[Z2A]Bimonthly malware challege – Emotet (Back From the Dead)

Published: 2022-12-19 · Archived: 2026-04-05 22:23:36 UTC

A quick check of information related to sections of this sample shows that it may be crypted/packed to conceal the real malware inside the original sample, besides there is an extra section with an unusual name: **.text**

Load the sample into x64dbg, set a breakpoint at the **VirtualAlloc** API function, run payload by press **F9**. It will break at the **VirtualAlloc** function:

Execute till return (**Ctrl+F9**) and follow the allocated memory, trace over the `ret` instruction to return the Dll's code will reach the code area like the following:

To quickly get the Emotet core payload, set a bp at the `ret` command below the loop, then press **F9** to let the payload finish decrypting and fill core payload content to the allocated memory. The resulting core payload is decrypted as shown below:

Now, dump the above memory to disk, then fix total size of the payload to `0x2B800`, we get the final Emotet core Dll (Md5: *`577118e39051f0678a52f871f74cd675`*):

Load fixed core Dll above into IDA, go to the export function `DllRegisterServer` we see there are 2 sub routines as follows:

At `et_retrieve_api_addr (0x18000F174)` function, the code snippet does the following:

Continuing to dive into the `et_get_dll_base_from_hash (0x0180002960)` function, the process of getting the base address of the Dll will be as follows:

Based on the above pseudocode, rewrite the hash function in Python for the name of the Dll as follows:

We can write an IDAPython script that recovers the names of the DLLs that Emotet uses from these pre-computed hashes. The script performs the following tasks:

```
import idc, ida_enum, idautils, ida_bytes, idaapi, ida_bytes
most_common_dlls =
[ 'kernel32.dll' , 'user32.dll' , 'ntdll.dll' , 'shlwapi.dll' , 'iphlpapi.dll' , 'urlmon.dll' , 'ws2_32.dll' , 'cry
'comctl32.dll' , 'comdlg32.dll' , 'msvcrt.dll' , 'oleaut32.dll' , 'srsvc.dll' , 'winhttp.dll' , 'advpack.dll' ,
def calc_hash(dll_name):
    hash_value = 0x0
    module_name_list = []
    module_name_list = list (dll_name)
    for i in range ( len (module_name_list)):
        ch = ord (module_name_list[i])
        hash_value = ((hash_value << 0x10 ) & 0xFFFFFFFF ) + ((hash_value << 0x6 ) & 0xFFFFFFFF ) + ch -
    return ((hash_value ^ 0x106308C0 ) & 0xFFFFFFFF )
def get_enum_const(constant):
    all_enums = ida_enum.get_enum_qty()
    for i in range ( 0 , all_enums):
        enum_id = ida_enum.getn_enum(i)
        mask = ida_enum.get_first_bmask(enum_id)
        enum_constant = ida_enum.get_first_enum_member(enum_id, mask)
        name = ida_enum.get_enum_member_name(ida_enum.get_enum_member(enum_id, enum_constant, 0 , mask))
        if int (enum_constant) = = constant: return [name, enum_id]
        while True :
```

```
            enum_constant = ida_enum.get_next_enum_member(enum_id, enum_constant, mask)
            name = ida_enum.get_enum_member_name(ida_enum.get_enum_member(enum_id, enum_constant, 0 , mask))
            if enum_constant = = 0xFFFFFFFF :
                break
            if int (enum_constant) = = constant: return [name, enum_id]
    return None
def convert_offset_to_enum(addr):
    n_operand = 0
    if idc.print_insn_mnem(addr) = = "push" :
        constant = idc.get_operand_value(addr, 0 ) & 0xFFFFFFFF
    elif idc.print_insn_mnem(addr) = = "mov" :
        constant = idc.get_operand_value(addr, 1 ) & 0xFFFFFFFF
        n_operand = 1
    enum_data = get_enum_const(constant)
    if enum_data:
        name, enum_id = enum_data
        idc.op_enum(addr, n_operand, enum_id, 0 )
        return True
    else :
        return False
def enum_for_xrefs(func_addr, eid):
    for x in idautils.XrefsTo(func_addr, flags = 0 ):
        call_address = x.frm
        if ida_bytes.is_code(ida_bytes.get_full_flags(call_address)):
            pre_module_hash_addr = idaapi.get_arg_addrs(call_address)[ 1 ]
            if idc.print_insn_mnem(pre_module_hash_addr) = = "mov" and idc.get_operand_type(pre_module_hash_addr, 1
                print ( "[+] Target instruction found at 0x{address:x}" . format (address = pre_module_hash_addr))
                pre_module_hash = idc.get_operand_value(pre_module_hash_addr, 1 ) & 0xFFFFFFFF
                module_hash_addr = pre_module_hash_addr
            for dll_name in most_common_dlls:
                calced_hash = calc_hash(dll_name)
                if calced_hash = = pre_module_hash:
                    print ( '   [+] Module name: %s ==> Hash: 0x%x' % (dll_name, calced_hash))
                    ida_enum.add_enum_member(eid, '%s_hash' % dll_name, int (calced_hash), idaapi.BADADDR)
                    if convert_offset_to_enum(module_hash_addr):
                        print ( "   [+] Converted 0x%x to %s enumeration" % (idc.get_operand_value(module_hash_addr, 1
def main():
    target_function = 0x018000F174
    if ida_enum.get_enum( "MODULE_HASHES" ) ! = 0xffffffffffffffff :
```

```
60          print ( 'Enum already exists ...' )
61          return 0xffffffffffffffff
62      else :
63          eid = ida_enum.add_enum( 0 , "MODULE_HASHES" , ida_bytes.hex_flag())
64      enum_for_xrefs(target_function, eid)
65  if __name__ = = '__main__' :
66      main()
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
```

The pseudocode at the `et_get_api_addr_from_hash (0x0180025D84)` function does the following task:

Based on the above pseudocode, it can be seen that this hash function is similar to the hash function for Dll name above, we can rewrite it in Python in another way as follows:

Following this article, we can write python script to perform the following tasks:

Once JSON file has been generated, we can write another IDAPython script (similar to above script or refer to this code) does the following tasks:

To find the function that decrypt the strings, the fastest way is to find the function that calls the `LoadLibraryW` API because this function will take as an argument the name of the module to be loaded.

As the figure above, `sub_18002629C` will return the name of the module. The pseudocode at `sub_18002629C` stores its encrypted string as stack string, then calls the `et_decrypt_string (0x180025C58)` function to decrypt:

The `et_decrypt_string` function accepts parameters for the decryption process, including:

As mentioned above, the encrypted string has a variable length and the values of the encrypted string are dynamically calculated by Emotet before being stored to the stack. Therefore, it is difficult to get these values for writing script to perform decryption. Therefore, one of the most possible ways is to write a script that uses IDA Appcall feature to execute a call to the decryption function and receive the decrypted string as the return result.

End.
