Reverse Engineering Hostile Code

by Joe Stewart, GCIH

Computer criminals are always ready and waiting to compromise a weakness in a system. When they do, they usually leave programs on the system to maintain their control. We refer to these programs as "Trojans" after the story of the ancient Greek Trojan horse. Often these programs are custom compiled and not widely distributed. Because of this, anti-virus software will not often detect their presence. It also means information about what any particular custom Trojan does is also not generally available, so a custom analysis of the code is necessary to determine the extent of the threat and to pinpoint the origin of the attack if possible.

This article outlines the process of reverse engineering hostile code. By "hostile code", we mean any process running on a system that is not authorized by the system administrator, such as Trojans, viruses, or spyware. This article is not intended to be an in-depth tutorial, but rather a description of the tools and steps involved. Armed with this knowledge, even someone who is not an expert at assembly language programming should be able to look at the internals of a hostile program and determine what it is doing, at least on a surface level.

Tools Required

As with most types of engineering, you'll need some tools. We'll cover tools native to both Unix and Windows. While Unix is the ideal platform to perform the initial reverse engineering process, you can still make do on Windows, especially if you install tools such as Cygwin, a Unix environment that runs on Win32 platforms. Most of these commands are also available for Windows when running Cygwin. However, when you get to the decompile/disassemble/debug steps ahead, going the Windows route will cost a lot of money, whereas the Unix solutions are all free. Be sure to weigh the costs of working on Windows versus the benefits before making it your reverse-engineering platform of choice.

Some useful commands are:

- dd - byte-for-byte copying of raw devices. Useful to perform analysis on a compromised system's hard drive without affecting the integrity of evidence of the intrusion.
- file - tries to identify the type of a file based on content
- strings - outputs the readable strings from an executable program.
- hexedit - allows you to read and edit binary files
- md5sum - creates a unique checksum for a file for comparison
- diff - outputs differences between files
- lsof - shows all open files and sockets by process
- tcpdump - network packet sniffer
- grep - search for strings within a file

Compressed Executables

Trojans are often compressed with an executable packer. This not only makes the code more compact, it also prevents much of the internal string data from being viewed by the strings or hexedit commands. The most commonly used executable packer is UPX, which can compress

Linux or Windows binaries. There are several other packers available, but they are typically Windows-only. Fortunately, UPX is one of the few that also provide a manual decompression to restore the original file. This prevents us from having to use advanced techniques to decompress the file into its original format.

In an ordinary executable, running the "strings" command or examining the Trojan with hexedit should show many readable and complete strings in the file. If you only see random binary characters or mostly truncated and scattered pieces of text, the executable has likely been packed. Using grep or hexedit, you should be able to find the string "UPX" somewhere in the file if it was packed by UPX. Otherwise you may be dealing with one of the many other executable packers. Dealing with these other formats is beyond the scope of this article, but you can find resources to help work with these files.

Decompiling

Occasionally you will get lucky and find that the Trojan was written in an interpreted or semi-interpreted language such as Visual Basic, Java or even compiled Perl. There are tools available to decompile these languages to varying degrees.

- Visual Basic - There is a decompiler floating around the Net for VB version 3. For newer versions, there are no decompilers known, but you can use a tool such as Compuware's SmartCheck to trace calls in the program. While its output is not a source code listing, you can see just about everything the program is doing internally.
- Java - There is the excellent decompiler jad, which decompiles to a complete source code listing which can be recompiled again. Several other java decompilers are also known to exist.
- Perl - Perl programs compiled into Windows executables can be reduced to their bare script using exe2perl.

Disassembly

If the Trojan was written in a true compiled language, you'll have to bite the bullet and disassemble the code into assembly language. For Unix executables, objdump is the way to go. For Windows executables, you'll want IDA Pro or W32dasm There is a free version of IDA that is just as powerful as IDA Pro but has a console-based interface. These programs will disassemble your code, then match up strings in the data segment to where they are used in the program, as well as show you separation between subroutines. They will attempt to show you Windows API calls by name instead of by offset. This kind of output is known as a deadlisting, and can give you a good idea of what the program is doing internally. The GNU objdump program does not provide such useful features, but there is a perl-based wrapper for objdump called dasm, which will give you much of the same functionality as the Windows disassemblers.

Debuggers

While a deadlisting can be quite valuable, you will still want to use a debugger to step through the program code, especially if the Trojan is communicating via network sockets. This gives you access to the memory and temporary variables stored in the program, as well as all data it is

sending and receiving from socket communications. On Unix, gdb is the debugger of choice. It has a long history on Unix, is well documented, and best of all, is available free of charge. Under Windows, the choices are far more varied, but most tutorials on reverse engineering under Win32 assume you are using SoftICE. It does cost a fair amount of money, but is worth getting if you can afford it.

Preparing to Debug

You must take precautions when running hostile code, even under a debugger. You should never debug a Trojan on a production network. Ideally, you should set up a lab network, as shown in figure 1.

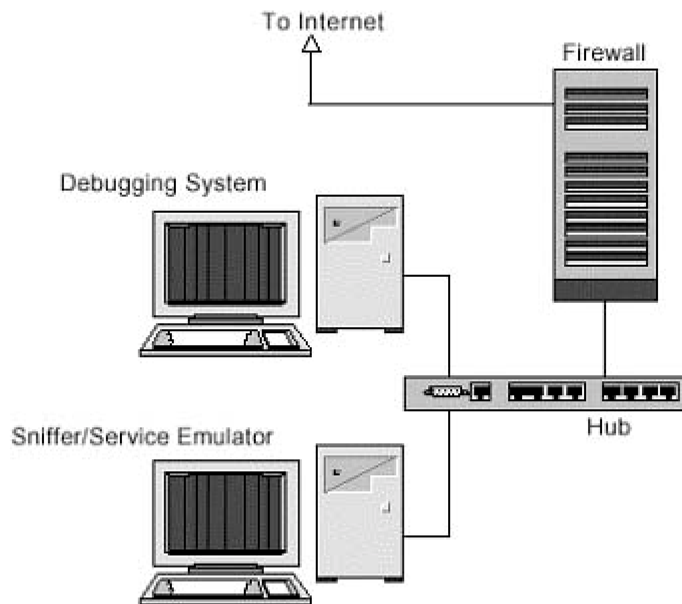

Figure 1: A typical debugging network

The debug system should have a clean install of whatever OS the Trojan is intended for, with a second box acting as a firewall. A third system on the network allows you to emulate services and capture the network traffic generated by the Trojan. Capturing this traffic can be invaluable in tracing the source of the infection. Ensure that you firewall all outbound connections, allowing only the Trojan's control connection through. If you don't want the master controller to know your lab network is running the Trojan, you can set up services to mimic the resources the Trojan needs, such as an IRC or FTP/TFTP server.

Stepping Through the Code

Now that we have constructed a proper quarantined lab environment, we can begin debugging the code. Using the deadlisting, we look for key functions in the program, such as Winsock and file I/O calls. The debugger allows us to set breakpoints in the program based on offset values, so we can interrupt the flow of the program and examine the program memory and CPU registers at

that point. The remainder of this article will look at an example of how such a debugging session might look on an x86 Linux platform.

Running the Debugger

We want to know how the Trojan communicates with its controller. Often, sniffing the network traffic will be sufficient. However, many newer Trojans are incorporating encryption into their network traffic, making network sniffing a lost cause. However, with some cleverness we can grab the messages from memory before they are encrypted. By setting a breakpoint on the "send" socket library call, we can interrupt the code just prior to the packet being sent. Then, by getting a stack trace, we can see where we are in the program. For example, the Trojan source code might look something like:

```
/* encrypt output to master */
elen = encrypt(crypted,buf,len);
/* write crypted output to socket */
send(s, crypted, elen, 0);
```

Examining the compiled Trojan in gdb might give us the following output [note that the **bolded** statement represent the author's comments on the output]:

[test@debugger test]$ gdb ./Trojan
GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of
it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
(no debugging symbols found)...
(gdb) set disassembly-flavor intel **[Switch syntax output from AT&T]**
(gdb) b send **[Set a breakpoint on the "send" library call]**
Breakpoint 1 at 0x400f5c10
(gdb) run
Starting program: /home/test/Trojan

Breakpoint 1, 0x400f5c10 in send () **[We hit a breakpoint]**
(gdb) where **[Do a stack trace to see where we are at in the program]**
#0 0x400f5c10 in send () from /lib/i686/libc.so.6
#1 0x080487fa in socket ()
#2 0x40040082 in __libc_start_main () from /lib/i686/libc.so.6

The above output from the "where" command in gdb shows us the offset each subroutine will return to after execution. Since we know that the "send" call was right after our encrypt call, we need only to examine the previous subroutine, which encompasses the return offset 0x080487fa.

We are interested in the assembly language code just prior to this offset. Using gdb, we can disassemble the code at this point.

(gdb) disas 0x080487d2 0x080487fa
Dump of assembler code from 0x80487d2 to 0x80487fa:
0x80487d2 <socket+622>: call 0x8048804 <socket+672>
0x80487d7 <socket+627>: add esp,0x10
0x80487da <socket+630>: mov DWORD PTR [ebp-836],eax
0x80487e0 <socket+636>: push 0x0
0x80487e2 <socket+638>: push DWORD PTR [ebp-836]
0x80487e8 <socket+644>: lea eax,[ebp-824]
0x80487ee <socket+650>: push eax
0x80487ef <socket+651>: push DWORD PTR [ebp-828]
0x80487f5 <socket+657>: call 0x8048534 <send>
End of assembler dump.

We see that just prior to the call to "send", there was a call to 0x8048804 < socket+672>. In reality, this is our "encrypt" subroutine. When programs are stripped of their symbols, gdb is often confused about where subroutines begin and end, so it continues the name of the last one it recognizes for all following subroutines, often the previous dynamic library call. In this case, it is mislabeled as being part of the "socket" function.

To examine the contents of the unencrypted packet, we need only know how the "call" instruction works. The arguments to our subroutine were pushed onto the "stack", a place where temporary data and return offsets are stored. We can access the contents of the variables by setting a breakpoint on the call and then using an offset from an internal CPU register known as the stack pointer, ESP. ESP+4 will be a pointer to the first argument, ESP+8 will be a pointer to the second argument, ESP+12 will be a pointer to the third argument, and so forth. Just keep poking at the stack until something useful comes up. In this case, the useful information (the plaintext data) is in the second argument to "encrypt". Let's set a breakpoint at the encrypt call, and examine the stack [Again, the **bolded** statement represent the author's comments on the output.]

(gdb) b * 0x80487d2 **[Set a breakpoint on the "encrypt" call]**
Breakpoint 2 at 0x80487d2
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/Trojan
(no debugging symbols found)...
Breakpoint 2, 0x080487d2 in socket ()
(gdb) x/x $esp+8 **[Get the offset of the second argument ESP+8]**
0xbffff5e4: 0x0806fe20
(gdb) x/fs 0x0806fe20 **[Examine the contents of the memory at 0x0806fe20]**
0x806fe20: "root pts/0 Oct 11 14:22\n"

From this output we can see that the Trojan is reporting back on who is currently logged on to the system. Of course, it could send any kind of data; network packet captures, keystroke logs, etc. Fortunately, we have our network set up so this traffic will be redirected to the sniffer host instead.

Conclusion

The Trojan above is not real. Had it been an actual Trojan, we might have followed additional courses of action. Often times a Trojan will use established channels such as IRC to reach its master. We can take advantage of this fact, and use it to track down the source of the attack, even gaining control of the entire network of Trojaned hosts if the Trojan writer has been careless. If the Trojan uses FTP to update itself, you might find additional code on the FTP server and possibly clues to the identity of the Trojan writer.

Although we've only scratched the surface of reverse engineering, you should be able to take the basic information above and put it to work. Read the documentation for your debugger; you'll be surprised at how powerful it can be, and how much it can tell you; even if you're not the best at reading assembly code. If it seems overwhelming at first, don't give up hope. The payoff can be quite gratifying. During one reverse-engineering session the author of this article found the real name of the Trojan author unintentionally embedded in the program's source code (hint: don't write Trojans in VB when logged in to your NT workstation at work). With a quick trip to Google the author's email address and picture was available, posted to a VB discussion site. One "whois" later and his home address and phone number was found. Somewhere in Brazil, a Trojan writer slaps his forehead and says (in Portuguese), Doh!