7

Phase 3: Gaining Access Using Application and Operating System Attacks

At this stage of the siege, the attacker has finished scanning the target network, and has developed an inventory of target systems and potential vulnerabilities on those machines. Next, the attacker wants to gain access on the target systems. The particular approach to gaining access will depend heavily on the skill level of the attacker, with simple script kiddies trolling for pre-packaged exploits and more sophisticated attackers using highly pragmatic approaches.

Script Kiddie Exploit Trolling

To try to gain access, the average script kiddie will just take the output from the vulnerability scanner and surf over to a Web site offering vulnerability exploitation programs to the public. Several organizations offer huge databases of canned exploits, with search engines allowing an attacker to look up a particular application, operating system, or discovered vulnerability. Some of the most useful Web sites offering up large databases chock full of exploits include:

- Packet Storm Security, run by Securify, Inc., at *packetstorm.securify.com*
- Technotronic Security Information, at *www.technotronic.com*
- Security Focus Bugtraq Archives, at *www.securityfocus.com*

Some controversy surrounds the organizations distributing these exploits. Most of the organizations offering these exploits have a philosophy of complete disclosure—if the attackers know about these exploits, they should be made public so that everyone can learn about the techniques to defend against them. With this mindset, these purveyors of explicit exploit information argue that they are providing a service to the Internet community. Others take the view that these exploits just make attacks easier and more prevalent. While I respect the arguments of both sides of this disclosure controversy, I tend to fall into the full-disclosure camp (but you could have guessed that, given this book on the same topic).

As shown in Figure 7.1, a script kiddie will search one of the exploit databases to find an exploit for a hole detected during a vulnerability scan. The script kiddie will download the prepackaged exploit, configure it to run against the target, and launch the attack, usually without even really understanding how the attack works. Although this indiscriminate attack technique fails against well-fortified systems, it is remarkably effective against huge numbers of machines on the Internet whose system administrators do not keep the systems patched and configured securely.

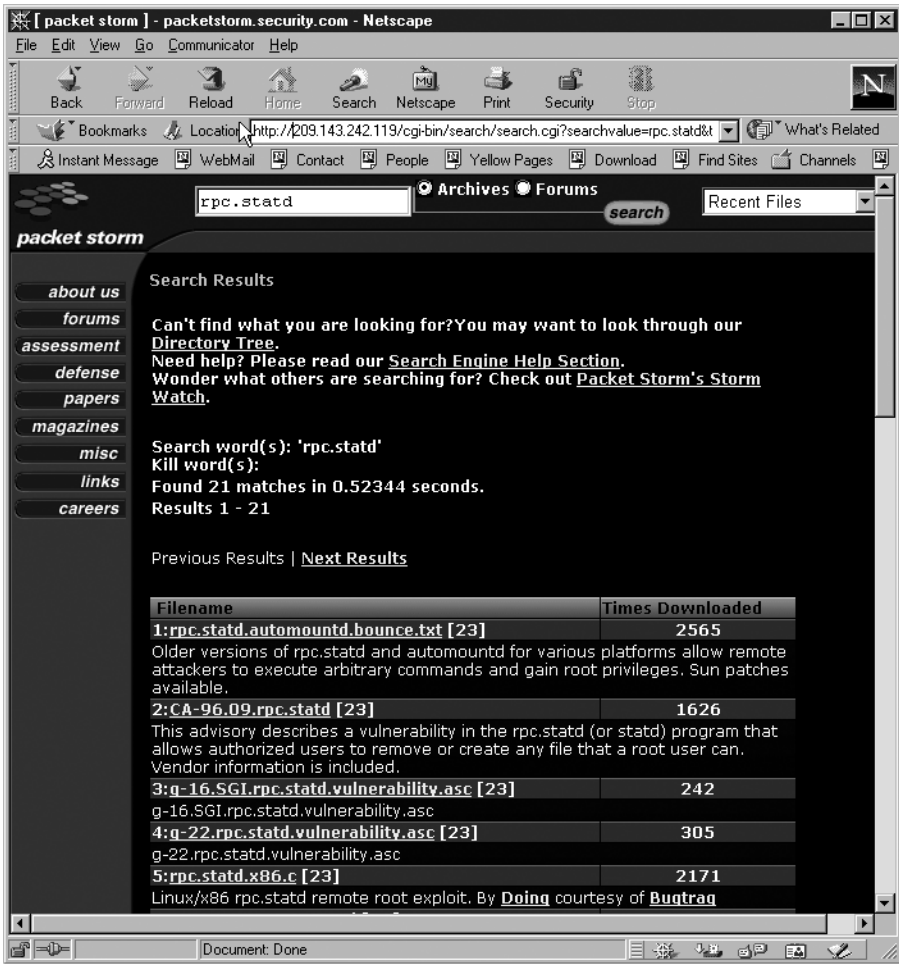


Figure 7.1
Searching Packet Storm for a common vulnerability exploit.

Pragmatism for More Sophisticated Attackers

While a script kiddie utilizes these Internet searches to troll for canned exploits without understanding their function, a more sophisticated attacker will employ far more complex techniques to gain access. Let's focus on these more in-depth techniques for gaining access and the ideas underlying many of the canned exploits.

Of the five phases of an attack described in this book, Phase 3, the gaining access phase, tends to be very free form in the hands of a more sophisticated attacker. Although the other phases of an attack (reconnaissance, scanning, maintaining access, and covering tracks) are often quite systematic, the techniques used to gain access depend heavily on the architecture and configuration of the target network, the attacker's own expertise and predilections, and the level of access that the attacker begins with. Because of all these dependencies, the more sophisticated attackers are very pragmatic during the gaining access phase, selecting from a variety of techniques based on the particulars of the target environment.

In this book, we discussed the reconnaissance and scanning phases in a roughly chronological fashion, stepping through each tactic in the order used by a typical attacker. However, given that gaining access is based so heavily on pragmatism, experience, and skill, there is no such clearly defined order for this phase of the attack. We will discuss this phase by describing a variety of techniques used to gain access. Our discussion of these techniques will start with attacks against operating systems and applications, followed, in the next chapter, by a discussion of network-based attacks.

There are dozens of popular operating systems and hundreds of thousands of different applications, and history has shown that each operating system and most applications are teeming with vulnerabilities. A large number of these vulnerabilities, however, can be attacked using variations on popular and recurring themes. In the remainder of this chapter, we will discuss some of the most widely used and damaging application and operating system attacks, namely stack-based buffer overflows, password attacks, and Web application attacks.

Stack-Based Buffer Overflow Attacks

Stack-based buffer overflow attacks are extremely common today and offer an attacker a way to gain access to and have a significant degree of control over a vulnerable machine. While they have been known for many years, this type of attack really hit the big time when a seminal paper on the topic called "Smashing the Stack for Fun and Profit" was written by Aleph One and published in the Phrack online magazine (issue no. 49). You can find this detailed and well-written paper at *packetstorm.securify.com/docs/hack/smashstack.txt*.

Any application or operating system component that is poorly written could have a stack-based buffer overflow. By exploiting a vulnerable application or operating system, an attacker can execute arbitrary commands on the target machine, potentially taking over the whole machine. Imagine if I could execute one or two commands on your valuable server, workstation, or palmtop computer. Depending on the privileges I'd have to run these commands, I could add accounts, change passwords, alter the system's configuration… anything I want to do, really.

Attackers love this ability to execute commands on a target computer. To understand how stack-based buffer overflows can yield this type of access, we need to understand an important element of most modern computing architectures, a stack.

What Is a Stack?

A *stack* is a data structure that stores important information for processes running on a computer. The stack acts kind of like a scratch pad for the system. The system writes down important little notes for itself to remember and places these notes on the stack, a special reserved area in memory. Stacks are similar to (and get their name from) stacks of dishes, in that they behave in a Last-In, First-Out manner (known as LIFO). That is, when you are creating a stack of dishes, you pile dish on top of dish to build the stack. When you want to remove dishes from the stack, you start by taking the top dish, which was the last one placed on the stack. The last one in is the first one out. Similarly, when the computer puts data onto its stack, it pushes data element after data element on the stack. When it needs to access data from the stack, the system will first take off the last element it placed on the stack.

So, what types of things does a computer store on a stack? Among other things, stacks are used to store information associated with function calls on the computer. *Function calls* are used by programmers to break code down into smaller pieces. Figure 7.2 shows some sample code written in the C programming language.

When the program starts to run, the `main` procedure gets executed first. The first thing the `main` procedure does is call our sample function. All processing by the program will now transition from the `main` procedure to the sample function. The system has to remember where it is operating in the `main` procedure, because after `sample_function` finishes running, the program flow must return back to the `main` procedure. The stack helps to orchestrate this process of moving to and from the function call.

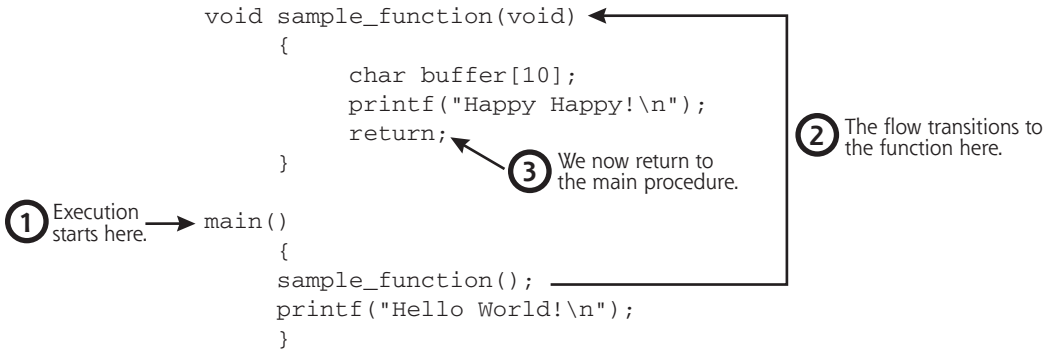


Figure 7.2
Sample code with function call.

As shown in Figure 7.3, the system will push various data elements onto the stack associated with making the function call. First, the system pushes the function call arguments onto the stack. This includes any data handed from the `main` procedure to the function. To keep things simple, our example includes no arguments in the function call. Next, the system pushes the return pointer onto the stack. This return pointer indicates the place in the system's memory where the next instruction to execute in the `main` procedure resides. The whole program itself is just a

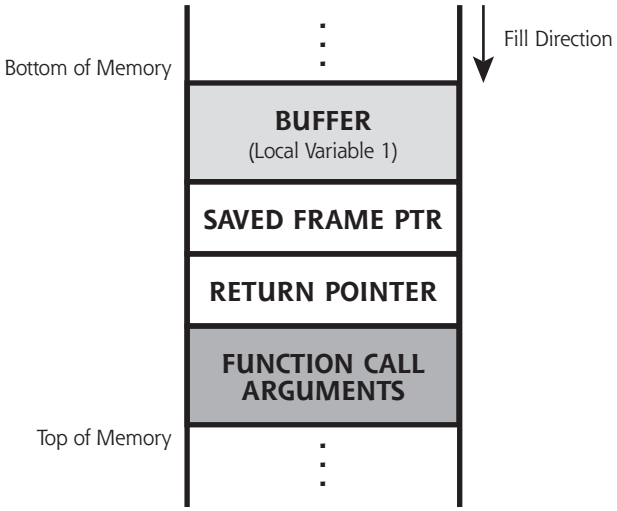


Figure 7.3
A normal stack.

bunch of bits in the computer's memory, in the form of a series of instructions for the processor. The processor has a register (just a small piece of fast memory in the processor itself) called the *instruction pointer* that indicates which instruction the processor should execute. This instruction pointer gets incremented as the program runs, going through instruction after instruction in a program and jumps in value when a function is called. For a function call, the system needs to remember the value of the instruction pointer in the `main` procedure so that it knows where to go back to for more instructions after the function finishes running. The instruction pointer is copied onto the stack as a return pointer.

Next, the system pushes the frame pointer on the stack. This value helps the system refer to various elements on the stack itself. Finally, space is allocated on the stack for the local variables that the function will use. In our example, we've got one local variable called *buffer* to be placed on the stack. These local variables are for the exclusive use of the function, which can store its local data in them and manipulate their value.

After the function finishes running, printing out its happy message, control returns to the main program. This transition occurs by popping the local variables from the stack (in our example, the variable "buffer"). For efficiency sake, the memory allocated to the variables is not erased. Data is removed from the stack just by changing the value of a pointer to the top of the stack. This stack pointer now moves down to its value before the function was called. The saved frame pointer is also removed from the stack and squirreled away in the processor. Then, the return pointer is removed from the stack and loaded into the processor's instruction pointer register. Finally, the function call arguments are removed, returning the stack to its original (prefunction call) state. At this point, the program begins to execute in the `main` procedure again, because that's where the instruction pointer tells it to go.

What is a Stack-Based Buffer Overflow?

Now that we understand how a system interacts with the stack, let's look at how an attacker can abuse this capability. A buffer overflow is rather like putting 10 liters of stuff into a bag that will only hold five liters. Clearly, something is going to spill out. Consider the sample program offered by Aleph One in his "Smashing the Stack" paper in Figure 7.4.

For this program, the main routine creates a big buffer containing 255 copies of the character A, which it passes to `sample_function`. In `sample_function`, the `big_buffer` is referred to as "string," and a local

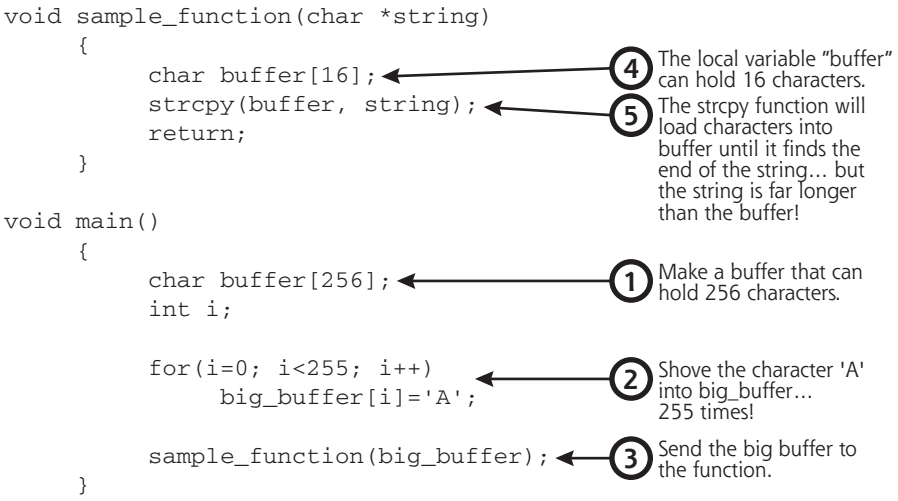


Figure 7.4
Bufer overflow sample program.

variable called "buffer" is allocated space on the stack to hold 16 characters. Next, we encounter the `strcpy` routine. This routine is used to copy information from one string of characters to another. In our program, `strcpy` will move characters from string to buffer. Unfortunately, `strcpy` is very sloppy, because it doesn't check the size of either string, and happily copies from one string to the other until it encounters a null character in the source string. A null character, which consists of a bunch of zero bits, usually indicates the end of a string. This sloppiness of `strcpy` is a well-known limitation found in many of the normal C language library functions, particularly string functions. When we created `big_buffer`, we did not put a null character at the end, and we also built the string (255 characters) to be far larger than the buffer (16 characters). This is bad news, because the system will allow `strcpy` to write far beyond where it's supposed to. That's one of the big problems with computers: They do exactly what we tell them to, no more and no less.

What happens to the stack when we do this? Well, it gets messed up. The A characters will spill over the end of buffer, running into the saved frame pointer, and even into the return pointer. The return pointer on the stack will be filled with a bunch of A's. When the program finishes executing the function, it will pop the local variables and saved frame pointer off of the stack, as well as the return pointer (with

all the A's). The return pointer is copied into the processor's instruction pointer, and the machine tries to fetch the next instruction from a memory location that is the binary equivalent of a bunch of A's. Most likely, this is a bogus memory location, and the program will crash.

So, after all this discussion, we've learned how to write a program that can crash. "Gee," you may be thinking, "Most of the programs I write crash anyway." I know mine do.

But let's look at this more closely. Although loading a bunch of A's into the return pointer made the program crash, what if we could overflow our buffer with something more meaningful? We could insert actual machine language code into the buffer, with commands that we want to get executed. But how can we get the system to execute these commands? Remember, when we run off the end of the local variables, we can modify the return pointer. By overflowing a buffer, we could overwrite the return pointer with a value that points back into the buffer, which contains the commands we want to execute. The resulting recipe, as shown in Figure 7.5, is a stack-based buffer overflow attack, and will allow us to execute an arbitrary command on the system.

Let's review how the smashed stack works. The attacker forces a program to fill one of its local variables (a buffer) with data that is longer than the space allocated on the stack, overwriting the local variables themselves with machine language code. But the system doesn't stop at

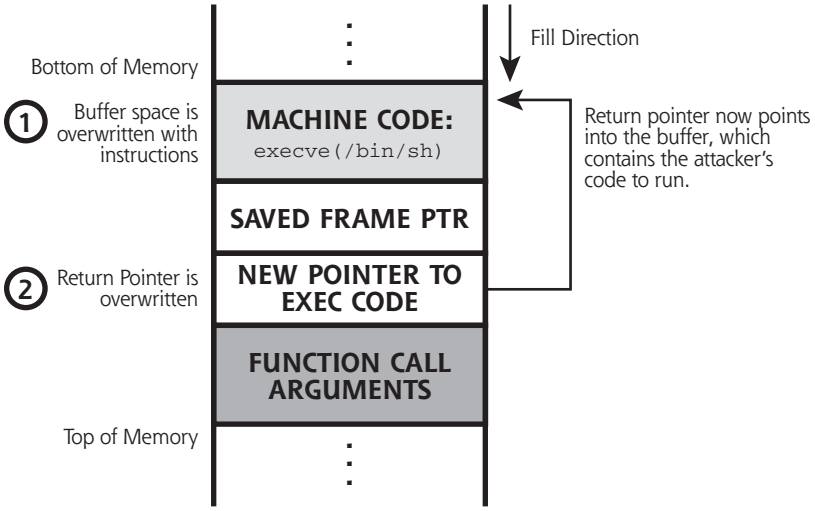


Figure 7.5
A smashed stack.

the end of the local variables. It keeps writing data over the end of the buffer, even overwriting the return pointer with a value that points back to the machine language instructions we've loaded into the stack. When the function call finishes, the local buffers containing the instructions will be popped off the stack, but the information we place in those memory locations will not be cleared. The system then loads the return pointer into the processor, and starts executing instructions where the return pointer tells it to. The processor will then start executing the instructions the attacker had put into the buffer on the stack. Voila! The attacker just made the program execute arbitrary instructions from the stack.

This whole problem is the result of a function not checking the size of the information it is putting into a local variable. Without carefully doing a size check of these buffers before manipulating them, a function call can easily blow away the end of the stack. Essentially, stack-based buffer overflows are a result of sloppy programming by not doing bounds checks on data being placed into local variables, or using a library function written by someone else with the same problem.

Now that we understand how an attacker puts code on the stack and gets it to execute, let's analyze the kind of instructions that an attacker will place on the stack. In UNIX, probably the most useful thing to force the machine to run is a command shell, because a command shell (such as `/bin/sh`) can be fed any other command to run. This can be achieved by placing the machine language code for executing (using the `execve` system call) `/bin/sh` on the stack. After spawning a command shell, the attacker can then automatically feed a few specific system commands into the shell, running any program or system call on the target machine. On Windows NT/2000 systems, attackers often use a buffer overflow to trigger a specific Dynamic Link Library (DLL) to get their work done on the target. A DLL is simply a small program used by a variety of applications on the system to accomplish some task. One of the most effective DLLs to call on a Windows NT/2000 machine with a buffer overflow is `WININET.DLL`, a program that allows an attacker to easily send requests to and get information from the network to download additional code or retrieve commands to execute.

Buffer overflow attacks are very processor- and operating system-dependent, because the raw machine code will run only on a specific processor, and techniques for executing commands differ on various operating systems. Therefore, a buffer overflow exploit against a Linux machine with an x86 processor will not run on a Windows NT box on an Alpha processor or Solaris system with a Sparc processor, even if the

same buggy program is used on all of these systems. The attack must be tailored to the target processor and operating system type.

Exploiting Stack-Based Buffer Overflows

This may all sound great, but how does an attacker actually exploit a target using this technique? Keep in mind that the vast majority of useful, modern programs are written with function calls, some of which do not do proper bounds checking when handling their local variables. A user enters data into a program by using the program's inputs. When running a program on a local system, these inputs could be through a GUI, command-line interface, or even command-line arguments. For programs accessed across the network, data enters through open ports listening on the network, usually formatted with specific fields that the program is looking for.

To exploit a buffer overflow, an attacker will enter data into the program by typing characters into a GUI or command line, or sending specially formatted packets across the network. In this input to the program, the attacker will include the machine language code and return pointer in a single package. If the attacker sends just the right code with the right return pointer formatted the right way to overflow a buffer of a vulnerable program, a function in the program will copy the buffer to the stack and ultimately execute the attacker's code. Because everything has to be formatted extremely carefully for the target program, creating new buffer overflow exploits is not easy.

Finding Buffer Overflow Vulnerabilities

Most stack-based buffer overflow attacks are carried out by simple script kiddie attackers that do not understand how their tools work. They just scan the target with an automated tool that detects the vulnerability, download the exploit code written by someone else, and point the exploit tool at the target. The exploit itself was likely written by someone with a lot more experience and understanding in discovering vulnerable programs and creating successful exploits.

How does the creator of a stack-based buffer overflow exploit find programs that are vulnerable to such attacks? These folks will carry out detailed analyses of programs looking for evidence of functions that do not properly bounds-check local variables. If the attackers have the source code for the program, they can look for a large number of often-used functions that are known to do improper bounds checking. The

`strcpy` routine we saw earlier is just such a function that programmers often misuse, resulting in a stack-based buffer overflow vulnerability. Other C-language functions that often cause such problems include:

- `fgets`
- `gets`
- `getws`
- `memcpy`
- `memmove`
- `scanf`
- `sprintf`
- `strcat`
- `strncpy`

An exploit creator will search the source code or use a debugger on an executable program to find evidence of the use of these functions. Also, if the attackers have the source code, they can utilize automated tools to find weak functions in the program.

Alternatively, if they do not have the source code, exploit creators may take a more brute force approach to finding vulnerable programs. They will run the program in a lab and configure an automated tool to cram massive amounts of data into every input of the program. The program's local user input fields, as well as network inputs, will be inundated with data. When cramming data into a program looking for a vulnerability, the attacker will make sure the data has a repeating pattern, such as using the character "A" repeated thousands of times. The exploit creator is looking for the program to crash under this heavy load of input, but to crash in a meaningful way. They'd like to see their repeated input pattern (like the character "A," which in hexadecimal format is 0x41) reflected in the instruction pointer when the program crashes.

Consider the example of a famous buffer overflow exploit widely hyped by the eEye security team in mid-1999. The team was looking for vulnerabilities in Microsoft's IIS server by bombarding it with input using their Retina security product. After cramming input for an hour, IIS crashed, leaving the following values in the processor's registers:

```
EAX = 00F7FCC8 EBX = 00F41130
ECX = 41414141 EDX = 77F9485A
ESI = 00F7FCC0 EDI = 00F7FCC0
EIP = 41414141 ESP = 00F4106C
EBP = 00F4108C EFL = 00000246
```

Don't worry about all the different values; just look at the instruction pointer (EIP). Attackers love this value. The pattern being entered into the program's input (a long series of 0x41) somehow made its way

into the instruction pointer. Therefore, most likely, user input overflowed a buffer, got placed into the return pointer, and then transferred into the processor's instruction pointer. Based on this tremendous clue about a vulnerability, the eEye team created a buffer overflow exploit that let an attacker gain command shell access of Windows NT systems running IIS. You can read more details of the eEye team's discovery and subsequent interactions with Microsoft at *www.eeye.com/html/advisories/AD19990608.html*.

When exploit creators find a vulnerable function call (either by inspecting the source, debugging, or cramming input), they will carefully analyze how the function gets input from a user to determine whether and how user data gets fed into the function. Based on this analysis, they will write specific code that provides the proper input to push machine instructions onto the stack and overwrite the return pointer. Again, properly positioning the machine language shell code instructions and setting the return pointer to the right value can be quite difficult. The shell code also has to fit into the buffer of the target program. Furthermore, the machine language instructions to be put on the stack must avoid any character filtering done on the buffers by the target program. If a vulnerable string function is being exploited, the machine language code and return pointer must not include null characters, which stop processing in many string functions. Aleph One covers some techniques for getting all of this right in his *Smashing the Stack* paper. Other excellent documents covering this topic include:

- Taeho Oh's *Advanced Buffer Overflow Exploit* paper, available at *ohhara.sarang.net/security/adv.txt*.
- A really well-done talk by Greg Hoglund on the same subject at *www.blackhat.com/presentations/bh-asia-00/greg/greg-asia-00-stalking.ppt*.
- Dark Spyrit's paper on Windows buffer overflows, available at *www.beavuh.org/dox/win32_oflow.txt*.

The Make up of a Buffer Overflow

Let's focus more on the data components of a buffer overflow exploit. What does the attacker send to the target to trigger the overflow? Clearly, the attacker must send the machine language code for the commands to be executed. Furthermore, the attacker must send information to write over the return pointer so that it points back into the stack, where the attacker's machine language code awaits to be executed. Set-

ting this return pointer to just the right value is extremely important. If it jumps to the wrong area of memory, the program might crash, or the attacker's code may not be properly executed. Making the task even more difficult for the attacker, the particular location in memory where the stack is working at a given instant is dynamic. Therefore, the attacker often has to guess the proper place in memory to jump to execute the machine language code on the stack.

To help improve the odds that the return pointer will jump to a good place to begin executing the attackers' code, attackers will often prepend a series of NOP instructions to their machine language code. A NOP (pronounced "no-op" or "nop," depending on whom you ask) is just a command telling the processor to do nothing. The processor takes the command, does nothing, and then loads the next command. Each CPU brand has one or more instruction types that implement a NOP, which is used to make the processor wait for a tick of its clock. The attackers will put a bunch of NOPs in front of their code on the stack. Several hundred or even a thousand or more NOPs will be included, depending on the buffer size. These NOPs in a buffer overflow exploit are sometimes called a NOP slide or sled. The data components that make up the buffer overflow then consist of the NOP sled, which is located on the stack first, followed by the machine language code of the instructions the attacker wants to execute, and finally the return pointer.

These NOPs help improve the odds that the return pointer will contain a valid jump to execute the attacker's code. Without the NOP sled, the attacker would have to jump exactly to the start of the attacker's instructions, calibrating the return pointer to an exact value. With the NOP sled, the attacker only has to jump somewhere into the sea of NOPs. As long as the attacker's guess at a return pointer is accurate enough to fall somewhere into the NOP sled, the NOPs will be processed one by one without any effect on the processor, until the attacker's code is reached. Then, the attacker's code will be executed, successfully completing the buffer overflow attack. For this reason, most buffer-overflow attacks include a NOP sled.

Intrusion Detection Systems and Stack-Based Buffer Overflows

Most network-based Intrusion Detection Systems (IDSs) identify stack-based buffer overflows by conducting signature matching, looking for NOP sleds, commonly used machine language code to get attackers'

commands executed, or frequently used return pointers associated with popular buffer overflows. Any one of these elements of buffer overflow exploits can be easily detected by an IDS. By monitoring the traffic on the network to see if a bunch of NOPs, typical exploit code, or common return pointers go by, the IDS can detect such attacks and alert an administrator. The most popular type of buffer overflow signature implemented in network-based IDS tools is the tell-tale NOP sled.

Application Layer IDS Evasion for Buffer Overflows

Because stack-based buffer overflows are so powerful and popular, attackers want to use them while avoiding detection. A recent area of activity in the computer underground involves evading the IDS signature matching capabilities for buffer overflows by implementing application-layer techniques for altering the appearance of buffer overflow exploits on the network. A software developer named K2 has released a powerful tool called ADMutate that implements several very clever techniques for modifying buffer overflow attacks to evade network-based IDS capabilities. ADMutate can be found at *www.ktwo.ca/security.html*.

ADMutate accepts a buffer overflow exploit as its input. Then, the tool modifies the exploit using a technique borrowed from the computer virus world called *polymorphism*. ADMutate modifies the buffer overflow exploit to create a new exploit that does not match the signature of the old exploit, but is otherwise functionally equivalent. How does it create polymorphic buffer overflow code? Remember, a buffer overflow exploit consists of three main components: a NOP sled, the machine language code with the attacker's commands, and the return pointer.

ADMutate alters each of these three components to create a different set of instructions with the same ultimate function. For the NOP sled, ADMutate randomly substitutes a bunch of functionally equivalent statements for the NOPs. For example, instead of implementing the exact processor command NOP, the tool will substitute an instruction that moves the contents of a register back to that same register. Essentially, nothing is done, but the instruction doesn't match the NOP that the IDS tool is looking for. ADMutate has a bunch of NOP-equivalent instructions built into it that it will randomly substitute in creating a functionally equivalent NOP sled that doesn't match any signatures.

For the machine language code part of the buffer overflow exploit, ADMutate uses a simple function to alter the machine language code. ADMutate applies the XOR function to the code to combine it with a

randomly generated key. The output of this process is a bunch of gibberish to both the IDS looking for the attack and the CPU it is destined to run on. The resulting data is completely dependent on the randomly generated key. Of course, to get the attacker's code to run on the target system, the XOR encoding must be removed when the attack gets to the target. To undo the XOR function, ADMutate inserts additional machine language instructions in the buffer overflow exploit to use the key to decode the attacker's exploit instructions. Now, you may be thinking, "Well, the IDS can just look for the decoder instructions on the network to detect the attack." However, K2 thought of that when implementing ADMutate. The decoder itself is polymorphic. It is randomly created by choosing from a bunch of functionally equivalent instructions, laced with various types of NOPs. Therefore, the decoder always has a different appearance on the network to evade IDS machines.

Finally, ADMutate alters the appearance of the return pointer by simply tweaking the least significant bits of the address used for the jump. As long as the jump still ends up in the NOP sled, the attack will still work, so ADMutate changes the least significant byte of the return address to some random value.

Finally, ADMutate combines the four polymorphic components together: the functionally equivalent NOP sled, the randomly generated XOR decoder with the key, the XOR'ed machine code for the exploit, and the modified return pointer. Now, for this to all work, ADMutate must make sure that it does not include any sequence of bits that the target program will filter out or will stop processing the exploit on the target. In particular, a sequence of seven or eight zero-bits will be interpreted as a null character, which will stop the function of an errant string function. Therefore, ADMutate automatically creates valid machine language code that doesn't include any sequences of null characters or any other characters configured by the attacker.

By using these techniques, an attacker can write a buffer overflow exploit program and feed it into ADMutate. ADMutate can then be used to generate hundreds or thousands of functionally equivalent exploit programs, each with a different signature to evade IDS mechanisms.

Once the Stack Is Smashed... Now What?

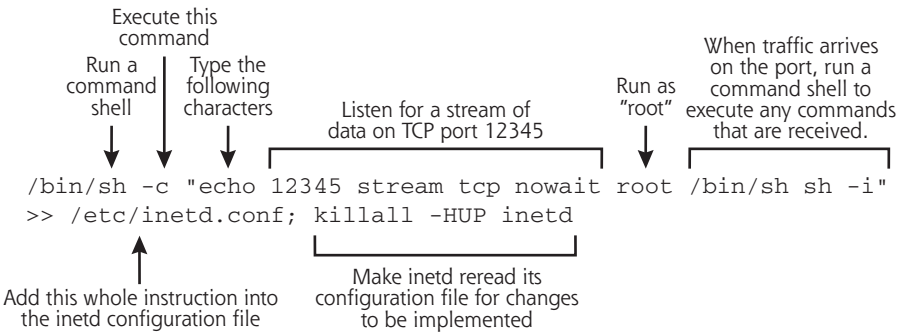
With a vulnerable program, the attacker can force the program to spawn a command shell, and enter a command or two into that command shell for execution. The shell and command will run under the context of the vulnerable process. If the process runs with super-user

privileges (root or administrator), the attacker will have those privileges for the commands to be executed through the buffer overflow. If the vulnerable process runs as another user, the attacker has that user's privileges. Because of this, attackers love to find vulnerable programs that run SUID root on a UNIX system or with administrator or system privileges on a Windows NT machine. Sometimes attackers will exploit one nonsuper-user buffer overflow vulnerability remotely to gain access to an account on a machine across the network. Then, having gained access to one account on the machine, they will escalate their privileges by exploiting a local buffer overflow vulnerability on the machine to gain super-user access.

When exploiting a stack-based buffer overflow vulnerability, what type of commands will an attacker feed into the command shell? There are an enormous number of possibilities, but let's look at the most popular techniques used to exploit a buffer overflow across a network: creating a backdoor using `inted`, backdooring with TFTP and Netcat, and shooting back an Xterm.

Creating a Backdoor Using Inetd

As we discussed in Chapter 3, on UNIX systems, the `inetd` process listens for connections on various ports and spawns a process to handle incoming network traffic. If attackers can find a network-accessible buffer overflow vulnerability in any program running with root privileges on the system, they can alter the configuration for `inetd`, stored in the `/etc/inetd.conf` file. For example, an attacker may overflow a buffer in some root-level program to get a command string like the following to be executed:



This string runs a command shell (`/bin/sh`), telling it to add a line to the end of the `/etc/inetd.conf` file. (The format of the `/etc/inetd.conf` file is described in more detail in the Chapter 3 section titled "Automatically Starting up Processes: Init, Inetd, and Cron.") This new line in the inetd configuration file will tell the `inetd` process to listen on TCP port 12345. When someone connects to this port, `inetd` will spawn an interactive command shell, running as root. Finally, the string includes the `killall` command, which sends the HUP signal to the `inetd` process. This `killall` command simply makes inetd reread the newly modified configuration file.

After making this modification to `inetd`, the attacker can use a tool called Netcat, which we discuss in detail in Chapter 8, to connect to the target system. Netcat allows the attacker to make a raw interactive connection to any port on another system. The attacker will use Netcat to connect and be presented with an interactive command prompt, having the ability to type any commands into a session. Essentially, the attacker has created a backdoor listener with inetd allowing root-level command line access to the system. The attacker could reconfigure the machine, steal data, or anything they desire, having gained root-level control over the box.

One downside to this technique from an attacker's perspective is that it requires a modification of the `/etc/inetd.conf` file on the target machine. A good system administrator will likely notice this modification quickly using a file system integrity checking tool, like Tripwire, exposing the attacker to rapid detection. Still, if the target administrator doesn't notice the change to the system, this technique works quite well.

Backdooring with TFTP and Netcat

While the `inetd` backdoor technique is UNIX-specific, another technique frequently used to gain control over Windows NT or UNIX systems is to utilize the trivial file transfer protocol (TFTP) client and Netcat to create a backdoor listener. TFTP, which is included with Windows NT and various UNIX varieties, is a very simple program used to transfer files across a network, kind of like a little sibling to FTP. It is often used by routers to retrieve their operating system and configuration across a network. Netcat, a tool we'll examine in detail in Chapter 8, can be used to push a command shell prompt across the network.

In this type of attack, an attacker will exploit a vulnerable program on the target system, getting it to execute the TFTP client. The TFTP client is then used to load the Netcat program on the target system. Net-

cat (which is called "nc" for short) can be configured to execute a command shell pushed to the attacker's machine for command input. This technique, illustrated in Figure 7.6, is very popular today and quite powerful. Before starting this attack, the attackers will load the Netcat executable on their own TFTP server so that it can be reached across the Internet.

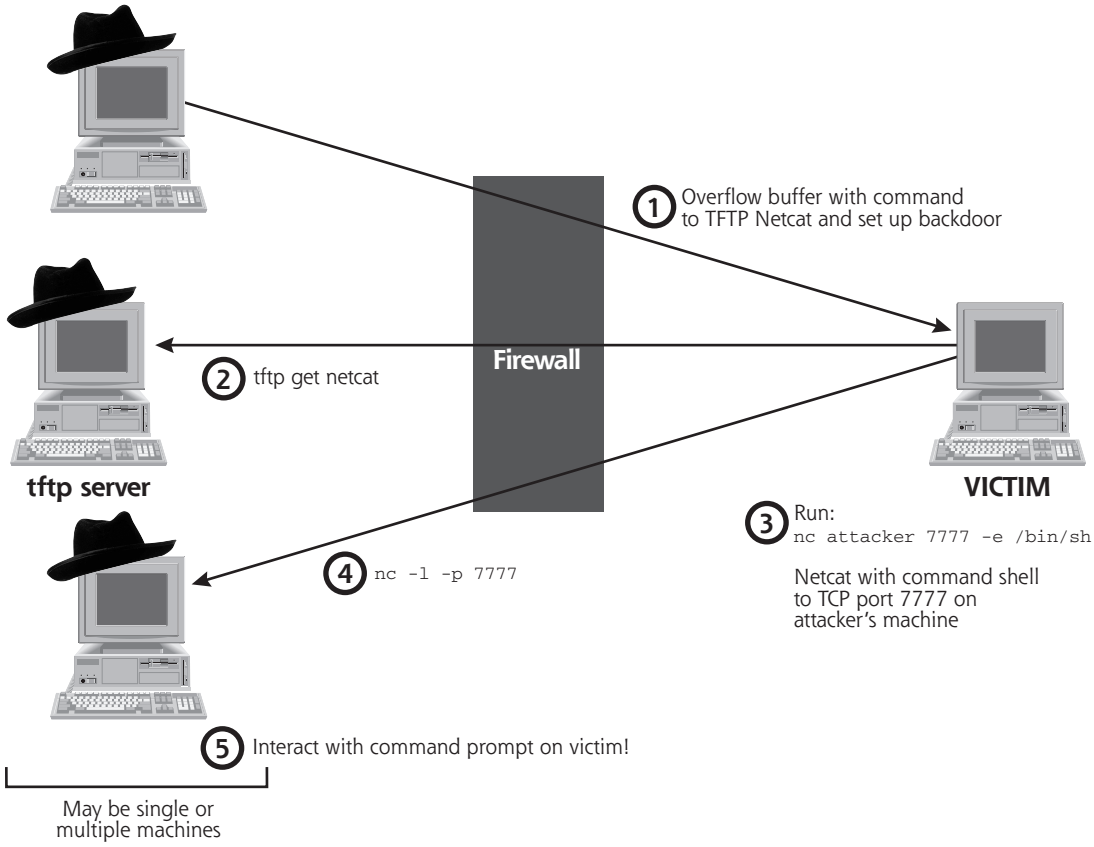


Figure 7.6
Placing a backdoor using buffer overflows, TFTP, and Netcat.

The steps of this attack include:

Step 1: The attacker overflows the buffer, getting the victim program to spawn a shell with a command to activate the TFTP client.

Step 2: The TFTP client on the victim machine downloads a copy of Netcat from the attacker's system and runs it.

Step 3: The victim machine runs Netcat configured to execute a shell and push it to the attacker's machine.

Step 4: Using a copy of Netcat on the attacker's machine, the attacker waits for a connection.

Step 5: The attacker now has interactive shell access on the target machine.

If outgoing TFTP is blocked at the firewall, the attacker could use the FTP client to transfer the Netcat executable. Using either TFTP or FTP, the attacker has gotten interactive command line access on the target system running with the privileges of the vulnerable process. One benefit for the attacker of this technique is that it leaves the configuration of the target system intact; no modifications of `inetd.conf` or any other system settings are required to gain access.

Shooting Back Xterms

Another popular method of gaining access using a buffer overflow is to use the X Window system, commonly referred to simply as "X." X is a popular GUI used on most UNIX systems, and a small number of Windows NT machines with a third-party X Window system program. This technique works against any target that has the X Window system package installed, with a firewall that allows outgoing X connections.

Many networks carefully filter incoming connections at a firewall, fearful that an attacker will get in. However, they ignore outgoing connections, letting them through unfettered. On many networks, an attacker can get a publicly available server to shoot back an X Window connection. Attackers frequently use this technique to run the Xterminal program (Xterm) to gain incoming command-line access using an outgoing X connection. The flow of this attack is shown in Figure 7.7.

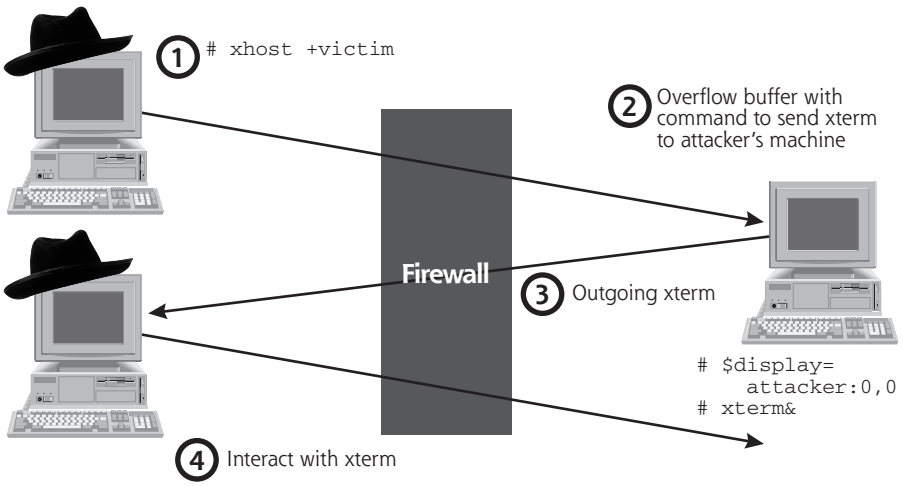


Figure 7.7
Getting an Xterm using a buffer overflow.

The steps of this attack are:

Step 1: The attackers configure their own machine to accept incoming X sessions from the target network.

Step 2: The attacker overflows the buffer of a vulnerable program on the target machine, executing a command shell.

Step 3: The shell on the victim machine is fed a command to run the Xterm program, directing its display to the attacker's machine.

Step 4: The attacker types commands into the Xterm, which are executed on the victim machine.

This attack has several benefits to an attacker. First, no modifications of the target's configuration are required. Additionally, no software (like the Netcat program) needs to be loaded onto the target. As long as the target has X installed and allows outgoing X connections, this attack is clean and simple.

Beyond Buffer Overflows

It is important to note that these three techniques—creating a backdoor using inetd, backdooring with TFTP and Netcat, and shooting back an

Xterm—are useful for attacks beyond just stack-based buffer overflows. While they work nicely against systems that have buffer overflow vulnerabilities, these basic techniques can apply to any vulnerability that allows an attacker to execute an arbitrary command on the target system. Besides buffer overflows, there are hundreds of vulnerabilities that allow an attacker to execute a command on a target. These vulnerabilities are usually caused by programming errors. The programs do not properly screen user input or include some other logic flaws that let an attacker forward commands to be executed into a command shell. Examples of widely used exploits that are not buffer overflows but could be teamed up with techniques like the inetd, TFTP/Netcat, and Xterm, gaining access techniques include:

- The IIS Unicode exploit, discovered in October 2000, which lets an attacker execute commands on a Windows NT/2000 machine running IIS. Rainforest Puppy's fantastic write-up/rant describing this attack can be found at *www.wiretrip.net/rfp/p/doc.asp?id=57*.
- The wu-ftpd string input validation problem, widely exploited against UNIX systems starting in mid-2000. You can read more about it at *www.kb.cert.org/vuls/id/29823*.
- Rainforest Puppy's RDS exploit, discovered in 1999, which lets an attacker execute commands on a Windows NT server running IIS. Another Rainforest Puppy description of this problem is located at *www.wiretrip.net/rfp/p/doc.asp?id=1*.

To learn more about these and other new exploits, you should keep up to speed by reading a variety of free information resources available on the Internet. The most valuable resource for this type of information is the BugTraq mailing list, housed at *www.securityfocus.com/frames/?content=/forums/bugtraq/intro.html*. If you don't have enough time for the great level of detail and traffic volumes of BugTraq, you can read the far less detailed (and also less timely) advisories from Carnegie Mellon's Computer Emergency Response Team (CERT), whose mailing list is described at *www.cert.org/contact_cert/certmaillist.html*. Another list that might suit your fancy is the SANS Newsbite mailing list, distributed by the SANS Institute, available at *www.sans.org*.

Stack-Based Buffer Overflow and Related Attack Defenses

There are a variety of ways to protect your systems from stack-based buffer overflow attacks. These defensive strategies fall into the following two categories:

- Defenses that can be applied by system administrators and security personnel during deployment, configuration, and maintenance of systems.
- Defenses applied by software developers during program development.

Both sets of defenses are very important in stopping these attacks, and they are not mutually exclusive. If you are a system administrator or security professional, you should not only adhere to the defensive strategies associated with your job, but you should also encourage your in-house software development personnel and your vendors to follow the defenses for software developers. By covering both bases, you can help minimize the possibility of falling victim to this type of nasty attack.

Defenses for System Administrators and Security Personnel

What can a system administrator or security professional do to prevent stack-based buffer overflows and similar attacks? As mentioned at several points throughout this book, you must, at a minimum, keep your systems patched. The computer underground and security professionals are constantly discovering new vulnerabilities. Vendors are scrambling to create fixes for these holes. You must have a regular routine that monitors various mailing lists, such as the BugTraq, CERT, and SANS mailing lists. Most vendors also have their own mailing lists to distribute information about newly discovered vulnerabilities and their associated fixes to customers. You need to be on these lists for the vendors whose products you use in your environment.

In addition to monitoring mailing lists looking for new vulnerabilities, you also must institute a process for testing newly patched systems and rolling them into production. You cannot just apply a vendor's security fix to a production system without trying it out in a test environment first. A new security fix could impair other system operations, so you need to work things out in a test lab first. However, once you determine that the fix operates in a suitable fashion in your environment, you need to make sure it gets quickly deployed. Deploying fixes

in a timely manner is quite important before the script kiddie masses come knocking at your doors trying to exploit a vulnerability recently made public.

In addition to keeping your machines patched, make sure your publicly available systems (Internet mail, DNS, Web, and FTP servers, as well as firewall systems) have configurations with a minimum of unnecessary services and software extras. During system build and regular maintenance, you must remove extra junk from these critical systems. In particular, you should remove unneeded TFTP clients, FTP clients, and X Window system components. Do you really need X on a headless Internet Web server, or a TFTP client on your DNS server? Of course not. Leaving this software installed on those machines is asking for trouble.

Also, you need to strictly control *outgoing* traffic from your network. Most organizations are really careful about traffic coming into their network from the Internet. This is good, but it only addresses part of the problem. You will likely require some level of incoming access to your network, at least into your DMZ, so folks on the Internet can access your public Web server or send you email. If attackers discover a vulnerability that they can exploit over this incoming path, they may be able to use it to send an outgoing connection that gives them even greater access. This scenario is exactly what we saw with the X Window attack.

To avoid this problem, you need to apply strict filters on your firewalls to allow only outgoing traffic for services with a defined business need. Sure, your users may require outgoing HTTP or FTP. But do they really need outgoing X Window access? Probably not. You should block unneeded services at external firewalls and routers. Deny all services except those your users really need, such as outgoing HTTP traffic.

A final defense against stack-based buffer overflows that can be applied by system administrators and security personnel is to configure your system with a nonexecutable stack. If the system is configured to refuse to execute instructions from the stack, most stack-based buffer overflows just won't work. There are some techniques for getting around this type of defense, but the vast majority of stack-based buffer overflows will fail if they cannot execute instructions from the stack. While this solution doesn't apply to all systems, it can help for particularly sensitive machines running the Solaris, Linux, or Windows NT/2000 operating systems. To set up a Solaris system so that it will never execute instructions from the stack, add the following lines to the `/etc/system` file:

```
set noexec_user_stack=1
set noexec_user_stack_log=1
```

To configure a Linux system with a nonexecutable stack, you'll have to apply a kernel patch. Solar Designer, a brilliant individual whom we'll encounter again later in this chapter, has written a Linux kernel patch that includes a nonexecutable stack as well as other security features. His handiwork can be downloaded from *www.openwall.com/linux/README*.

For Windows NT machines, a tool called SecureStack is available from SecureWave that will prevent execution of code from the stack. The free version of SecureStack generates a warning message for the administrator when someone tries to run a program that executes code from the stack. The commercial version generates a warning message and prevents the program from executing the instructions from the stack. You can find both the free and commercial version of SecureStack at *www.securewave.com/products/securestack/secure_stack.html*.

Unfortunately, some legitimate programs actually require putting instructions on the stack for execution. These programs will not run properly if you configure the machine with a nonexecutable stack, so make sure to test your systems thoroughly before implementing this change.

Stack-Based Buffer Overflow Defenses for Software Developers

"An ounce of prevention is worth a pound of cure."
–Anonymous

While system administrators and security personnel can certainly do a lot to prevent stack-based buffer overflow attacks, the problem ultimately stems from sloppy programming. Software developers are the ones who can really stop this type of attack by avoiding programming mistakes involving the allocation of memory space and checking the size of all user input as it flows through their applications. Software developers must be trained to understand what buffer overflows are and how to avoid them. They should refrain from using functions with known problems, especially the weak string and memory functions cited earlier in this Chapter, instead using equivalent functions without the security vulnerabilities. The code review component of the software development cycle should include an explicit step to look for security-related mistakes, including buffer overflow problems.

To help this process, there are a variety of automated code-checking tools that search for known problems, such as the appearance of frequently misused functions that lead to buffer overflows like strcpy. A free tool called ITS4 (which stands for It's the Software, Stupid—Security Scanner) is available at *www.cigital.com/its4/*. Also, the folks at the L0pht have released SLINT, a commercial tool that includes similar source code security check capabilities at *www.l0pht.com/slint.html*.

A final defensive technique for software developers can be implemented while compiling programs, altering the way the stack functions. Two tools, StackGuard and Stack Shield, can be invoked at compile time for Linux programs to create stacks that are more difficult to attack with buffer overflows. You can find StackGuard at *immunix.org*, while Stack Shield is at *www.angelfire.com/sk/stackshield*.

StackGuard, available for Linux platforms for free, changes the stack by inserting an extra field called a "canary" next to the return pointer on the stack. The canary operates much like its namesake used by coal miners in the past. In a coal mine, if the canary died, the miner had a pretty good warning that there was a problem with the air in the tunnel. The miners would then evacuate the area. Similarly, if the canary on the stack gets altered, the system knows something has gone wrong with the stack, and will stop execution of the program, thereby foiling a buffer overflow attack.

Stack Shield, which is also free and runs on Linux, handles the problem in a slightly different way than StackGuard. Stack Shield stores return pointers for functions in various locations of memory outside of the stack. Because the return pointer is not on the stack, it cannot be overwritten by a buffer overflow.

Both Stack Shield and StackGuard offer significant protection against buffer overflows, and are definitely worth considering to prevent such attacks. However, they aren't infallable. Some techniques for creating buffer overflows on systems with StackGuard and Stack Shield were documented by Bulba and Kil3r in Phrack 56 at *phrack.infonexus.com/search.phtml?issueno=56&r=0*.

While none of the techniques discussed in this section for preventing buffer overflows is completely foolproof, they can, if applied together in a judicious manner, be used to minimize this common and nasty type of attack.

Password Attacks

Passwords are the most commonly used computer security tool in the world today. In many organizations, the lowly password often protects some of the most sensitive secrets imaginable, including healthcare information, confidential business strategies, sensitive financial data, and so on. Unfortunately, with this central role in security, easily guessed passwords are often the weakest link in the security of our systems. By simply guessing a single password, an attacker could gain access to very sensitive information or shut down critical computing systems.

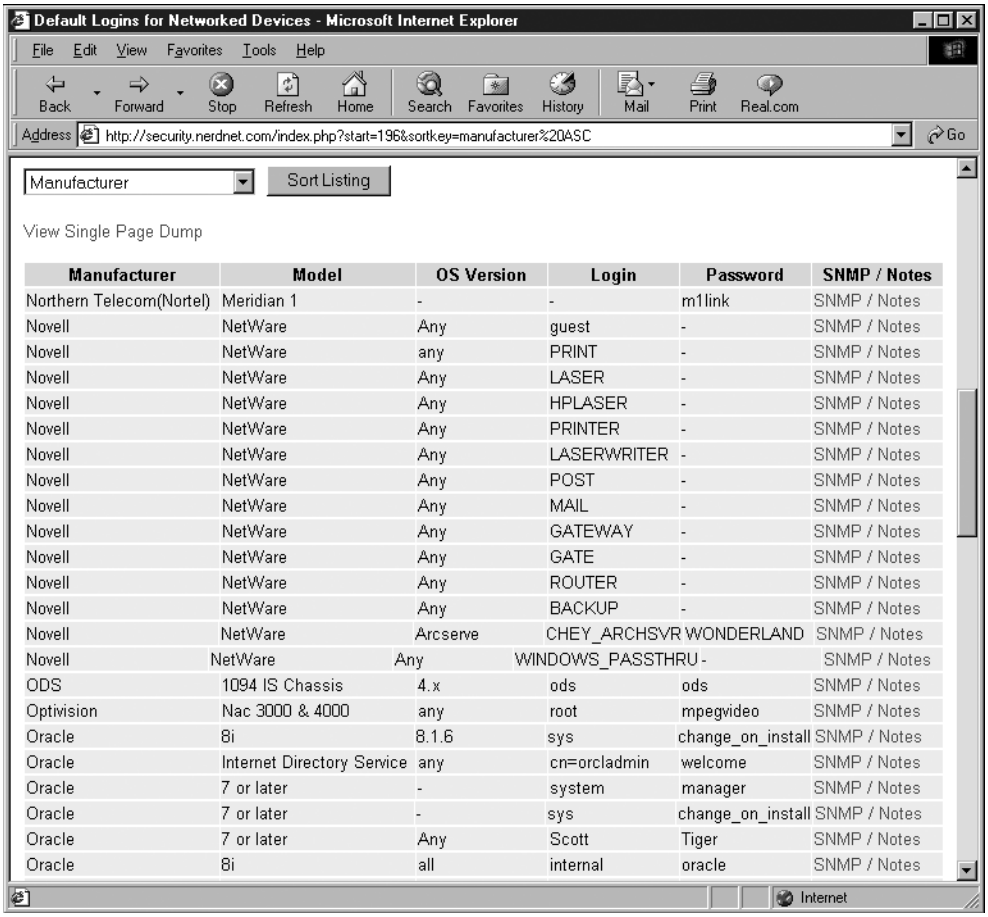
Compounding this problem with passwords is the fact that every user has at least one password, and most users have dozens of passwords. Users are forced to remember and maintain passwords for logging into the network, signing on for numerous applications, accessing frequently used external Web sites, logging into voice mail, and even for making long-distance calls with a calling card. On almost all systems, the users themselves choose the passwords, placing the burden of security on end users who either do not know or, sometimes, do not care about sound security practices. Users often choose passwords that are easy to remember, but are also very easily guessed. We frequently encounter passwords that are set to days of the week, the word "password," or simple dictionary terms. A single weak password for one user on one account could give an attacker a toehold on a system. Most users have the same password for every password-protected system they access allowing an attacker to quickly gain access to multiple systems. After guessing one weak password, the attacker can move to take over the rest of the system, using further password guessing or exploiting some other vulnerability to escalate privileges.

For even a low-skill attacker, guessing such passwords and gaining access can be quite trivial. Numerous freely available tools automatically guess passwords at extremely high rates, looking for a weak password to enter a system. Let's explore how these password guessing tools work.

Guessing Default Passwords

Many applications and operating systems include built-in default passwords established by the vendor. Oftentimes, overworked, uninformed, or lazy administrators fail to remove default passwords from systems. An attacker can quickly and easily guess these default passwords to try to gain access to the target. A huge database of default passwords for a

variety of platforms is maintained by Joe Jenkins and is publicly available at *security.nerdnet.com/*. This Web site, shown in Figure 7.8, includes default passwords for systems ranging from 3COM switches to Zyxel's modem-routers, and everything in between.



Manufacturer	Model	OS Version	Login	Password	SNMP / Notes
Northern Telecom(Nortel)	Meridian 1	-	-	m1link	SNMP / Notes
Novell	NetWare	Any	guest	-	SNMP / Notes
Novell	NetWare	any	PRINT	-	SNMP / Notes
Novell	NetWare	Any	LASER	-	SNMP / Notes
Novell	NetWare	Any	HPLASER	-	SNMP / Notes
Novell	NetWare	Any	PRINTER	-	SNMP / Notes
Novell	NetWare	Any	LASERWRITER	-	SNMP / Notes
Novell	NetWare	Any	POST	-	SNMP / Notes
Novell	NetWare	Any	MAIL	-	SNMP / Notes
Novell	NetWare	Any	GATEWAY	-	SNMP / Notes
Novell	NetWare	Any	GATE	-	SNMP / Notes
Novell	NetWare	Any	ROUTER	-	SNMP / Notes
Novell	NetWare	Any	BACKUP	-	SNMP / Notes
Novell	NetWare	Arcserve	CHEY_ARCHSVR	WONDERLAND	SNMP / Notes
Novell	NetWare	Any	WINDOWS_PASSTHRU	-	SNMP / Notes
ODS	1094 IS Chassis	4.x	ods	ods	SNMP / Notes
Optivision	Nac 3000 & 4000	any	root	mpegvideo	SNMP / Notes
Oracle	8i	8.1.6	sys	change_on_install	SNMP / Notes
Oracle	Internet Directory Service	any	cn=orcladmin	welcome	SNMP / Notes
Oracle	7 or later	-	system	manager	SNMP / Notes
Oracle	7 or later	-	sys	change_on_install	SNMP / Notes
Oracle	7 or later	Any	Scott	Tiger	SNMP / Notes
Oracle	8i	all	internal	oracle	SNMP / Notes

Figure 7.8
An online database of default passwords.

Password Guessing through Login Scripting

But what if none of the default passwords work? Another technique for guessing weak passwords is to simply write a script that runs on the attacker's machine and repeatedly tries to login to the target system across the network. The attacker will configure the script to guess a common or known userID. The script will also select a password guess, perhaps by using a dictionary. The attacker points the script to the target machine, which may have a command-line login prompt, Web front-end login dialogue box, or other method of requesting a password. The attacker's script will transmit its userID and password guess and then automatically determine if the guess was successful. If not, another guess is tried. Many attackers create their own scripts that attempt to login across the network. Others use the THC-Login Hacker tool (available at *thc.inferno.tusculum.edu*) that we discussed in the Chapter 6 section titled "War Dialing." Also, some canned tools have this login guessing capability, including:

- Authforce, by Zachary P. Landau, which attempts to guess passwords for basic HTTP authentication by logging into a Web server, available at *kapheine.hypa.net/authforce/index.php*.
- brute_ssl and brute_web by BeastMaster, which also guess passwords for HTTP and HTTPS authentication, available at *packetstorm.security.com/Exploit_Code_Archive/brute_ssl.c* and *packetstorm.security.com/Exploit_Code_Archive/brute_web.c*.
- A tool that remotely guesses Windows NT passwords, by Somarsoft, available at *packetstorm.securify.com/NT/audit/nt.remotely.crack.nt.passwords.zip*.
- Xavier, by LithiumSoft, a flexible tool that supports guessing plaintext passwords for a variety of applications, available at *www.btinternet.com/~lithiumsoft/*.
- Hypnopaedia, by NullString, a password guesser for email using the POP3 protocol, available at *packetstorm.securify.com/Crackers/hypno.zip*.

You can find many of these and dozens of other password-guessing tools at the Packet Storm Web site at *packetstorm.securify.com/Crackers/*.

Password guessing through login scripting can be a slow process. Each login attempt could take 5 or 10 seconds. To go through an entire 40,000-word dictionary could take many days, and guessing random combinations of characters could require weeks or months before a usable password is discovered. However, the greatest asset the attackers

have is time. They can be very determined when focused on a given target, and often don't mind spending many months trying to gain access.

Beyond being time consuming, there are additional limitations with this technique. The constant attempts to login to the target generate a significant amount of regular network traffic and log activity, which could easily be noticed by a system administrator or an intrusion detection system. An additional challenge an attacker faces when trying to guess a password through a scripting process is account lockout. Some systems are configured to disable a user account after a given number of incorrect login attempts with faulty passwords. The account is reenabled only by a user calling the help desk, or through an automated process after a period of time expires. Either way, the attacker's guessing can be detected or at least slowed down significantly. Account lockout features are a good idea in preventing password-guessing attacks through login scripting. However, with account lockout in place, an attacker could conduct a denial-of-service attack by locking out all of your accounts using a script.

The Art and Science of Password Cracking

Guessing default passwords usually doesn't work. At its best, password guessing through login scripting could take a very long time, while at its worst, it could get an attacker detected. A much more sophisticated approach to determining passwords that avoids these problems is known as *password cracking*. To analyze how password cracking works, you need to understand how passwords are stored on most systems.

When you login to a machine, whether it is a UNIX system, NT box, Novell server, Cisco router, or any other type of machine, you provide a userID and password to authenticate. The system has to check whether your authentication information is accurate to make the decision whether to log you in or not. It could base this decision by having a local file of the passwords for all users and comparing the password you just typed in with your password in the file. Unfortunately, a file with every user's password would be an incredible security liability. An attacker gaining access to such a password file would be able to login as any user of the system.

System designers, realizing the dilemma of requiring a list of passwords to compare for user login while not having a huge security hole, decided to solve the problem by applying cryptographic techniques to protect each password in the password file. Thus, the password file contains a list of userIDs and representations of the passwords that are

encrypted or hashed. I use the words "encrypted or hashed" because a variety of different cryptographic algorithms are applied. Some systems use pure encryption algorithms, like the Data Encryption Standard (DES), which require a key for the encryption. Others use hash algorithms, such as Message Digest 4 (MD4), which are one-way functions that transform data with or without a key. Either way, the password is altered using the crypto algorithm so that an attacker cannot determine the password by directly looking at its encrypted or hashed value in the password file.

When a user wants to login to the system, the system gathers the password, applies the same cryptographic transformation used to generate the password file, and compares the results. If the encrypted or hashed value of the password you typed matches the encrypted or hashed value in the file, you are allowed to login. Otherwise, you are denied access.

Let's Crack Those Passwords!

> *"Lather. Rinse thoroughly. Repeat."*
> —directions from a shampoo bottle, that, if followed literally, would leave you in the shower for eternity.

Most systems include a password file that contains encrypted or hashed passwords. Password cracking involves stealing the encrypted passwords and trying to recover the clear-text password using an automated tool. A password-cracking tool operates by setting up a simple loop, as shown in Figure 7.9.

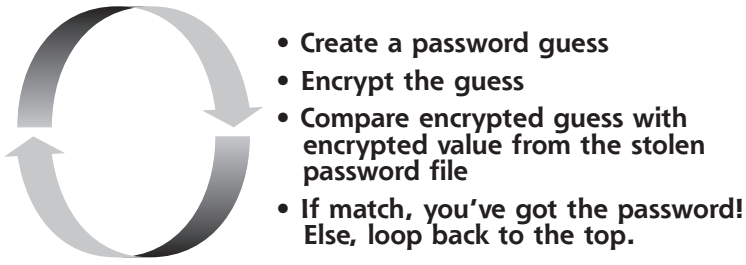


Figure 7.9
Password cracking is really just a loop.

A password-cracking tool can form its password guesses in a variety of ways. Perhaps the simplest method is to just throw the dictionary at the problem, guessing one term after another from a dictionary. A large number of dictionaries are available online, in many languages, including English, Russian, Japanese, French, and, for you *Star Trek* fans, even Klingon! Of course, if the target's passwords are not dictionary terms, this technique will fail. Happily for attackers, it almost always succeeds.

Beyond guessing dictionary terms, many password-cracking tools support brute-force cracking. For this type of attack, the tool will guess every possible combination of characters to determine the password. The tool may start with alphanumeric characters (a–z and 0–9), and then progress to special characters (!@#$, etc.) Even for a fast password-cracking tool, this brute-force guessing process can take an enormous amount of time, ranging from weeks to centuries. However, if the target password is short enough, this technique can retrieve it in a few weeks.

Hybrid password-cracking attacks are a nice compromise between quick but limited dictionary cracks and slow but effective brute-force cracks. In a hybrid attack, the password-cracking tool will start guessing passwords using a dictionary term. Then, it will create other guesses by appending or prepending characters to the dictionary term. By methodically adding characters to words in a brute-force fashion, these hybrid attacks are often extremely successful in determining a password.

From an attacker's perspective, password cracking is fantastic, because the cracking loop does not have to run on the victim machine. If the attackers can steal the encrypted/hashed password file, they can run the password cracking on their own systems in the comfort of their own homes or on any other machine that suits their fancy. This makes things much faster than password guessing through login scripting. While using a script to attempt a login across the network requires many valuable seconds to evaluate each guess, a password cracking tool can guess hundreds or even thousands of passwords a second! The password cracker only has to operate on the stolen password file stored locally, applying quick and optimized cryptographic algorithms. Every word in a 50,000-word dictionary can be attempted in only a minute.

Furthermore, the more CPU cycles the attackers throw at the problem, the more guesses they can make and the faster they can recover passwords. So, an attacker who has taken over dozens of machines throughout the world and is looking to crack the passwords of

a new victim can divide up the password-cracking task among all of these machines to set up a virtual password-cracking super computer.

Password-cracking tools have been around for over a decade, and an enormous number of them are available. Some of the most notable password-cracking tools include:

- L0phtCrack, an easy-to-use Windows NT/2000 password cracker by the folks at the L0pht, available at *www.l0pht.com/l0phtcrack/*.
- John the Ripper, a fantastic UNIX password cracker by Solar Designer, available at *www.openwall.com/john/*.
- Crack, by Alec Muffett, one the earliest really powerful UNIX password-cracking tools, which is still useful today, available at *www.users.dircon.co.uk/~crypto/*.
- Pandora, a tool for testing Novell Netware, including password cracking, written by Simple Nomad, and available at *www.nmrc.org/pandora/*.
- PalmCrack, a cool tool for cracking Windows NT and UNIX passwords that runs on the PalmOS PDA platform, by Noncon, Inc., available at *www.noncon.org/noncon/download.html*.

To understand how these tools work in more detail, let's explore two of the most powerful password crackers available today, L0phtCrack and John the Ripper.

Cracking Windows NT/2000 Passwords Using L0phtCrack

L0phtCrack is one of the most hyped security/attack tools of all time, and with good reason. It is trivially easy to use and blazingly fast in cracking passwords from Windows NT and 2000 machines. With its fancy GUI, the tool runs on Windows 9x, NT, and 2000 systems, and is available for a free trial period of 15 days. After that, you must pay $249 to the L0pht to run the tool.

Retrieving the Password Representations

To use L0phtCrack, the attacker must first get a copy of the encrypted/hashed password representations stored in the SAM database of the target machine. To accomplish this, L0phtCrack includes an integrated tool called "pwdump" for dumping Windows NT password representations from the local system or any other machine on the network. However, this built-in password dump capability requires administrator

privileges on the system with the target SAM database. Another alternative for getting these passwords is to use the Pwdump3 program, available at *www.ebiz-tech.com/pwdump3/*. This tool allows an attacker to dump passwords from a SAM database or a Windows 2000 Active Directory system. To use Pwdump3, the attacker must have administrative privileges on the target system.

Attackers also have many other options for getting a copy of the password representations. They could search the system looking for files used during a system backup and steal the password representations. For example, when a system is backed up, by default, a copy of the SAM database with the password representations is usually placed in the `%systemroot%\repair\sam._` file. This file is readable by everyone on the system.

Another option for getting the password representations is to steal the administrator recovery floppy disks. When a Windows NT system is built, a good administrative practice is to create floppy disks that can be used to recover the machine more quickly if the operating system gets corrupted. These floppy disks include a copy of the SAM database with at least a representation of the administrator's password. Alternatively, an attacker with physical access to the target machine could simply boot the system from a Linux or DOS floppy disk, and retrieve the SAM database located at `%systemroot%\system32\config`. Because DOS cannot natively read an NTFS partition, the attacker will have to use the NTFSDOS program available at *packetstorm.securify.com/NT/hack/ntfsdos.zip* to access the SAM database. A handy tool for retrieving and altering Windows NT and 2000 passwords using a Linux boot disk can be found at *home.eunet.no/~pnordahl/ntpasswd/bootdisk.html*.

L0phtCrack offers one final option for getting password representations: sniffing them off of the network. L0phtCrack includes a very powerful integrated network capture tool, SMB Packet Capture, that will monitor the LAN looking for Windows challenge/response authentication packets. Whenever users try to authenticate to a domain or mount a remote file share, their Windows machine will authenticate to the server using a challenge/response protocol. Taken together, the challenge and response are based cryptographically on the user's password. After grabbing the challenge/response from the network using its integrated sniffing tool, L0phtCrack can crack it to determine the users' password. We'll discuss sniffers in more detail in Chapter 8.

Configuring L0phtCrack

L0phtCrack is very easy to configure, as shown in Figure 7.10. The attacker can set up the tool to do dictionary attacks (using any wordlist as a dictionary, but L0phtCrack is distributed with an English dictionary with 50,000 words). L0phtCrack also supports hybrid attacks with a user-selectable number of brute-force characters to add to the dictionary terms. It also offers complete brute-force password cracking attacks, letting the user select a particular character set to use, including alphanumerics and special characters.

Additionally, L0phtCrack can be configured to crack either the LM representations or NT hashes retrieved from the target system. As described in Chapter 4, the LM representations are far weaker and can be cracked much more quickly than the NT hashes.

Figure 7.10
Configuration options for L0phtCrack.

Cracking Passwords

After loading the password representations, selecting a dictionary, and configuring the options, the attacker can run L0phtCrack by selecting the "Run Crack" option. L0phtCrack generates and tests guesses for passwords very quickly. The L0pht Web site includes some benchmark statistics for running L0phtCrack against LM representations, based on using a machine with quad-Xeon processors running at 400 MHz to crack the passwords. Certainly this is a speedy system, but not unattainable by today's standards. Using this machine, the L0pht obtained the following numbers for the attack against the LM password representations:

Character Set	Time
alpha-numeric	5.5 hours
alpha-numeric-some symbols	45 hours
alpha-numeric-all symbols	480 hours

That's pretty impressive performance! A full brute-force attack (every possible keystroke character) against the weak LM representations takes 480 hours, or 20 days, to recover *any* password, regardless of its value. And, if the attacker has more processing horsepower, the attack requires even less time. Of course, the NT hashes are more difficult to crack and require much more time.

The main L0phtCrack screen, illustrated in Figure 7.11, shows the information dumped from the target's SAM database (including User Name, LM representation, and NT Hash). While running, this screen displays a very useful status indicator, which shows what percentage of the configured attack the system has completed so far. Finally, as L0phtCrack runs, each successfully cracked password is highlighted in the display in real time as it is determined.

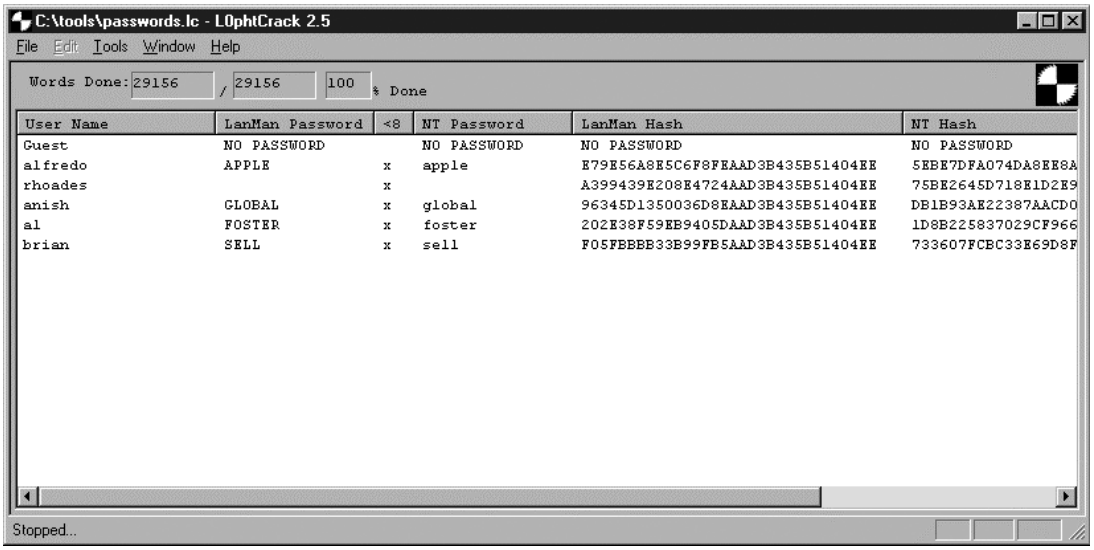


Figure 7.11
Successful crack using L0phtCrack.

Using L0phtCrack's Integrated Sniffer

As we discussed earlier, L0phtCrack allows an attacker to sniff challenge/response information off of the network for cracking. But how can an attacker force users to send this information across the network? Well, attackers could position their machine or take over a system on the network at a point where they will see all traffic for users authenticating to the domain or a very popular file server. In such a strategic position, whenever anyone authenticates to the domain or tries to access a share, the attacker can run L0phtCrack in sniffing mode to snag user authentication information from the network.

Of course, it may be very difficult for attackers to insert themselves in such a sensitive location. To get around this difficulty, the L0phtCrack FAQ suggests: "You just have to make the hashes come to you. Send out an email to your target, whether it is an individual or a whole company. Include in it a URL in the form of *file://yourcomputer/sharename/message.html.* When people click on that URL they will be sending their password hashes to you for authentication."

Consider the email shown in Figure 7.12, which was sent by an attacker, pretending to be the boss. Note that the message includes a link to a file share on the machine "SOMESERVER." On this machine, the attacker has installed L0phtCrack and is running the integrated sniffing tool.

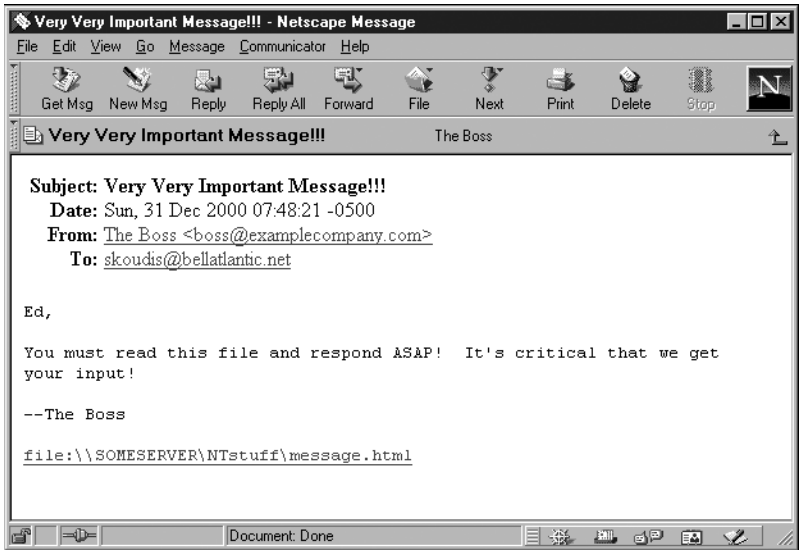


Figure 7.12
Would you trust this email?

When the victim clicks on the "file:\\" link, the victim's machine will attempt to mount the share on the attacker's server, interacting with the server using a challenge/response protocol. Once the victim clicks on the link, the attacker's sniffer will display the gathered challenge and response, as shown in Figure 7.13.

To complete the attack, the attacker can save this captured data and feed it into the main L0phtCrack tool to retrieve the user's password, as shown in Figure 7.14. This technique, which combines social engineering via email, sniffing data from the network, and password cracking, really demonstrates the power of L0phtCrack.

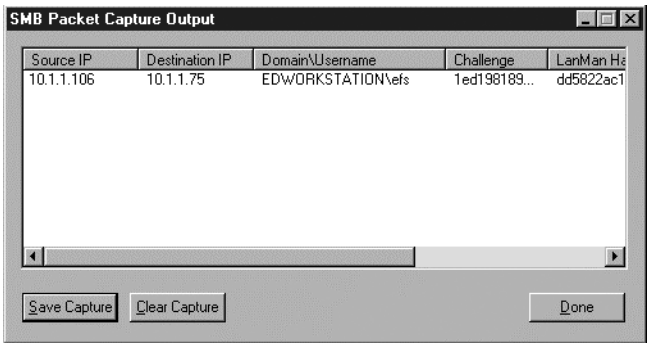


Figure 7.13
L0phtCrack's integrated sniffer captures the challenge/response from the network for cracking.

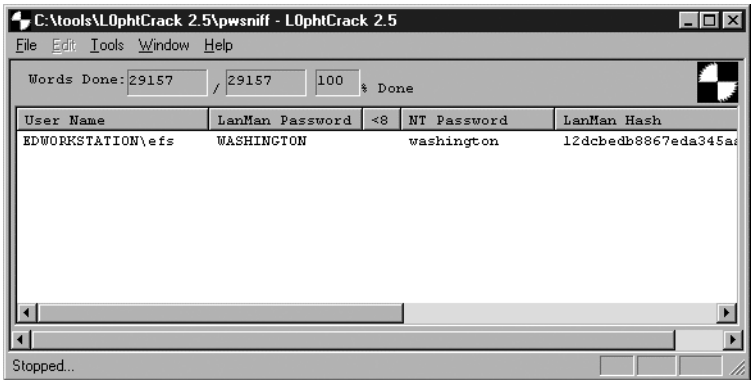


Figure 7.14
Successful crack of sniffed challenge/response.

Cracking UNIX (and Other) Passwords Using John the Ripper

L0phtCrack is certainly a powerful tool, but it focuses only on cracking Windows passwords. To crack passwords for other platforms, other tools are required. An extremely effective program named "John the Ripper" is one of the best password-cracking tools designed to determine UNIX passwords. John the Ripper (called "John" for short) is a free tool developed by Solar Designer, the chap we discussed in the last section who wrote the nonexecutable kernel patch for Linux to defend against stack-based buffer overflows.

John runs on a huge variety of platforms, including many variants of UNIX, DOS, Win9X, NT, and 2000 systems. To boost its speed, John even includes optimized code to take advantage of advanced CPU capabilities, such as Intel's MMX™ technology and specific features of AMD's K6® processor. Such capabilities are pretty impressive, given how few commercial programs actually support those processor features.

Further showing its great flexibility, John can be used to crack passwords from a variety of UNIX variants, including Linux, FreeBSD, OpenBSD, Solaris, Digital UNIX, AIX, HP-UX, and IRIX. Although it was designed to crack UNIX passwords, John can also attack NT hashes from a Windows NT machine. Also, Dug Song, the author of the FragRouter IDS evasion tool that we discussed in Chapter 6, has written modular extensions for John that will crack files associated with the S/Key one-time-password system and AFS/Kerberos Ticket Granting Tickets, which are used for cryptographic authentication.

Retrieving the Encrypted Passwords

As described in Chapter 3, UNIX systems store password information in the `/etc` directory. Older UNIX systems store encrypted passwords in the `/etc/passwd` file, which can be read by any user with an account on the system. For these types of machines, an attacker can grab the encrypted passwords very easily, just by copying `/etc/passwd` using an account on the machine or a buffer overflow to snag the password file.

Most modern UNIX variants include an option for using shadow passwords. In such systems, the `/etc/passwd` file still contains general user information, but all encrypted passwords are moved into another file, usually named `/etc/shadow` or `/etc/secure`. Figure 7.15 shows the `/etc/passwd` file from a system configured to use shadow pass-

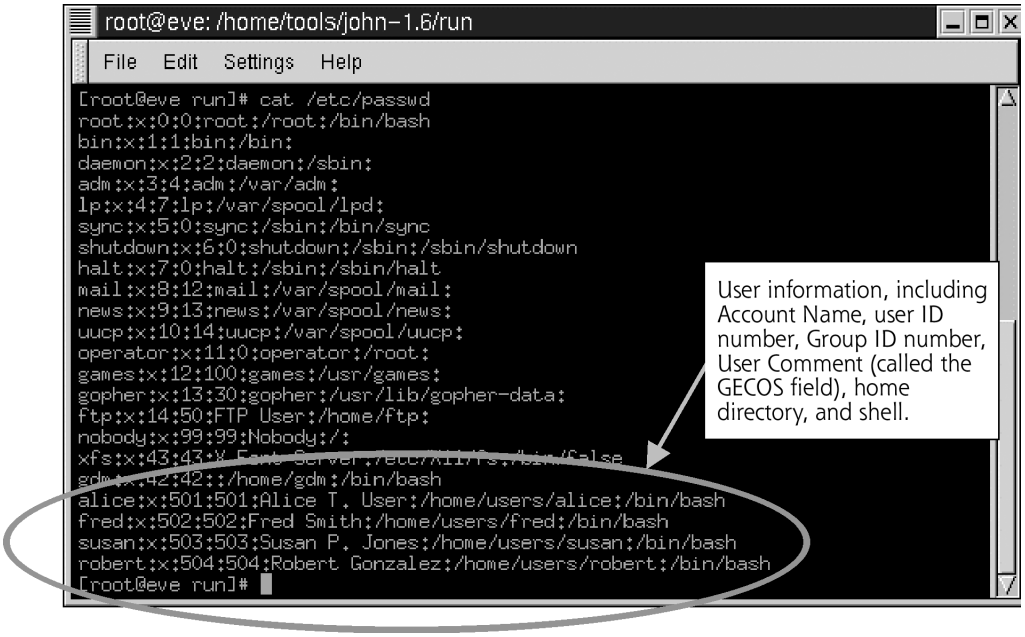


Figure 7.15
When password shadowing is used, the `/etc/passwd` file contains no passwords.

words. Figure 7.16 shows the corresponding `/etc/shadow` file. A shadow password file (`/etc/shadow` or `/etc/secure`) is only readable by users with root-level privileges. To grab a copy of a shadow password file, an attacker must find a root-level exploit, such as a stack-based buffer overflow of an SUID root program or related technique, to gain root access. After achieving root-level access, the attacker will make a copy of the shadow password file.

Another popular technique used on systems with or without shadow passwords involves causing a process that reads the encrypted password file to crash, generating a core dump file. On UNIX machines, the operating system will often write a core file containing a memory dump of a dying process (for debugging purposes and to store unsaved data). After retrieving a copy of a core file from a process that read the encrypted passwords before it died, the attacker can comb through it to look for the encrypted passwords. This technique for mining core dumps is particularly popular in attacking FTP servers. If attackers can crash one instance of the FTP server, causing it to create a core dump, they can then use another instance of the FTP server to transfer the core file

Figure 7.16
The corresponding `/etc/shadow` file contains the encrypted passwords.

from the target machine. They'll then pour through the core file looking for passwords to crack to gain access to the FTP server.

Configuring John the Ripper

John is trivially easy to configure. The attacker must feed John with a file that includes all user account and password information. On a UNIX system without shadow passwords, all of this information is available in the `/etc/passwd` file itself, so that's all John requires. On a system with shadow passwords, this information is stored in `/etc/passwd` and `/etc/shadow` (or `/etc/secure`). To merge these two files into a single file for input, John includes a program called "unshadow," which is shown in Figure 7.17.

Another very nice feature of John is its ability to automatically detect the particular encryption algorithm for the target UNIX system variety to use during a crack. In this way, the tool practically automatically configures itself. Although the autodetect function is nifty, the absolute greatest strength of John is its ability to quickly create many

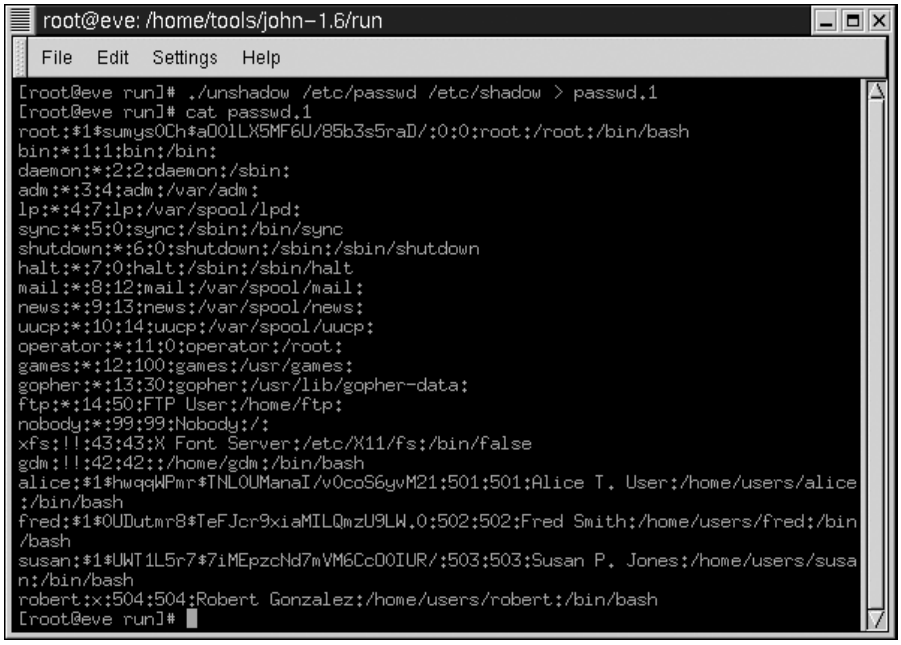


Figure 7.17
Running the unshadow program from John the Ripper.

permutations for password guesses based on a single word list. Using a word list in a hybrid-style attack, John will append and prepend characters, and attempt dictionary words forward, backward, and typed in twice. It will even truncate dictionary terms and append/prepend characters to the resulting strings. This capability lets the tool create many combinations of password guesses, foiling most users' attempts to create strong passwords by slightly modifying dictionary terms.

With all of this slicing and dicing of words to create password guesses, John acts like a dictionary vegematic. The process of creating permutations for password guesses is defined in a user-configurable rule set. The default rules that John ships with are exceptionally good, and most users won't have to tinker with the rules.

When conducting a password-cracking attack, John supports several different modes of operation, including:

- *Wordlist Mode:* As its name implies, this mode guesses passwords based on a dictionary, creating numerous permutations of the words using the rule set.

- *"Single Crack" Mode:* This mode is the fastest and most limited mode supported by John. It bases its guesses only on information from the user account, including the login name, GECOS field, and so on.
- *Incremental Mode:* This is John's mode for implementing a complete brute-force attack, trying all possible character combinations as password guesses. A brilliant feature of this mode is to use character frequency tables to ensure the most widely used characters (such as "e" in English) have a heavier weighting in the guessing.
- *External Mode:* You can create custom functions to generate guesses using this external mode.

By default, John starts using Single Crack mode, moves onto Wordlist mode, and finally, tries Incremental mode.

Even in the face of all of this flexibility, John's default values are well tuned for most password-cracking attacks. By simply executing the John program and feeding it an unshadowed password file, the attacker can quickly and easily crack passwords, as shown in Figure 7.18. While John is running, it displays successfully cracked passwords on the screen, and stores them in a local file. Also while John is running, the attacker can press any key on the keyboard to get a one-line status check, which displays the amount of time John has been running, the

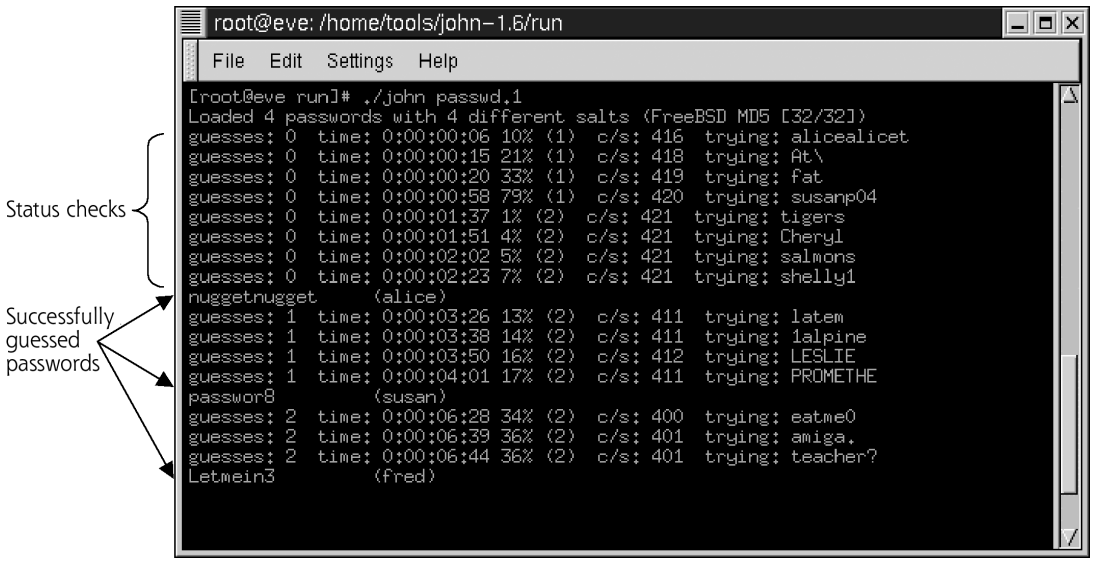


Figure 7.18
Running John the Ripper to crack passwords.

percentage of the current mode that is completed, as well as the current password guess John has just created.

Defenses against Password-Cracking Attacks

L0phtCrack and John the Ripper represent the best of breed password-cracking tools, and can quickly determine passwords in most environments. In my experience at numerous organizations, L0phtCrack or John often return dozens of passwords after running for a couple of minutes. Given the obvious power of these tools, together with the widespread use of passwords as security tools, how can we successfully defend our systems? To defend against password-cracking attacks, you must make sure your users do not select passwords that can be easily guessed by an automated tool. You must employ several defensive techniques that work together to help eliminate weak passwords, starting by establishing an effective password policy.

Strong Password Policy

A strong password policy is a crucial element in ensuring the security of your systems. Your organization must have an explicit policy regarding passwords, specifying a minimum length and prohibiting the use of dictionary terms. Passwords should be at least nine characters long, and should be required to include nonalphanumeric characters. Furthermore, passwords should have a defined maximum lifetime of 90, 60, or 30 days, depending on the particular security sensitivity and culture of your organizations. I tend to recommend a 60- or 90-day policy, because, in my experience, users nearly always write down passwords that expire every 30 days on Post-it™ notes. Of course, your culture may vary. Finally, make sure that your password policy is readily accessible by employees on an internal network Web site and through employee orientation guides.

User Awareness

To comply with your password policy, users must be aware of the security issues associated with weak passwords and trained to create memorable, yet difficult-to-guess passwords. A security awareness program covering the use of passwords is very important. Such a program could take several forms, ranging from posters in the work place to explicit training for users in how to create good passwords and protect them.

In your password awareness program (as well as your password policy), tell users how to create good, difficult-to-guess passwords. You

should recommend that they use the first letters of each word from a memorable phrase, mixing in numbers and special characters. When training users in selecting good passwords, I like to use an example from the theme song from the television show *Gilligan's Island*: "Just sit right back, and you'll hear a tale." A password derived from this phrase would be "Jsrb,Ayhat." As you may recall, there were seven stars from the TV program, so, we can add that information to the password, coming up with "Jsrb,Ayhat7*", which would be reasonably difficult to guess, as it contains alphabetic and numeric characters, mixed cases, and special characters. Using the same technique, your users should be able to create their own memorable passwords. Of course, if you use this example from *Gilligan's Island*, make sure to warn your users not to set their password to the example "`Jsrb,Ayhat7*`." If you don't warn them, a large number of them will just use the password from your example!

Password-Filtering Software

To help make sure users do not select weak passwords, you can use password-filtering tools that prevent them from setting their passwords to easily guessed values. When a user establishes a new account or changes their password on a system, these filtering programs check the password to make sure that it meets your organization's password policy (i.e., the password is sufficiently complex and is not just a variation of the user name or a dictionary word). With this kind of tool, users are simply unable to create passwords that violate your password policy rules. However, by being creative enough, some users will be able to come up with something that gets through the password filter yet is still easily crackable. Still, the vast majority of your user population will have strong passwords, significantly improving the security of your organization.

For filtering software to be effective, it must be installed on all servers where users establish passwords, including UNIX servers, Windows NT primary domain controllers, and other systems. Many modern variants of UNIX include built-in password-filtering software. For those that do not, you can use a variety of third-party tools to add this capability, including:

- Npasswd, at *ftp.cc.utexas.edu/pub/npasswd*
- Passwd+, available at *ftp.dartmouth.edu/pub/security*

For Windows NT environments, you can select from numerous password-filtering tools as well, including:

- Passprop, a tool from Microsoft available on the Windows NT Resource Kit Server Supplement 4.
- Passfilt.dll, a simple password-filtering tool included in Service Pack 2.
- Password Guardian, available at *www.georgiasoftworks.com*
- Strongpass, available at *ntsecurity.nu/toolbox/*
- Fast Lane, available at *www.fastlanetech.com*

Where Possible, Use Authentication Tools Other Than Passwords

Of course, one of the main reasons we have this password-cracking problem in the first place is our excessive use of traditional reusable passwords. If you get rid of access through passwords, you deal a significant blow to attackers trying to utilize password-cracking programs. For particularly sensitive systems and/or authentication across untrusted networks, you should avoid using traditional password authentication. Instead, consider one-time password tokens or smart cards for access.

Conduct Your Own Regular Password-Cracking Tests

To make sure your users are selecting difficult-to-guess passwords and to find weak passwords before an attacker does, you should conduct your own periodic password-cracking assessments. Using a high-quality password-cracking tool, like L0phtCrack or John the Ripper, check for crackable passwords every month or every quarter. As always, avoid using programs from untrusted sources. While your own organization policies may differ, I personally trust L0phtCrack and John the Ripper. L0phtCrack is a commercial tool that has been in widespread use for years without any concerns. John ships with source code, so it can be reviewed for any security compromises.

Before conducting this type of assessment, make sure you have explicit permission from management. Otherwise, you may damage your career path by cracking the password of some very cranky employees, possibly in senior management positions. When weak passwords are discovered, make sure you have clearly defined, management-approved procedures for interacting with users whose passwords can be easily guessed.

Protect Your Encrypted/Hashed Password Files

A final very important technique for defending against password-cracking tools is to protect your encrypted/hashed passwords. If the attackers

cannot steal your password file or SAM database, they will not be able to crack your passwords en masse. You must carefully protect all system backups that include password files (or any other sensitive data, for that matter). Such backups must be stored in locked facilities and possibly encrypted. Similarly, lock up any system recovery floppy disks in a safe location.

On all of your UNIX systems, make sure that you activate password shadowing. On Windows NT and 2000 systems, apply the SYSKEY tool from Microsoft to provide a modicum of extra protection for passwords at all domain controllers, as described by Microsoft at *support.microsoft.com/support/kb/articles/Q143/4/75.ASP*. Furthermore, if you do not have to support Windows for Workgroups or Windows 95/98 clients, disable the incredibly weak LM authentication. In an environment that includes only Windows NT and 2000 machines, you can get rid of the weak LM representations by applying Microsoft's LM-Fix, described at *www.microsoft.com/technet/support/kb.asp?ID=147706*. Finally, when you make a backup, delete or alter the permissions on the copy of the SAM database stored in the `%systemroot%\repair\sam._` file. Using these techniques, you can significantly lower the chances of an attacker grabbing your password hashes.

Web Application Attacks

Now that we understand how the frequently exploited buffer overflow and password cracking attacks operate, let's turn our attention to a class of attacks that is rapidly growing in prominence: World Wide Web application exploits. More and more organizations are placing applications on the Internet for all kinds of services, including electronic commerce, trading, information retrieval, voting, government services, and so on. New applications are being built with native Web support, and legacy applications are being upgraded with fancy new Web front-ends. As we Webify our world, the World Wide Web has proven to be a particularly fruitful area for attackers to exploit.

All of the attack techniques we've discussed throughout this book apply to Web-based systems. However, there are several additional techniques that have particular relevance in Web applications. In particular, in my investigations of a large number of Web sites, I have frequently encountered Web applications that are subject to account harvesting, undermining session-tracking mechanisms, and SQL piggybacking. The concepts behind these vulnerabilities are not inherently

Web specific, as these same problems have plagued all kinds of applications for decades. However, because Web applications seem particularly prone to these types of errors, it is important to understand them and defend against these attacks.

All of the Web attack techniques described in this section can be conducted even if the Web server uses the Secure Sockets Layer (SSL) protocol. So often I hear someone say, "Sure, our Web application is secure… we use SSL!" SSL can indeed help by strongly authenticating the Web server to the browser and preventing an attacker from intercepting traffic, when it is used properly. It can even be used to authenticate clients if you deploy client-side certificates. You should definitely employ SSL to protect your Web application. However, SSL doesn't do the whole job of protecting a Web application. There are still a large number of attacks that function perfectly well over an SSL-encrypted connection. We will discuss several such techniques in this section.

Account Harvesting

Account harvesting is a good example of a technique that has been applied to all kinds of systems for decades, but now seems to be a particular problem with Web applications. Using this technique, an attacker can determine legitimate userIDs and even passwords of a vulnerable application. Account harvesting is really a simple concept, targeting the authentication process when an application requests a userID and password. The technique works against applications that have a different error message for users who type in an incorrect userID than for users who type an incorrect password.

Consider the error message screens for the application shown in Figures 7.19 and 7.20. These screens are from a proprietary Web application called Mock Bank, written by Arion Lawrence, a brilliant colleague of mine who has developed several interesting security testing tools. Our company uses Mock Bank internally to show common real-world problems with online applications, as well as to train new employees in the methods of ethical hacking. Figure 7.19 shows what happens when a user types in a wrong userID, while Figure 7.20 shows the output from a correct userID and an incorrect password. The actual HTML and appearance in the browser of both pages are identical. However, look at the location line in the browser a bit more closely. Notice that when the userID is incorrect, error number 1 is returned. When the userID is valid and the password is wrong, error number 2 is returned. This discrepancy is exactly what an attacker looks for when harvesting accounts.

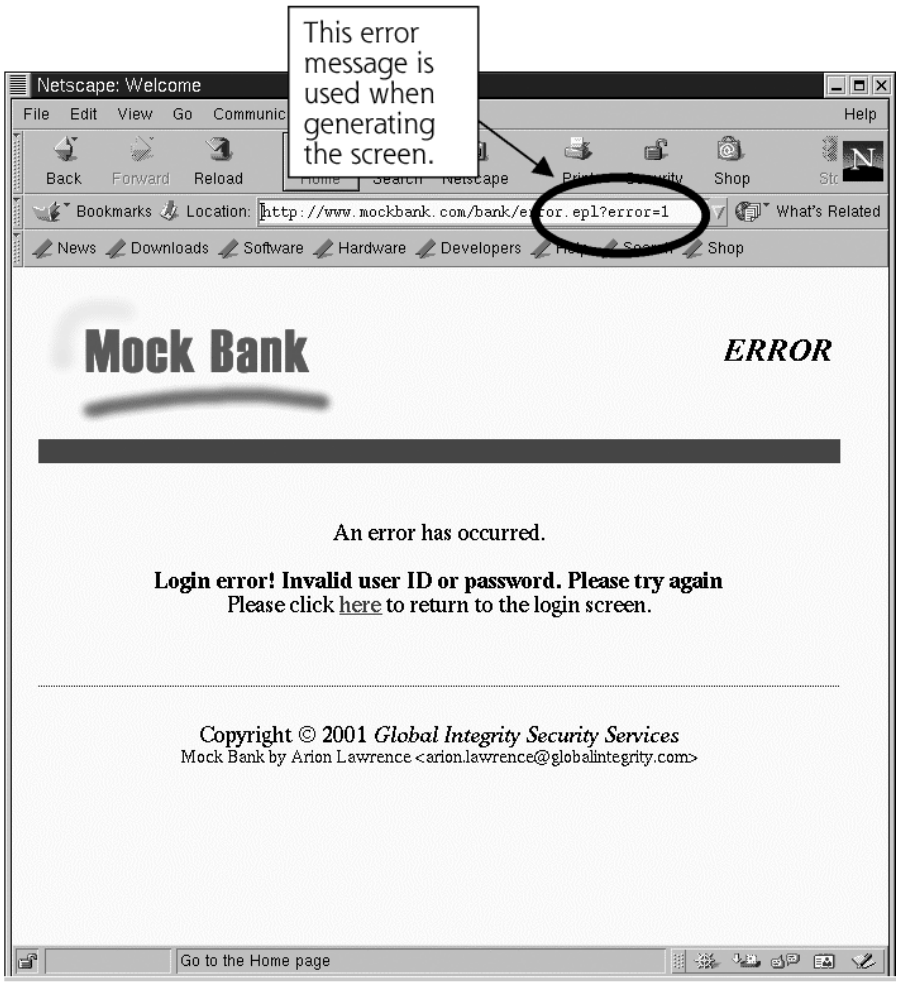


Figure 7.19
Mock Bank's error message when a user types an invalid userID.

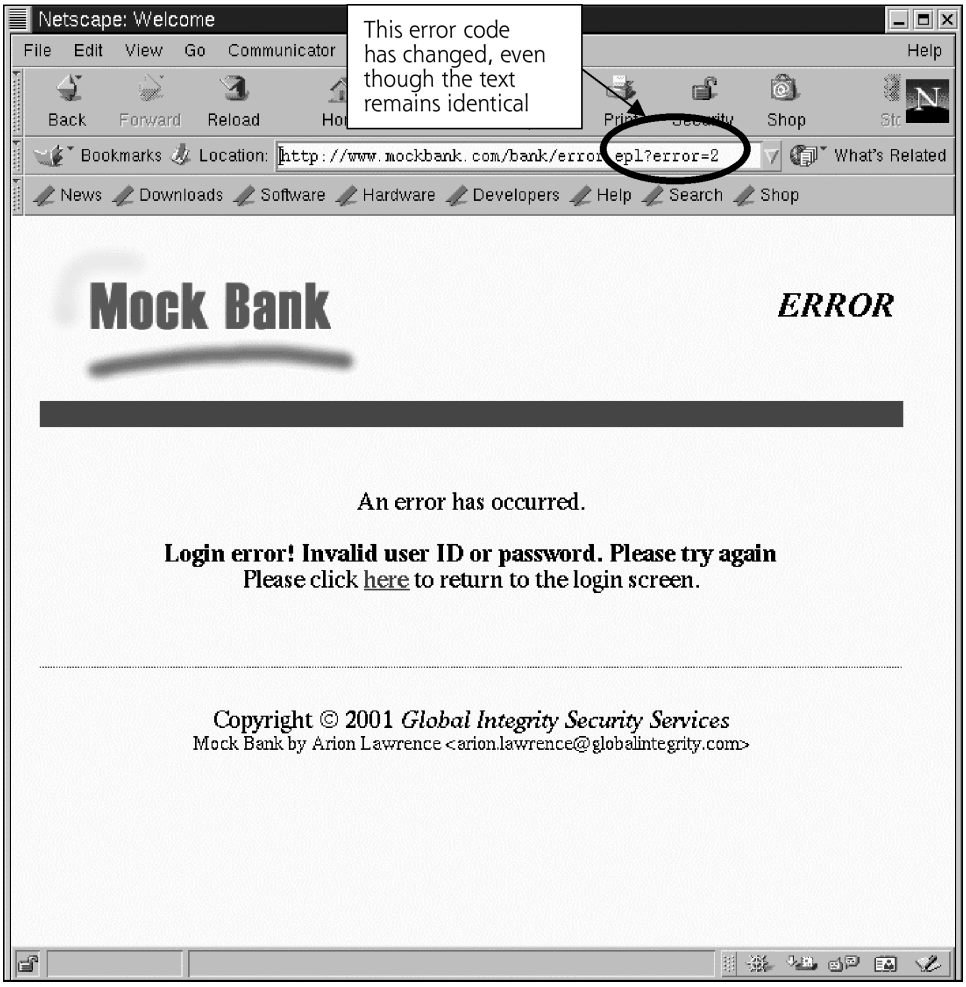


Figure 7.20
Mock Bank's error message when a user types a valid userID, but the wrong password.

Based on this difference in error messages, an attacker will write a custom script to interact with the Web application across the network, conducting a dictionary or brute-force attack guessing all possible userIDs, and using an obviously false password (such as "z"). The script will try each possible userID. If an error message is returned indicating that the userID is valid, the attacker will write the userID to a file. Otherwise, the next guess is tested. This is pure userID guessing through scripting, adding a bit of intelligence to discriminate between invalid

and valid userIDs. In this way, an attacker can harvest a large number of valid userIDs from the target application.

Next, the attacker can try to harvest passwords. If the target application doesn't lock out user accounts due to a given number of invalid password attempts, the attacker can write another script to try password guessing across the network. The attacker will take the userIDs previously harvested and try guessing all passwords for those accounts using login scripting. If the target application does lock out accounts, the attacker can easily conduct a denial-of-service attack using the harvested userID information.

Account Harvesting Defenses

For all of your Web applications (or any other application, for that matter), you must make sure that you use a consistent error message when a user types in an incorrect userID or password. Rather than telling the user, "Your userID was incorrect," or "Your password was incorrect," your application should contain a single error message for improper authentication information. You could display a message saying, "Your userID or password were incorrect. Please enter them again, or call the help desk." Note that all accompanying information sent back to the browser must be completely consistent for the two scenarios, including the raw HTML, URL displayed in the browser, cookies, and any hidden form elements.

Undermining Web Application Session Tracking

Another technique commonly used to attack Web applications deals with undermining the mechanisms used by the Web application to track user actions. After a user authenticates to a Web application (by providing a userID and password, or through a client-side certificate on an HTTPS session), most Web applications generate a session ID to track the user's session. The Web application generates a session ID and passes it to the client's browser, essentially saying, "Here, hold this now and give it back to me every time you send another request for the rest of this session." This session ID is passed back and forth across the HTTP or HTTPS connection for all subsequent interactions that are part of the session, such as browsing Web pages, entering data into forms, or conducting transactions. The application uses this information to track who is submitting the request. In essence, the session ID allows the Web application to maintain the state of a session with a user.

Note that a session ID can have any name the application developer assigns to it. It does not have to be called "sessionID," or "sid," or anything else in particular. A Web application developer could call the variable "Joe," but it would still be used to track the user through a series of interactions.

Furthermore, a session ID is completely independent of the SSL connection. The session ID is Application-level data, generated by the application and exchanged by the Web browser and Web server. While it is encrypted by SSL as it moves across the network, the session ID can be altered without impacting the underlying SSL connection.

Implementing Session IDs in Web Applications

How do Web applications implement session IDs? Three of the most popular techniques for transmitting session IDs are URL session tracking, hidden form elements, and cookies. For URL session tracking, the session ID is written right on the browser's location line, as shown in Figure 7.21. For all subsequent Web requests, the URL will be passed back to the server, which can read the session ID from this HTTP field, and determine who submitted the request.

A second technique for tracking session IDs involves putting the session ID information into the HTML itself, using hidden form elements. Using this technique, the Web application sends the browser an HTML form with elements that are labeled as hidden. One of these form elements includes the session ID. When it displays the Web page, the browser will not show the user these hidden elements, but the user can readily see them simply by invoking the browser's "view source"

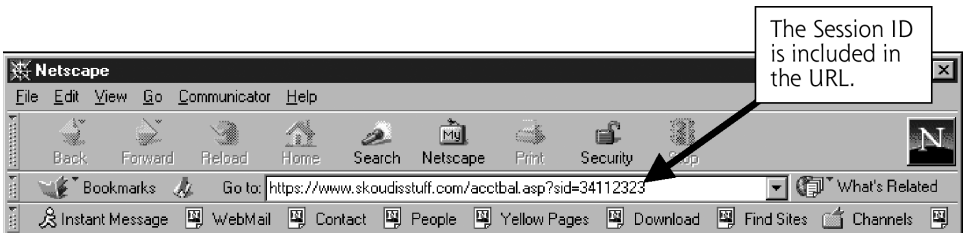


Figure 7.21
Session tracking using the URL.

function for the page. In the raw HTML, a hidden form element will have the following appearance:

```
<INPUT TYPE="HIDDEN" NAME="Session" VALUE="22343">
```

Cookies are the most widely used session-tracking mechanisms. A cookie is simply an HTTP field that the browser stores on behalf of a Web server. A cookie contains whatever data the server wants to put into it, which could include user preferences, reference data, or a session ID. There are two types of cookies: per-session cookies and persistent cookies. A per-session cookie is stored in the browser's memory and is deleted when the browser is closed. This type of cookie has a short but useful life, and is often used to implement session IDs. A persistent cookie, on the other hand, is written to the local file system when the browser is closed, and will be read the next time the browser is executed. Persistent cookies, therefore, are most often used to store long-term user preferences.

Attacking Session Tracking Mechanisms

Many Web-based applications have vulnerabilities in properly allocating and controlling these session IDs. An attacker may be able to establish a session, get assigned a session ID, and alter the session ID in real time. For applications that don't handle session tracking properly, if the attacker changes the session ID to a value currently assigned to another user, the application will think the attacker's session belongs to that other user! In this way, the attacker usurps the legitimate user's session ID. As far as the application is concerned, the attacker *becomes* the other user. Of course, both the legitimate user and the attacker are using the same session ID at the same time. Still, many Web-based applications won't even notice this problem, accepting and processing transactions from both the attacker and the legitimate user.

An application with this vulnerability will allow an attacker to do anything a legitimate user can do. In an online banking application, the attacker could transfer funds or possibly write online checks. For online trading, the attacker could make trades on behalf of the user. For an online health care application…well, you get the idea.

An attacker first needs to determine another user's session ID. To accomplish this, the attacker will login to the application using an account assigned to the attacker, and observe the session ID assigned to that session. The attacker will look at how long the session ID is and the types of characters (numeric, alphabetic, or others) that make it up. The

attacker will then write a script to login again and again, gathering hundreds of session IDs to determine how they change over time. Then, applying some statistical analysis to the sampled session IDs, the attacker will attempt to predict future session IDs that belong to other users.

How does an attacker actually manipulate the session ID? First, the attacker will login to the application using his/her own account to be assigned a session ID. Then, the attacker will attempt to modify this session ID to take over the session of another user. For many session-tracking mechanisms, such modifications are trivial. With URL session tracking, the attacker simply types over the session ID in the URL line of the browser. If hidden form elements are used to track sessions, the attacker will save the Web page sent by the server to the local file system. The attacker will then edit the session ID in the hidden form elements of the local copy of the Web page, and reload the local page into the browser. By simply submitting the form back to the server, the attacker will send the new session ID and could become another user.

If sessions are tracked using persistent cookies, the attacker can simply edit the local cookie file. In Netscape browsers, all persistent cookies are stored in a single file called "`cookies.txt`," as shown in Figure 7.22. For Internet Explorer, cookies from different servers are stored in their own individual files in the "Cookies" directory. Despite the dire warning at the top of Netscape's cookie file, an attacker can edit these persistent cookies using any text editor. To exploit a session ID based on a persistent cookie, the attacker will log into the application to get a session ID, close their browser to write the cookie file, edit the cookies using their favorite text editor, and relaunch the browser, now

Figure 7.22
Editing nonpersistent cookies to modify a session ID using notepad.

using the new session ID. The browser must be closed and relaunched during this process because persistent cookies are only written and read when the browser is closed and launched.

Editing persistent cookies is trivial. But how can an attacker edit per-session cookies, which are stored in the browser's memory and are not written to the local file system? Many Web application developers just assume that a user cannot view or alter per-session cookies, so they don't bother worrying about protecting the information stored in them. They think that just because the per-session data is encrypted with SSL and is never written to the hard drive, it cannot be edited. Unfortunately, there are techniques for altering per-session cookies, and a good deal of active work is being conducted in the computer underground in this area.

Achilles is one of the best tools for editing per-session cookies (or any HTTP field, for that matter). Written by the DigiZen Security Group and released in October 2000, Achilles is available at *www.digizen-security.com.* As illustrated in Figure 7.23, Achilles is actually a Web proxy. Remember, the attacker cannot directly edit per-session cookies in the browser's memory. However, a proxy sitting between the browser and the server can edit these cookies easily by grabbing onto them in the raw communication stream between browser and server. The attacker will configure a browser to send all HTTP and HTTPS data to and from the target Web server through Achilles. Achilles will let the attacker edit the raw HTTP/HTTPS fields and HTML information including per-session and persistent cookies, hidden form elements, URLs, frame definitions, and so on.

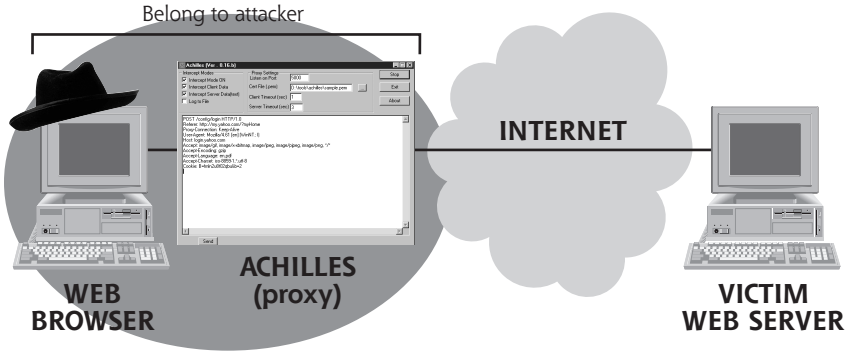


Figure 7.23
Achilles is used to proxy connections for the attacker.

The attacker runs the browser and the Achilles proxy, either on separate systems or on a single machine. Figure 7.24 shows the simple yet powerful Achilles interface. In the main window of the GUI, all information from the HTTP or HTTPS session is displayed. When the browser or server sends data, Achilles intercepts it, allowing it to be edited before passing it on. In this way, Achilles pauses the browsing session, giving the attacker a chance to alter it. The attacker can simply point to and click on any information in this session in the main window and type right over it. The attacker then hits the "Send" button, which transfers the data from Achilles to the server or browser.

Achilles also supports HTTPS connections, which are really just HTTP connections protected using SSL. To accomplish this, as displayed in Figure 7.25, Achilles sets up two SSL connections: one session between the browser and Achilles, and the other between Achilles and the Web server. Achilles even comes with a built-in digital certificate to establish the connection with the Web browser. The Web server never knows that there is a proxy in the connection. The attacker's browser

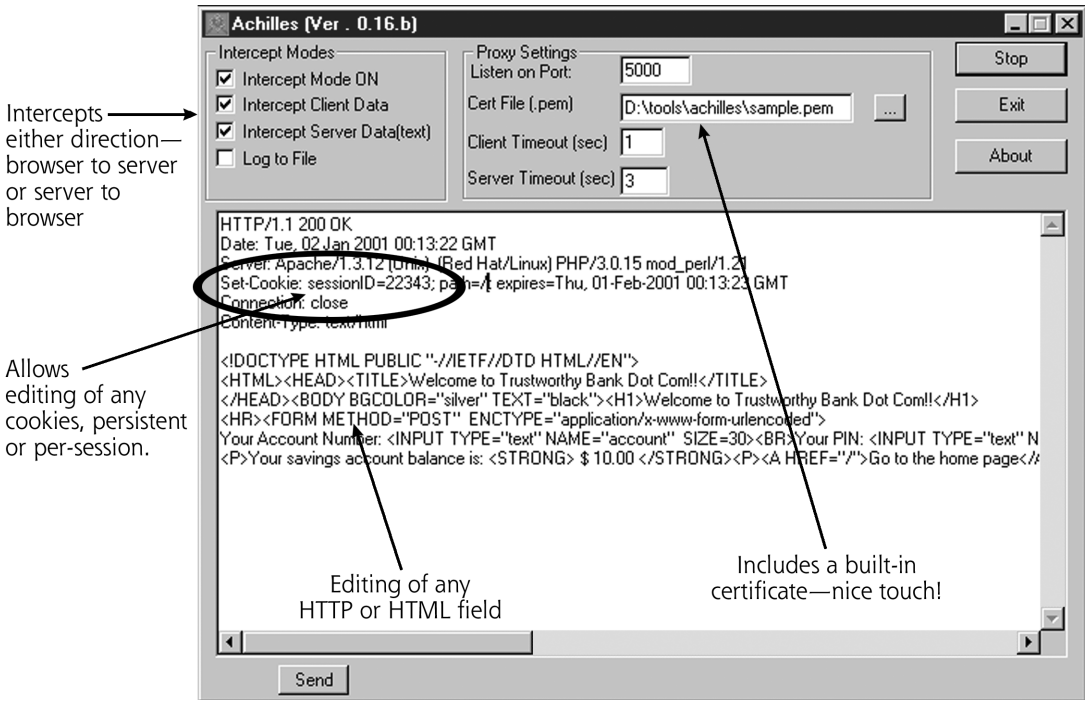


Figure 7.24
The Achilles screen.

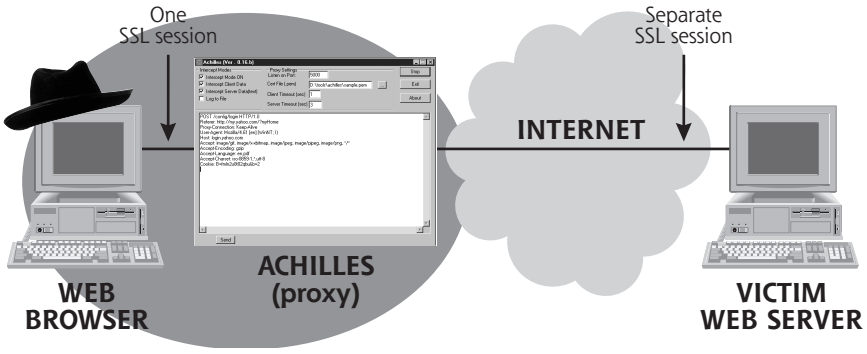


Figure 7.25
Handling HTTPS with Achilles.

will display a warning message saying that the certificate from the server isn't signed by a trusted certificate authority. However, the attacker is running both the browser and Achilles, so the warning message can be ignored.

Defending against Web Application Session-Tracking Attacks

To defend your Web applications from this type of attack, you must ensure the integrity of all session-tracking elements, whether they are implemented using URLs, hidden form elements, or cookies. To accomplish this, use the following techniques for creating your session-tracking elements:

- Digitally sign or hash session-tracking information using a cryptographic algorithm
- Encrypt the information in the URL, hidden form element, or cookie; don't just rely on SSL
- Make sure your session IDs are long enough to prevent accidental collision (at least 10 characters are recommended)
- Consider making your session IDs dynamic, changing from page to page throughout your Web application
- Apply a timestamp within the session ID variable and encrypt it

Going beyond session IDs, you should use these same techniques to protect *any* information sent to the browser that you do not want a user to see or alter. It is extremely important to understand that unless you protect the data sent to the browser, an attacker will be able to access it and even alter it. Some Web applications send pricing or other

information to a browser in a cookie, and then trust that data when the browser sends it back. Using Achilles, an attacker can alter all data sent to the browser.

When applying these mechanisms to secure the variables passed to the browser, you have to make sure that you cover the entire application. Sometimes, 99.9% of all session-tracking information in an application is securely handled, but on one screen, a single variable is passed in the clear without being encrypted or hashed. Perhaps the Web developer got lazy on one page, or had a raucous night partying before writing that particular code. Alternatively, maybe the page was deemed unimportant, so an inexperienced summer intern wrote the code. Regardless, if a session ID is improperly protected on a single page, an attacker could find this weakness, usurp another user's session on that page, and move on to the rest of the application as that other user. With just one piece of unprotected session-tracking information, the application is very vulnerable, so you have to make sure you are protected throughout the application.

Additionally, you need to give your users the ability to terminate their sessions by providing a logout feature in your Web application. When users click on the logout button, their session should be terminated and the application should invalidate the session ID. Therefore, an attacker will not be able to steal the session ID, because it's no longer valid. Also, if a user's session is inactive for a certain length of time (for example, 15 minutes), your application should automatically time out the connection and terminate the session ID. That way, when users close their browsers without gracefully logging out of the session, an attacker will still not be able to usurp a live session after the time-out period expires.

I recommend that you assess the security of the session-tracking mechanisms of your own Web applications. You could use a tool like Achilles to manually comb through your application to make sure you properly handle all session IDs, as well as other information exchanged with the browser. Additionally, a commercial tool called AppScan by Sanctum, Inc. (at *www.sanctuminc.com*), will automatically scan your Web site looking for problems with information exchanged with the browser and warn you before an attacker can exploit them.

SQL Piggybacking

Another weakness of many Web applications involves problems with accepting user input and interacting with back-end databases. Most

Web applications are implemented with a back-end database that uses the Structured Query Language (SQL). Based on interactions with a user, the Web application accesses the back-end database to search for information or update fields. For most user actions, the application sends one or more SQL statements to the database that include search criteria based on information entered by the user. By carefully crafting a statement in a user input field of a vulnerable Web application, an attacker could extend an application's SQL statement to extract or update information that the attacker is not authorized to access. Essentially, the attacker wants to piggyback extra information onto the end of a normal SQL statement to gain unauthorized access. Rainforest Puppy used a variation on this technique to attack the Packetstorm security Web site, as he describes in his paper "How I Hacked Packetstorm" at *www.wiretrip.net/rfp/p/doc.asp?id=42*.

To accomplish this SQL piggybacking attack, the attackers will explore how the Web application interacts with the back-end database by finding a user-supplied input string that they suspect will be part of a database query (e.g., user name, account number, product SKU, etc.). The attacker will then experiment by adding quotation characters (i.e., ', ", and `) and command delimiters (i.e., ;) to the user data to see how the system reacts to the submitted information. In many databases, quotation characters are used to terminate values entered into SQL statements. Additionally, semicolons often act as separating points between multiple SQL statements. Using a considerable amount of trial and error, the attacker will attempt to determine how the application is interacting with the SQL database. A trial-and-error process is involved because each Web application formulates queries for a back-end database in a unique fashion.

Figure 7.26 displays the Mock Bank Web application feature that allows a user to conduct a database search for specific accounts owned by a user. Users should only be able to view accounts that they own; all other customer accounts should be inaccessible. In our example, to explore how the Web application interacts with the back-end database, the attacker will start by logging into the Web application with the attacker's own userID of 10001. The attacker might then start analyzing the search function by typing in a bogus value for an account search, such as an extra-long account number made up of all 1's.

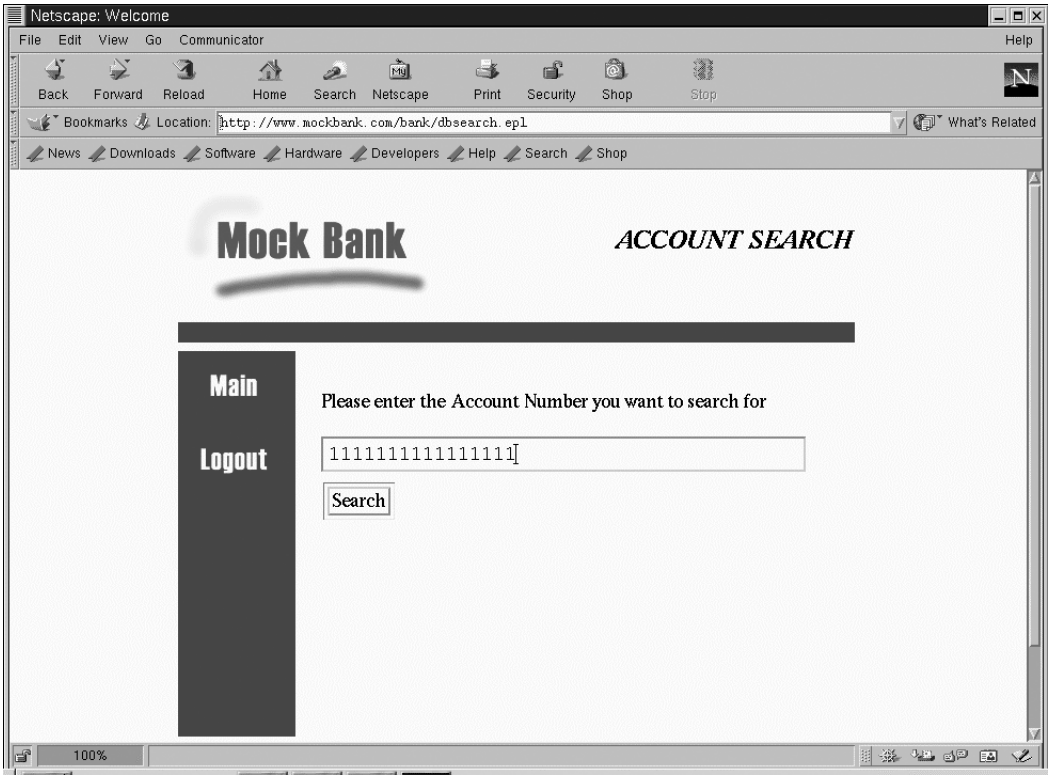


Figure 7.26
Figuring out how the Web application interacts with a database.

As can be expected, the attacker's search did not yield an actual account in the target system. However, as shown in Figure 7.27, we can see that the browser's location line does contain the search string used by the attacker.

Now, the attacker will start playing with the search element on the location line, entering various combinations of quote characters and semicolons to try to reverse-engineer the way the application creates SQL queries for the database based on user input. At this stage of the attack, raw error messages from the back-end database are extremely helpful.

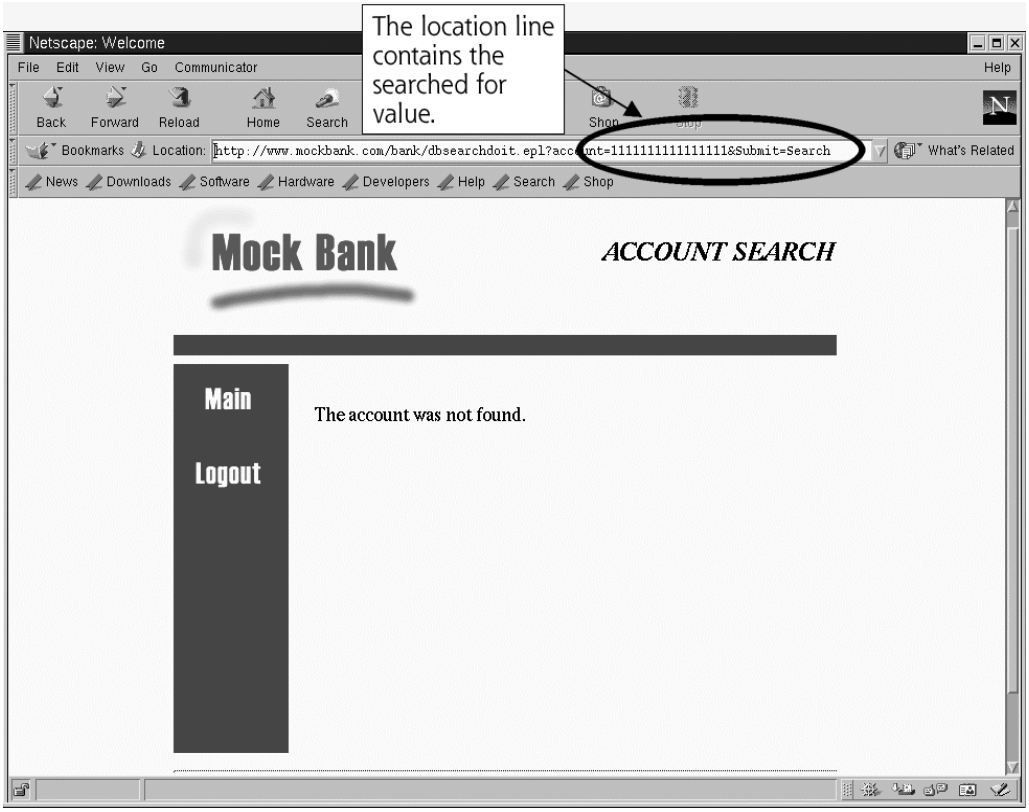


Figure 7.27
The location line contains the account number searched for.

At some point in our example, the attacker stumbles upon the simple idea of adding a single quote to the end of the account number, entering in the value `1111111111111111'` on the location line. Our example Web application returns the error message shown in Figure 7.28.

This error message is just what the attacker is looking for. The error message is the result of the two consecutive quote characters at the end of the statement. One quote mark was added by the attacker typing on the location line, while the other was generated by the Web application itself. More importantly, the error message comes right from the

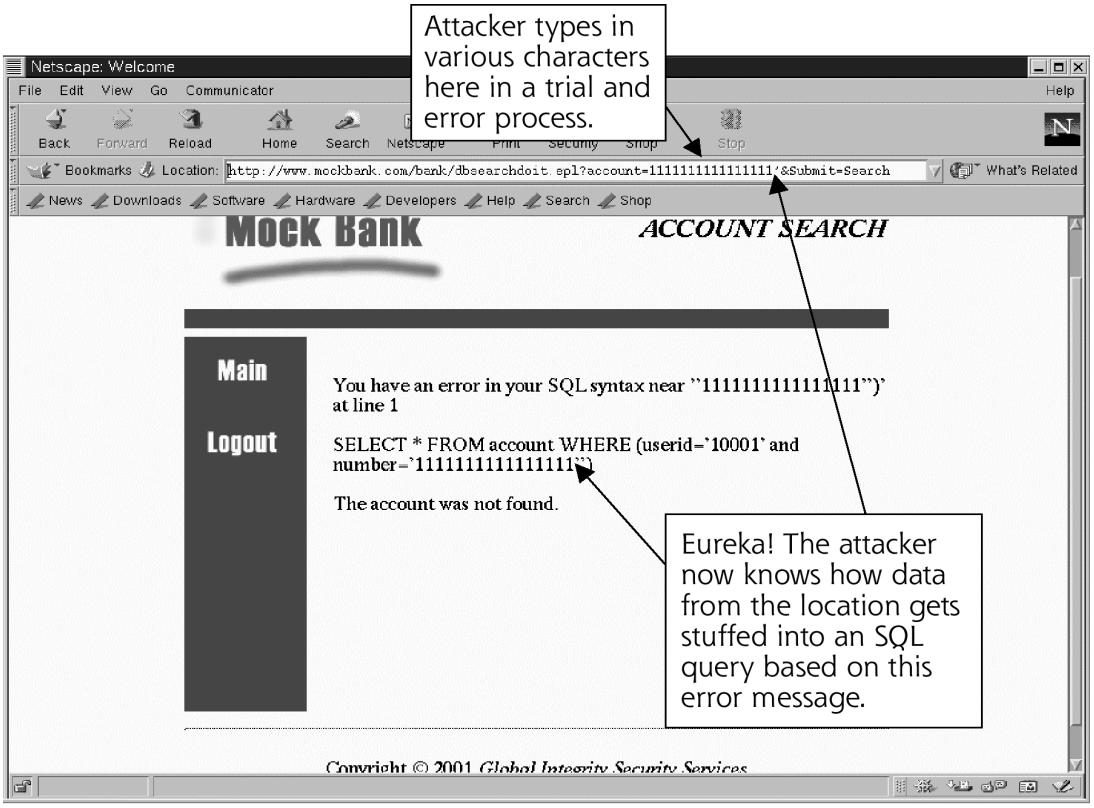


Figure 7.28
A very useful error message.

database itself and shows how the application formulates a query. The basic SQL statement used by the application is:

This value is the attacker's userID, automatically entered into the SQL query by the Web application.

```
SELECT * FROM account WHERE (userid='10001' and number
='INPUT_FROM_LOCATION LINE')
```

Here is where the input from the location line is entered into the SQL statement.

We can see that the application takes the information from the browser's location line and drops it into the SQL query. In SQL piggybacking attacks, the attacker will try to extend the SQL query, again using trial and error. For example, suppose that the attacker has used account harvesting against the Web application and knows that another customers' userID is 10002. The attacker wants to access unauthorized information associated with this target userID. By analyzing the SQL statement from the error message, the attacker will again use trial and error to add characters to the location line to feed them to the database.

In our example, shown in Figure 7.29, the attacker types the characters `1111111111111111'+or+userid%3d'10002` onto the location line. The Web application will drop this line into its SQL query by translating the + characters into spaces, and the code `%3d` into an equals sign ("="). Therefore, by entering this string into the location line of the browser, the attacker will force the application to formulate the following SQL query:

```
SELECT * FROM account WHERE (userid='10001' and number
='1111111111111111' or userid='10002')
```

Added by the attacker to the browser's location line.

With this SQL statement, the attacker hits pay dirt! The SQL statement looks up account information based on where the account number is bogus (111111111111) *or* the account's userID is 10002. The resulting response will include account information associated with userID 10002, giving the attacker an unauthorized view of this other users' data.

Our example showed piggybacking techniques for SQL query statements (a SELECT command in particular). Piggybacked UPDATE commands can allow an attacker to modify data in the database, adding accounts or altering sensitive user information.

SQL piggybacking can be extremely useful, but it is limited because all returned data is formatted and displayed by the Web application. Therefore, while attackers may be able to get the database to do all kinds of strange tricks with piggybacked SQL elements, they will only be able to see the results that the Web application is coded to deliver. So, in our previous example, the Web application may print the response from the lookup for the bogus account (111111111111111) and the accounts for userID 10002, or it may just print out the first response it receives (i.e., the blank data from the bogus account). In

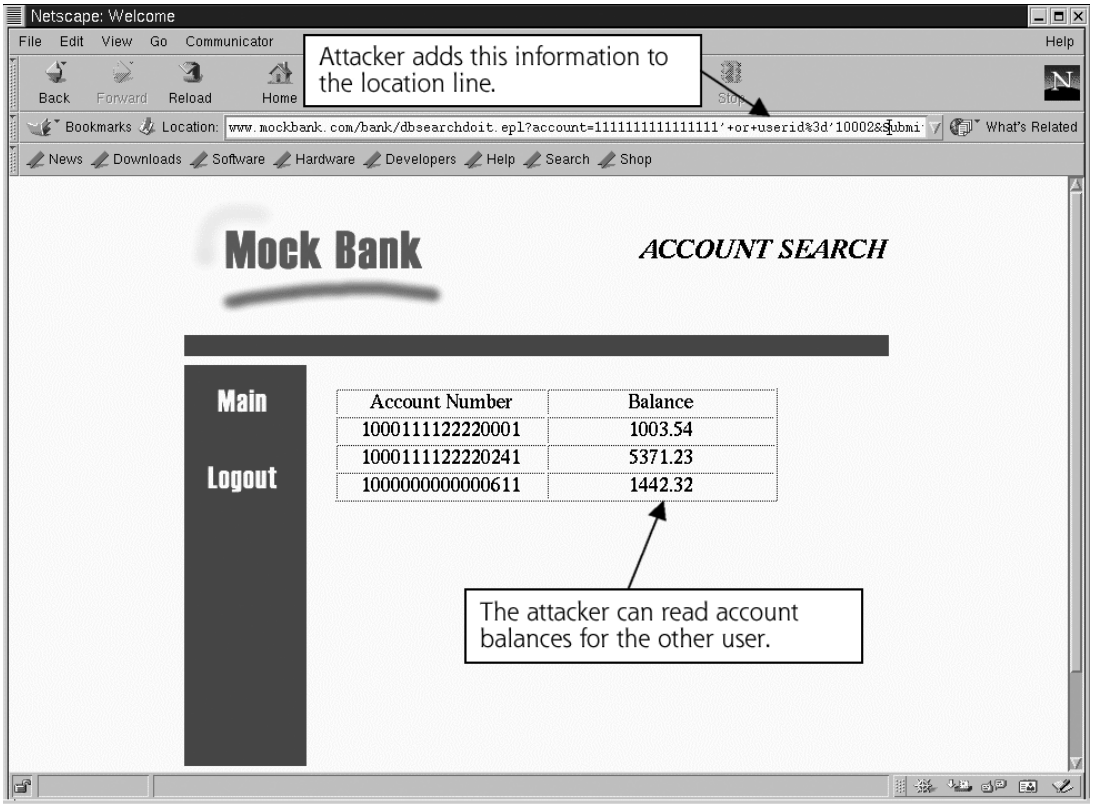


Figure 7.29
Gaining unauthorized access with SQL piggybacking.

essence, while the attackers can arbitrarily extend SQL statements going to the database using this technique, they can only view their results through the screen of the Web application. Still, even with this limitation, this technique can offer an attacker a profound level of access into a database.

Defenses against Piggybacking SQL Commands

To defend against piggybacked SQL statements and related attacks through user input, your Web application must be developed to carefully filter user-supplied data. Remember, the application should never trust raw user input. It could contain escape characters, piggybacked commands, and all kinds of general nastiness. Wherever data is entered into the application by a user, the application must strongly enforce the

content type of data entered. A numerical user input should really only be a number; all non-numerical characters must be filtered. Furthermore, the application must remove unneeded special characters before further processing the user input. In particular, the application should screen out the following list of scary characters:

- Quotes of all kinds (', ", and `)—String terminators
- Semicolons (;)—Query terminators
- Asterisks (*)—Wildcard selectors
- Percents (%)—Matches for substrings
- Underscores (_)—Matches for any character
- Other shell metacharacters (&\|*?~<>^()[]{}$\n\r), which could get passed through to a command shell, allowing an attacker to execute arbitrary commands on the machine.

These potentially damaging characters must be filtered at the server side. Many applications filter input on the browser, using Javascript or other techniques. As we discussed in the previous section, an attacker can bypass any client-side filtering using Achilles to inject arbitrary data into the HTTP/HTTPS connection.

To defend against this attack and other Web application problems, you should also arm your Web application developers with the World Wide Web Security FAQ, by Lincoln Stein, available at *www.w3.org/Security/Faq/www-security-faq.html.* This fantastic document describes many important details for developing secure Web applications, as well as securing a Web server.

Conclusions

Throughout this chapter, we've seen powerful techniques that an attacker can use to gain access to a target machine by attacking operating systems and applications. New vulnerabilities in these areas are being discovered on a daily basis and are widely shared within the computer underground. Therefore, it is important that you consider the defenses highlighted in this chapter in your own security program to protect your systems and vital information.

Now that we understand the most common operating system and application attacks, let's move down the protocol stack to analyze network-based attacks.

Summary

Using information gained from the reconnaissance and scanning phases, attackers attempt to gain access to systems. The techniques employed during Phase 3, Gaining Access, depend heavily on the skill level of the attacker. Less-experienced attackers use exploit tools developed by others, available at a variety of Web sites. More sophisticated attackers write their own customized attack tools and employ a good deal of pragmatism to gain access.

Stack-based buffer overflows are among the most common and damaging of attacks today. They exploit software that is poorly written, allowing an attacker to enter data into programs to execute arbitrary commands on a target machine. When a program does not check the length of input supplied by a user before entering the input into memory space on the stack, a buffer overflow could result. Without this proper bounds checking, an attacker provides input that consists of executable code for the target system to run, along with a new return pointer for the stack. By rewriting the return pointer on the stack, the attacker can make the target system run the executable code.

On systems with stack-based buffer overflow vulnerabilities, attackers employ a variety of techniques to gain access. They may create a backdoor using the inetd process. Another popular technique is using the TFTP program to upload Netcat, a tool that can be used to create a backdoor. Attackers also exploit the X Window system to get Xterminal access to target systems. They also use a variety of additional techniques.

Defenses against stack-based buffer overflows include applying security patches in a timely manner, filtering incoming and outgoing traffic, and configuring systems so that their stacks cannot be used to run executable code. Software developers can also help stop stack-based buffer overflows by utilizing automated code-checking and compile-time stack protection tools.

Password attacks are also very common. Attackers often try to guess default passwords for systems to gain access, by hand or through using automated scripts. Password cracking involves taking the encrypted/hashed passwords from a system and using an automated tool to determine the original passwords. Password-cracking tools create password guesses, encrypt/hash the guesses, and compare the result with the encrypted/hashed password. The password guesses can come from a dictionary, brute-force routine, or a hybrid technique.

L0phtCrack is one of the best tools for cracking Windows NT/2000 passwords. On UNIX systems, John the Ripper is excellent.

To defend against password attacks, you must have a strong password policy that requires users to have nontrivial passwords. You must make users aware of the policy, employ password-filtering software, and periodically crack your own users' passwords to enforce the policy. You may also want to consider authentication tools stronger than passwords, such as hardware tokens.

Attackers employ a variety of techniques to undermine Web-based applications. Some of the most popular techniques are account harvesting, undermining Web application session tracking, and SQL piggybacking. Account harvesting allows an attacker to determine account numbers based on different error messages returned by an application. To defend against this technique, you must make sure your error messages regarding incorrect userIDs and passwords are consistent. Attackers can undermine Web application session tracking by manipulating URL parameters, hidden form elements, and cookies to try to usurp another user's session. To defend against this technique, make sure your applications use strong session tracking information that cannot easily be determined by an attacker. SQL piggybacking allows attackers to extend SQL statements in an application by appending SQL elements to user input. The technique allows attackers to extract or update additional information in a back-end database behind a Web server. To protect your applications from this technique, you must carefully screen special characters from user input.