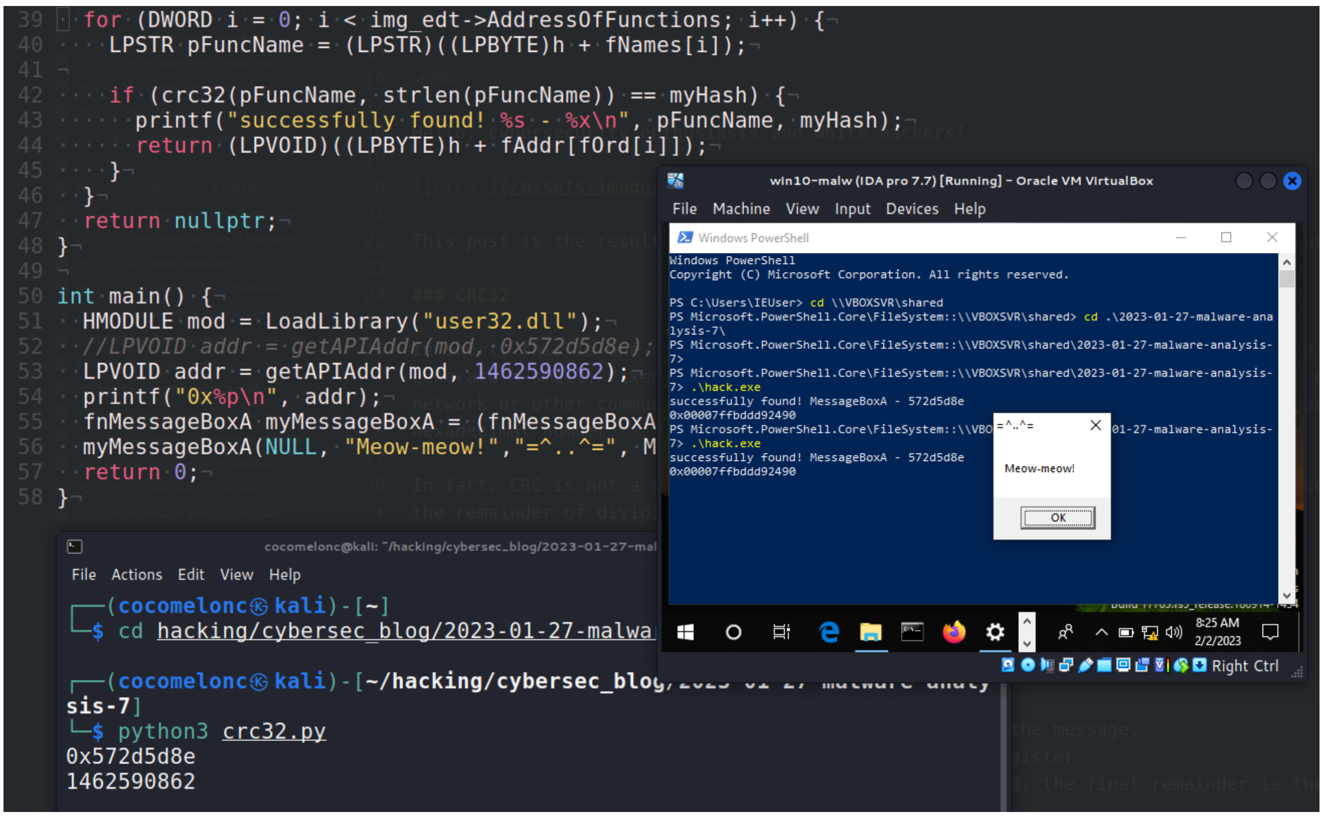
Malware analysis: part 7. Yara rule example for CRC32. CRC32 in REvil ransomware

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

February 2, 2023

6 minute read

Hello, cybersecurity enthusiasts and white hackers!



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.

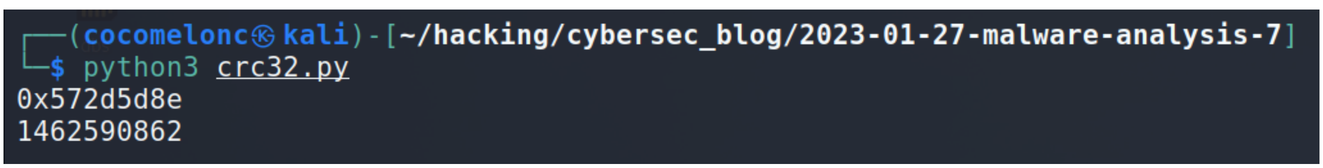
I used my example from the previous article and just replaced the hashing algorithm to `CRC32`:

```
/*
 * hack.cpp - hashing Win32API functions via CRC32. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware/2023/01/27/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 fNames = (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 + fNames[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")
```



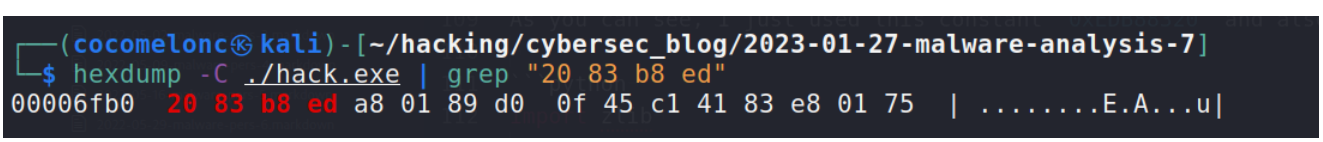
yara rule

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
}
```

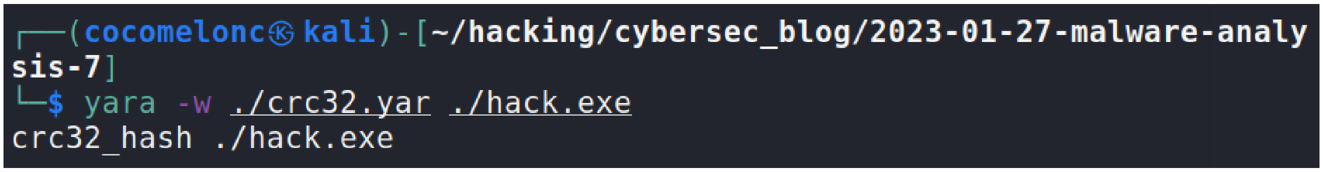
As you can see, we just add algorithm's constant for identity:

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



Let's check it:

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

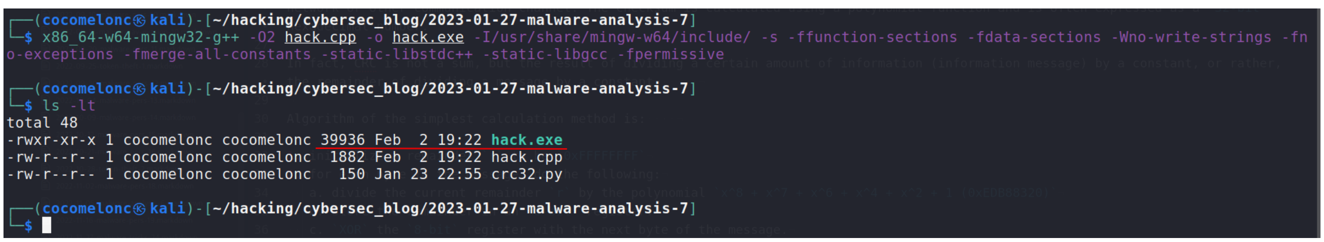


Despite the fact that this rule may provide a large number of false-positive matches, it is useful to be aware that a sample may have implemented `CRC32`, since this can speed up malware sample analysis.

demo

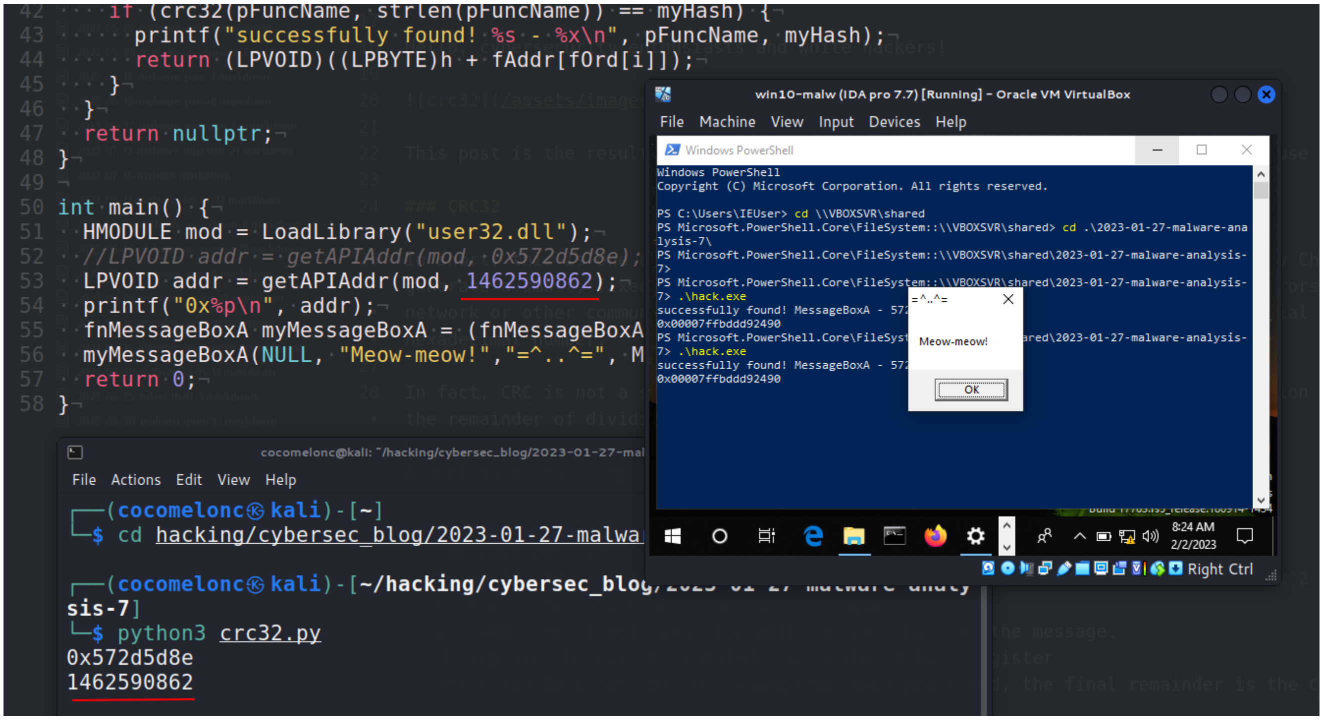
First of all, compile our "malware":

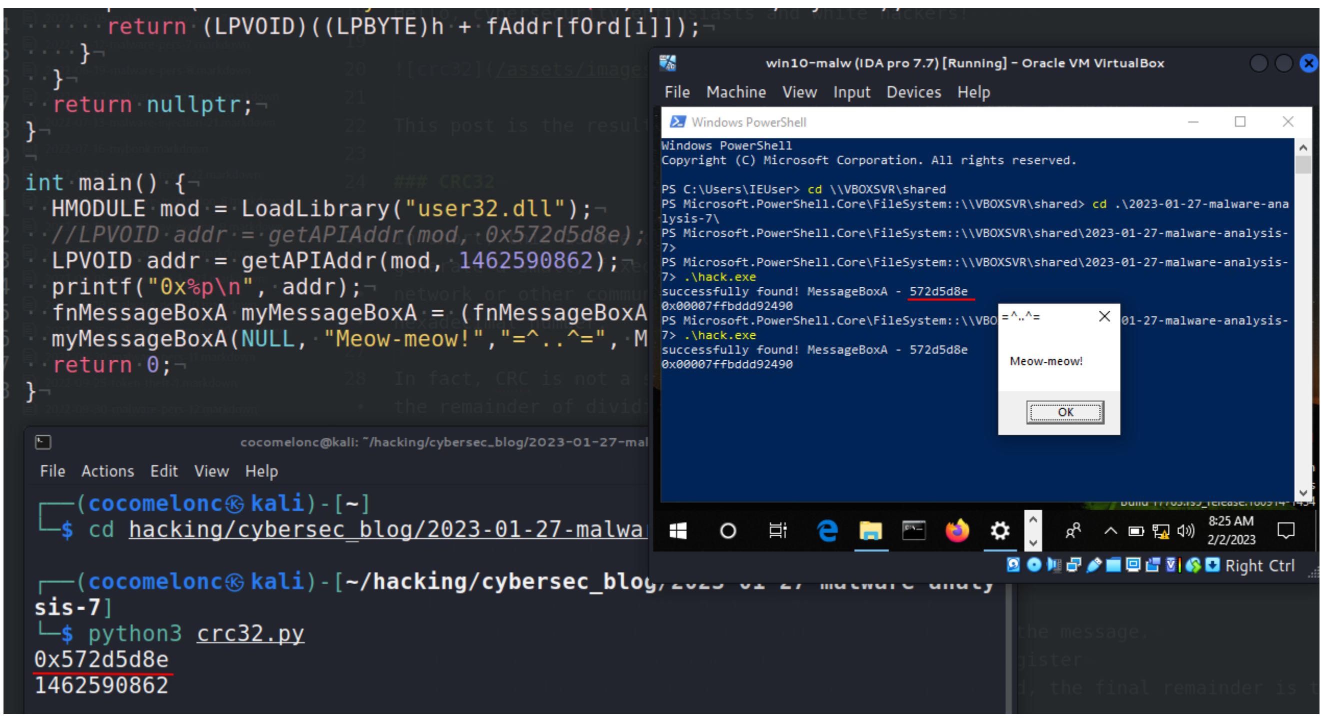
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```



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

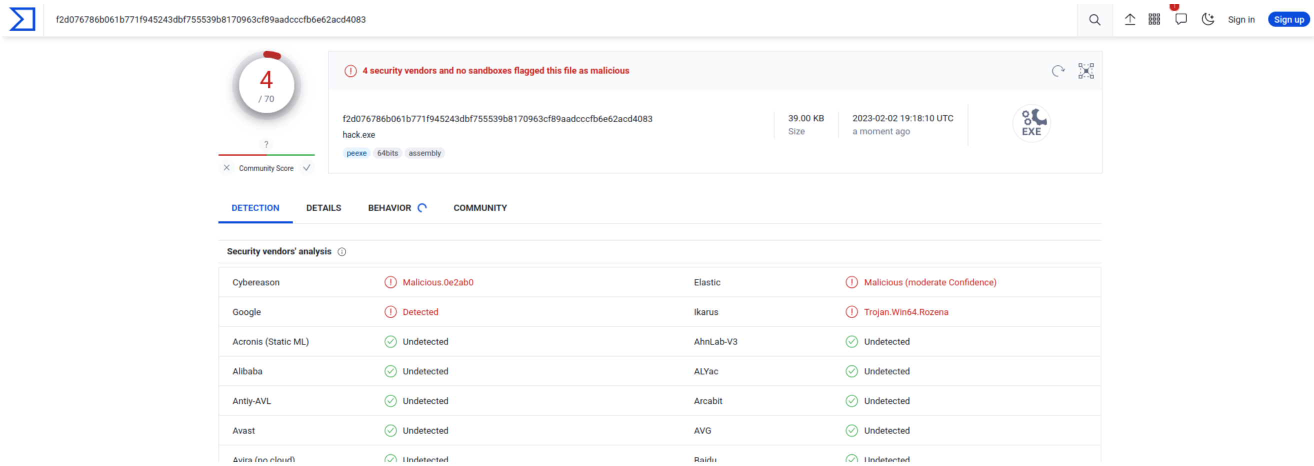
```
.\hack.exe
```





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

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



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

https://www.virustotal.com/gui/file/f2d076786b061b771f945243dbf755539b8170963cf89aadcccfb6e62acd4083/details

This is 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);
```

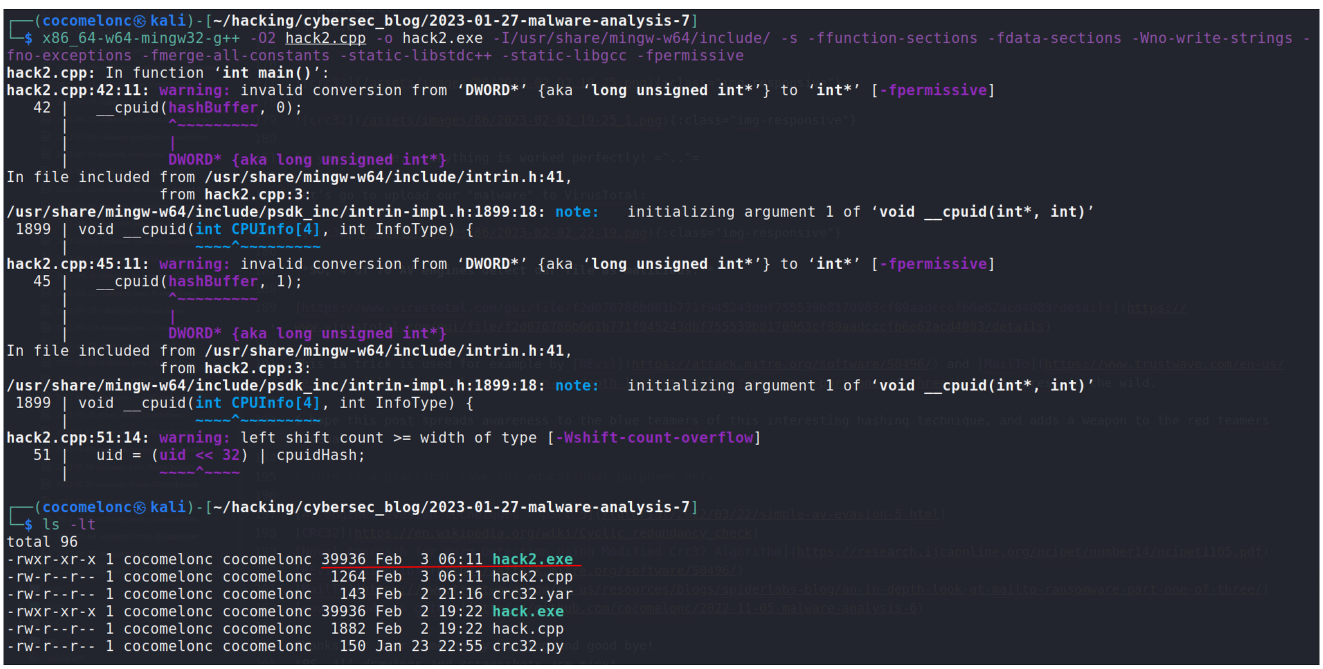
```
  cpuidHash = crc32(volumeSerial, cpuidData, cpuidDataSize);

  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

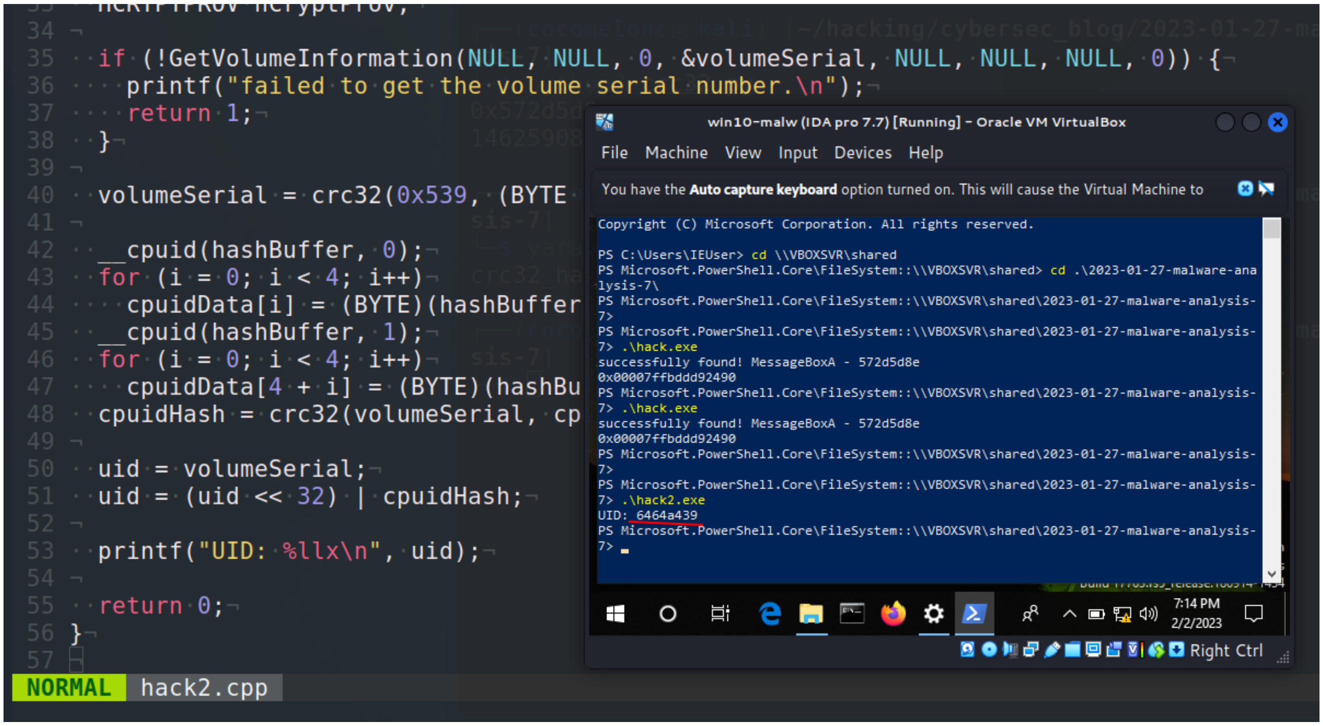
Let's go to see in action. Compile it:

```
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
```



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

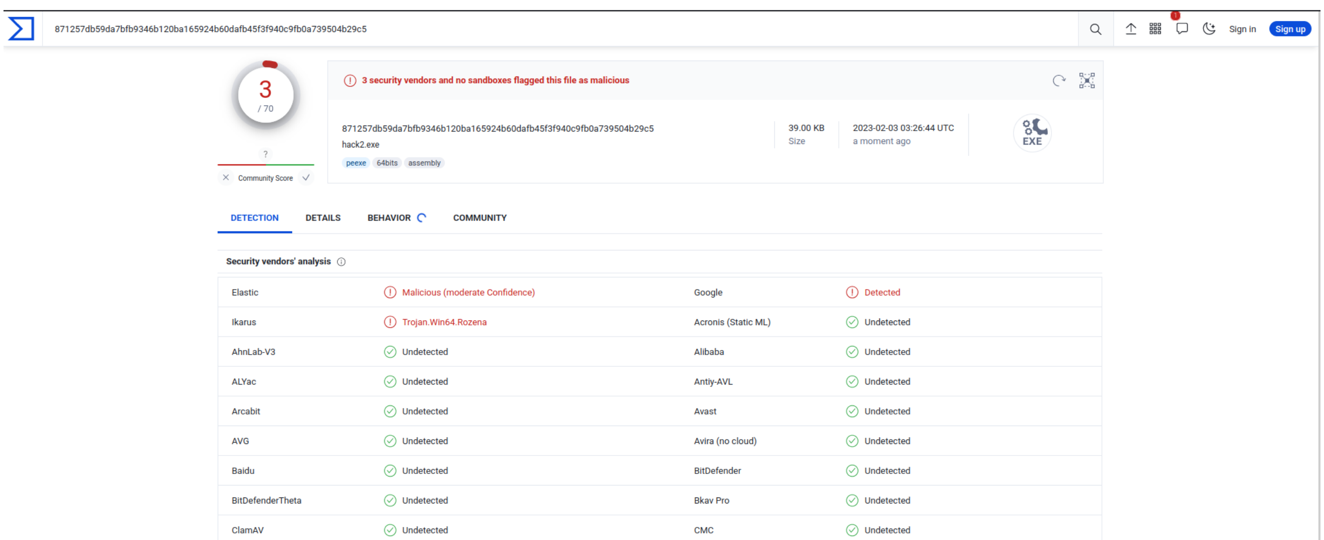
```
.\hack2.exe
```



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:



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

https://www.virustotal.com/gui/file/871257db59da7bfb9346b120ba165924b60dafb45f3f940c9fb0a739504b29c5/details

Let's check it via YARA:

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

```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-01-27-malware-analysis-7]
└─$ 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