

Anti-Hacker Tool Kit: Reverse Engineering Binaries

del.icio.us

Discuss in Forums {mos_smf_discuss:Book Reviews}

Your computer seems to be running slower than normal. The router shows that your computer is transmitting data out to the Internet without you knowing it. Friends are complaining about you sending them e-mails you never composed. Determined to see if you have a Trojan running on your computer, you take a look at your process list to see if there is anything out of the ordinary. Much to your dismay, you notice a program running that you have never seen before and didn't explicitly start. You have been backdoored by malware.

There are many questions you should be asking in these situations. What does the program do? Does it use network resources? Can outside hackers now access my computer? Am I being used as a zombie for DDoS attacks? This chapter will focus on methods and tools you can use to determine what these programs do and how they do them, without having the source code. In the past, reverse engineering was something of a black art. Typically it involved some type of decompilation using a tool such as IDA or GDB to extract the assembly out of the binary, and the best you could hope for was to have that assembly converted into a low-level C code that you could use to understand what was going on. These tools have evolved, however, and you no longer need a PhD in Computer Science to be able to reverse engineer binaries. That being said, however, a brief primer will go miles in helping you understand when to use certain tools and when to use others.

This chapter (26) is excerpted from the book titled "Anti-Hacker Toolkit 3rd Edition" By Mike Shema, published by McGraw-Hill Osborne Media. ISBN: 0072262877; Published: Feb 9, 2006; Pages: 378; Edition: 3rd.

THE ANATOMY OF A COMPUTER PROGRAM

As anyone who has written a script or program knows, the user never actually creates the binary themselves. They create a program in some higher-level language (C, C#, Java, etc.) and then use a compiler or interpreter to actually convert the code into the lower-level instruction set.

Hello World in C

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

```
main()
```

```
{
```

```
printf("hello\n").
```

```
}
```

Hello World in Assembly (x86)

```
txt db "hello world!", 0Dh,0Ah, 24h
```

```
lea dx, txt
```

```
mov ah, 09h
```

```
int 21h
```

```
mov ah, 0
```

This instruction set can be one of two things: either the native ISA (Instruction Set Architecture) of the platform the program is intended to run on or an instruction set that is used by a virtual machine to execute (think C# and Java). The ISA instructions are then encoded into a binary number, and all the instructions are appended together to create the binary file. From a reverse engineering standpoint, it is important to discover how the binary was compiled and the platform it was intended to run on so you don't waste a lot of time chasing your tail.

Determining a Binary File Type

GNU has a utility named `file` which can be useful in determining the makeup of a binary executable. `file` looks at the header of the file, and from the signature of that header (the so-called "magic" number) it determines whether the file is a natively compiled executable, a byte-code compilation, or just static nonexecutable binary data. Let's take a look at what a typical `file` command looks like:

```
$ file backdoor
```

```
backdoor: ELF 32-bit LSB executable, Intel 80386, version 1
```

(SYSV), dynamically linked (uses shared libs), stripped

As you can see, when we run the file utility on the executable backdoor, we are given lots of useful information, albeit somewhat cryptic at first glance. Breaking the results down: ELF 32-bit LSB executable tells us this is a Linux ELF binary, compiled for the x86 platform. Dynamically linked is important to us because that means the executable relies on other files besides itself to run. If you are trying to track down all the files that have been placed on your computer, you can look to see if the executable uses any nonstandard libraries and where they are stored on the file system for an idea of where else to look for root kit remnants. The last detail, stripped, is the most important one if you are going to actually reverse engineer the binary. When a program is compiled, you have the option of including the source code for ease of debugging. If you have ever written a program and used a debugger such as GDB or the one in Visual Studio, you have worked with the code embedded within the executable. A hacker can remove the source code (also known as the symbols) either at compile time by using a flag to the compiler or after creation by using the GNU utility strip. If you are lucky enough to be working with a binary that has all symbols included, you can use the debugger to re-create the symbols. If not, your best course of action will be to try to determine as much as you can without looking at the binary programmatically.

BLACK BOX ANALYSIS

Reverse engineering using only the assembly or byte code is a very arduous, time-intensive process. If you find yourself in the situation where you can't access the source because it has been stripped or obfuscated, the best course of action is to start looking at the things you can easily see. What text strings does the file contain? Does the program try to access the network? What other files does the program rely on and is there any information you can glean from the support files? The amount of nonprogrammatic analysis you can perform varies from using a few command-line utilities to determine things about the file all the way to creating a true "sandbox" that tracks every movement of the binary as it executes on the system. If you are in a position where you will find yourself doing this more than once or twice every few months, it would be well worth your time to set up a lab sandbox to execute suspicious binaries in so you can quickly and easily diagnose them. We will discuss the creation of such a sandbox later in the chapter.

Viewing the Text Strings in a Binary

Seeing what text strings exist in a binary can be extraordinarily useful. These text strings can give you clues to what the binary does as well as information that the programmer thought was secret. For instance, you can determine if a program accesses the Internet because the addresses, if in canonical form (www.google.com, for example), will be stored in the executable as a text string. In addition, say, if the programmer has set a password to access a backdoor in the executable (this is common if say the program runs in the background and listens for someone connecting to the machine with correct credentials before activating), that password may be stored as a text string in the file. Let's take a look at a sample:

```
$ strings backdoor
```

```
&hellip;
```

```
l33t0wn3d
```

```
&hellip;
```

As you can see by the output, there is a string in the executable l33t0wn3d, which is more than likely some kind of password for the program. This can go a long way in helping us if we can figure out where to input it.

Using LSOF to Determine What Files and Ports a Binary Uses

LSOF is an open-source utility which can be extremely useful in determining what a program does. LSOF is short for List Open Files, and it shows what files each program running has open. This is useful not only for determining what supporting files a program uses, but since network sockets are treated like files, you can also see what network connections a program has open for both transmission and listening. Let's take a look at the

program output:

```
lsuf -p 600 COMMAND PID USER
FD
TYPE
DEVICE
SIZE/OFF
NODE NAME

&hellip; backdoor 600 root backdoor 600 root (IDLE)
8u

7u
inet 0x30002432228 0t0 inet 0x300031f1410 0t0
TCP *:2950 (LISTEN) TCP out:*->199.1.90.2:*

&hellip;
```

As you can see, there are two network sockets associated with backdoor. One seems to be an outgoing connection that is idle. The other is of more interest to us. It is a listener on port 2950 via the TCP protocol. This could be the way that outside hackers communicate with the backdoor. Now that we know the ports and communication tunnels that the binary uses, we can look at how it communicates with the outside world.

Determining Ports Using NMAP

Sometimes the easy way can be just as effective as more complicated measures. Nmap is a popular port scanner used to determine which TCP/IP ports are open on a machine. While the binary is running, execute a port scan on your external network interface to see if it is listening on a port. There is a caveat here, however. Subverting port scanning, while not trivial, is not a difficult task. We have seen some root kits that use obsolete protocols that are overlooked in a port scan (covert channels that are neither TCP nor UDP). The other method is to use some kind of knock-knock protocol, such as sending an ICMP packet of a certain size before the backdoor will reveal itself. If you find a port using nmap, great, but don't assume that just because you don't see it at first glance it isn't there. If you are graphically inclined, you can use the GUI version of nmap, nmapFE, to help with the process.

Using a Sniffer to Determine Network Traffic

Since we now have the ports used by the backdoor, we can easily set up a sniffer to monitor the traffic flow in and out of the program. This can be as simple as setting up tcpdump or Ethereal to monitor the inflow and outflow, looking for patterns and data. The other thing that you should think about doing is setting up an IDS system such as snort to see if

there is anything the program does that matches a signature for a known Trojan or backdoor. This can help identify both the genus of the malware, as well as determine if it is using some covert channel for communication that LSOFF may have missed (a few years back the HoneyNet project had a contest in which they had a binary that used an obsolete protocol to transmit information, and most entries completely whiffed on it because it was so nonstandard).

Looking at the System Calls

No man is an island. The same holds true for software. Whenever a program writes to the screen, accesses the network, accesses a file, or does any number of other things, these calls are made to the system libraries. These libraries are the core of the operating system, which allows for the layer of abstraction between the hardware and application software. If you have a binary that you think has been modified (i.e., we have seen versions of `ls` which have been modified to transmit out sensitive data when run), and you want to figure out with some certainty what it is doing, looking at the way it calls system routines can help. In the Unix world, there is a tool that started out as `truss` on Solaris and is now called `strace` on the other Unix flavors (Linux included). It allows you to do exactly that. Let's look at an example of some sample output so we can see how to interpret the results:

```
strace -p 123
```

```
Process 123 attached - interrupt to quit
```

```
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
```

```
read(5, "l33t0wn3d", 9) = 1
```

```
write(6, "welcome Dr. Faulken", 15) = 1
```

As you can see, we attached to the process and monitored it for system calls. We have two of specific interest: the program called the system command `read` with the input `"l33t0wn3d"`. As you discovered earlier using strings, this was built into the binary and is possibly an activation password. The suspicion is confirmed with the next line, where the program then pushes the string `"welcome Dr. Faulken"` back across the network, indicating the hacker has activated some kind of backdoor routine.

Identifying Kernel-hiding Techniques

Kernel-hiding methods for identifying and analyzing binaries have been around for tens of years. Trojan writers understand that employing these methods is the first thing a responder will do when looking for backdoors. As such, they often write code into their programs that will modify the kernel or somehow modify the system the code will run on to hide the backdoors. Depending on the sophistication of the hiding technique, it can make locating and identifying these Trojans a very difficult prospect. In the best case, they will simply modify the process listing program so that it doesn't display their Trojan as running. In the worst case, they actually modify the kernel so that it actively protects the secrecy of the program running. If you are using Linux and have a package manager such as RPM, you can actually check to see if any of the vital binaries have been modified. Here's a program in action:

```
# rpm -V ps
```

```
S.5...T /sbin/ps
```

So what does this mean? Let's look at the flags that RPM gives us in front of the filename:

S = size change

M = permissions change

5 = MD5 changed

L = Symlink changed

D = Device change

U = User change

G = Group change

T = Date/Time change

missing = file is gone

As you can see, since ps has been flagged as S, 5, and T: the size, the time/datestamp, and the md5 checksum all have changed since the default install was made. This is a very good indication that the binary has been modified, and if you didn't make the change, it's an even better indication that you have been thoroughly rooted and it may be time to scrap the machine and start over.

If the kernel has been modified, however, you have a much more difficult task ahead. The easiest way to proceed will be to create a quarantined sandbox machine that you can use to monitor the actions of the program without worry of a compromised machine.

Creating a Sandbox Machine

The obvious way to do this is to build a machine, load it up with monitoring tools such as LIDS, snort, and whatever else you can throw at it, fire up the program, and go to town. However, in my experience it is sometimes better in the long run to use a tool such as VMware to simulate a virtual machine and then monitor everything through the virtual machine. This approach has several benefits: First, it is very easy to start over with a clean image if the program obliterates everything on your test machine; second, since everything is a virtual software replication of hardware, you are afforded a level of access that would be hard and/or cumbersome to gain if you are just dealing with a hardware machine alone. Since VMware acts as a virtual bridge between the networking in the virtual machine and the true networking on the host computer, you can monitor, modify, and manipulate the data in ways that would be hard to impossible over a real network bridge/router configuration. Also, if you want to see how multiple computers running the binary interact with each other (such as the case with DDoS software), you can easily use VMware to create multiple machines and place them all on the same virtual subnet. With a real machine you would have to spend time and money getting everything up and running.

GETTING YOUR HANDS DIRTY: WORKING WITH THE CODE

Sometimes just looking at the external actions of a binary are not enough. We need to be able to break it down and find out what's going on internally so we can understand the true nature of the program. This is not an easy process. The majority of the work you will do will be done, in the best case scenario, in low-level C. The majority of the time you will be working in assembly or whatever low-level language the object code was created in. Make sure you brush up on these skills before you begin this process.

Getting at the Memory

In the Unix environment (and in Windows via the dumper utility in the Cygwin package), you have the ability to write out the entire memory space for an active program to a file. In the Unix vernacular, this is known as a core dump. Core dumps are extremely important in that they can store valuable information about what the program is doing and storing. To dump out the memory to a core file in Linux, you can use the kill command with a special signal:

```
$ kill -S SIGSEGV <processid>
```

NOTE: If the above command doesn't work, check the environment and make sure core dumps are allowed. In Linux this is done using the ulimit command.

A SIGSEGV signal tells the process that a segmentation violation has occurred and tricks it into dumping out the contents of memory when the process terminates. The resulting core file is a flat binary representation of what was stored in memory at the time of termination. You can perform the same types of analysis on these core files that you would apply to any binary file. Start off with a quick strings run to see if you can locate any relevant text that may have been obfuscated in the binary. You can also then use the core dump to help when you run the binary in a debugger such as GDB to cleanly sandbox and diagnose what the program is doing.

Working with objdump

Objdump is the GNU Binutils disassembler. It is a very powerful tool that you can use to take an object file and break it down into its assembly instructions. In addition, if you are lucky enough to be working with a file that has the symbols left in, you can actually re-create parts of the C code using objdump. Let's take a look at some of the options that objdump has in detail:

```
$ objdump <options> <filename>
```

```
--demangle[=style]
```

```
--debugging
```

```
--disassemble
```

```
--source
```

```
--info
```

Demangle

When a binary is compiled from C++ source, the names of the functions are changed. This is an artifact of the way that C++ works, where you can have two completely different functions named the exact same thing, just in different namespaces. The solution is to append what looks like gibberish onto the end of the function name, and then store the new function name in a lookup table. Unfortunately, this makes it very difficult for a human who is looking at the code after the fact to understand what in the world is going on. Using demangle is the best way to get the name back to a human-readable form that you can make sense out of.

Debugging

If a binary is compiled using GCC, there can be special debugging metadata included in the file that will last even if the binary is stripped using the strip command. Using this metadata, objdump will attempt to reconstruct low-level C code for the program. This feature is by no means a slam dunk, but does offer a great place to start if you need to attack the source by hand.

Disassemble

In accordance with the Von Neumann architecture, a program holds both instructions and data, and the two are intertwined. For a human to look at the binary and determine which is which is darn near impossible. Using disassemble, objdump will parse out which elements are data and which elements are instructions, and only interpret the ones that are instructions. This can be extraordinarily useful in pairing down large executables into manageable code segments.

Source

If you are lucky enough to have a binary that hasn't been stripped and still has the source code intact, source will automatically extract it for you and place it back into the original files. The usefulness of this feature can't be understated; if you can get back to the original source code, you'll have everything you need to perform an accurate and complete analysis.

Info

You aren't always going to be dealing with executables for well-known platforms. The day may come when you are handed an RS/6000 running AIX that has been rooted. The info flag shows you all the platforms that objdump can decompile. This is useful because you won't have to re-create an entire environment to analyze a binary on some exotic platform. You can do it from the comfort of your Linux box.

IDA Pro

If you are interested in a commercial application that can help your reverse engineering efforts, I cannot recommend IDA Pro enough. It automates many of the tasks I have discussed previously, such as creating a graph showing the dependencies of the program, the execution tree, and other things to aid in your task. The software has a signature database that can help identify common functions in the program, giving you the ability to fencepost certain areas in the code. In addition, it has support for a wide variety of binaries, including ELF (the Linux format) and the Win32 architecture.

GNU DeBugger (GDB)

As mentioned earlier, GDB can be an extremely effective tool in determining postmortem what an application does. If you can create a core file, GDB will allow you to navigate the file and poke through the memory contents. You can also place watches on file handles and network sockets to see what an application is accessing when. The one downside to GDB, however, is its steep learning curve. It is a program that is now over 20 years old and it shows with the sheer enormity of functionality the program holds. There is a fairly large time commitment required to learn its ins and outs, but this curve can be mitigated somewhat by using an external GUI such as DDD to help organize what you are trying to do.

JAVA PROGRAMS

If you are working with a Java program, the concepts discussed up to here will work but the tools will not. To fully understand why, let's look at how Java programs differ from traditional native code. Instead of compiling directly to a native platform object binary, Java instead is either interpreted or compiled into byte code (JIT or Just In Time compilers are a third option that wait until the very last second to compile code, thus giving the benefits of both interpreting and compiling), for which a virtual machine (commonly referred to as the Java Virtual Machine or JVM) then acts as a translation layer between the Java program and the computer.

This process allows a program to be compiled once and then be run on many different platforms. The good news for someone who wants to reverse engineer Java code, however, is that this intermediary step makes it much easier to figure out what is going on, unless, that is, the code has been obfuscated by a third-party utility.

Obfuscation

Because of the way Java is structured, it is much easier to reverse engineer the Java byte code back into high-level Java code than it is to take a natively compiled program back into its respective language. Thus, tools that are known as Java byte code obfuscators have become popular. These tools, in essence, jumble the code around so that it can't be easily reversed, or if it is, the high-level code makes no sense to a programmer. There are many tools that can perform this obfuscation, and just about as many that can undo it. Truly protecting Java source with obfuscation is a hard problem and it is one that hasn't completely been answered yet.

Decompiling a Java Program

Compared to the hoops you have to jump through to be able to view native code, Java decompilation is a breeze. Just fire up your favorite decompiler (we prefer Jad) and let it go. The tool will then create the Java source files from the class file. The source may not be identical to what was written, but it will be close enough so that you can understand exactly what is going on. The only problem that can arise is that the class file might have been obfuscated. To illustrate the ease with which an obfuscated program can be de-obfuscated, consider RetroGuard, a well-known open-source obfuscation program. Some industrious programmers took the open-source tool and reworked it so that it could be used to de-obfuscate its own handy work. For complete details on how this was achieved, check out <http://multimedia.cx/pre/re-retroguard.html>.