thực hiện đọc toàn bộ nội dung của core Dll từ disk vào vùng nhớ, sau đó thực hiện ghi vào một random dll trong một thư mục đã tạo ở trên tại `%APPDATA%`:



Thực hiện tạo persistence key tại `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`:



12. Tham khảo

♦ 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