

Malware analysis: part 7. Yara rule example for CRC32. CRC32 in REvil ransomware

cocomelonc.github.io/malware/2023/02/02/malware-analysis-7.html

February 2, 2023

6 minute read

Hello, cybersecurity enthusiasts and white hackers!

The screenshot shows a terminal window on a Kali Linux system with a Windows PowerShell window running in the background. The terminal window displays the following code:

```
39 for (DWORD i = 0; i < img edt->AddressOfFunctions; i++) {  
40     LPSTR pFuncName = (LPSTR)((LPBYTE)h + fNames[i]);  
41  
42     if (crc32(pFuncName, strlen(pFuncName)) == myHash) {  
43         printf("successfully found! %s - %x\n", pFuncName, myHash);  
44         return (LPVOID)((LPBYTE)h + fAddr[f0rd[i]]);  
45     }  
46 }  
47 return nullptr;  
48 }  
49  
50 int main() {  
51     HMODULE mod = LoadLibrary("user32.dll");  
52     //LPVOID addr = getAPIAddr(mod, 0x572d5d8e);  
53     LPVOID addr = getAPIAddr(mod, 1462590862);  
54     printf("0x%p\n", addr);  
55     fnMessageBoxA myMessageBoxA = (fnMessageBoxA)  
56     myMessageBoxA(NULL, "Meow-meow!", "=^..^=", M  
57     return 0;  
58 }
```

The PowerShell window shows the command being run and the output:

```
PS C:\Users\IEUser> cd \\VBOXSVR\shared  
PS Microsoft.PowerShell.Core\FileSystem:\\VBOXSVR\shared> cd .\2023-01-27-malware-analysis-7\  
PS Microsoft.PowerShell.Core\FileSystem:\\VBOXSVR\shared\2023-01-27-malware-analysis-7>  
PS Microsoft.PowerShell.Core\FileSystem:\\VBOXSVR\shared\2023-01-27-malware-analysis-7> .\hack.exe  
successfully found! MessageBoxA - 572d5d8e  
0x00007ffbd9d2490  
PS Microsoft.PowerShell.Core\FileSystem:\\VBOXSVR\shared\2023-01-27-malware-analysis-7> .\hack.exe  
successfully found! MessageBoxA - 572d5d8e  
0x00007ffbd9d2490
```

A message box titled "01-27-malware-analysis" displays the text "Meow-meow!" with an "OK" button.

This post is the result of my own research on Yara rule for **CRC32** hashing. How to use it for malware analysis in practice.

At first I wanted to focus on the WinAPI hashing method by **CRC32** at malware development. But then this article would differ from [this one](#) only in the hashing algorithm. Then I decided to see how to create a Yara rule which indicate using this algorithm at malware samples. I also consider the implementation of this algorithm in the REvil ransomware.

CRC32

In short, this is one of the checksum calculation methods. CRC32 (Cyclic Redundancy Check 32) is a type of hashing algorithm used to generate a small, fixed-size checksum value from any data. It is used to detect errors in data stored in memory or transmitted over a

network or other communication channel. The checksum is calculated using a polynomial function and is often expressed as a **32-bit** hexadecimal number.

In fact, CRC is not a sum, but the result of dividing a certain amount of information (information message) by a constant, or rather, the remainder of dividing a message by a constant.

Algorithm of the simplest calculation method is:

1. initialize a remainder **r** to be **0xFFFFFFFF**
2. for each byte in the message, do the following:
 - a. divide the current remainder **r** by the polynomial $x^8 + x^7 + x^6 + x^4 + x^2 + 1$ (**0xEDB88320**)
 - b. store the remainder in an **8-bit** register.
 - c. **XOR** the **8-bit** register with the next byte of the message.
 - d. replace the current remainder with the **8-bit** register
3. after the last byte of the message has been processed, the final remainder is the CRC result.

practical example

And, where can this be applied in the malware development? This algorithm is often used for hashing function names.

I used my example from the previous article and just replaced the hashing algorithm to **CRC32**:

```

/*
 * hack.cpp - hashing Win32API functions via CRC32. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware-analysis-7.html
 */
#include <windows.h>
#include <stdio.h>

typedef UINT(CALLBACK* fnMessageBoxA)(
    HWND    hWnd,
    LPCSTR  lpText,
    LPCSTR  lpCaption,
    UINT    uType
);

unsigned int crc32(const char *data, size_t len) {
    unsigned int crc_table[256], crc;

    for (int i = 0; i < 256; i++) {
        crc = i;
        for (int j = 0; j < 8; j++) crc = (crc >> 1) ^ (crc & 1 ? 0xEDB88320 : 0);
        crc_table[i] = crc;
    }

    crc = 0xFFFFFFFF;
    while (len--) crc = (crc >> 8) ^ crc_table[(crc ^ *data++) & 0xFF];
    return crc ^ 0xFFFFFFFF;
}

static LPVOID getAPIAddr(HMODULE h, unsigned int myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header = (PIMAGE_NT_HEADERS)((LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)(
        (LPBYTE)h + img_nt_header-
    >OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PDWORD fAddr = (PDWORD)((LPBYTE)h + img_edt->AddressOfFunctions);
    PDWORD fName = (PDWORD)((LPBYTE)h + img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)((LPBYTE)h + img_edt->AddressOfNameOrdinals);

    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

        if (crc32(pFuncName, strlen(pFuncName)) == myHash) {
            printf("successfully found! %s - %x\n", pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]));
        }
    }
    return nullptr;
}

int main() {

```

```

HMODULE mod = LoadLibrary("user32.dll");
//LPVOID addr = getAPIAddr(mod, 0x572d5d8e);
LPVOID addr = getAPIAddr(mod, 1462590862);
printf("0x%p\n", addr);
fnMessageBoxA myMessageBoxA = (fnMessageBoxA)addr;
myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
return 0;
}

```

As you can see, I just used this constant `0xEDB88320` and also hardcoded `MessageBoxA` string:

```

import zlib

# crc32
def crc32(data):
    hash = zlib.crc32(data)
    print ("0x%08x" % hash)
    print (hash)
    return hash

crc32(b"MessageBoxA")

```

```

└─(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
$ python3 crc32.py
0x572d5d8e
1462590862

```

yara rule

So in the simplest implementation, the Yara rule will look like this:

```

rule crc32_hash
{
    meta:
        author = "cocomelonc"
        description = "crc32 constants"
    strings:
        $c = { 2083B8ED }
    condition:
        $c
}

```

As you can see, we just add algorithm's constant for identity:

```
hexdump -C ./hack.exe | grep "20 83 b8 ed"
```

```

└─(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
$ hexdump -C ./hack.exe | grep "20 83 b8 ed"
00006fb0  20 83 b8 ed a8 01 89 d0  0f 45 c1 41 83 e8 01 75  | .....E.A...u|

```

Let's check it:

```
yara -w ./crc32.yar ./hack.exe
```

```
[cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
└$ yara -w ./crc32.yar ./hack.exe
crc32_hash ./hack.exe
```

Despite the fact that this rule may provide a large number of false-positive matches, it is useful to be aware that a sample may have implemented CRC32, since this can speed up malware sample analysis.

demo

First of all, compile our “malware”:

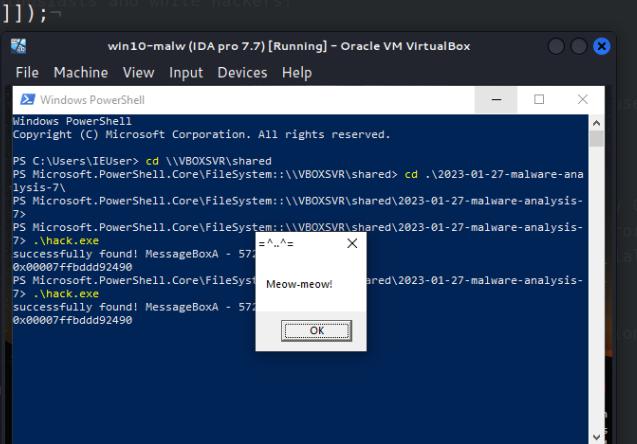
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

```
[cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
└$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
└$ ls -lt
total 48
drwxr-xr-x 1 cocomelonc cocomelonc 39936 Feb 2 19:22 .
drwxr-xr-x 1 cocomelonc cocomelonc 1882 Feb 2 19:22 .
-rw-r--r-- 1 cocomelonc cocomelonc 150 Jan 23 22:55 crc32.py
[cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
└$
```

Run it at victim's machine ([Windows 10 x64](#)):

```
.\hack.exe
```

```
42     if (crc32(pFuncName, strlen(pFuncName)) == myHash) {-
43         printf("successfully found! %s - %x\n", pFuncName, myHash);
44         return (LPVOID)((LPBYTE)h + fAddr[f0rd[i]]);-
45     }-
46     return nullptr;-
47 }-
48 }-
49 }-
50 int main() {-
51     HMODULE mod = LoadLibrary("user32.dll");-
52     LPVOID addr = getAPIAddr(mod, 0x572d5d8e);-
53     LPVOID addr = getAPIAddr(mod, 1462590862);-
54     printf("%x%p\n", addr);-
55     fnMessageBoxA myMessageBoxA = (fnMessageBoxA)-
56     myMessageBoxA(NULL, "Meow-meow!", "=^..^=", M-
57     return 0;-
58 }-
```



```
win10-malw (IDA pro 7.7) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\Users\IEUser> cd \VBOXSVR\shared
PS Microsoft.PowerShell.Core\FileSystem::\VBOXSVR\shared> cd .\2023-01-27-malware-analysis-7
PS Microsoft.PowerShell.Core\FileSystem::\VBOXSVR\shared\2023-01-27-malware-analysis-7> .\hack.exe
successfully Found! MessageBoxA - 572d5d8e
0x00007ffffdd2490
PS Microsoft.PowerShell.Core\FileSystem::\VBOXSVR\shared\2023-01-27-malware-analysis-7> .\hack.exe
successfully Found! MessageBoxA - 572d5d8e
0x00007ffffdd2490
OK
PS Microsoft.PowerShell.Core\FileSystem::\VBOXSVR\shared\2023-01-27-malware-analysis-7>
```

```
cocomelonc@kali: ~/hacking/cybersec_blog/2023-01-27-malware-analysis-7
└$ cd hacking/cybersec_blog/2023-01-27-malware-analysis-7
└$ python3 crc32.py
0x572d5d8e
1462590862
```

The screenshot shows a terminal session on a Kali Linux host (cocomelonc㉿kali) running a Python script named `crc32.py`. The script takes two arguments: `0x572d5d8e` and `1462590862`. The output of the script is a hex string: `f2d076786b061b771f945243dbf755539b8170963cf89aadccfb6e62acd4083`. This hex string is then used to construct a file path in a Windows PowerShell window titled "Windows PowerShell". The PowerShell session runs on a Windows 10 VM (win10-malw). It navigates to the shared folder and executes the file `hack.exe`. A message box appears with the text "Meow-meow!". The terminal session ends with the command `python3 crc32.py 0x572d5d8e 1462590862`.

```

.... return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);  

.... }  

.... }  

.... return nullptr;  

}  

} // This post is the result  

int main() {  

    HMODULE mod = LoadLibrary("user32.dll");  

    //LPVOID addr = getAPIAddr(mod, 0x572d5d8e);  

    LPVOID addr = getAPIAddr(mod, 1462590862);  

    printf("0x%p\n", addr);  

    fnMessageBoxA myMessageBoxA = (fnMessageBoxA)  

        myMessageBoxA(NULL, "Meow-meow!", "=^..^=", M  

    return 0;  

}  

} // the remainder of divide by 0  

cocomeonc@kali:~/hacking/cybersec_blog/2023-01-27-mal  

File Actions Edit View Help  

[~(cocomeonc㉿kali)-[~]  

$ cd hacking/cybersec_blog/2023-01-27-malwa  

[~(cocomeonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malwa  

sis-7]  

└$ python3 crc32.py  

0x572d5d8e  

1462590862

```

As you can see, everything is worked perfectly! =^..^=

Let's go to upload our “malware” to VirusTotal:

The screenshot shows the VirusTotal analysis page for the file `f2d076786b061b771f945243dbf755539b8170963cf89aadccfb6e62acd4083`. The file is identified as `hack.exe` and is a 39.00 KB 64-bit assembly file. The analysis was performed a moment ago at 2023-02-02 19:18:10 UTC. The community score is 4/70. The detection table shows the following results:

Vendor	Detection	Confidence
Cybereason	Malicious.0e2ab0	Malicious (moderate Confidence)
Google	Detected	Trojan.Win64.Rozena
Acronis (Static ML)	Undetected	
Alibaba	Undetected	
Anti-AVL	Undetected	
Avast	Undetected	
Baidu (non-avsoft)	Undetected	
Elastic		
ikarus		
AhnLab-V3		
ALYac		
Arcabit		
AVG		
Raijin		

So, 4 of 70 AV engines detect our file as malicious.

<https://www.virustotal.com/gui/file/f2d076786b061b771f945243dbf755539b8170963cf89aadccfb6e62acd4083/details>

This trick is used for example by REvil and MailTo ransomwares in the wild.

practical example 2. REvil ransomware

REvil generates a unique identifier (**UID**) for the host using the following process. The **UID** is part of the payment URL referenced in the dropped ransom note:

- obtains the volume serial number for the system drive

- generates a **CRC32** hash of the volume serial number using the hard-coded seed value of **0x539**
- generates a **CRC32** hash of the value returned by the **CPUID** assembly instruction using the **CRC32** hash for the volume serial number as a seed value
- appends the volume serial number to the **CPUID CRC32** hash.

In the simplest implementation, it is look like (**hack2.cpp**):

```

/*
 * hack2.cpp - get UID via CRC32 as REvil ransomware. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware/2023/01/27/malware-analysis-7.html
*/
#include <stdio.h>
#include <windows.h>
#include <intrin.h>
#include <wincrypt.h>

DWORD crc32(DWORD crc, const BYTE *buf, DWORD len) {
    DWORD table[256];
    DWORD i, j, c;
    for (i = 0; i < 256; i++) {
        c = i;
        for (j = 0; j < 8; j++) {
            if (c & 1)
                c = 0xEDB88320 ^ (c >> 1);
            else
                c = c >> 1;
        }
        table[i] = c;
    }

    crc = ~crc;
    while (len--)
        crc = table[(crc ^ *buf++) & 0xFF] ^ (crc >> 8);

    return ~crc;
}

int main(void) {
    DWORD volumeSerial, cpuidHash, uid, i;
    char volumePath[MAX_PATH];
    BYTE cpuidData[16];
    DWORD cpuidDataSize = sizeof(cpuidData);
    DWORD hashBuffer[4];
    HCRYPTPROV hCryptProv;

    if (!GetVolumeInformation(NULL, NULL, 0, &volumeSerial, NULL, NULL, NULL, 0)) {
        printf("failed to get the volume serial number.\n");
        return 1;
    }

    volumeSerial = crc32(0x539, (BYTE *)&volumeSerial, sizeof(volumeSerial));

    __cpuid(hashBuffer, 0);
    for (i = 0; i < 4; i++)
        cpuidData[i] = (BYTE)(hashBuffer[i] & 0xff);
    __cpuid(hashBuffer, 1);
    for (i = 0; i < 4; i++)
        cpuidData[4 + i] = (BYTE)(hashBuffer[i] & 0xff);
}

```

```

cpuidHash = crc32(volumeSerial, cpuidData, cpuidDataSize);

uid = volumeSerial;
uid = (uid << 32) | cpuidHash;

printf("UID: %llx\n", uid);

return 0;
}

```

This implementation calls `GetVolumeInformation` to retrieve the volume serial number for the system drive, `crc32` to build the `CRC32` hash, and `__cpuid` to obtain the value returned by the `CPUID` assembly instruction.

The resulting `uid` is a `64-bit` value that combines the serial number of the volume and the `CPUID` hash.

demo 2

Let's go to see in action. Compile it:

```
x86_64-w64-mingw32-g++ -O2 hack2.cpp -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

```

[cocomelonc@kali:~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
$ x86_64-w64-mingw32-g++ -O2 hack2.cpp -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
hack2.cpp: In function 'int main()':
hack2.cpp:42:11: warning: invalid conversion from 'DWORD*' {aka 'long unsigned int*'} to 'int*' [-fpermissive]
  42 |     __cpuid(hashBuffer, 0);
      |     ^
      |     |
      |     DWORD* {aka long unsigned int} (warning is worked perfectly)
In file included from /usr/share/mingw-w64/include/intrin.h:41,
                 from hack2.cpp:3:
/usr/share/mingw-w64/include/pdk_inc/intrin-impl.h:1899:18: note:   initializing argument 1 of 'void __cpuid(int*, int)'
 1899 |     void __cpuid(int CPUInfo[4], int InfoType) {
      |             ^
      |             |
      |             DWORD* {aka long unsigned int}
hack2.cpp:45:11: warning: invalid conversion from 'DWORD*' {aka 'long unsigned int*'} to 'int*' [-fpermissive]
  45 |     __cpuid(hashBuffer, 1);
      |     ^
      |     |
      |     DWORD* {aka long unsigned int}
In file included from /usr/share/mingw-w64/include/intrin.h:41,
                 from hack2.cpp:3:
/usr/share/mingw-w64/include/pdk_inc/intrin-impl.h:1899:18: note:   initializing argument 1 of 'void __cpuid(int*, int)' he will.
 1899 |     void __cpuid(int CPUInfo[4], int InfoType) {
      |             ^
      |             |
      |             DWORD* {aka long unsigned int}
hack2.cpp:51:14: warning: left shift count >= width of type [-Wshift-count-overflow]
  51 |     uid = (uid << 32) | cpuidHash;
      |             ^
[cocomelonc@kali:~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
$ ls -lt
total 96
-rwxr-xr-x 1 cocomelonc cocomelonc 39936 Feb  3 06:11 hack2.exe (Modified Create Algorithem)(https://research.jcanthine.org/jcnet/number14/jcnet105.pdf)
-rw-r--r-- 1 cocomelonc cocomelonc 1264 Feb  3 06:11 hack2.cpp (https://research.jcanthine.org/jcnet/number14/jcnet105.pdf)
-rw-r--r-- 1 cocomelonc cocomelonc 143 Feb  2 21:16 crc32.yar (https://research.jcanthine.org/jcnet/number14/jcnet105.pdf)
-rwxr-xr-x 1 cocomelonc cocomelonc 39936 Feb  2 19:22 hack.exe (https://research.jcanthine.org/jcnet/number14/jcnet105.pdf)
-rw-r--r-- 1 cocomelonc cocomelonc 1882 Feb  2 19:22 hack.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 150 Jan 23 22:55 crc32.py (good bye!
```

Run it at victim's machine (`Windows 10 x64`):

```
.\hack2.exe
```

```

33     if (!GetVolumeInformation(NULL, NULL, 0, &volumeSerial, NULL, NULL, NULL, 0)) {
34         printf("failed to get the volume serial number.\n");
35         return 1;
36     }
37
38     volumeSerial = crc32(0x539, (BYTE*)
39     _cpuid(hashBuffer, 0);
40     for (i = 0; i < 4; i++)
41         cpuidData[i] = (BYTE)(hashBuffer
42     _cpuid(hashBuffer, 1);
43     for (i = 0; i < 4; i++)
44         cpuidData[4 + i] = (BYTE)(hashBu
45     cpuidHash = crc32(volumeSerial, cp
46
47     uid = volumeSerial;
48     uid = (uid << 32) | cpuidHash;
49
50     printf("UID: %llx\n", uid);
51
52     return 0;
53 }

```

NORMAL | hack2.cpp

As you can see, everything is worked perfectly!

Of course, this is just “dirty PoC” of part of the REvil ransomware’s logic.

Let’s go to upload this to VirusTotal:

Security vendor	Detection
Elastic	Malicious (moderate Confidence)
Ikarus	Trojan.Win64.Rozena
AhnLab-V3	Undetected
ALYac	Undetected
Arcabit	Undetected
AVG	Undetected
Baidu	Undetected
BitDefenderTheta	Undetected
ClamAV	Undetected
CMC	Undetected
Google	Detected
Acronis (Static ML)	Undetected
Alibaba	Undetected
Anti-AVL	Undetected
Avast	Undetected
Avira (no cloud)	Undetected
BitDefender	Undetected
Bkav Pro	Undetected

In this example, 3 of 70 AV engines detect our file as malicious.

<https://www.virustotal.com/gui/file/871257db59da7fb9346b120ba165924b60dafb45f3f940c9fb0a739504b29c5/details>

Let’s check it via YARA:

yara -w ./crc32.yar -r ./

```
(cocomelonc㉿kali) - [~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]o to upload this to VirusTotal:  
└$ yara -w ./crc32.yar -r ./  
crc32_hash ./hack2.exe  
crc32_hash ./hack.exe
```

I hope this post spreads awareness to the blue teamers of this interesting hashing technique, and adds a weapon to the red teamers arsenal.

| This is a practical case for educational purposes only.

[AV engines evasion techniques - part 5](#)

[CRC32](#)

[Novel Approach for Worm Detection using Modified Crc32 Algorithm](#)

[REvil/Sodinokibi](#)

[MailTo](#)

[GetVolumeInformation](#)

[source code in github](#)

Thanks for your time happy hacking and good bye!

PS. All drawings and screenshots are mine