

9

Exploring Hash Algorithms

Hash algorithms play a crucial role in malware, and they are often used for various tasks, from checking the integrity of downloaded components to evading detection by changing the hash of a file. In this chapter, we'll delve into common hash algorithms that are used in malware and provide examples of their implementation. The overarching theme of this chapter is to provide you with a holistic understanding of hash algorithms in the context of malware development. By combining theoretical insights with practical implementations, you'll gain not only conceptual knowledge but also the skills to apply these principles in real-world scenarios.

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

- Understanding the role of hash algorithms in malware
- A deep dive into common hash algorithms
- Practical use of hash algorithms in malware development

Technical requirements

For 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 purposes and Windows 10 (<https://www.microsoft.com/en-us/software-download/windows10ISO>) as the victim's machine.

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

I'll be using NeoVim (<https://github.com/neovim/neovim>) with syntax highlighting as a text editor. Neovim is a great choice if you want a lightweight, efficient text editor. However, you can use any other, such as VS Code (<https://code.visualstudio.com/>).

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

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

So, let's delve a little deeper into the role of hashing algorithms in malware development.

Understanding the role of hash algorithms in malware

Within the complex realm of malicious software, hash algorithms exert a greater impact than conventional integrity verification methods. The algorithms are utilized by malicious actors to implement intricate methods, including function call obfuscation and invoking WinAPI functions via hashes. These algorithms furnish the actors with potent instruments to elude detection and strengthen their malevolent undertakings.

In this chapter, we will look at some simple hashing examples and show their application in malware development.

In the enormous field of computer science, **hashing** stands as a fundamental concept with broad applications and profound implications. At its core, hashing is a process that transforms input data of arbitrary size into a fixed-size string of characters, often referred to as a hash value or hash code. This transformative operation is accomplished using a hash function, a mathematical algorithm specifically designed for this purpose.

Cryptographic hash functions

Cryptographic hash functions add an extra layer of security by possessing properties such as collision resistance, meaning it's computationally infeasible to find two different inputs that produce the same hash. Cryptographic hashing is fundamental in digital signatures and certificates, as well as ensuring data integrity in secure communications.

In addition to cryptographic hash functions, hashing algorithms serve various purposes across different domains of computer science. Here are some additional functions of hashing:

- **Data retrieval optimization:** Hashing is commonly used in data structures such as hash tables to optimize data retrieval operations. Non-cryptographic hash functions are employed to quickly map keys to their corresponding values in a data structure, enhancing efficiency in tasks such as database querying and information retrieval.
- **Password hashing (non-cryptographic):** In addition to cryptographic hashing for password storage, non-cryptographic hash functions are sometimes employed for password hashing in less security-sensitive applications. While not as robust as cryptographic hash functions, non-cryptographic hashing can still provide a basic level of protection for stored passwords.

In this chapter, we will consider various cryptographic and non-cryptographic hash functions and show their application in practice.

Applying hashing in malware analysis

Hashing also finds extensive application in the realm of malware analysis. Malware analysts leverage hashing techniques to enhance various aspects of their investigative processes, offering both efficiency and reliabil-

ity. Here are the key applications of hashing in the context of malware analysis:

- Verifying the integrity of files during malware analysis
- Signature-based detection
- Threat intelligence
- De-duplication of malware samples

Let's delve into some practical implementations of hashing algorithm techniques and the practical application of hashing within the realm of malware development.

A deep dive into common hash algorithms

In this section, we'll take a closer look at some common hash algorithms that are frequently employed in various applications, including security, data integrity verification, and password hashing. Here, we'll explore the characteristics and typical usage scenarios of MD5, SHA-1, SHA-256, and Bcrypt.

MD5

Message Digest Method 5 (MD5) is a cryptographic hash algorithm that transforms a string of any length into a 128-bit digest. These digests are represented as hexadecimal integers with 32 digits. Developed by Ronald Rivest in 1991, this algorithm can verify digital signatures.

Practical example

A complete re-implementation of hash functions is not the goal of this chapter. Instead, we will consider a simple example of an MD5 hash. The full source code for the PoC in Python looks like this:

```
import hashlib
def calc_md5(data):
    md5_hash = hashlib.md5()
    md5_hash.update(data)
    return md5_hash.hexdigest()
def main():
    input_data = b'meow-meow'
    md5_hash = calc_md5(input_data)
    print(f"MD5 Hash: {md5_hash}")
if __name__ == "__main__":
    main()
```

You can find the full source code in C here:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter09/02-dive-into-hashing/md5.c>

Upon compiling this, our PoC source code in C looks as follows:

```
$ x86_64-w64-mingw32-g++ -O2 md5.c -o md5.exe -I/usr/share/mingw-w64/include/ -s -ffunction-section
```

On my Kali Linux machine, it looks like this:

```
(cocomelon@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/02-dive-into-hashing]
$ x86_64-w64-mingw32-g++ -O2 md5.c -o md5.exe -I/usr/share/mingw-w64/include/ -s -f
function-sections -fddata-sections -Wno-write-strings -fno-exceptions -fmerge-all-cons
tants -static-libstdc++ -static-libgcc -fpermissive -lcrypt32
(cocomelon@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/02-dive-into-hashing]
$ ls -lt
total 48
-rwxr-xr-x 1 cocomelon cocomelon 40960 Dec  6 23:06 md5.exe
-rw-r--r-- 1 cocomelon cocomelon 1636 Dec  6 22:37 md5.c
-rw-r--r-- 1 cocomelon cocomelon 277 Dec  6 22:35 calc-md5.py
```

Figure 9.1 – Compiling our PoC

Then, execute it on any Windows machine by running the following command:

```
> .\md5.exe
```

On my Windows 10 x64 v1903 virtual machine, it looks like this:

```
win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\packtpub\chapter09\02-dive-into-hashing> .\md5.exe
MD5 Hash: 5686690ee0f029677a1b6cb0fd60612b
PS Z:\packtpub\chapter09\02-dive-into-hashing>
```

Figure 9.2 – Running our example on a Windows machine

As we can see, the example worked as expected.

SHA-1

Secure Hash Algorithm 1 (SHA-1) is a cryptographic algorithm that generates a hash value of 160 bits (20 bytes) from an input. The term for this hash value is **message digest**. This message digest is typically represented as a 40-digit hexadecimal number. It is a **Federal Information Processing Standard (FIPS)** of the USA that was developed by the National Security Agency. The security of SHA-1 has been compromised since 2005. By 2017, major technology companies' web browsers, including those of Microsoft, Google, Apple, and Mozilla, had ceased accepting SHA-1 SSL certificates. Let's look at further improvements:

- The SHA-2 hash functions, developed by the NSA, represent a significant improvement over SHA-1. The SHA-2 family includes hash functions that generate digests of 224, 256, 384, or 512 bits and are known as SHA224, SHA256, SHA384, and SHA512, respectively.
- SHA-512 operates on 64-bit words, while SHA-256 operates on 32-bit words. SHA-384 is similar to SHA-512 but truncated to 384 bytes, and SHA-224 is akin to SHA-256 but truncated to 224 bytes.
- SHA-512/224 and SHA-512/256 are shortened versions of SHA-512, with their initial values determined according to the guidelines outlined in FIPS PUB 180-4.

Practical example

In Python 3, we can implement this algorithm like this:

```
import hashlib
def sha1_hash(data):
    sha1 = hashlib.sha1()
    sha1.update(data.encode('utf-8'))
    return sha1.hexdigest()
# Example Usage
data_to_hash = "Hello, World!"
hashed_data = sha1_hash(data_to_hash)
print(f"SHA-1 Hash: {hashed_data}")
```

To learn how to implement SHA-256 in C using WINAPI, go to

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter09/02-dive-into-hashing/sha256.c>.

Bcrypt

Bcrypt, developed by Niels Provos and David Mazières in 1999, is a password hashing algorithm built upon the **Blowfish cipher**. It was introduced at USENIX to enhance security. Noteworthy features include the inclusion of a salt to safeguard against rainbow table attacks. Bcrypt is considered adaptive, allowing iteration counts (rounds) to be adjusted over time. This adaptability ensures that, despite advancements in computational power, the algorithm remains robust against brute-force search attacks by slowing down the hashing process.

Practical example

In Python 3, we can implement this algorithm like this:

```
import bcrypt
def hash_password(password):
    salt = bcrypt.gensalt()
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed_password
# Example Usage
password_to_hash = "mysupersecretpassword"
hashed_password = hash_password(password_to_hash)
print(f"Hashed Password: {hashed_password.decode('utf-8')}")
```

You can find the version in C in this book's GitHub repository:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter09/02-dive-into-hashing/bcrypt.c>.

At this point, we have an idea of how to implement these algorithms in practice. The practical implementation of other algorithms is not very different from these examples.

Of course, simple cryptographic algorithms are not limited to only the examples that we have considered. At the time of writing, there are a bunch of libraries and modules that implement most cryptographic hash functions. I just wanted to show you that it is not very difficult and you can

implement something yourself. Let's move on to their use in malware development.

Practical use of hash algorithms in malware

As mentioned previously, in the realm of malware and cyber threats, hash algorithms serve as indispensable tools, wielding both protective and subversive capabilities. Malware developers strategically exploit hash functions to obscure malicious code, enabling them to evade detection mechanisms and foster the surreptitious execution of harmful payloads. Conversely, security practitioners leverage hash algorithms as powerful tools for malware analysis so that they can identify, categorize, and mitigate malicious software. This section delves into the practical applications of hash algorithms in the context of malware from the real world.

Hashing WINAPI calls

I want to show you an interesting and effective technique for using hashing algorithms for malware development purposes. Implementing this easy yet effective method will mask WinAPI calls. It invokes functions via hash names. It is straightforward and frequently encountered in practice.

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

Practical example

Let's look at a simple message box example:

```
#include <windows.h>
#include <stdio.h>
int main() {
    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}
```

Compile it using **mingw** (you can use any Linux distribution):

```
$ i686-w64-mingw32-g++ meow.c -o meow.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-
```

On my Kali Linux machine, the result of this command is as follows:

```

(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
└─$ i686-w64-mingw32-g++ meow.c -o meow.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
In file included from /usr/share/mingw-w64/include/windows.h:70,
                 from meow.c:7:
/usr/share/mingw-w64/include/winbase.h:1098: warning: "InterlockedCompareExchangePointer" redefined
 1098 | #define InterlockedCompareExchangePointer __InlineInterlockedCompareExchangePointer
      |
In file included from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/unistd.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69:
/usr/share/mingw-w64/include/winnt.h:2409: note: this is the location of the previous definition
 2409 | #define InterlockedCompareExchangePointer(Destination, Exchange, Comperand) (PVOID) (LONG_PTR)InterlockedCompareExchange ((LONG volatile *) (Destination), (LONG) (LONG_PTR) (Exchange), (LONG) (LONG_PTR) (Comperand))
      |
(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
└─$ ls -lt
total 116
-rwxr-xr-x 1 cocomelonc cocomelonc 14336 Apr 16 15:43 meow.exe

```

Figure 9.3 – Compiling the meow.c code

We can ignore warnings here.

Next, run **meow.exe** on the victim's Windows machine:

```

PS C:\Users\user> cd Z:\packtpub\chapter09\03-practical-use-hashing\
PS Z:\packtpub\chapter09\03-practical-use-hashing> .\meow.exe

```

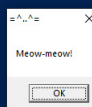


Figure 9.4 – Running meow.exe

As projected, it's simply a pop-up window.

Now, run **strings**. The **strings** command in Linux is used to extract readable strings from a binary file:

```
$ strings -n 8 meow.exe | grep MessageBox
```

On Kali Linux, the result of running this command is as follows:

```

(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
└─$ strings -n 8 meow.exe | grep MessageBox
MessageBoxA

```

Figure 9.5 – Running the strings command for meow.exe

It seems that the WinAPI functions are invoked explicitly during the basic static malware analysis and are visible in the application's import table:

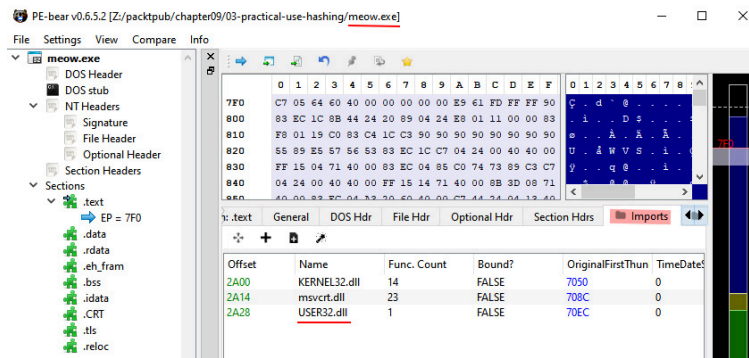


Figure 9.6 – USER32.dll visible in the import address table

Here, we'll mask the **MessageBoxA** WinAPI function so that it can't be detected by malware analysts. We are using our hash value by running a simple Python script:

```
# simple hashing example
def myHash(data):
    hash = 0x35
    for i in range(0, len(data)):
        hash += ord(data[i]) + (hash << 1)
    print (hash)
    return hash
myHash("MessageBoxA")
```

Run it using the following command:

```
$ python3 myhash.py
```

On my Kali Linux machine, it successfully printed the hash:

```
(cocomelon@kali)-[~/.../packtpub/Malware-Development/09-03-practical-use-hashing]
$ python3 myhash.py
17036696
```

Figure 9.7 – Running the myhash.py script

As we can see, this Python code defines a custom hashing function, which is a non-cryptographic hashing algorithm.

The concept behind this involves determining the address of a WinAPI function by its hashing name by enumerating exported WinAPI functions.

Let's write malware that uses this technique so that you understand this.

First, let's declare a hash function that's logically identical to the Python code:

```
DWORD calcMyHash(char* data) {
    DWORD hash = 0x35;
    for (int i = 0; i < strlen(data); i++) {
        hash += data[i] + (hash << 1);
    }
}
```



```
    return hash;
}
```

Then, declare a function that compares the hash of a given Windows API function to determine its address:

```
static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header = (PIMAGE_NT_HEADERS)((LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)(
        (LPBYTE)h + img_nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Virtual
    );
    PDWORD fAddr = (PDWORD)((LPBYTE)h + img_edt->AddressOfFunctions);
    PDWORD fNames = (PDWORD)((LPBYTE)h + img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)((LPBYTE)h + img_edt->AddressOfNameOrdinals);
    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fNames[i]);
        if (calcMyHash(pFuncName) == myHash) {
            printf("successfully found! %s - %d\n", pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
        }
    }
    return nullptr;
}
```

The logic is quite straightforward. We begin by traversing the PE headers of the required exported functions. We will exit the loop as soon as we discover a match between the hashes of the functions in the export table and the hash that's passed to our function within the iteration:

```
for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
    LPSTR pFuncName = (LPSTR)((LPBYTE)h + fNames[i]);
    if (calcMyHash(pFuncName) == myHash) {
        printf("successfully found! %s - %d\n", pFuncName, myHash);
        return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
    }
}
```

Then, the prototype of our function is declared through something like this:

```
typedef UINT(CALLBACK* fnMessageBoxA)(
    HWND    hWnd,
    LPCSTR  lpText,
    LPCSTR  lpCaption,
    UINT    uType
);
```

Finally, take a look at the **main()** function:

```
int main() {
    HMODULE mod = LoadLibrary("user32.dll");
    LPVOID addr = getAPIAddr(mod, 17036696);
    printf("0x%p\n", addr);
    fnMessageBoxA myMessageBoxA = (fnMessageBoxA)addr;
    myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}
```

Please note that the hash value in our main function and the value from our Python script are the same:

```

(cocomelon@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
$ python3 myhash.py
17036696

hack.c
chapter09 > 03-practical-use-hashing > C hack.c
80  ...return (LPVOID)((LPBYTE)h + tAddr[tUrd[1]]);
81  ...}
82  ...}
83  return nullptr;
84  }
85
86  int main() {
87      HMODULE mod = LoadLibrary("user32.dll");
88      LPVOID addr = GetProcAddress(mod, "17036696");
89      printf("0x%p\n", addr);
90      fnMessageBoxA myMessageBoxA = (fnMessageBoxA)addr;
91      myMessageBoxA(NULL, "Meow-meow!", "Meow-meow!", MB_OK);
92      return 0;
93  }

```

Figure 9.8 – The hash in the main function is the same as what’s in our Python script

The full source code for our malware can be found in this book’s GitHub repository: <https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter09/03-practical-use-hashing/hack.c>.

Demo

Let’s see this malware in action. First, compile it on an attacker’s machine:

```
$ i686-w64-mingw32-g++ hack.c -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-
```

On my Kali Linux machine, it’s compiled successfully and looks like this:

```

(cocomelon@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
$ i686-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
In file included from /usr/share/mingw-w64/include/windows.h:70,
                 from hack.c:10:
/usr/share/mingw-w64/include/winbase.h:1098: warning: "InterlockedCompareExchangePointer" redefined
1098 | #define InterlockedCompareExchangePointer __InlineInterlockedCompareExchangePointer
      |
In file included from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/unistd.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69:
/usr/share/mingw-w64/include/winnt.h:2409: note: this is the location of the previous definition
2409 | #define InterlockedCompareExchangePointer(Destination, Exchange, Comperand) (LONG_PTR)InterlockedCompareExchange((LONG volatile *) (Destination), (LONG) (LONG_PTR) (Exchange), (LONG) (LONG_PTR) (Comperand))
      |
(cocomelon@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing]
$ ls -lt
total 116
-rwxr-xr-x 1 cocomelon cocomelon 43008 Apr 16 15:52 hack.exe

```

Figure 9.9 – Compiling the hack.c code

As you can see, it also shows warnings; we can ignore these.

Run it on your Windows 10 x64 virtual machine by running the following command:

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

Note that we are printing the hash to check for correctness:

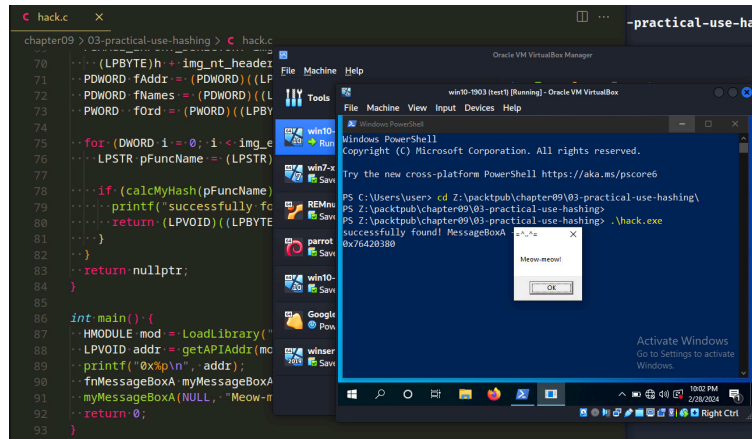


Figure 9.10 – Running hack.exe on a Windows machine

As we can see, our logic has been executed! Excellent!

Recheck our PE file with **strings** by running the following command:

```
$ strings -n 8 hack.exe | grep MessageBox
```

Here's the result:

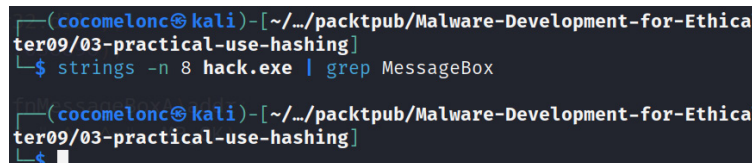


Figure 9.11 – Running strings for hack.exe

Here's what the **import address table** looks like:

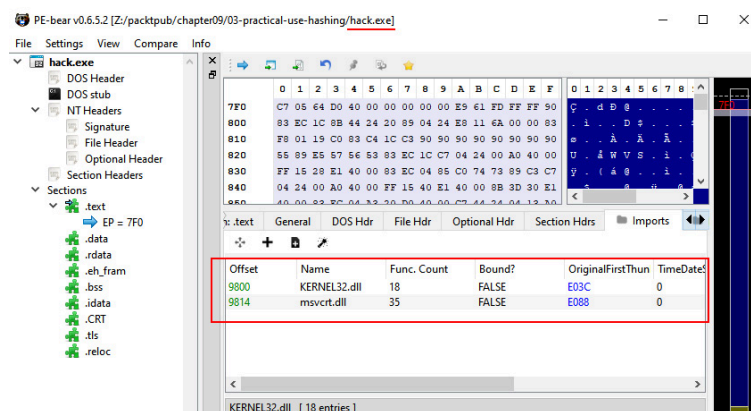


Figure 9.12 – Imports in hack.exe

As you can see, the **user32.dll** library isn't visible. If we dig deeper into the malware investigation, we will find our hashes, strings such as **user32.dll**, and so on. But this is just an example to understand the concept.

MurmurHash

In real malware, developers often use not entirely well-known and standard hashing algorithms. One such popular example is the **MurmurHash** algorithm, which was created and optimized by Austin Appleby.

Practical example

Here's some simple PoC code in C that demonstrates the logic of hashing via MurmurHash:

```
unsigned int MurmurHash2A(const void *input, size_t length, unsigned int seed) {
    const unsigned int m = 0x5bd1e995;
    const int r = 24;
    unsigned int h = seed ^ length;
    const unsigned char *data = (const unsigned char *)input;
    while (length >= 4) {
        unsigned int k = *(unsigned int *)data;
        k *= m;
        k ^= k >> r;
        k *= m;
        h *= m;
        h ^= k;
        data += 4;
        length -= 4;
    }
    switch (length) {
        case 3:
            h ^= data[2] << 16;
        case 2:
            h ^= data[1] << 8;
        case 1:
            h ^= data[0];
            h *= m;
    };
    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;
    return h;
}
```

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

```
$ x86_64-w64-mingw32-g++ murmurhash.c -o murmurhash.exe -s -ffunction-sections -fdata-sections -Wn
```

On my Kali Linux machine, it's compiled successfully:

```
(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing
$ x86_64-w64-mingw32-g++ murmurhash.c -o murmurhash.exe -s -ffunction-sections -fdata-sections -Wno-write-strings -fexceptions -fmerge-all-constants -static-libstdc++ -static-libgcc

(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter09/03-practical-use-hashing
$ ls -lt
total 116
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 16 15:59 murmurhash.exe
```

Figure 9.13 – Compiling the PoC code

Now, run it on the victim's machine (Windows 10 x64, in my case):

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

As we can see, the same trick worked perfectly:

```
PS C:\Users\user> cd Z:\packtpub\chapter09\03-practical-use-hashing\
PS Z:\packtpub\chapter09\03-practical-use-hashing>
PS Z:\packtpub\chapter09\03-practical-use-hashing> .\murmurhash.exe
successfully found! MessageBoxA - -179898011 0x00007ffaf7ac17f0
```

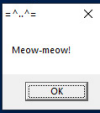


Figure 9.14 – Running murmurhash.exe

As we can see, our logic worked, and the meow message box successfully popped up as expected.

MurmurHash2 serves as a versatile cross-platform algorithm that can be implemented across diverse programming languages and environments. The following example shows how it can be implemented in Python:

```
def murmurhash2(key: bytes, seed: int) -> int:
    m = 0x5bd1e995
    r = 24
    h = seed ^ len(key)
    data = bytearray(key) + b'\x00' * (4 - (len(key) & 3))
    data = memoryview(data).cast("I")
    for i in range(len(data) // 4):
        k = data[i]
        k *= m
        k ^= k >> r
        k *= m
        h *= m
        h ^= k
    h ^= h >> 13
    h *= m
    h ^= h >> 15
    return h

h = murmurhash2(b"meow-meow", 0)
print ("%x" % h)
print ("%d" % h)
```

As we will see in future chapters, the **MurmurHash2A** hashing algorithm is used in real-life malware.

Summary

In this chapter, we explored the pivotal role that hash algorithms play in the realm of malware. This chapter encompassed three primary sections, each shedding light on distinct aspects of hash algorithm utilization in the context of malware.

Here, we covered prevalent hash algorithms. You learned how these algorithms function by exploring practical examples implemented in C/C++ and Python 3. The algorithms that were covered included MD5, SHA-1, SHA-256, and others. Each example equipped you with hands-on experience, fostering a comprehensive understanding of these widely used hash functions.

Finally, we took a hands-on approach to demonstrate the practical implementation of hash algorithms in concealing WinAPI calls. Through detailed examples, you learned how hash algorithms can be leveraged to obfuscate function calls, adding a layer of complexity to malware and enhancing its ability to evade detection.

We hope that the trick of hiding WinAPI calls will be useful not only to red team operators but also to specialists such as malware analysts from the blue team.

In the next chapter, we will learn about another application of classical cryptography in malware development. We'll start by looking at simple ciphers.