API Hashing in the Zloader malware – nullteilerfrei

By born

Published: 2021-01-04 · Archived: 2026-04-05 12:58:03 UTC

Directing your attention as a reverse engineer is key for not wasting your life looking at irrelevant code. This blag post will use an anti-analysis technique used in the Zloader malware as an example to practice this art. We will also take a short detour into code-level obfuscation and are going to re-implement the API hashing function from Zloader in Python.

This post is aimed towards reverse engineering beginners that have already heard about API hashing. If you don't know, what Ghidra is or how to use it, you will need to brush over some parts of this post.

What is API Hashing

In case, you don't follow this blag closely, I'll quickly summarize, what I mean with API hashing: if a malware author doesn't want to include API function names in the malware - neither in the import address table nor as strings somehow passed to `GetProcAddress` for example - they can use API hashing. This involves calculating some sort of hash for each combination of DLL file name and API function name (often only the latter) and inclusion of those hashes in the malware instead.

Given an API hash, the malware can enumerate all loaded DLLs and their exported functions to calculate hashes with the same custom algorithm and compare the result to the given hash, ultimately enabeling resolution of the corresponding API function. The topic is covered more thoroughly in the post about API hashing in the REvil ransomware.

Identifying the API Resolution Function

We will be looking at the Zloader sample with SHA256 hash

```
4029f9fcba1c53d86f2c59f07d5657930bd5ee64cca4c5929cbd3142484e815a
```

In another blag post about string obfuscation, we stumbled upon the API hashing function of Zloader: The function `FUN_030a3170` is called in 190 places and each time, it receives some small integral number and a larger value fitting into a `DWORD`. This alone slightly smells like API hashing but a dead give-away is the fact that the returned value of the function is always `CALL`ed shortly after:

```
 pcVar1 = (code *)FUN_030a3170(0,0x6aa0e84);
iVar2 = (*pcVar1)(2,0);
```

So let us rename `FUN_030a3170` to `ev_ResolveApi` and take note of a few argument combinations:

First Argument	Second Argument	Call
`0`	`0x6aa0e84`	`f(2,0)`
`1`	`0xf3c7b77`	`f(0,puVar5,puVar4,0xcf0000,0x80000000,0x80000000,0x80000000,0x80000000,0,0,uVar3,0);`

First Argument	Second Argument	Call
`9`	`0xabc78f7`	`f(puVar1,1,&local_14,0)`

A quick look at the decompiled code of the function should instantly make you loose interested in reverse engineering it top to bottom: It looks very convoluted and long. But let us not give up but leap our way to the goal.

The First Argument

So let's skip over everything and only realize that the first argument is used to index the array named `PTR_DAT_030bc2ec`. The data `PTR_DAT_030bc2ec[param_1]` is then passed to the string deobfuscation function, analyzed in a previous blag post. Double clicking the array will show the following in the assembly listing view:

```
                PTR_DAT_030bc2ec                        XREF[1]:     FUN_030a3170:030a3224(R)
030bc2ec e8 c3 0b 03                    addr        DAT_030bc3e8                                    = 32h
030bc2f0 f5 c3 0b 03                    addr        DAT_030bc3f5                                    = 2Ch
030bc2f4 00 c4 0b 03                    addr        DAT_030bc400                                    = 37h
...
030bc348 e8 c3 0b 03                    addr        DAT_030bc3e8                                    = 32h
030bc34c db c4 0b 03                    addr        DAT_030bc4db                                    = 3Bh
```

Ghidra identified each of the array entries as a pointer. So let's interpret each entry of this array as an obfuscated string and decrypt it:

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`

Index	DLL Name
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`

Hence the first argument to the function is an index into the above listed array of DLL names and hence almost certainly used to specify the DLL to use when resolving an API function.

Note the choice of words here: I did not say that I am sure that the argument is used to specify the DLL which is used to resolve a function; but only, that I am almost certain. When reverse engineering like this, you should keep the amount of certainty for every statement in the back of your head. So if something doesn't make sense anymore, you can track back and more easily assess, where to dig deeper. In this case, I don't see a lot of other possibilities, what this DLL name will be used for otherwise.

The second argument

Let us do a similar trick with the second argument: don't reverse engineer the whole function but just look at the four places, where the second argument appears while keeping in mind that we believe it to specify an API hash of the function to be resolved:

```
 uVar1 = param_2 % uVar1;
...
uVar2 = FUN_030b9a70(uVar2,param_2,0,0);
...
pcVar5 = FUN_030a3620(iVar4,param_2);
...
*local_14 = param_2;
```

The alleged API hash is passed into the two functions `FUN_030b9a70` and `FUN_030a3620`. We will now take a look at the two, keeping an eye out for code that calculates an API hash to then compare it to the passed argument.

On first glance, the first of the two functions looks promising: it contains some arithmetic operations and calls a few other functions. But looking at the return value, one can instantly see that it either returns `0xa1` or `0`. So these are probably not the droids we are looking for. The second function - `FUN_030a3620` - looks at least as promising as the first one: it contains the two constants `0x60` and `0x18` at the very top and also uses a few (nested) loops.

So if someone would point a gun to my head and ask me for an opinion, which of the two you should investigate further, I'd definitely choose the second. And you should always imagine that someone is pointing a gun to your head while reverse engineering. We don't have no time for anything else.

I nearly forgot to repeat a life hack from the very same blag post already referenced a few times, which should clear up, why I got so excited about the two constants `0x60` and `0x18`: `0x18` is the offset of the *Optional Header* within the *PE header* and `0x60` is the offset of the *Data Directories* within that Optional Header. We don't need to understand everything here but can simply assume that there is some sort of PE parsing going on (that is parsing of the Windows *Portable Executable* file format). And you need PE parsing to list exports from loaded DLLs, hence you need PE parsing to calculate API hashes of loaded functions.

Lucky for us, the API hash passed in as an argument is only used in one single line:

```
if (uVar3 == param_2) {
```

And since it does not make much sense to compare an API hash with anything else but another API hash, it is reasonable to assume that `uVar3` also contains an API hash. It is also plausible that it contains the API hash calculated by the malware based on loaded DLL names and their exported functions. Since the value of `uVar3` comes out of `FUN_030a3140` let's rename that function to `pr_ApiHash`. It receives `local_90` and `-1` as arguments. So let's just assume for now that `local_90` is somehow derived from DLL and function names and dive into `pr_ApiHash`.

The API Hashing Function

Lazy time is over now. We finally need to understand some code and what exactly, `pr_ApiHash` does with its arguments to arrive at an API hash. Since we already assumed that the first argument contains some data derived from DLL and function names, let us focus on the second argument for now: It is first compared with `-1` - which makes sense because we already observed this value as an argument - and another function, `FUN_030a2fe0`, is called with the alleged DLL and function names as arguments. Let's look into `FUN_030a2fe0` and retype its argument to `BYTE *`:

```
int __cdecl FUN_030a2fe0(BYTE *param_1) {
 int iVar1;
 int iVar2;

 if (param_1 != (BYTE *)0x0) {
   iVar2 = -1;
   do {
     iVar1 = iVar2 + 1;
     iVar2 = iVar2 + 1;
   } while (param_1[iVar1] != '\0');
   return iVar2;
 }
 return 0;
}
```

If the passed data is the `NULL` pointer, the function will return `0`. Otherwise, it will initialize the variable `iVar2` with `-1` and increase value passed into the function until it is the `NULL` terminator. During each iteration, the return variable `iVar2` is incremented by one. Since this is a do-while loop, this incrementation happens at least one time. Staring at this code a bit more, you can see that this function will interpret the passed argument as a string and return its length. This is

huge because we can now guess the type of the argument and also the type of the variable passed into this function: it probably is just `char *` as opposed to some complex data structure derived from DLL and function names.

So let us rename `FUN_030a2fe0` to `strlen` and retype the two arguments to `pr_ApiHash` according to what we just learned. While we are at it, realize that `uVar4` is the value returned from `pr_ApiHash` and rename that variable to `ApiHash`.

```
uint __cdecl pr_ApiHash(char *SomeString,int StrLen) {
 byte bVar1;
 uint uVar2;
 uint uVar3;
 uint ApiHash;

 if (StrLen == -1) {
   StrLen = strlen(SomeString);
 }
 ApiHash = 0;
 if ( (SomeString != (char *)0x0) && (0 < StrLen) ) {
   ApiHash = 0;
   do {
     bVar1 = FUN_030a5260();
     ApiHash = (uint)(byte)*SomeString + (ApiHash << (bVar1 & 0x1f));
     if ( (ApiHash & 0xf0000000) != 0 ) {
       uVar3 = (ApiHash & 0xf0000000) >> 0x18;
       uVar2 = FUN_030a9b90(0xffffffff,0xffffffff,0);
       ApiHash = FUN_030aeef0(~(uVar2 | ~ApiHash | uVar3),(uVar2 | ~ApiHash) & uVar3,(HINSTANCE)0x0);
     }
     SomeString = (char *)( (byte *)SomeString + 1);
     StrLen = StrLen + -1;
   } while (StrLen != 0);
 }
 return ApiHash;
}
```

Code-Level Obfuscation

Now we need to get *really* un-lazy. There are three functions used during calculation of the API hash with names `FUN_030a5260`, `FUN_030a9b90` and, `FUN_030aeef0`. Each of these functions needs special attention.

- Even though the return value of `FUN_030a5260` is used, Ghidra did not correctly guess the function signature and somehow determine that it is a `void` function. Change the signature (Hotkey `F`), check "Use Custom Storage" and change the returned data type to `int` and the storage location to `EAX`. Choosing `EAX` is often correct and I suggest to just try it and justify later if the resulting decompiled code makes sense. Again, purely for time-efficiency reasons. The result will be a convoluted function that ends with `return _DAT_030be374 ^ 0xa2df808b`. Follow `_DAT_030be374` and change the type to `ddw` (Hotkey `D` three times). This reveals that this global variable contains the value `0xA2DF808F`. Xor-ing with `0xa2df808b` results in `4`. Hence, we can rename `FUN_030a5260` to `Return4`.
- Similarly, `FUN_030a9b90` is identified to be a `void` function. Performing the same procedure as above (adapt the signature to return an `int` in `EAX`) will lead to a very simple decompiled function that only Xors the first two arguments. Hence you can rename it to `Xor` (and also remove the last parameter if you feel tidy).

- Finally, `FUN_030aeef0` only seems to calculate the binary or of the two parameters, hence rename it to `Bor` (and, again, remove the third parameter if you like).

The above three functions are probably caused by anti-analysis techniques employed by the malware author. The technique used in the first function is called "constant unfolding" because it is the opposite of the compiler optimization technique called constant folding. Constant folding evaluates constant expressions during compile time to avoid unnecessary calculations during run time. Constant unfolding does the reverse: it identifies constants - `4` in this case - and replaces them with some sort of calculation - `_DAT_030be374 ^ 0xa2df808b` in this case - during compile/build time.

Similarly, the other two function employ the opposite of the compiler optimization technique called inlining: Instead of performing the arithmetic operation in-line (here, a simple Xor / Binary Or), a function is called that performs this operation. In addition to that, unnecessary instructions where inserted into this un-in-lined function that make the code harder to read. Specifically the condition of a branch like

```
if ( ( ( (param_2 == 0xb4c6d61) && (fuLoad != param_1)) &&
   (in_stack_0000000c != (HINSTANCE)0xb4c6d61)) &&
   ( ( (int)in_stack_0000000c << 7 | (uint)in_stack_0000000c) == 0)) {
```

that is never taken is called opaque predicate. In addition to that, both function contain some jung instructions without any side effects. A lot of them have been removed by the Ghidra decompiler, but since identifying those is hard - even heuristically - some still remain.

Re-Implementing the Hashing Function

Offentimes you want to emulate API hashing in a different language because it enables you to annotate API resolution calls during static analysis. Re-implementing an algorithm will often also get rid of any implementational details that may have even been introduces by a compiler or obfuscator during built. This in turns eases identification of overlaps in the hashing method between different malware families, which in turn may indicate a link between the families.

After performing the above-described steps and some minor adjustments to variable names in the `pr_ApiHash` function, we end up with the following:

```
uint __cdecl pr_ApiHash(char *SomeString,int StrLen) {
 byte Four;
 uint Mask;
 uint HighNibble;
 uint ApiHash;

 if (StrLen == -1) {
   StrLen = strlen(SomeString);
 }
 ApiHash = 0;
 if ( ( SomeString != (char *)0x0 ) && (0 < StrLen) ) {
   ApiHash = 0;
   do {
     _Four = Return4();
     ApiHash = (uint)(byte)*SomeString + (ApiHash << ( (byte)_Four & 0x1f ));
     if ( (ApiHash & 0xf0000000) != 0 ) {
       HighNibble = (ApiHash & 0xf0000000) >> 0x18;
       Mask = Xor(0xffffffff,0xffffffff);
```

```
        ApiHash = Bor(~(Mask | ~ApiHash | HighNibble),(Mask | ~ApiHash) & HighNibble);
      }
      SomeString = (char *)( (byte *)SomeString + 1 );
      StrLen = StrLen + -1;
    } while (StrLen != 0);
  }
  return ApiHash;
}
```

You can just copy this into a text editor and change the syntax a bit until it is valid code of your language of choice. If your language natively supports bigints (like Python), better make sure to sprinkle it with enough `& 0xffffffff`. I decided to use Python for now and since I very much enjoy totally unnecessary optimizations, I ended up with the following:

```
def calc_hash(function_name):
    mask = 0xf0000000
    ret = 0
    for c in function_name:
        ret = ord(c) + (ret << 0x4)
        if ret & mask:
            ret = (~ret | mask) ^ (~ret | ~mask) >> 0x18
    return ret & 0xffffffff
```

API Hash Lookup

We already know that the API hash `0x6aa0e84` from the `kernel32.dll` should resolve to a function that accepts two arguments like so `f(2,0)`. So let us plug all exports from the `kernel32.dll` into the hashing function and check the result:

```
import pefile

pe = pefile.PE(data=open('C:\\Windows\\SysWOW64\\kernel32.dll', 'rb').read())
export = pe.DIRECTORY_ENTRY_EXPORT
dll_name = pe.get_string_at_rva(export.struct.Name)
for pe_export in export.symbols:
    export_name = pe_export.name.decode('utf-8')
    if calc_hash(export_name) == 0x6aa0e84:
        print(export_name)
```

Which ... fails by giving no result. Since I was pretty sure about everything *but* the data actually passed into `pr_ApiHash`, I decided to do some reversing around that next: Ghidra determined the type of variable `local_90` to be `undefined2 local_90 [50]`. This is Ghidra's way of telling you that it thinks it is an array with 50 entries where each entry has a length of 2 bytes. Since we already established that the array is actual a string, I decided to retype it to `char[100]`:

```
...
FUN_0309ea50(local_90, uVar3);
cVar1 = *(char *)(iVar5 + param_1);
if (cVar1 != '\0') {
    i = 0;
    do {
```

```
    local_90[i] = FUN_0309a690(cVar1);;
    cVar1 = *(char *)(iVar5 + param_1 + 1 + i);
    i = i + 1;
    } while (cVar1 != '\0');
}
uVar4 = pr_ApiHash(local_90,-1);
...
```

So the values of that array come out of the function `FUN_0309a690`. Let's take a close look at it: the function receives a value and either returns it or adds `0x20` and returns the result. Because the condition looks complicated and was hit pretty hard by the obfuscator the author probably uses, I was just lucky to know what adding the number `0x20` in the context of strings may mean: converting upper-case characters to lower-case characters. So my leap of faith was to assume that `FUN_0309a690` actually is `pr_toLower`. And heureca! Running the above Python code with lower-cased `export_name` results in a single hit, namely `CreateToolhelp32Snapshot` which accepts two `DWORD` arguments according to the documentation. This is in-line with our observation from the table at the start of this post.

Summary

What can you take away from this post? Maybe it is that I'm just as lazy as a sloth and don't even reverse engineer. Maybe, that adding or subtracting `0x20` means converting between upper and lower case strings. Maybe, that offsets `0x18` and `0x60` indicate PE parsing. Or maybe, that it is sometimes possible to understand a lot of about a malware without going into every single line of code and understanding everything.
