Deobfuscating DanaBot’s API Hashing

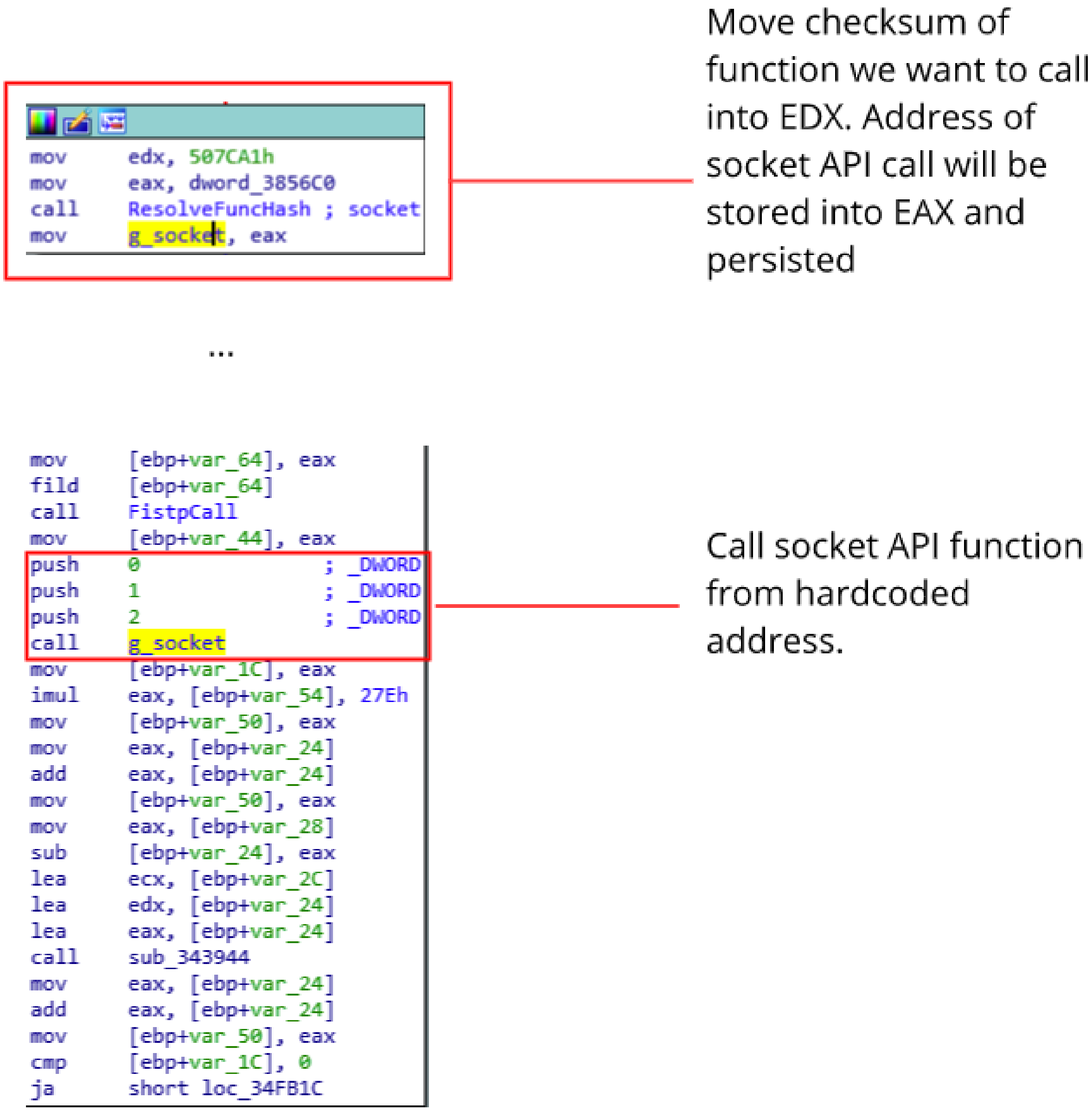
malwareandstuff.com/deobfuscating-danabots-api-hashing/

July 12, 2020

Published by **hackingump** on July 12, 2020

You probably already guessed it from the title’s name, API Hashing is used to obfuscate a binary in order to hide API names from static analysis tools, hindering a reverse engineer to understand the malware’s functionality.
A first approach to get an idea of an executable’s functionalities is to more or less dive through the functions and look out for API calls. If, for example a `CreateFileW` function is called in a specific subroutine, it probably means that cross references or the routine itself implement some file handling functionalities. This won’t be possible if API Hashing is used.

Instead of calling the function directly, each API call has a corresponding checksum/hash. A hardcoded hash value might be retrieved and for each library function a checksum is computed. If the computed value matches the hash value we compare it against, we found our target.



```
mov     edx, 507CA1h
mov     eax, dword_3856C0
call    ResolveFuncHash ; socket
mov     g_socket, eax
```

Move checksum of function we want to call into EDX. Address of socket API call will be stored into EAX and persisted

...

```
mov     [ebp+var_64], eax
fild    [ebp+var_64]
call    FistpCall
mov     [ebp+var_44], eax
push    0               ; _DWORD
push    1               ; _DWORD
push    2               ; _DWORD
call    g_socket
mov     [ebp+var_1C], eax
imul    eax, [ebp+var_54], 27Eh
mov     [ebp+var_50], eax
mov     eax, [ebp+var_24]
add     eax, [ebp+var_24]
mov     [ebp+var_50], eax
mov     eax, [ebp+var_28]
sub     [ebp+var_24], eax
lea     ecx, [ebp+var_2C]
lea     edx, [ebp+var_24]
lea     eax, [ebp+var_24]
call    sub_343944
mov     eax, [ebp+var_24]
add     eax, [ebp+var_24]
mov     [ebp+var_50], eax
cmp     [ebp+var_1C], 0
ja      short loc_34FB1C
```

Call socket API function from hardcoded address.

API Hashing used by DanaBot

In this case a reverse engineer needs to choose a different path to analyse the binary or deobfuscate it. This blog article will cover how the DanaBot banking trojan implements API Hashing and possibly the easiest way on how this can be defeated. The `SHA256` of the binary I am dissecting here is added at the end of this blog post.

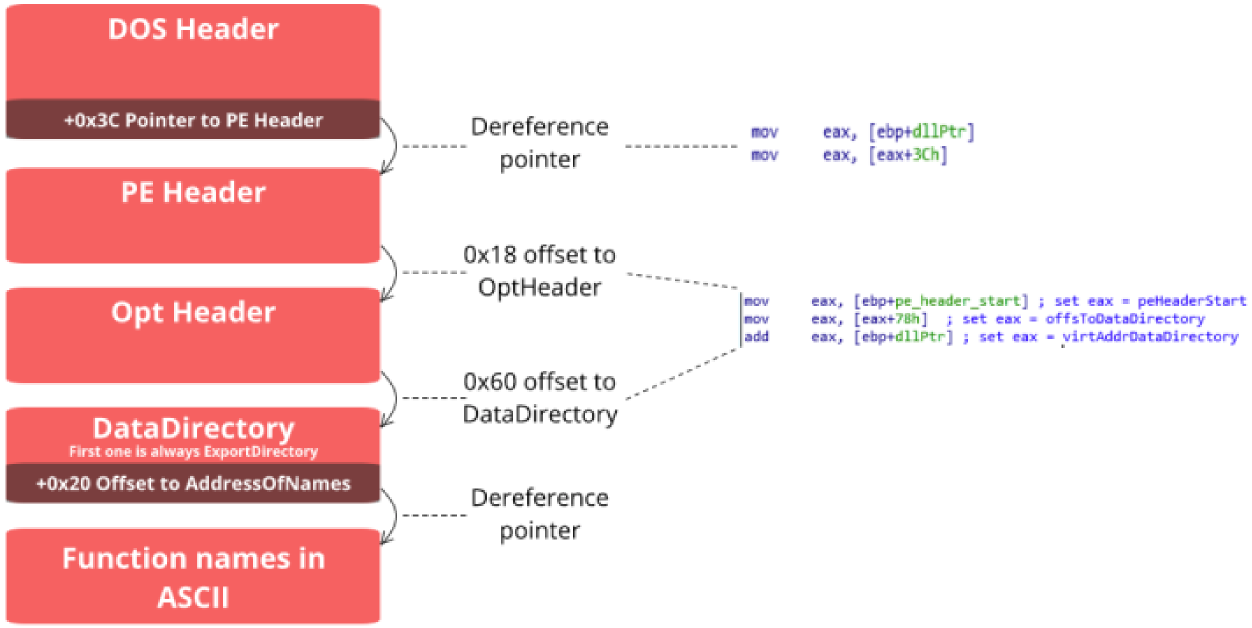
Deep diving into DanaBot

DanaBot itself is a banking trojan and has been around since atleast 2018 and was first discovered by ESET[1]. It is worth mentioning that it implements most of its functionalities in plugins, which are downloaded from the C2 server. I will focus on deobfuscating API Hashing in the first stage of DanaBot, a DLL which is dropped and persisted on the system, used to download further plugins.

Reversing the ResolvFuncHash routine

At the beginning of the function, the `EAX` register stores a pointer to the `DOS` header of the Dynamic Linked Library which, contains the function the binary wants to call. The corresponding hash of the yet unknown API function is stored in the `EDX` register. The routine also contains a pile of junk instructions, obfuscating the actual use case for this function.

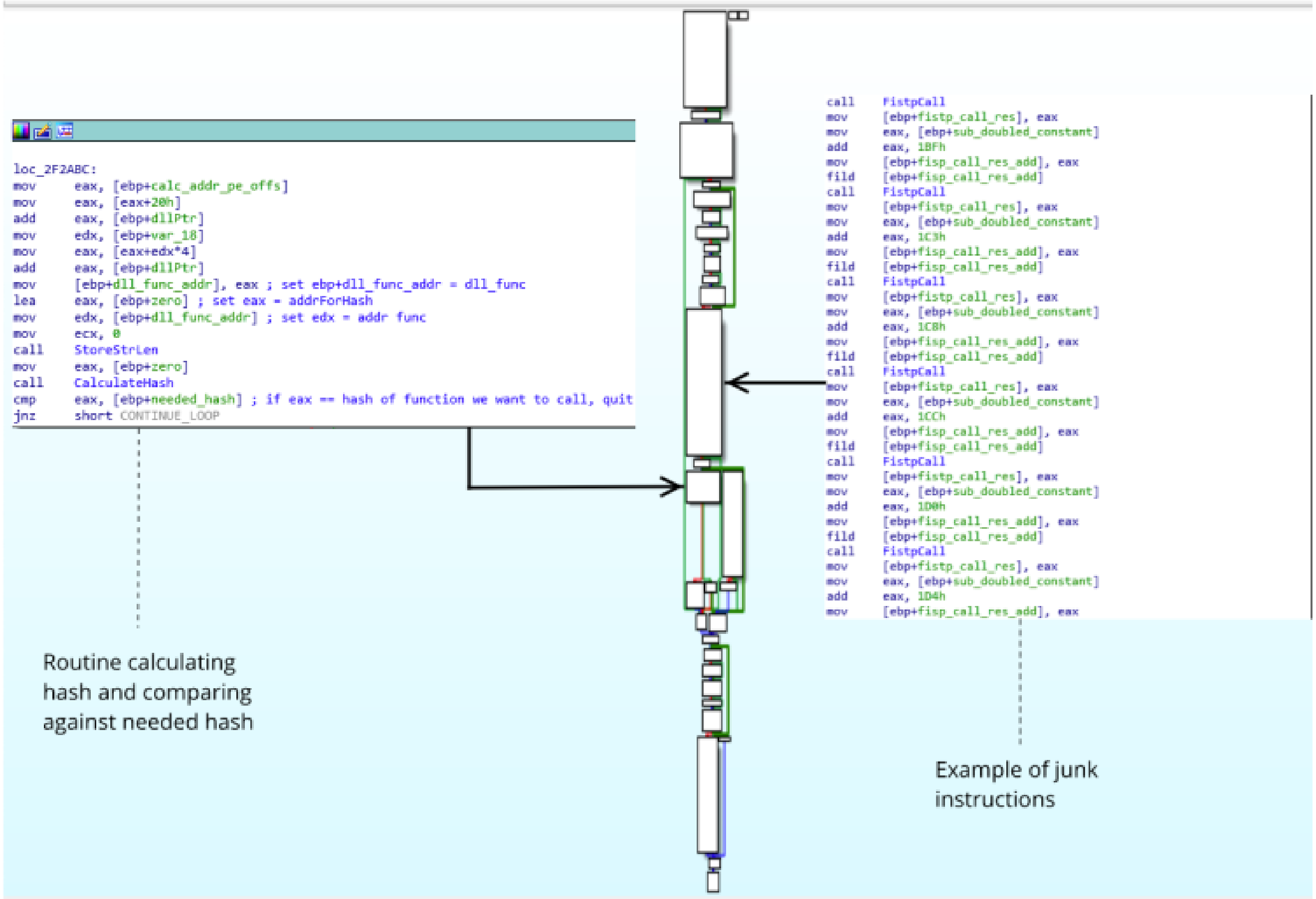
The hash is computed solely from the function name, so the first step is to get a pointer to all function names of the target library. Each DLL contains a table with all exported functions, which are loaded into memory. This Export Directory is always the first entry in the Data Directory array. The PE file format and its headers contain enough information to reach this mentioned directory by parsing header structures:



```
mov     eax, [ebp+dllPtr]
mov     eax, [eax+3Ch]
```

```
mov     eax, [ebp+pe_header_start] ; set eax = peHeaderStart
mov     eax, [eax+78h]  ; set eax = offsToDataDirectory
add     eax, [ebp+dllPtr] ; set eax = virtAddrDataDirectory
```

Cycling through the PE headers to obtain the ExportDirectory and AddressOfNames
In the picture below, you can see an example of the mentioned junk instructions, as well as the critical block, which compares the computed hash with the checksum of the function we want to call. The routine iterates through all function names in the Export Directory and calculates the hash.
The loop breaks once the computed hash matches the value that is stored in the `EDX` register since the beginning of this routine.



```
loc_2F2ABC:
mov     eax, [ebp+calc_addr_pe_offs]
mov     eax, [eax+20h]
add     eax, [ebp+dllPtr]
mov     edx, [ebp+var_18]
mov     eax, [eax+edx*4]
add     eax, [ebp+dllPtr]
mov     [ebp+dll_func_addr], eax ; set ebp+dll_func_addr = dll_func
lea     eax, [ebp+zero] ; set eax = addrForHash
mov     edx, [ebp+dll_func_addr] ; set edx = addr func
mov     ecx, 0
call    StoreStrLen
mov     eax, [ebp+zero]
call    CalculateHash
cmp     eax, [ebp+needed_hash] ; if eax == hash of function we want to call, quit
jnz     short CONTINUE_LOOP
```

Routine calculating hash and comparing against needed hash

```
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1BFh
mov     [ebp+fisp_call_res_add], eax
fild    [ebp+fisp_call_res_add]
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1C3h
mov     [ebp+fisp_call_res_add], eax
fild    [ebp+fisp_call_res_add]
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1C8h
mov     [ebp+fisp_call_res_add], eax
fild    [ebp+fisp_call_res_add]
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1CCh
mov     [ebp+fisp_call_res_add], eax
fild    [ebp+fisp_call_res_add]
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1D0h
mov     [ebp+fisp_call_res_add], eax
fild    [ebp+fisp_call_res_add]
call    FistpCall
mov     [ebp+fistp_call_res], eax
mov     eax, [ebp+sub_doubled_constant]
add     eax, 1D4h
mov     [ebp+fisp_call_res_add], eax
```

Example of junk instructions

Graph overview of obfuscated API Hashing function

Reversing the hashing algorithm

The hashing algorithm is fairly simple and nothing too complicated. Junk instructions and opaque predicates complicate the process of reversing this routine.

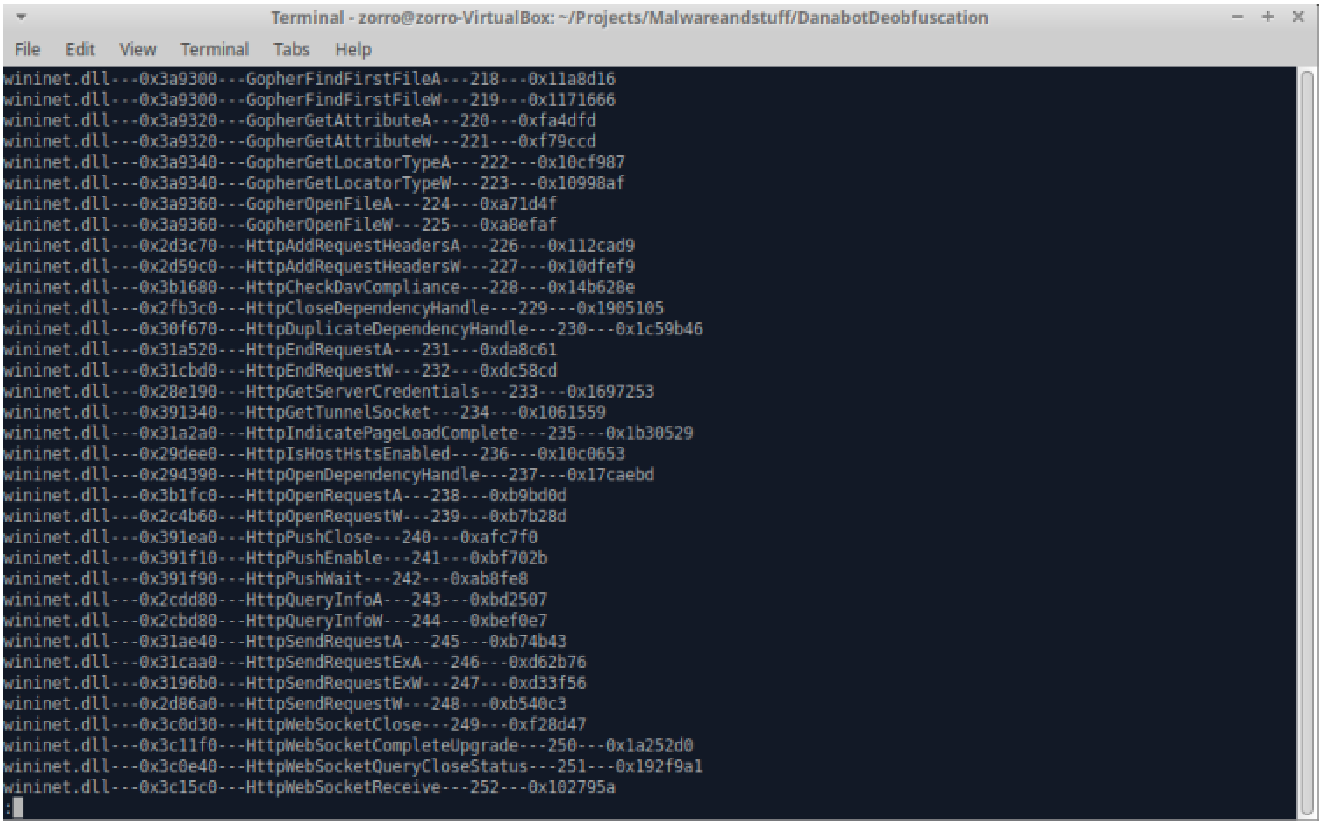
The algorithm takes the `nth` and the `stringLength-n-1th` char of the function name and stores them, as well as capitalised versions into memory, resulting in a total of 4 characters. Each one of those characters is `XOR'd` with the string length. Finally they are multiplied and the values are added up each time the loop is run and result in the hash value.

```
def get_hash(funcname):
    """Calculate the hash value for function name. Return hash value as integer"""
    strlen = len(funcname)
    # if the length is even, we encounter a different behaviour
    i = 0
    hashv = 0x0
    while i < strlen:
        if i == (strlen - 1):
            ch1 = funcname[0]
        else:
            ch1 = funcname[strlen - 2 - i]
        # init first character and capitalize it
        ch = funcname[i]
        uc_ch = ch.capitalize()
        # Capitalize the second character
        uc_ch1 = ch1.capitalize()
        # Calculate all XOR values
        xor_ch = ord(ch) ^ strlen
        xor_uc_ch = ord(uc_ch) ^ strlen
        xor_ch1 = ord(ch1) ^ strlen
        xor_uc_ch1 = ord(uc_ch1) ^ strlen
        # do the multiplication and XOR again with upper case character1
        hashv += ((xor_ch * xor_ch1) * xor_uc_ch)
        hashv = hashv ^ xor_uc_ch1
        i += 1
    return hashv
```

A python script for calculating the hash for a given function name is also uploaded on my github page[2] and free for everyone to use. I’ve also uploaded a text file with hashes for exported functions of commonly used DLLs.

Deobfuscation by Commenting

So now that we cracked the algorithm, we want to update our disassembly to know which hash value represents which function. As I’ve already mentioned, we want to focus on simplicity. The easiest way is to compute hash values for exported functions of commonly used DLLs and write them into a file.


Terminal - zorro@zorro-VirtualBox: ~/Projects/Malwareandstuff/DanabotDeobfuscation
File Edit View Terminal Tabs Help

```
wininet.dll---0x3a9300---GopherFindFirstFileA---218---0x11a8d16
wininet.dll---0x3a9300---GopherFindFirstFileW---219---0x1171666
wininet.dll---0x3a9320---GopherGetAttributeA---220---0xfa4dfd
wininet.dll---0x3a9320---GopherGetAttributeW---221---0xf79ccd
wininet.dll---0x3a9340---GopherGetLocatorTypeA---222---0x10cf987
wininet.dll---0x3a9340---GopherGetLocatorTypeW---223---0x10998af
wininet.dll---0x3a9360---GopherOpenFileA---224---0xa71d4f
wininet.dll---0x3a9360---GopherOpenFileW---225---0xa8efaf
wininet.dll---0x2d3c70---HttpAddRequestHeadersA---226---0x112cad9
wininet.dll---0x2d59c0---HttpAddRequestHeadersW---227---0x10dfef9
wininet.dll---0x3b1680---HttpCheckDavCompliance---228---0x14b628e
wininet.dll---0x2fb3c0---HttpCloseDependencyHandle---229---0x1905105
wininet.dll---0x30f670---HttpDuplicateDependencyHandle---230---0x1c59b46
wininet.dll---0x31a520---HttpEndRequestA---231---0xda8c61
wininet.dll---0x31cbd0---HttpEndRequestW---232---0xdc58cd
wininet.dll---0x28e190---HttpGetServerCredentials---233---0x1697253
wininet.dll---0x391340---HttpGetTunnelSocket---234---0x1061559
wininet.dll---0x31a2a0---HttpIndicatePageLoadComplete---235---0x1b30529
wininet.dll---0x29dee0---HttpIsHostHstsEnabled---236---0x10c0653
wininet.dll---0x294390---HttpOpenDependencyHandle---237---0x17caebd
wininet.dll---0x3b1fc0---HttpOpenRequestA---238---0xb9bd0d
wininet.dll---0x2c4b60---HttpOpenRequestW---239---0xb7b28d
wininet.dll---0x391ea0---HttpPushClose---240---0xafc7f0
wininet.dll---0x391f10---HttpPushEnable---241---0xbf702b
wininet.dll---0x391f90---HttpPushWait---242---0xab8fe8
wininet.dll---0x2cdd80---HttpQueryInfoA---243---0xbd2507
wininet.dll---0x2cbd80---HttpQueryInfoW---244---0xbef0e7
wininet.dll---0x31ae40---HttpSendRequestA---245---0xb74b43
wininet.dll---0x31caa0---HttpSendRequestExA---246---0xd62b76
wininet.dll---0x3196b0---HttpSendRequestExW---247---0xd33f56
wininet.dll---0x2d86a0---HttpSendRequestW---248---0xb540c3
wininet.dll---0x3c0d30---HttpWebSocketClose---249---0xf28d47
wininet.dll---0x3c11f0---HttpWebSocketCompleteUpgrade---250---0x1a252d0
wininet.dll---0x3c0e40---HttpWebSocketQueryCloseStatus---251---0x192f9a1
wininet.dll---0x3c15c0---HttpWebSocketReceive---252---0x102795a
:
```

Generated hashes

With this file, we can write an `IdaPython` script to comment the library function name next to the Api Hashing call. Luckily the Api Hashing function is always called with the same pattern:

- Move the wanted hash value into the `EDX` register
- Move a `DWORD` into `EAX` register

First we retrieve all `XRefs` of the Api Hashing function. Each `XRef` will contain an address where the Api Hashing function is called at, which means that in atleast the 5 previous instructions, we will find the mentioned pattern. So we will fetch the previous instruction until we extract the wanted hash value, which is being pushed into `EDX` . Finally we can use this immediate to extract the corresponding api function from the hash values we have generated before and comment the function name next to the `Xref` address.

```
def add_comment(addr, hashv, api_table):
    """Write a comment at addr with the matching api function.Return True if a
corresponding api hash was found."""
    # remove the "h" at the end of the string
    hashv = hex(int(hashv[:-1], 16))
    keys = api_table.keys()
    if hashv in keys:
        apifunc = api_table[hashv]
        print "Found ApiFunction = %s. Adding comment." % (apifunc,)
        idc.MakeComm(addr, apifunc)
        comment_added = True
    else:
        print "Api function for hash = %s not found" % (hashv,)
        comment_added = False
    return comment_added


def main():
    """Main"""
    f = open(
        "C:\\Users\\luffy\\Desktop\\Danabot\\05-07-
2020\\Utils\\danabot_hash_table.txt", "r")
    lines = f.readlines()
    f.close()
    api_table = get_api_table(lines)
    i = 0
    ii = 0
    for xref in idautils.XrefsTo(0x2f2858):
        i += 1
        currentaddr = xref.frm
        addr_minus = currentaddr - 0x10
        while currentaddr >= addr_minus:
            currentaddr = PrevHead(currentaddr)
            is_mov = GetMnem(currentaddr) == "mov"
            if is_mov:
                dst_is_edx = GetOpnd(currentaddr, 0) == "edx"
                # needs to be edx register to match pattern
                if dst_is_edx:
                    src = GetOpnd(currentaddr, 1)
                    # immediate always ends with 'h' in IDA
                    if src.endswith("h"):
                        add_comment(xref.frm, src, api_table)
                        ii += 1
    print "Total xrefs found %d" % (i,)
    print "Total api hash functions deobfuscated %d" % (ii,)


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

Conclusion

As reverse engineers, we will probably continue to encounter Api Hashing in various different ways. I hope I was able to show you some quick & dirty method or give you at least some fundament on how to beat this obfuscation technique. I also hope that, the next time a blue team fellow has to analyse DanaBot, this article might become handy to him and saves him some time reverse engineering this banking trojan.

IoCs

- Dropper = `e444e98ee06dc0e26cae8aa57a0cddab7b050db22d3002bd2b0da47d4fd5d78c`
- DLL = `cde01a2eeb558545c57d5c71c75e9a3b70d71ea6bbeda790a0b871fcb1b76f49`