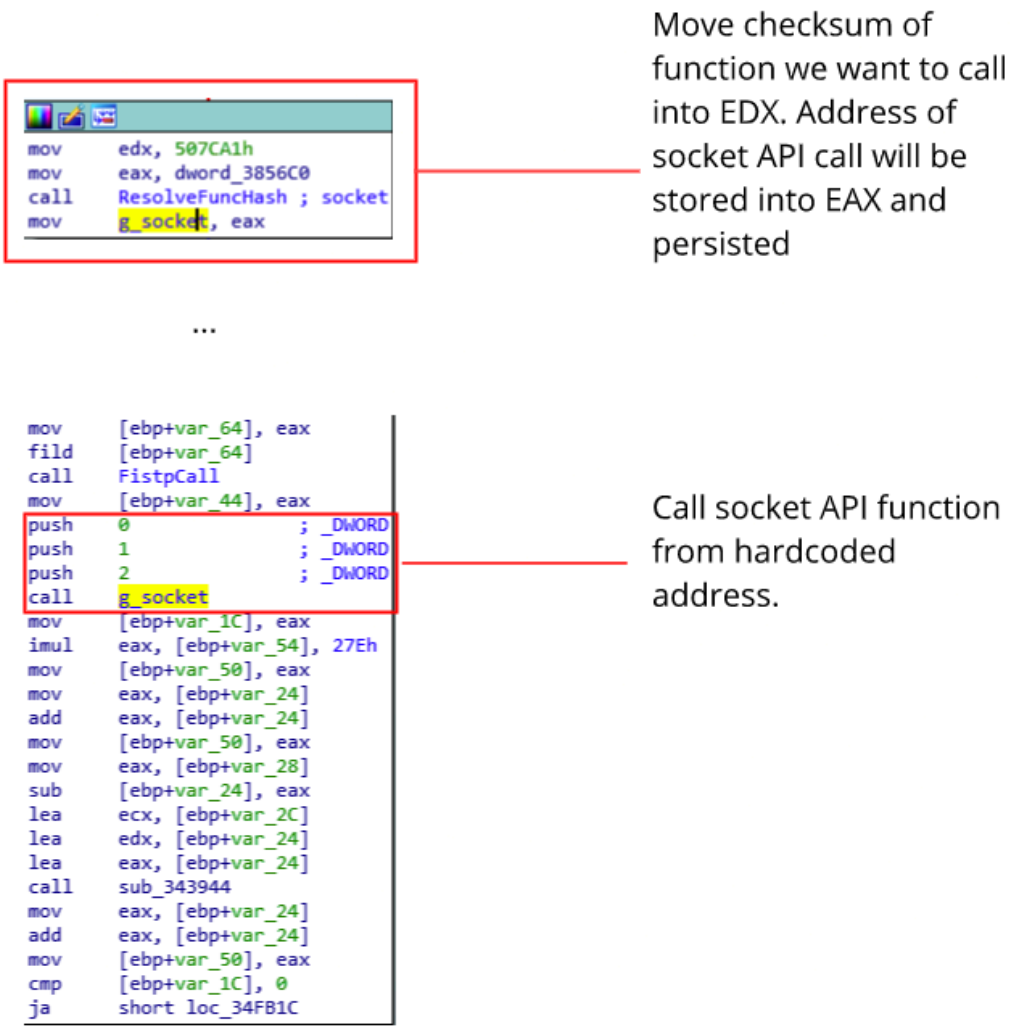
Deobfuscating DanaBot's API Hashing

Published: 2020-07-12 · Archived: 2026-04-05 23:43:10 UTC

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.



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.

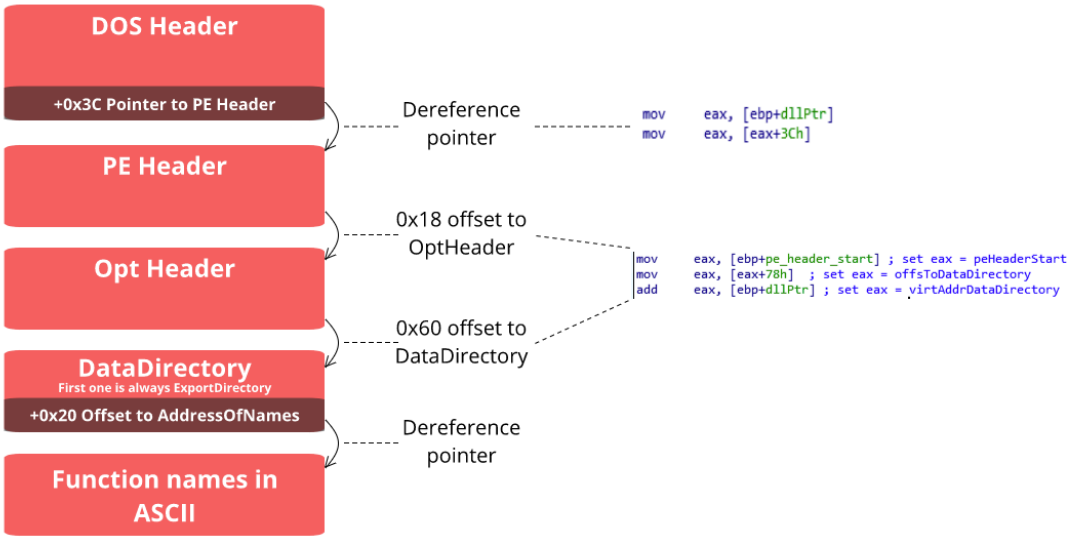
Deep diving into DanaBot

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

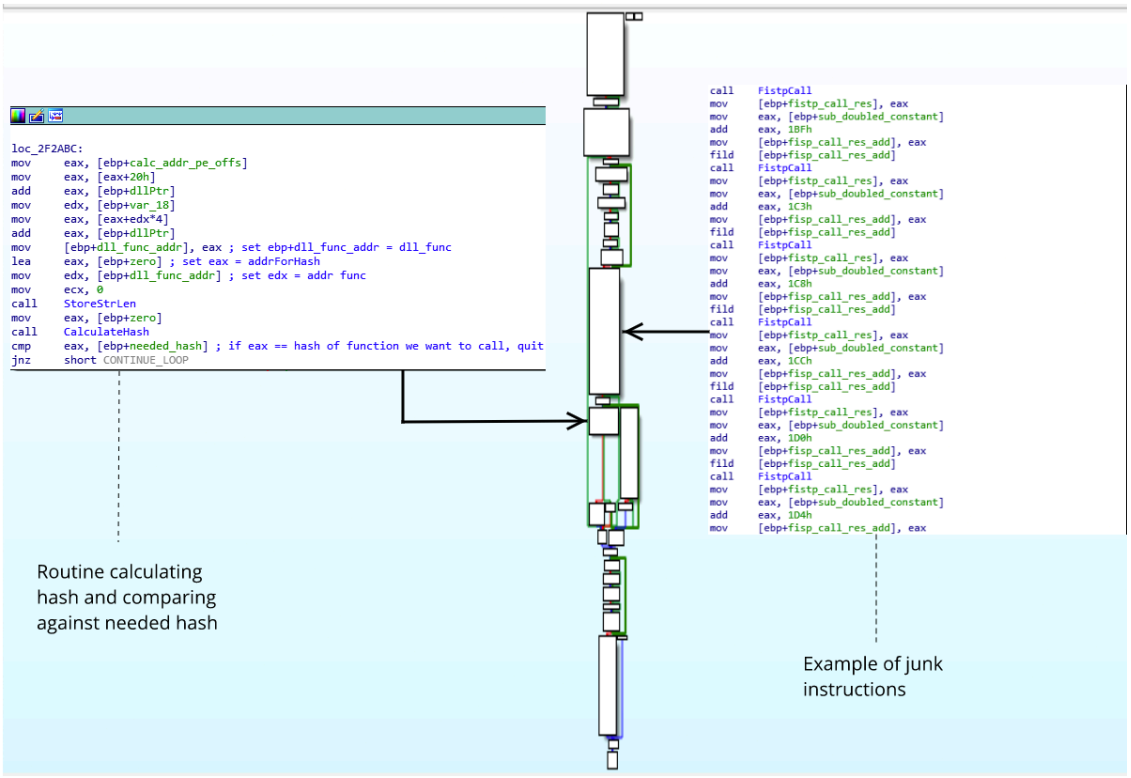
At the beginning of the function, the `EAX` register stores a pointer to the `DOS` header of the Dynamic Linked Library which, contains the function the binary wants to call. The corresponding hash of the yet unknown API function is stored in the `EDX` register. The routine also contains a pile of junk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



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.



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.

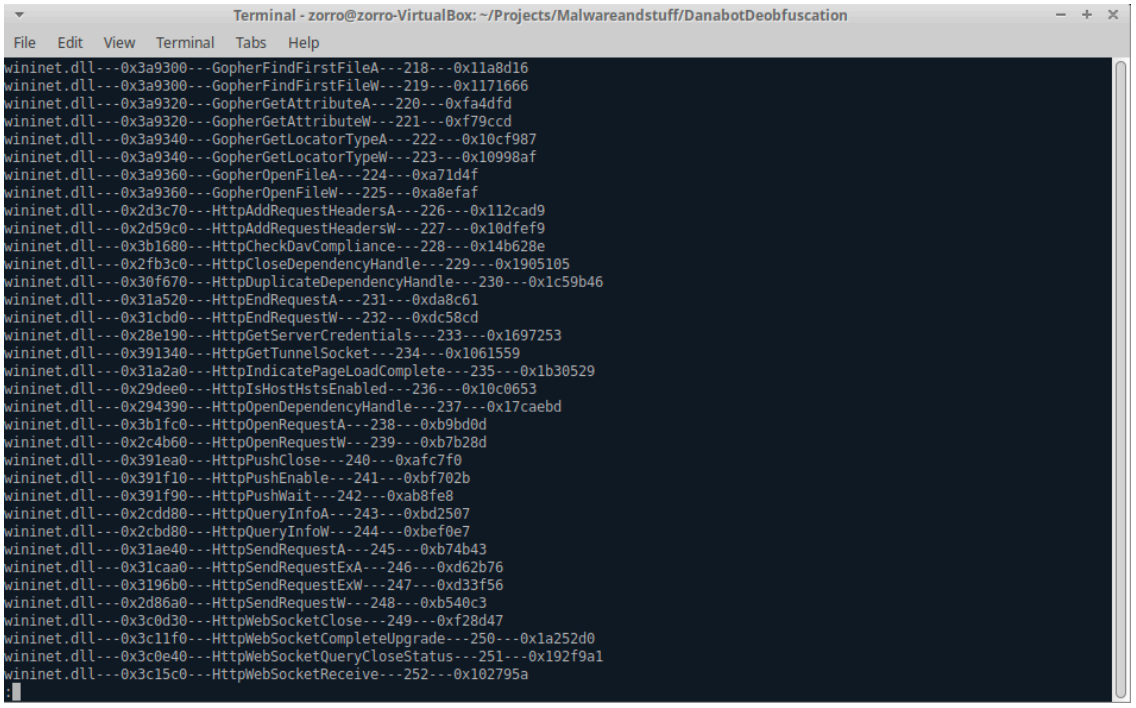
```
1  def get_hash(funcname):
2      strlen = len (funcname)
3      i = 0
4      hashv = 0x0
5      while i < strlen:
6          if i = = (strlen - 1 ):
7              ch1 = funcname[ 0 ]
8          else :
```

```
9                ch1 = funcname[strlen - 2 - i]
10           ch = funcname[i]
11           uc_ch = ch.capitalize()
12           uc_ch1 = ch1.capitalize()
13           xor_ch = ord (ch) ^ strlen
14           xor_uc_ch = ord (uc_ch) ^ strlen
15           xor_ch1 = ord (ch1) ^ strlen
16           xor_uc_ch1 = ord (uc_ch1) ^ strlen
17           hashv += ((xor_ch * xor_ch1) * xor_uc_ch)
18           hashv = hashv ^ xor_uc_ch1
19           i += 1
20       return hashv
21
22
23
24
25
26
```

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.



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.

```
1  def add_comment(addr, hashv, api_table):
2      hashv = hex ( int (hashv[: - 1 ], 16 ))
3      keys = api_table.keys()
4      if hashv in keys:
5          apifunc = api_table[hashv]
6          print "Found ApiFunction = %s. Adding comment." % (apifunc,)
7          idc.MakeComm(addr, apifunc)
8          comment_added = True
```

```
9       else :
10          print "Api function for hash = %s not found" % (hashv,)
11          comment_added = False
12      return comment_added
13  def main():
14      f = open (
15          "C:\\Users\\luffy\\Desktop\\Danabot\\05-07-2020\\Utils\\danabot_hash_table.txt" ,
    "r" )
16
17      lines = f.readlines()
18      f.close()
19      api_table = get_api_table(lines)
20      i = 0
21      ii = 0
22      for xref in idautils.XrefsTo( 0x2f2858 ):
23          i + = 1
24          currentaddr = xref.frm
25          addr_minus = currentaddr - 0x10
26          while currentaddr > = addr_minus:
27              currentaddr = PrevHead(currentaddr)
28              is_mov = GetMnem(currentaddr) = = "mov"
29              if is_mov:
30                  dst_is_edx = GetOpnd(currentaddr, 0 ) = = "edx"
31                  if dst_is_edx:
32                      src = GetOpnd(currentaddr, 1 )
33                      if src.endswith( "h" ):
34                          add_comment(xref.frm, src, api_table)
                          ii + = 1
```

```
35      print "Total xrefs found %d" % (i,)
36      print "Total api hash functions deobfuscated %d" % (ii,)
37  if __name__ == '__main__':
38      main()
39
40
41
42
43
44
45
46
47
```

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`

Source: https://malwareandstuff.com/deobfuscating-danabots-api-hashing/