Understanding BlackMatter's API Hashing

blog.digital-investigations.info/2021-08-05-understanding-blackmatters-api-hashing.html

04 Aug 2021

tl;dr:

The ransomware **BlackMatter** aims at stepping into the void, which was left by REvil's and DarkSide's (temporary) retreat. At this point in time this new ransomware seems to pose a serious threat. In this blogpost BlackMatter's API hashing mechanism is described in detail and a Ghidra-script is supplied [1] to aid future analyses.

The API hashing algorithm: To calculate the API hashes for function names, each and every character is added up, while a binary rotation to the right by 13 bits is performed in each iteration. The hash of the housing module, which was calculated in the same manner before, serves as a seed value for the respective hash.

Storing addresses: Notably, BlackMatter does not store the function addresses in clear after resolving them. Instead it uses an array of "trampoline-pointers", which point to small, dynamically allocated assembly blocks (12 bytes in size), which perform the XOR-decoding of the encoded version of the respective function address, that was placed in there during import resolution and call it afterwards.

Motivation

A new ransomware gang named *BlackMatter* appeared in July 2021 and started to recruit affiliates at the underground forums *Exploit* and *XSS*[2]. They fill the void, which is left by DarkSide's shutdown after the Colonial Pipeline attack[3] and REvil's disappearing in the mid of July after pwning Kaseya[4].

On the 2nd of August 2021 an interview of D. Smilyanets (*The Record*) with the alleged threat actor behind *BlackMatter* was published[5]. Within this interview, the actor states, that he is neither the successor of DarkSide nor REvil, instead he proclaims, that *BlackMatter* tries to unify the best of the ransomwares LockBit, REvil and DarkSide, which all have their individual strengths in the opinion of the alleged actor behind "the new ransomware on the block".

At this point in time it seems to be a valid assumption, that *BlackMatter* will keep the DFIR-community and the law enforcement agencies busy for the next few weeks, therefore initial analyses might be helpful to get to know the threat imposed by this probably rebranded actor.

Scope

For this analysis the *BlackMatter*-sample with SHA256

`7f6dd0ca03f04b64024e86a72a6d7cfab6abccc2173b85896fc4b431990a5984`

was used. It has a compilation timestamp of 23rd of July 2021 20:51:18 UTC and was published on the 2nd of August 2021 at MalwareBazaar[6].

This blog post deals with the API hashing found in this sample and shows a way to defeat it with the help of Ghidra scripting. Resolving the hidden imports, is the main prerequisite for a static analysis of *BlackMatter*-binaries. However, further steps, like the decoding of its eventually available config data, are not in scope of this blog post.

Findings

Directly after the entry point of the executable, the function at address `00405e5c`, which is responsible for initializing the import resolution is called, as the following figure of the decompiled code illustrates.

Decompile: performImportResolution - (7f6dd0ca03f04b64024e86a72a6d7cfab6abccc2173b85896fc4b431990a

```
 1
 2 void performImportResolution(void)
 3
 4 {
 5   HeapCreate *HeapCreate;
 6   HANDLE mem;
 7   HeapAlloc *HeapAlloc;
 8
 9                     /* Resolve HeapCreate */
10   HeapCreate = (HeapCreate *)resolveHashedImport(0x260b0745);
11   if (HeapCreate != (HeapCreate *)0x0) {
12     mem = (*HeapCreate)(0x40000,0,0);
13     if (mem != (HANDLE)0x0) {
14                     /* Resolve HeapAlloc; */
15       HeapAlloc = (HeapAlloc *)resolveHashedImport(0x6e6047db);
16       if (HeapAlloc != (HeapAlloc *)0x0) {
17                     /* Load functions from Kernel32.dll */
18         resolveApiHash((ImportEntry **)&DAT_004112ac,&DWORD_00405afc,mem,HeapAlloc);
19                     /* Load functions from ntdll.dll */
20         resolveApiHash((ImportEntry **)&DAT_00411368,&DWORD_00405bbc,mem,HeapAlloc);
21                     /* Load functions from advapi32.dll */
22         resolveApiHash((ImportEntry **)&DAT_00411428,&DWORD_00405c80,mem,HeapAlloc);
23                     /* Load functions from user32.dll */
24         resolveApiHash((ImportEntry **)&DAT_00411480,&DWORD_00405cdc,mem,HeapAlloc);
25                     /* Load functions from GDI32.dll */
26         resolveApiHash((ImportEntry **)&DAT_004114b4,&DWORD_00405d14,mem,HeapAlloc);
27                     /* Load functions from shell32.dll */
28         resolveApiHash((ImportEntry **)&DAT_004114ec,&DWORD_00405d50,mem,HeapAlloc);
29                     /* Load functions from ole32.dll */
30         resolveApiHash((ImportEntry **)&DAT_004114fc,&DWORD_00405d64,mem,HeapAlloc);
31                     /* Load functions from shlwapi.dll */
32         resolveApiHash((ImportEntry **)&DAT_00411518,&DWORD_00405d84,mem,HeapAlloc);
33                     /* Load functions from oleaut32.dll */
34         resolveApiHash((ImportEntry **)&DAT_00411540,&DWORD_00405db0,mem,HeapAlloc);
35                     /* Load functions from wtsapi32.dll */
36         resolveApiHash((ImportEntry **)&DAT_0041154c,&DWORD_00405dc0,mem,HeapAlloc);
37                     /* Load functions from RstrtMgr.dll */
38         resolveApiHash((ImportEntry **)&DAT_00411554,&DWORD_00405dcc,mem,HeapAlloc);
39                     /* Load functions from netapi32.dll */
40         resolveApiHash((ImportEntry **)&DAT_00411568,&DWORD_00405de4,mem,HeapAlloc);
41                     /* Load functions from activeds.dll */
42         resolveApiHash((ImportEntry **)&DAT_00411594,&DWORD_00405e14,mem,HeapAlloc);
43                     /* Load functions from wininet.dll */
44         resolveApiHash((ImportEntry **)&DAT_004115a8,&DWORD_00405e2c,mem,HeapAlloc);
45       }
46     }
47   }
48   return;
49 }
```

Figure 1: Decompilation of the setup function at `00405e5c`, which kicks off the import resolution

At l. 10 and l. 15 of this function the actual import resolution is started by calling another function at `0040581d`, named `resolveHashedImport` in the figure above. In this function all the heavy lifting required to resolve symbols is performed, e.g. the loaded modules are traversed in memory by utilizing the doubly-linked list named `InLoadOrderModuleList` of the `PEB_LDR_DATA`-struct and so on.

The goal of those initial calls is to retrieve `HeapCreate` and `HeapAlloc` (l. 10 and 15) at first. This is only possible since the called function at `0040581d` ensures that `LoadLibraryA` and `GetProcAddress` are loaded on the first invocation by recursive calls to itself, as it is shown in the following figure exemplary for `LoadLibraryA`.

```
0040581d 55          PUSH     EBP
0040581e 8b ec       MOV      EBP,ESP
00405820 83 c4 f0    ADD      ESP,-0x10
00405823 53          PUSH     EBX
00405824 56          PUSH     ESI
00405825 57          PUSH     EDI
00405826 83 3d       CMP      dword ptr [LOADLIBRARY],0x0
         14 12
         41 00 00
0040582d 75 1f       JNZ      LAB_0040584e
0040582f b8 5f       MOV      EAX,0x5d6015f
         01 d6 05
00405834 35 ed       XOR      EAX,0x22065fed
         5f 06 22
00405839 a3 14       MOV      [LOADLIBRARY],EAX
         12 41 00
0040583e ff 35       PUSH     dword ptr [LOADLIBRARY]
         14 12
         41 00
00405844 e8 d4       CALL     resolveHashedImport
         ff ff ff
00405849 a3 14       MOV      [LOADLIBRARY],EAX
         12 41 00
```

Figure 2: Recursive call from within `0040581d` to load `LoadLibraryA` on the first invocation

So how is the hash, which is passed to the function called at `00405844` calculated by *BlackMatter*?

Hash calculation

For the calculation of the API hash each character is added up one by another. In each iteration a seeded ROR-13-operation is performed, as the following figure illustrates.

```
                    ******************************************
                    *                FUNCTION                *
                    ******************************************
                    uint __fastcall calcFuncHash(undefined4…
     uint           EAX:4        <RETURN>
     undefined4     ECX:4        param_1
     undefined4     EDX:4        param_2
     byte *         Stack[0x4…funcName                    XREF[1]: 004010a0(R)
     uint           Stack[0x8…modHash                     XREF[1]: 0040109d(R)
     undefined1     HASH:5ff6…curChar
                    calcFuncHash                      XREF[1]: resolveHashedImport
00401096 55           PUSH     EBP
00401097 8b ec        MOV      EBP,ESP
00401099 52           PUSH     param_2
0040109a 56           PUSH     ESI
0040109b 33 c0        XOR      EAX,EAX
0040109d 8b 55 0c     MOV      param_2,dword ptr [EBP + modHash]
004010a0 8b 75 08     MOV      ESI,dword ptr [EBP + funcName]

                    LAB_004010a3                      XREF[1]: 004010b1(j)
004010a3 ac           LODSB    ESI
004010a4 80 c6 61     ADD      param_2,0x61
004010a7 80 ee 61     SUB      param_2,0x61
004010aa c1 ca 0d     ROR      param_2,0xd
004010ad 03 d0        ADD      param_2,EAX
004010af 85 c0        TEST     EAX,EAX
004010b1 75 f0        JNZ      LAB_004010a3
004010b3 8b c2        MOV      EAX,param_2
004010b5 5e           POP      ESI
004010b6 5a           POP      param_2
004010b7 5d           POP      EBP
004010b8 c2 08 00     RET      0x8
```

Figure 3: Algorithm to calculate the API hash

Because of the fact, that the hash of the module name is used as a seed, a two step process has to be employed to construct the final API hash for a single function.

First, the module name is hashed in a similar manner with a seed of 0. This happens in the function at `004010bb`, which is not shown here. It is looped over the characters, which are transformed to lower case. In each iteration a rotation by 13 bits of the dword value resulting from the previous iteration is performed and the current character value is added. This leads to the following Python implementation:

```
def calc_mod_hash(modname):
    mask = 0xFFFFFFFF
    h = 0
    for c in modname + "\x00":
        cc = ord(c)
        if (0x40 < cc and cc < 0x5b):
            cc = (cc | 0x20) & mask
        h = (h >> 0xd) | (h << 0x13)
        h = (h + cc) & mask

    return h
```

The resulting hash of the module name is then used as a seed for the similar but simpler function presented at fig. 3, which finally calculates the actual function hash. The following Python code shows the logic found in this function at `00401096` :

```
def calc_func_hash(modhash, funcname):
    mask = 0xFFFFFFFF
    h = modhash
    for c in funcname + "\x00":
        cc = ord(c)
        h = (h >> 0xd) | (h << 0x13)
        h = (h + cc) & mask

    return h
```

***Note**: It is important to add the nullbyte, so that for a function name of n characters, n+1 ROR-operations are performed.*[7]

In summary this leads to the following calculation of a function hash as it is used by *BlackMatter*:

```
def get_api_hash(modname, funcname):
    return calc_func_hash(calc_mod_hash(modname), funcname)
```

Let’s test it:

```
mn = "kernel32.dll"
fn = "GetProcAddress"
print(hex(get_api_hash(mn, fn)))

mn = "kernel32.dll"
fn = "LoadLibraryA"
print(hex(get_api_hash(mn, fn)))

#+Result
: 0xbb93705c
: 0x27d05eb2
```

Indeed, both hashes can be found in the binary, as fig. 3 shows:

```

  if (LOADLIBRARY == (int *)0x0) {
    LOADLIBRARY = (int *)0x27d05eb2;
    LOADLIBRARY = resolveHashedImport(0x27d05eb2);
  }
  if (GETPROCADDRESS == (int *)0x0) {
    GETPROCADDRESS = (int *)0xbb93705c;
    GETPROCADDRESS = resolveHashedImport(0xbb93705c);
  }
```

Figure 4: Function hashes of `LoadLibraryA` and `GetProcAdress`

Actually only `0x5d6015f ^ 0x22065fed`, wich results in `0x27d05eb2` can be found, since all API hashes are stored XORed with `0x22065fed` and are XORed again with this value before a comparison with the calculated hash.

(Re)storing imports

After the a/m and absolutely required functions like `HeapAlloc`, `LoadLibraryA`, etc. have been loaded. *BlackMatter* resolves blocks of hashed functions stored as dwords in global memory (2nd arg to function[8]) and stores pointers to dynamically allocated "structs" in global memory as well (1st arg to function[9]):

```
00405ea7 57          PUSH    EDI
00405ea8 56          PUSH    ESI
00405ea9 68 fc       PUSH    DWORD_00405afc
         5a 40 00
00405eae 68 ac       PUSH    DAT_004112ac
         12 41 00
                 Load functions from Kernel32.dll
00405eb3 e8 ce       CALL    resolveApiHash
         fb ff ff
```

Figure 5: Resolving array of hashes (here for Kernel32.dll)

Line 18 in fig. 1 already showed this code in a decompiled representation.

Fig. 6 shows the decompilation of the called function beginning at `00405a86`. Within there, it is looped over the array of function hashes until the value `0xCCCCCCCC` is reached. This serves as an indicator of the end of the list of function hashes, so the loop stops in l. 19, when this value is read.

```
Decompile: resolveApiHash - (7f6dd0ca03f04b64024e86a72a6d7cfab6abccc2173b85896fc4b431990a5984
1
2  int resolveApiHash(ImportEntry **result,uint *hash,HANDLE mem,HeapAlloc *HeapAlloc)
3
4  {
5    uint addr;
6    int *funcAddr;
7    ImportEntry *entry;
8    ImportEntry **resultBuf;
9    ImportEntry **placeHolder;
10
11                       /* Load module and then resolve the following function
12                          hashes until the value 0xCCCCCCCC is reached */
13   addr = loadModFromMemOrSystem32(*hash ^ 0x22065fed);
14   if (addr != 0) {
15     resultBuf = result + 1;
16     while( true ) {
17       hash = hash + 1;
18       addr = *hash;
19       if (addr == 0xcccccccc) break;
20       funcAddr = resolveHashedImport(addr ^ 0x22065fed);
21       entry = (ImportEntry *)(*HeapAlloc)(mem,0,0xc);
22                       /* Advance result ptr, if alloc was successful */
23       placeHolder = resultBuf;
24       if (*(int *)(entry + 1) != -0x54545455) {
25         placeHolder = resultBuf + 1;
26         *resultBuf = entry;
27       }
28                       /* Build up assembly to decode */
29       entry->movInstr = 0xb8;
30       entry->xoredFuncAddr = (uint)funcAddr ^ 0x22065fed;
31       entry->xorInstr = 0x35;
32       entry->xorKey = 0x22065fed;
33       entry->callInstr = 0xe0ff;
34       resultBuf = placeHolder;
35     }
36   }
37   return addr;
38 }
```

Figure 6: Storing XORed function address with Assembly instructions

Line 29 ff. looks very interesting here. To further complicate analysis, *BlackMatter* does not store the function address itself in the result array. Instead it stores a pointer to 12 bytes of dynamically allocated memory. In these 12 bytes it does not store the function address in clear. Instead the results of XOR-operations (here XORed with `0x22065fed`) are stored together with assembly instructions to decode the real function address on the fly, when the function is called as fig. 6 suggests.

So the global array of pointers which is passed as a buffer to hold the results of the import resolution (e.g. l. 18 ff. in fig. 1 and fig. 5) acts as trampoline, so that on each call, it is jumped to a 12 byte “function-struct”, which is comprised of the following opcode sequence on the heap, where the questionmarks resemble the XORed-function address in question:

```
B8 ?? ?? ?? ?? 35 ED 5F 06 22 E0 FF
```

Upon execution, these instructions load the XORed-function address into EAX and perform the XOR-operation again to reverse it and to finally call the decoded function address, so that the actual libary-call is performed without storing the function-addresses in memory.

Import resolution with Ghidra scripting

The labelling of the a/m "trampoline-pointers", whose call ultimately leads to the execution of the a/m opcode-sequence should be automated with Ghidra's scripting capabilities. To accomplish this, have a look at the following Java-code in my Gist:

https://gist.github.com/jgru/c58851bde4ee4778d83c84babb2f69d1#file-blackmatterapihashing-java

Note, that this Ghidra-script is based on L. Wallenborn's and J. Hüttenhain's template code[10]. (Thank you guys for your invaluable teaching!)

Upon execution the script asks for the name of the resolving function (the one called in fig. 5), which takes the two pointers to global memory regions (here at `00405a86`). In the next GUI-dialog, that pops up, the XOR-key has to be specified (here `0x22065fed`). Afterwards you have to choose the file, containing the precomputed hashes, which should be used for name resolution. This list can be found at my Gist as well:

https://gist.github.com/jgru/c58851bde4ee4778d83c84babb2f69d1#file-blackmatter_api-hash-json

If you stumble upon a *BlackMatter*-sample, that uses the same ROR-13-hashing, this script might help to get you started quickly with the analysis.

Conclusion

This blog post detailed the API-hashing mechanism employed by the new ransomware *BlackMatter*.

To hash a function name, *BlackMatter* employs a seeded ROR13 in an iterative manner. That is a rotation of the dword by 13 bits to the right. The name of the housing module, hashed in the same way, but with an initial value of 0, is used as a seed for this trivial hashing algorithm. It has to be noted, that due to the implementation with a do-while-loop, for a function name of length *n* (terminating zero-byte excluded) *n+1* ROR-operations will be performed. The API hashes are initially stored as dwords in global arrays XORed with `0x22065fed`.

Interestingly, the imported function addresses are stored in a dynamically allocated memory region. To further complicate analysis, *BlackMatter* does not store the function address itself, but the result of an XOR-operation (here again XORed with `0x22065fed`) together with assembly instructions to decode it on the fly, when the function is called by a pointer to this memory location housing these instructions.

During the import resolution-routine at `00405a86`, which is called multiple times with different arrays of API hashes, pointers to those opcode-sequences are stored in a global array, which is then referenced for executing the single functions, when needed.

If you have any notes, errata, hints, feedback, etc., please send a mail to `ca473c19fd9b81c045094121827b3548 at digital-investigations.info`.

Tags: TI REM