skoudis.book Page 339 Wednesday, November 30, 2005 10:04 PM

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, developing 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 depends heavily on the skill level of the attacker, with simple script kiddies trolling for exploits and more sophisticated attackers using highly pragmatic approaches.

SCRIPT KIDDIE EXPLOIT TROLLING

To try to gain access, the average script kiddie typically just takes the output from a vulnerability scanner and surfs to a Web site offering vulnerability exploitation programs to the public. These exploit programs are little chunks of code that craft very specific packets designed to make a vulnerable program execute commands of an attacker's choosing, cough up unauthorized data, or even crash in a DoS attack. Several organizations offer huge arsenals of these free, 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 the following:

- The French Security Incident Response Team (Fr-SIRT) exploit list at *www.frsirt.com/exploits*
- Packet Storm Security at *www.packetstormsecurity.org*

skoudis.book Page 340 Wednesday, November 30, 2005 10:04 PM

- The Security Focus Bugtraq Archives *at www.securityfocus.com/bid*
- The Metasploit Project at *www.metasploit.com*

Some controversy surrounds the organizations distributing these exploits. Most of them have a philosophy of complete disclosure: If some attackers know about these exploits, they should be made public so that everyone can analyze, understand, and defend against them. With this mindset, these purveyors of explicit exploit information argue that they are merely providing a service to the Internet community, helping the good guys keep up with the bad guys. Others take the view that these exploits just make evil attacks easier and more prevalent. Although 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 the nature of this book).

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

PRAGMATISM FOR MORE SOPHISTICATED ATTACKERS

Whereas a script kiddie utilizes these Internet searches to troll for canned exploits without understanding their function, a more sophisticated attacker sometimes employs 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

skoudis.book Page 341 Wednesday, November 30, 2005 10:04 PM

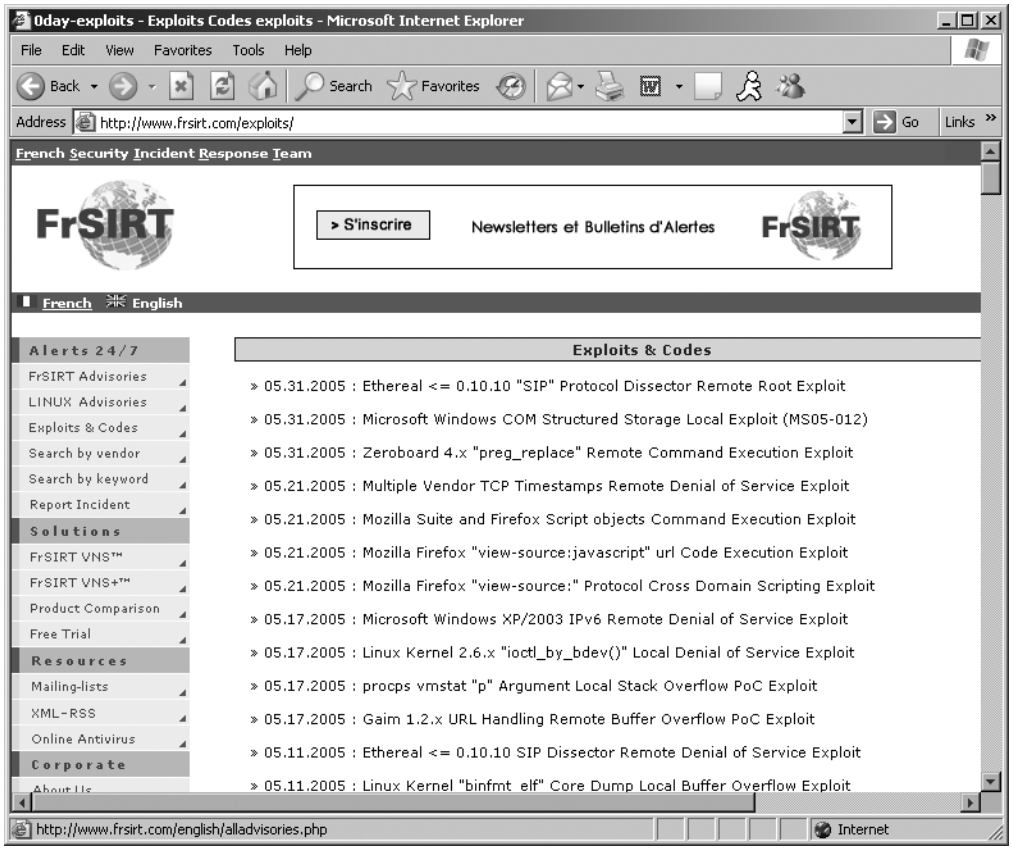


Figure 7.1 Searching FrSIRT for an exploit.

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 with which the attacker begins. 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. Thus, we discuss this phase by describing a variety of techniques used to gain access, without regard to the particular order in which an attacker might apply them. Our discussion of these techniques

skoudis.book Page 342 Wednesday, November 30, 2005 10:04 PM

starts with attacks against operating systems and applications in this chapter, followed, in the next chapter, by a discussion of network-based attacks.

There are several 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 discuss some of the most widely used and damaging application and operating system attacks, namely buffer overflow exploits, password attacks, Web application manipulation, and browser flaw exploits.

BUFFER OVERFLOW EXPLOITS

Buffer overflows 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. Although the infosec community has known about buffer overflows for decades, this type of attack really hit the big time in late 1996 with the release of a seminal paper on the topic called "Smashing the Stack for Fun and Profit" by Aleph One. You can find this detailed and well-written paper, which is still an invaluable read even today, at *www.packetstormsecurity.org/docs/hack/smashstack.txt*. Before this paper, buffer overflows were an interesting curiosity, something we talked about but seldom saw in the wild. Since the publication of this paper, the number of buffer overflow vulnerabilities discovered continues to skyrocket, with several brand new flaws and exploits to take advantage of them released almost every single day.

By exploiting vulnerable applications or operating systems, attackers can execute commands of their choosing on target machines, potentially taking over the victim machines. 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, access a command prompt, remotely control the GUI, alter the system's configuration … anything I want to do, really. Attackers love this ability to execute commands on a target computer.

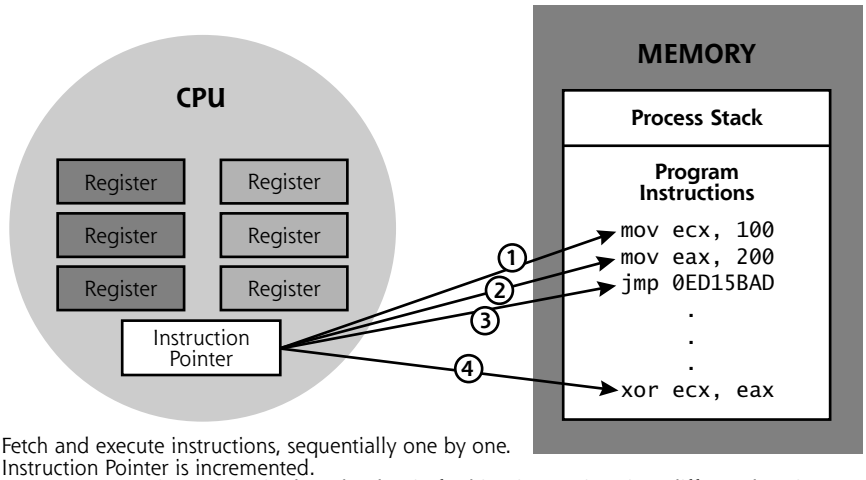
Buffer overflow vulnerabilities are based on an attacker sending more data to a vulnerable program than the original software developer planned for when writing the code for the program. The buffer that is overflowed is really just a variable used by

skoudis.book Page 343 Wednesday, November 30, 2005 10:04 PM

the target program. In essence, these flaws are a result of sloppy programming, with a developer who forgets to create code to check the size of user input before moving it around in memory. Based on this mistake, an attacker can send more data than is anticipated and break out of the bounds of certain variables, possibly altering the flow of the target program or even tweaking the value of other variables. There are a variety of buffer overflow types, but we look at two of the most common and popular: stack-based buffer overflows and heap overflows.

STACK-BASED BUFFER OVERFLOW ATTACKS

To understand how stack-based buffer overflow attacks work, we first need to review how a computer runs a program. Right now, if your computer is booted up, it is processing millions of computer instructions per second, all written in machine language code. How does this occur? Consider Figure 7.2, which highlights the relationship of a system's processor and memory during execution. When running a program, your machine's Central Processing Unit (CPU) fetches instructions from memory, one by one, in sequence. The whole program itself is just a bunch of bits in the computer's memory, in the form of a series of instructions for the processor. The CPU contains a very special register called the



Fetch and execute instructions, sequentially one by one.
Instruction Pointer is incremented.
At Jump, Instruction Pointer is altered to begin fetching instructions in a different location.

Figure 7.2 How programs run.

skoudis.book Page 344 Wednesday, November 30, 2005 10:04 PM

Instruction Pointer, which tells it where to grab the next instruction for the running program. The CPU grabs one program instruction from memory by using the Instruction Pointer to refer to a location in memory where the instruction is located within the given segment of code. The CPU executes this instruction, and the Instruction Pointer is incremented to point to the next instruction. The next instruction is then fetched and run. The CPU continues stepping through memory, grabbing and executing instructions sequentially, until some type of branch or jump is encountered. These branches and jumps are caused by if–then conditions, loops, subroutines, goto statements, and related conditions in the program. When a jump or branch is encountered, the instruction pointer's value is altered to point to the new location in memory, where sequential fetching of instructions begins anew.

In my opinion, the idea of the stored-program-controlled computer illustrated in Figure 7.2 is one of the most important technical concepts of the last century. Sure, splitting the atom was cool, but that feat has, so far, had less impact on my life than this idea. Let's hope it stays that way! Putting a person on the moon was sure nifty, but I feed my family because of the concepts in Figure 7.2, and you probably do, too. In fact, we might not have made it to the moon had we not already come up with this idea, given the primitive computers that were required for the moon shots. In fact, all a computer consists of is a little engine (the CPU) that moves data around in a memory map, based on instructions that are located in that same memory map. And that's where the problem is. By carefully manipulating elements in that memory, an attacker can redirect the flow of execution to the attacker's own instructions loaded into memory.

Function Calls and the Stack

Now that we've seen the microscopic level of how programs run, we've got to step up to a higher view of the system. Most modern programs aren't written directly in machine language, those low-level instructions we illustrated in Figure 7.2. Instead, they are written in a higher level language, such as C, C++, Java, or Perl. They are then converted into machine language (either by a compiler for languages like C and C++ or a real-time interpreter for stuff like Java and Perl) and executed. Most high-level languages include the concept of a function call, used by programmers to break the code down into smaller pieces. Figure 7.3 shows some sample code written in the C programming language.

skoudis.book Page 345 Wednesday, November 30, 2005 10:04 PM

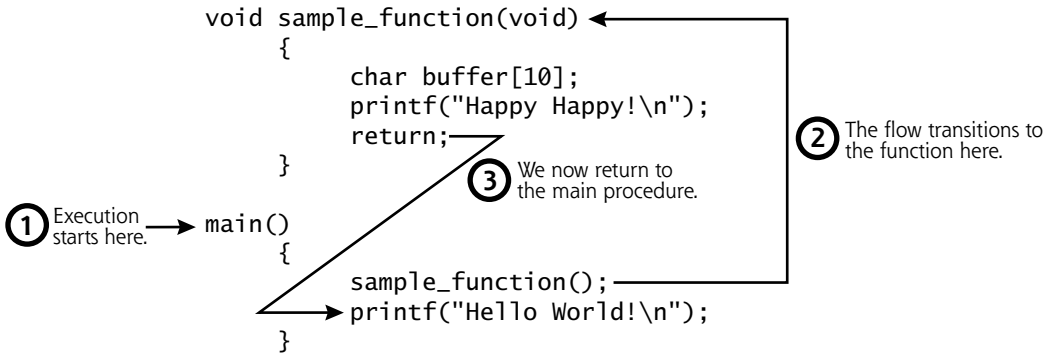


Figure 7.3 Some C code.

When the program starts to run, the `main` procedure is executed first. The first thing the `main` procedure does is to 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 was operating in the `main` procedure, because after `sample_function` finishes running, the program flow must return back to the `main` procedure. But how does the system remember where it should return after the function call is done? The system uses a stack to remember this information associated with function calls.

A stack is a data structure that stores important information for each process 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 and places these notes on the stack, a special reserved area in memory for each running program. Stacks are similar to (and get their name from) stacks of dishes, in that they behave in a Last-In, First-Out (LIFO) manner. That is, when you are creating a stack of dishes, you pile plate on top of plate 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 first takes off the last element it placed on the stack, a process known as popping an item off of the stack. Depending on the computing architecture, the stack may grow upward (toward higher memory addresses) or downward (toward lower addresses) in memory. The direction of growth isn't really important to us here; it's the LIFO property that matters.

skoudis.book Page 346 Wednesday, November 30, 2005 10:04 PM

Now, what types of things does a computer store on a stack? Among other things, stacks are used to store information associated with function calls. As shown in Figure 7.4, a system pushes various data elements onto the stack associated with making a 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 code of Figure 7.3 included 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. 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. That return pointer is a crucial element, isn't it? It later controls the flow of the program, directing where execution resumes after the function call is completed.

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

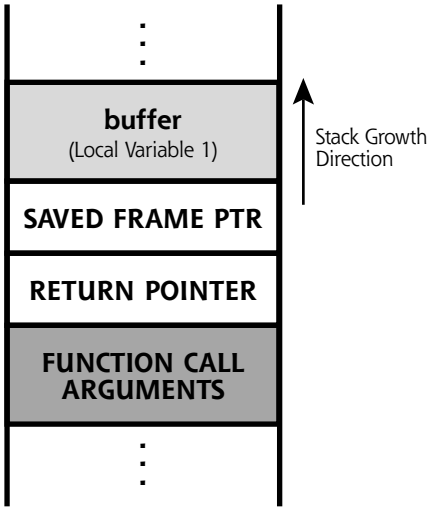


Figure 7.4 A normal stack.

skoudis.book Page 347 Wednesday, November 30, 2005 10:04 PM

are supposed to be for the exclusive use of the function, which can store its local data in them and manipulate their values.

After the function finishes running, printing out its happy message of "Hello World," control returns to the main program. This transition occurs by popping the local variables from the stack (in our example, the `buffer` variable). For the sake of efficiency, the memory locations on the stack allocated to these local variables are not erased. Data is removed from the stack just by changing the value of a pointer to the top of the stack, the so-called Stack Pointer. 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 copied 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 (pre-function-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. Everything works beautifully, as function calls get made and completed. Sometimes one function calls other functions, which in turn call other functions, all the while with the stack growing and shrinking as required.

What Is a Stack-Based Buffer Overflow?

Now that we understand how a program interacts with the stack, let's look at how an attacker can abuse this capability. A buffer overflow is rather like putting ten liters of stuff into a bag that will only hold five liters. Clearly something is going to spill out. Let's see what happens when an attacker provides too much input to a program. Consider the sample vulnerable program of Figure 7.5.

For this program, the main routine prints a "Hello World" greeting and then calls the `sample_function`. In `sample_function`, we create two buffers, `bufferA`, which is 50 characters in length, and `bufferB`, which can hold 16 characters. Both of these are local variables of the `sample_function`, so they will be allocated space on the stack, as shown in Figure 7.6. We then prompt the user for input by printing "Where do you live?" The `gets` function (which is pronounced "get-ess") from a standard C library will pull input from the user. Next, we encounter the `strcpy` library call. This routine is used to copy information from one string of characters to another. In our program, `strcpy` moves characters from `bufferA` to `bufferB`.

skoudis.book Page 348 Wednesday, November 30, 2005 10:04 PM


```
void sample_function()
{
        char bufferA[50];
        char bufferB[16];

        printf("Where do you live?\n");

        gets(bufferA);

        strcpy(bufferB, bufferA);

        return;
}

main()
{
        printf("Hell World!\n ");
        sample_function();
        printf("All Done!\n ");
}
```

3. Execution begins in the sample_function.
4. Create two strings. bufferA can hold 50 characters, while bufferB can hold 16 characters.
5. Ask the user where he or she lives.
6. Get input from the user. Note that gets puts no restrictions on the amount of data that can be entered!
7. Copy the contents of bufferA to bufferB.
8. Return (intended to go back to the main program that called the function!)
1. Print "Hello World!"
2. Call the sample_function.

Figure 7.5 Some very vulnerable C code.

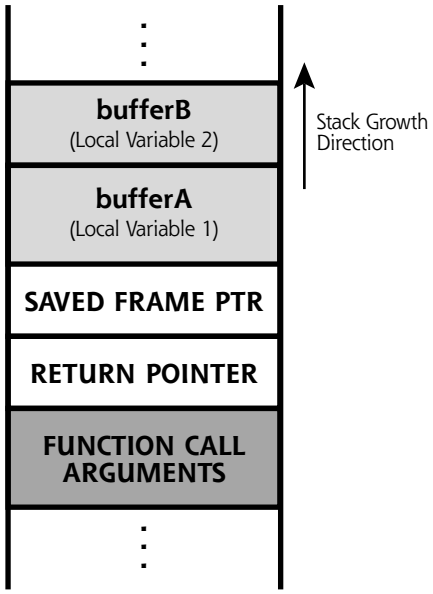


Figure 7.6 A view of the stack of the vulnerable program.

skoudis.book Page 349 Wednesday, November 30, 2005 10:04 PM

However, we've got a couple of problems here. Can you see them? First, the `gets` library puts no limitation on the amount of data a user can type in. If the user types in more than 50 characters, `bufferA` will be overflowed, letting the attacker change other nearby places on the stack. In fact, the `gets` call is extremely dangerous and should be avoided at all costs, because it doesn't put any limitation on user input, thereby almost guaranteeing a buffer overflow flaw.

But wait, there's more. Beyond `gets`, the `strcpy` library call is also 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 eight zero bits in a row aligned in a single byte, usually indicates the end of a string for the various C-language string-handling libraries. This sloppiness of `strcpy` is a well-known limitation found in many of the normal C language library functions associated with strings. This is bad news because the system will allow the `strcpy` to write far beyond where it's supposed to write. That's one of the big problems with computers: They do exactly what we tell them to do, no more and no less. Even if the attacker doesn't overflow `bufferA` with more than 50 characters of user input in the `gets` call, the attacker has a shot at overflowing `bufferB` by simply typing between 17 and 50 characters into `bufferA`, which will be written to `bufferB`. Thus, we've got two buffer overflow flaws in this sample code: the `gets` problem indicated by item number 6, and the `strcpy` indicated by item number 7 in Figure 7.5. Ouch!

Now, let's suppose the user entering the input is an evil attacker, and types in the capital A character a couple hundred times when prompted about where he or she lives. What happens to the stack when the bad guy does this? Well, it gets messed up. The A characters will spill over the end of `bufferA`, `bufferB`, or both, running into the saved Frame Pointer, and even into the return pointer on the stack. The return pointer on the stack will be filled with a bunch of As. 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 As in it). The return pointer is copied into the processor's Instruction Pointer, and the machine tries to resume execution, thinking it's back at the main program. It tries to fetch the next instruction from a memory location that is the binary equivalent of a bunch of As (that would be hexadecimal 0x41414141 … you can look it up!). Most likely, this is a bogus memory location that the program

skoudis.book Page 350 Wednesday, November 30, 2005 10:04 PM

doesn't have permission to access or that contains data and not real executable code. With a bogus Instruction Pointer value, we'll likely get a nasty segmentation fault, an indication that the program is trying to access a place in memory that it is not allowed to access, so the operating system shuts it down. Thus, most likely, the program will crash.

So, after all this discussion, we've learned how to write a program that can be easily crashed by a nefarious user. "Gee," you might 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 As into the return pointer made the program crash, what if an attacker could overflow `bufferA` or `bufferB` with something more meaningful? The attacker could insert actual machine language code into the buffers, with commands that he or she wants to get executed. When prompted for where they live, clever attackers might type in the ASCII characters corresponding to machine language code to run some evil command on the victim machine.

So, in this way, the attacker can load commands on the target machine that the attacker wants to run. But how can the bad guy get the system to execute these commands? If only there was a way to control the flow of execution of the program, so the bad guy could say, "When you are done with your nice stuff, Mr. Vulnerable Program, I want you to run my evil stuff." Now, we get to that beautiful return pointer down below the local variables and saved Frame Pointer. Remember, when the attacker's input runs off the end of the local variables, that extra input can modify the return pointer (as well as the saved Frame Pointer). The bad guy could overwrite the return pointer with a value that points back into the buffer, which contains the commands he or she wants to execute. The resulting recipe, as shown in Figure 7.7, is a stack-based buffer overflow attack, and will allow the attacker to execute arbitrary commands on the system. Cha-ching! It's almost like the stack was designed to foster buffer overflow attacks, with that highly important return pointer lining up nicely a little bit below the local variables on the stack!

Let's review how the smashed stack works, focusing on just cramming too much input into `bufferA` via that vulnerable `gets()` call. The attacker gets a program to

skoudis.book Page 351 Wednesday, November 30, 2005 10:04 PM

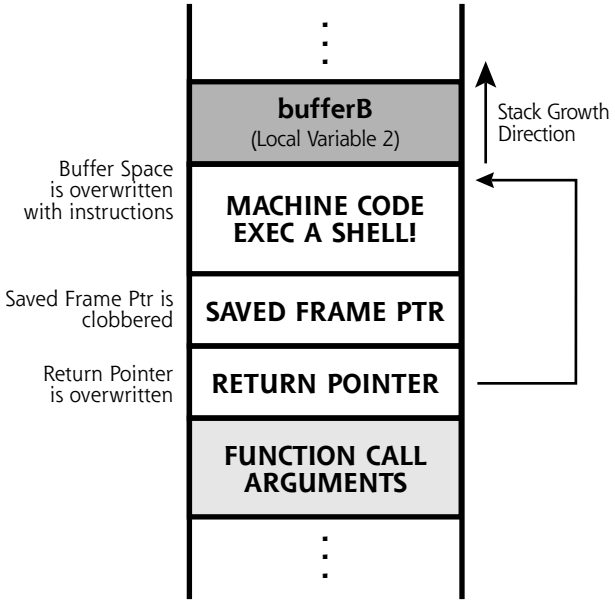


Figure 7.7 A smashed stack.

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 the end of the local variables. It keeps writing data over the end of the buffer, clobbering the saved Frame Pointer, and even overwriting the return pointer with a value that points back to the machine language instructions the attacker loaded into the `bufferA` on 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 now-modified return pointer into the processor, and starts executing instructions where the return pointer tells it to resume execution. 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 developer not checking the size of the information he or she is moving around in memory when making function calls. Without carefully doing a bounds check of these buffers before manipulating

skoudis.book Page 352 Wednesday, November 30, 2005 10:04 PM

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 usually places on the stack. Probably the most useful thing to force the machine to run for the attacker is a command shell, because then the attacker can feed the command shell (such as the UNIX and Linux `/bin/sh` or Windows `cmd.exe`) any other command to run. This can be achieved by placing the machine language code for executing a command prompt in the user input. Most operating systems include an exec system call to tell the operating system to run a given program. Thus, the attacker includes machine language code in the user input to exec a shell. After spawning a command shell, the attacker can then automatically feed a few specific system commands into the shell, running any program on the target machine. Some attackers force their shell to make a connection to a given TCP or UDP port, listening for the attacker to connect and get a remote command prompt. Others prefer to add a user to the local administrator's group on behalf of the attacker. Still other attackers might force the shell to install a backdoor program on the victim system.

Alternatively, instead of invoking the attacker's code in the stack, the bad guy could change a return pointer so that it doesn't jump into the stack, but instead resumes execution at another point of the attacker's choosing. Some attackers clobber a return pointer so that it forces the program to resume execution in the heap, another area of memory we discuss a little later. Or, the attacker could have the program jump into a particular C library the attacker wants to invoke, a technique known as a "return to libc" attack.

It's important to note that the attacker's code will run with the permissions of the vulnerable program. Thus, if the vulnerable program is running as root on UNIX or Linux or SYSTEM on Windows, the attacker will have complete administrative control of the victim machine. Lesser privileges are still valuable, though, as the attacker will have gotten a foot in the door with the ability to run limited privileged commands on the target.

skoudis.book Page 353 Wednesday, November 30, 2005 10:04 PM

Buffer overflow attacks are very processor and operating system dependent, because the raw machine code will only run on a specific processor, and techniques for executing a command shell differ on various operating systems. Therefore, a buffer overflow exploit against a Linux machine with an x86 processor will not run on a Windows 2003 box on an x86 processor or a 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 might 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 command-line arguments. For programs accessed across the network, data enters through open ports listening on the network, usually formatted with specific fields for which the program is looking.

To exploit a buffer overflow, an attacker enters 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 includes the machine language code and new return pointer in a single package. If the attacker sends just the right code with the right return pointer formatted just 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 trivial.

FINDING BUFFER OVERFLOW VULNERABILITIES

Simple script kiddie attackers who do not understand how their tools work carry out most stack-based buffer overflow attacks. These attackers just scan the target with an automated tool that detects the vulnerability, download the exploit code

skoudis.book Page 354 Wednesday, November 30, 2005 10:04 PM

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.

Beyond these script kiddies, how does the creator of a stack-based buffer overflow exploit find programs that are vulnerable to such attacks? These folks usually 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. Alternatively, they can peer into an executable program looking for evidence of the use of these library calls with a good debugger. The `gets` and `strcpy` routines we saw earlier are just some of the commonly used functions that programmers often misuse, resulting in a buffer overflow vulnerability. Other C and C++ functions that often cause such problems include the various string and memory handling routines like these:

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

Beyond these function calls, the developer of the program might have created custom calls that are vulnerable. Some exploit developers reverse engineer executables to find such flaws.

Alternatively, exploit creators might take a more brute force approach to finding vulnerable programs. They sometimes run the program in a lab and configure an

skoudis.book Page 355 Wednesday, November 30, 2005 10:04 PM

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 makes sure the entered data has a repeating pattern, such as the character A repeated thousands of times. Exploit creators are 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, remember, in hexadecimal format is 0x41) reflected in the instruction pointer when the program crashes. This technique of varying user input to try to make a target system behave in a strange fashion is sometimes called *fuzzing*. For buffer overflows, attackers fuzz the input by varying its size. Note that you can't just plop a billion characters into the input field to successfully fuzz most buffer overflows. It's possible that a billion characters will be filtered, but 10,000 might not. Therefore, for successful size fuzzing with buffer overflows, attackers typically start with small amounts of input (such as 1,000 characters or so) and then gradually increase the size in increments of 1,000 or 10,000, looking for a crash.

Consider this example of the output dump of a debugger showing the contents of a CPU's registers when a fuzzer triggers an overflow using a bunch of A characters.

```
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 (called EIP on modern x86 processors). Attackers love this value! The pattern being entered into the program (a long series of As; that is, 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, attackers can then create a buffer overflow exploit that lets them control a target machine running this program.

Once the attackers find out that some of the user input made it into the instruction pointer, they next need to figure out which part of all those As was the element

skoudis.book Page 356 Wednesday, November 30, 2005 10:04 PM

that landed on the return pointer. They determine this by playing a little game. They first fuzz with all As, as we saw before. Then, they fuzz with an incrementing pattern, perhaps of all of the ASCII characters, including ABCDEF and all of the other characters repeated again and again. I call this the ABCDEF game. They then wait for another crash. Now, suppose that the attacker sees that DEFG is in the return pointer slot. The attacker then fuzzes with each DEFG pattern of the input tagged, such as DEF1, DEF2, DEF3, and so on. Finally, the attacker might discover that DEF8 is the component of the user input that hits the return pointer. Voila! The attacker now knows where in the user input to place the return pointer. There are automated tools attackers can use to play this little game, which will identify the location in the user input where the new return pointer should be placed. Of course, the attacker still doesn't know what value to place there, but at least he or she knows where it will go in the user input once the value is determined.

So how does an attacker know what value to slide into our hypothetical DEF8 slot for the return pointer so that it will jump back into the stack to execute the attacker's instructions? With most programs, the stack is a rather dynamic place. An attacker usually doesn't know for sure what function calls were made before the vulnerable function is invoked. Thus, because the stack is very dynamic, it can be difficult to find the exact location of the start of the executable code the bad guy pushes onto the stack. The attacker could simply run the program 100 or more times, and make an educated guess of the address, a reasonable approach for some programs. However, the odds might still be 1 in 10,000 that the attacker gets the right address to hit the top of the evil code exactly in the stack.

To address this dilemma, the attackers usually prepend their machine language code with a bunch of No Operation (NOP) instructions. Most CPUs have one or more NOP instruction types, which tell the processor to do nothing for a single clock cycle. After doing nothing, execution will resume at the next instruction. By putting a large number of NOP instructions at the beginning of the machine language code, the attacker improves the odds that the guessed return pointer will work. This grouping of NOP instructions is called a NOP sled. As long as the guessed address jumps back into the NOP sled somewhere, the attacker's code will soon be executed. The code will do nothing, nothing, nothing, nothing, and then run the attacker's code to exec a shell.

skoudis.book Page 357 Wednesday, November 30, 2005 10:04 PM

You can think about the value of a NOP sled by considering a dart game. When you throw a dart at the target, you'd obviously like to hit the bull's eye. The guess of the return pointer is something like throwing a dart. If you guess the proper location of the start of the machine language code on the stack, that code will run. You've hit the bull's eye. Otherwise the program will crash, something akin to your dartboard exploding. A NOP sled is like a cone placed around the bull's eye on the dartboard. As long as your dart hits the cone (the NOP sled), the dart will slide gently into the bull's eye, and you'll win the game!

Attackers prepend as many NOP instructions at the front of their machine language code as they can, based on the size of the buffer itself. If the buffer is 1,024 characters long, and the machine language code the attacker wants to run takes up 200 bytes, that leaves 824 characters for NOPs. The simplest NOP is only one byte long for x86 processors. Thus, the bad guy can improve the odds in guessing the return pointer value 825-fold (that's one for each NOP, plus one for the very start of the attacker's machine language code to exec a shell). You don't have to be a gambler to realize that's a pretty good increase in odds, and it only gets better with bigger buffers. In fact, for this very reason, it's far easier for an attacker to exploit a larger buffer successfully than a smaller buffer. Remember, allocating more space to make bigger buffers doesn't fix buffer overflows. Bigger buffers ironically only make it easier to attack a program with a buffer overflow exploit. The real fix here involves checking the size of user input and managing memory more carefully, as we discuss later.

The NOP instructions used by an attacker in the NOP sled could be implemented using the standard NOP instruction for the given target CPU type, which might be detected by an IDS when a large number of NOPs move across the network. Craftier attackers might choose a variety of different instructions that, in the end, still do nothing, such as adding zero to a given register, multiplying a register by one, or jumping down to the next instruction in memory. Such variable NOP sleds are harder to detect.

As we have seen, the fundamental package for a buffer overflow exploit created by an attacker consists of three elements: a NOP sled, machine language code typically designed to exec a shell, and a return pointer to make the whole thing execute. This structure of a common buffer overflow exploit is shown in Figure 7.8.

skoudis.book Page 358 Wednesday, November 30, 2005 10:04 PM

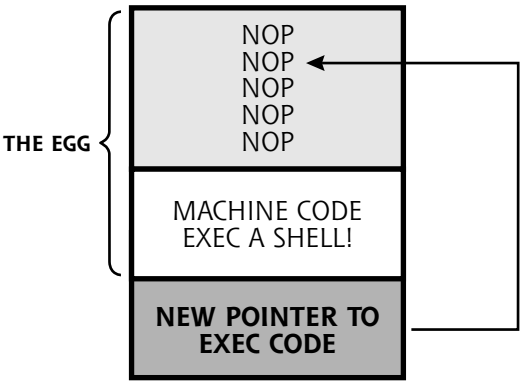


Figure 7.8 The structure of an exploit (also known as a sploit) for a buffer overflow vulnerability.

Note that the combined NOP sled and machine language code are sometimes called the exploit's *egg*. The entire package, including the code that alters a return pointer, along with the egg, is formally called an *exploit*, and informally referred to as a *sploit*.

HEAP OVERFLOWS

So far, our analysis of buffer overflow flaws has centered on the stack, the place where a process stores information associated with function calls. However, there's another form of buffer overflow attack that targets a different region of memory: the heap. The stack is very organized, in that data is pushed onto the stack and popped off of it in a coordinated fashion in association with function calls, as we've seen.

The heap is quite different. Instead of holding function call information, the heap is a block of memory that the program can use dynamically for variables and data structures of varying sizes at runtime. Suppose you're writing a program and want to load a dictionary in memory. In advance, you have no idea how big that dictionary might be. It could have a dozen words, or 6 million. Using the heap, you can dynamically allocate memory space as your program reads different dictionary terms as it runs. The most common way to allocate space in the heap in a C program is to use the `malloc` library call. That's short for memory allocation, and this function grabs some space from the heap so your program can tuck data there.

skoudis.book Page 359 Wednesday, November 30, 2005 10:04 PM

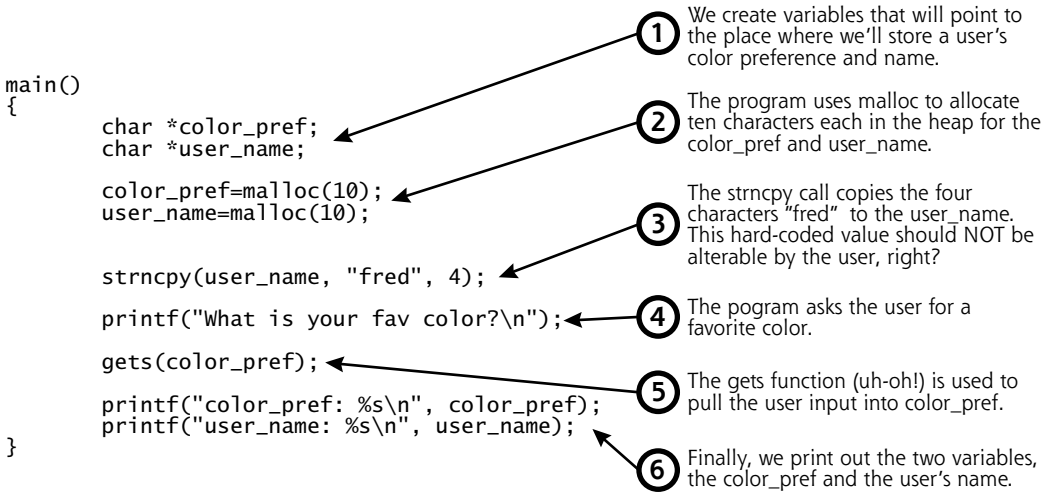


Figure 7.9 A program with a heap-based buffer overflow vulnerability.

So what happens if a developer uses `malloc` to allocate space in the heap where user input will be stored, but again forgets to check the size of the user input? Well, we get a heap-based buffer overflow vulnerability, as you'd no doubt expect. To illustrate this concern, consider the code in Figure 7.9.

Our program starts to run and creates some pointers where we'll later allocate memory to hold a user's color preference and name, called `color_pref` and `user_name`, respectively. We then use the `malloc` call to allocate ten characters in the heap to each of these variables, as illustrated in Figure 7.10. Note that the heap typically grows in the opposite direction as the stack in most operating systems and processors.

Next, our program uses the `strncpy` call, which copies a fixed number of characters into a string. We copy into the `user_name` a fixed value of "fred," only four characters in length. This `user_name` is hard coded, and shouldn't be alterable by the user in any way.

Next, we quiz our user, asking his or her favorite color. Uh-oh … the program developer used that darned `gets` function again, the poster child of buffer overflow flaws, to load the user input into the `color_pref` variable on the heap. Then, the program finishes by displaying the user's favorite color and user name on the screen.

skoudis.book Page 360 Wednesday, November 30, 2005 10:04 PM

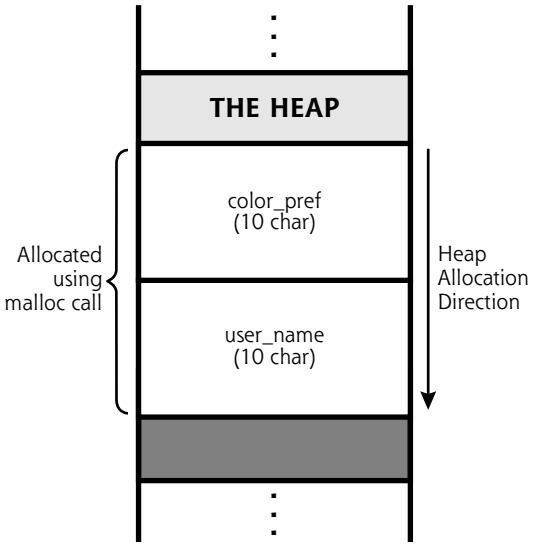


Figure 7.10 The heap holds the memory we malloc'ed.

To see what happens when this program runs, consider Figure 7.11, which shows two sample runs of the program. In the first run, shown on the left of Figure 7.11, the user types a favorite color of blue. The program prints out a favorite color of blue and a user name of fred, just like we'd expect. For the next run, the user is an evil attacker, who types in a favorite color of bluebluebluebluerоot. That's 16 characters of blue followed by root. Check out that display! Because the developer put no limitation on the size of the user input with that very lame `gets` call, the bad guy was able to completely overwrite all space in the `color_pref` location on the heap, breaking out of it and overwriting the `user_name` variable with the word root! Now, this wouldn't change the user ID of the running program itself in the operating system, but it would allow the attacker to impersonate another user named root within the program itself. Note that the attacker has to type in more than just ten characters (in fact, 16 characters are required, as in blueblueblue-blue) to scoot out of the `color_pref` variable, instead of just the ten characters we allocated. That's because the `malloc` call sets aside a little more space than we ask for to keep things lined up in memory for itself. Still, by exploring with different sizes of input using the fuzzing techniques we discussed earlier, the attacker can change this variable and possibly others on the heap.

skoudis.book Page 361 Wednesday, November 30, 2005 10:04 PM

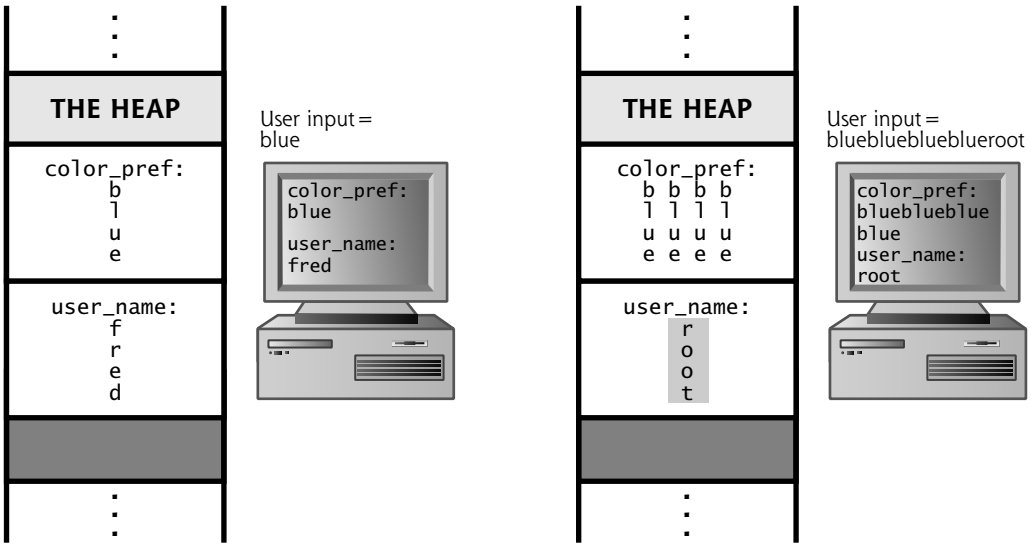


Figure 7.11 Running the vulnerable program with two different inputs.

THE EXPLOIT MESS AND THE RISE OF EXPLOITATION ENGINES

We've seen both stack- and heap-based buffer overflows and how they could let an attacker redirect the flow of program execution or change other variables in a vulnerable program. However, there's a problem for the bad guys. Historically, when a new vulnerability was discovered, such as a buffer overflow flaw, crafting an exploit to take advantage of the flaw was usually a painstaking manual process. Developing an exploit involved handcrafting software that would manipulate return pointers on a target machine, load some of the attacker's machine language code into the target system's memory (the egg), and then calculate the new value of the return pointer needed to make the target box execute the attacker's code. Some exploit developers then released each of these individually packaged exploit scripts to the public, setting off a periodic script kiddie feeding frenzy on vulnerable systems that hadn't yet been patched. But due to the time-consuming exploit development process, defenders had longer time frames to apply their fixes.

Also, the quality of individual exploit scripts varied greatly. Some exploit developers fine-tuned their wares, making them highly reliable in penetrating a target.

skoudis.book Page 362 Wednesday, November 30, 2005 10:04 PM

Other exploit creators were less careful, turning out garbage sploits that sometimes wouldn't work at all or would even crash a target service most of the time. The functionality of eggs varied widely as well. Some developers would craft exploits that created a command shell listener on their favorite TCP or UDP port, whereas others focused on adding an administrative user account for the attacker on the target machine, and others had even more exotic functionality embedded in their sploits. Making matters worse, a really good egg from one exploit wouldn't easily interoperate with another exploit, making it hard to reuse some really choice code. The developers and users of exploits were faced with no consistency, little code reuse, and wide-ranging quality; in other words, the exploit world was a fractured mess.

To help tame this mess of different exploits, two extremely gifted software developers named H. D. Moore and spoonm released Metasploit, an exploit framework for the development and use of modular exploits to attack systems, available for free at *www.metasploit.com*. Metasploit is written in Perl, and runs on Linux, BSD, and Microsoft Windows. To run it on Windows, the user must first install a Perl interpreter, such as the ActiveState Perl environment, available for free at *www.activestate.com/Perl.plex*. Beyond the free, open-source Metasploit tool, some companies have released high-quality commercial exploit frameworks for sale, such as the IMPACT tool by Core Security Technologies (*www.coresecurity.com*) and the CANVAS tool by Immunity (*www.immunitysec.com*).

In a sense, Metasploit and these commercial tools act as an assembly line for the mass production of exploits, doing about 75 percent of the work needed to create a brand new, custom sploit. It's kind of like what Henry Ford did for the automobile. Ford didn't invent cars. Dozens of creative hobbyists were handcrafting automobiles around the world for decades when Ford arrived on the scene. However, Henry revolutionized the production of cars by introducing the moving assembly line, making auto production faster and cheaper. In a similar fashion, exploit frameworks like Metasploit partially automate the production of sploits, making them easier to create and therefore more plentiful.

Some people erroneously think exploit frameworks are simply another take on vulnerability scanners, like the Nessus scanner we discussed in Chapter 6, Phase 2: Scanning. They are not. A vulnerability scanner attempts to determine if a target

skoudis.book Page 363 Wednesday, November 30, 2005 10:04 PM

machine has a vulnerability present, simply reporting on whether or not it thinks the system could be subject to exploitation. An exploit framework goes further, actually penetrating the target, giving the attacker access to the victim machine.

To understand how Metasploit works, let's look at its different parts, as shown in Figure 7.12. First, the tool holds a collection of exploits, little snippets of code that force a victim machine to execute the attacker's payload, typically by overwriting a return pointer in a buffer overflow attack. Most exploit frameworks have more than 100 different exploits today, including numerous stack- and heap-based buffer overflow attacks, among several other vulnerability types. The current Metasploit exploit inventory includes some of the most widespread and powerful attacks, such as the Windows RPC DCOM buffer overflow (that was the exploit used by the Blaster worm, by the way), the Samba trans2open Overflow, the War-FTPD passive flaw, and the good old WebDAV buffer overflow in

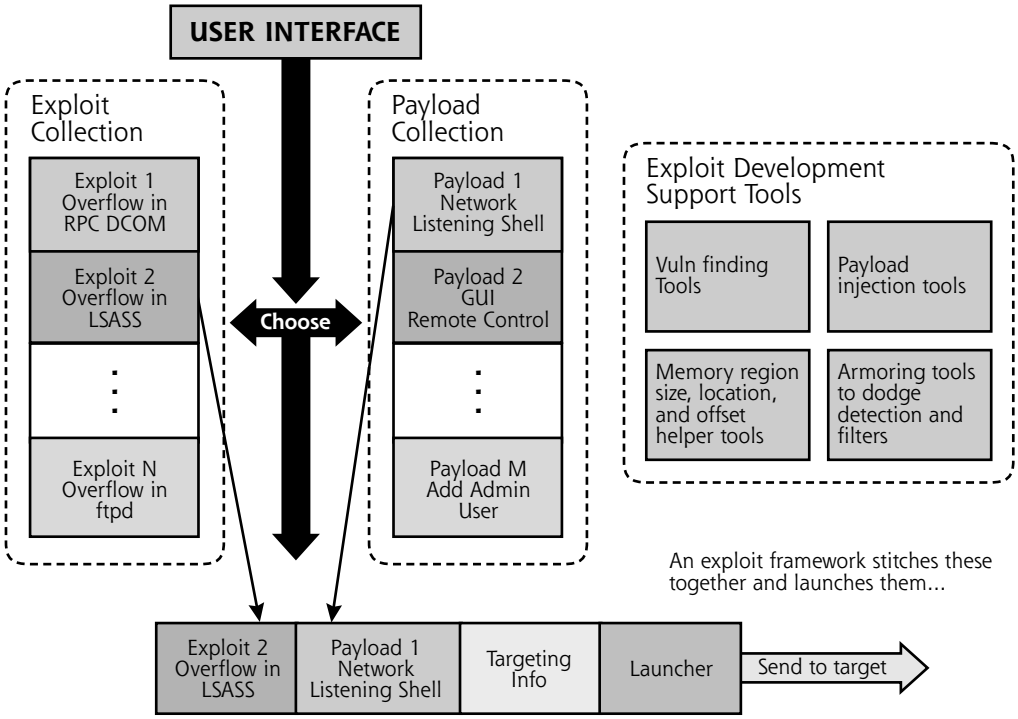


Figure 7.12 The components of Metasploit.

skoudis.book Page 364 Wednesday, November 30, 2005 10:04 PM

NTDLL.DLL used by the Nachi/Welchia worm. The Windows LSASS buffer overflow exploit is a particularly nasty one as well, used by the Sasser worm. There are several other exploits, including some that work against Solaris (the sadmind exploit), Linux (against Real Server on Linux), and many more. It's important to note that the Metasploit framework can attack any type of operating system for which it has exploits and payloads, regardless of the operating system on which Metasploit itself is running. So, for example, Metasploit running on Linux can attack Linux, Windows, and Solaris machines, and possibly many others.

Next, Metasploit offers a huge set of payloads, that is, the code the attacker wants to run on the target machine, triggered by the exploit itself. An attacker using Metasploit can choose from any of the following payloads to foist on a target:

- *Bind shell to current port.* This payload opens a command shell listener on the target machine using the existing TCP connection of a service on the machine. The attacker can then feed commands to the victim system across the network to execute at a command prompt.
- *Bind shell to arbitrary port.* This payload opens a command shell listener on any TCP port of the attacker's choosing on the target system.
- *Reverse shell.* This payload shovels a shell back to the attacker on a TCP port. With this capability, the attacker can force the victim machine to initiate an outbound connection, sent to the attacker, polling the bad guy for commands to be executed on the victim machine. So, if a network or host-based firewall blocks inbound connections to the victim machine, the attacker can still force an outbound connection from the victim to the attacker, getting commands from the attacker for the shell to execute. As we discuss in Chapter 8, Phase 3: Gaining Access Using Network Attacks, the attacker will likely have a Netcat listener waiting to receive the shoveled shell.
- *Windows VNC Server DLL Inject.* This payload allows the attacker to control the GUI of the victim machine remotely, using the Virtual Network Computing (VNC) tool sent as a payload. VNC runs inside the victim process, so it doesn't need to be installed on the victim machine in advance. Instead, it is inserted as a DLL inside the vulnerable program to give the attacker remote control of the machine's screen and keyboard.

skoudis.book Page 365 Wednesday, November 30, 2005 10:04 PM

- *Reverse VNC DLL Inject.* This payload inserts VNC as a DLL inside the running process, and then tells the VNC server to make a connection back to the attacker's machine, in effect shoveling the GUI to the attacker. That way, the victim machine initiates an outbound connection to the attacker, but allows the attacker to control the victim machine.
- *Inject DLL into running application.* This payload injects an arbitrary DLL of the attacker's choosing into the vulnerable process, and creates a thread to run inside that DLL. Thus, the attacker can make any blob of code packaged as a DLL run on the victim.
- *Create Local Admin User.* This payload creates a new user in the administrators group with a name and password specified by the attacker.
- *The Meterpreter.* This general-purpose payload carries a very special DLL to the target box. This DLL implements a simple shell, called the Metasploit Interpreter, or Meterpreter for short, to run commands of the attacker's choosing. However, the Meterpreter isn't just a tool that executes a separate shell process on the target. On the contrary, this new shell runs inside of the vulnerable program's existing process. Its power lies in three aspects. First, the Meterpreter does not create a separate process to execute the shell (such as `cmd.exe` or `/bin/sh` would), but instead runs it inside the exploited process. Thus, there is no separate process for an investigator or curious system administrator to detect. Second, the Meterpreter does not touch the hard drive of the target machine, but instead gives access purely by manipulating memory. Therefore, there is no evidence left in the file system for investigators to locate. Third, if the vulnerable service has been configured to run in a limited environment so that the vulnerable program cannot access certain commands on the target file system (known as a chroot environment), the Meterpreter can still run its built-in commands within the memory of the target machine, regardless of the chroot limitation. Thus, this Meterpreter payload is incredibly valuable for the bad guys.

To support a user in selecting an exploit and payload to launch at a target, Metasploit includes three different user interface options: a command-line tool suitable for scripting, a console prompt with specialized keywords, and even a point-and-click Web interface accessible via a browser. The Web interface, shown in Figure 7.13, is probably the easiest to use of all three, letting the attacker navigate

skoudis.book Page 366 Wednesday, November 30, 2005 10:04 PM

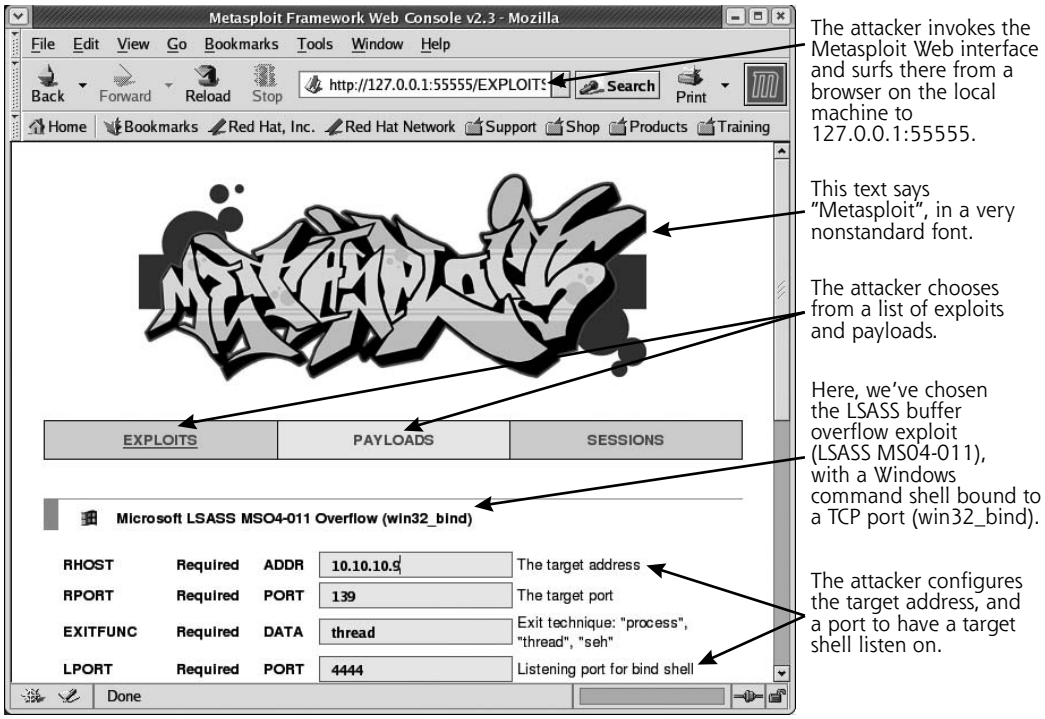


Figure 7.13 Metasploit's Web-based interface.

using a browser to select the components of the attack. However, my favorite Metasploit interface is the console, which includes a specialized language for launching attacks. It's my favorite because it is the most flexible way to attack one system and then rapidly alter the configuration to attack another system, a really useful functionality when performing penetration tests. The Metasploit console includes a nifty lingo with keywords as simple as `use [exploit]`, `set [payload]`, and the very lovely `exploit` command, which launches the attack against a target. In the days before Metasploit, a script kiddie often had to figure out how each individual exploit script should be configured to hit a target, a sometimes difficult process of trial and error. Now, the attacker merely needs to learn a single Metasploit user interface, and can then choose, configure, and launch exploits in a consistent manner.

Metasploit users don't even have to understand how the exploit or payload works. They simply run the user interface, select an appropriate exploit and

skoudis.book Page 367 Wednesday, November 30, 2005 10:04 PM

payload, and then fire the resulting package at the target. The tool bundles the exploit and payload together, applies a targeting header, and launches it across the network. The package arrives at the target, the exploit triggers the payload, and the attacker's chosen code runs on the victim machine. These are the things of which script kiddie dreams are made.

Script kiddies aside, in addition to the exploits and payloads, Metasploit also features a collection of tools to help developers create brand new exploits and payloads. Some of these tools review potentially vulnerable programs to help find buffer overflow and related flaws in the first place. Others help the developer figure out the size, location, and offset of memory regions in the target program that will hold and run the exploit and payload, automating the ABCDEF game we discussed earlier in this chapter. Some of the exploit development support tools include code samples to inject a payload into the target's memory, and still others help armor the resulting exploit and payload to minimize the chance it will be detected or filtered at the target. These pieces make up the partially automated assembly line for the creation of exploits.

And here's the real power of Metasploit: If a developer builds an exploit or payload within the Metasploit framework, it can be used interchangeably with other payloads or exploits as well as the overall exploit framework user interfaces. Using Perl, developers can write and then publish their new modules, and thousands of exploit framework users around the globe can easily import the new building block into their own attacks, relying on the same, consistent interface. Right now, hundreds of developers around the world are coding new exploits and payloads within Metasploit. Some of these people are even releasing their new attack code, created within Metasploit, publicly.

ADVANTAGES FOR ATTACKERS

Exploit frameworks like Metasploit offer significant advantages for the bad guys, including those who craft their own custom exploits and even the script kiddies just looking for low-hanging fruit. For the former, exploit frameworks shorten the time needed to craft a new exploit and make the task a lot easier. In the good old days of the 1990s, we often had many months after finding out about a new vulnerability before an exploit was released in the wild. Now, increasingly, we

skoudis.book Page 368 Wednesday, November 30, 2005 10:04 PM

have only a couple of days before a sploit is publicly unleashed. Exploit frameworks are helping to fuel that shorter duration. As exploit frameworks are further refined, this time frame could shrink even more. Some researchers are working on further automating the reverse engineering of security patches to create an exploit for a framework within a matter of hours or minutes after a new patch or flaw is discovered and announced. Because of these trends, we need to patch more diligently than ever before.

Furthermore, while shortening development time and effort, exploit frameworks like Metasploit have simultaneously increased the quality of exploit code, making the bad guys much more lethal. Unlike the handcrafted, individual exploit scripts of the past, the sploits written in an exploit framework are built on top of time-tested, interchangeable modules. Some seriously gifted exploit engineers created these underlying modules and have carefully refined their stuff to make sure it works reliably. Thus, an attacker firing an exploit at a target can be much more assured of a successful compromise.

At the SANS Institute's Internet Storm Center (*http://isc.sans.org*), when a new vulnerability is announced, we often see widespread port scanning for the vulnerable service begin immediately, even before an exploit is released publicly. Developers who have already quickly created an exploit might cause some of this scanning, but a lot of it is likely due to anticipatory scanning. That is, even script kiddie attackers know that an exploit will likely soon be created and released for a choice vulnerability, so they want an inventory of juicy targets as fast as possible. When the exploit is then actually released, they pounce. Today, quite often, the exploit is released as part of an exploit framework first.

BENEFITS FOR THE GOOD GUYS, TOO?

Exploit frameworks aren't just evil. Tools like Metasploit can also help us security professionals to improve our practices as well. One of the most valuable aspects of these tools to infosec pros involves minimizing the glut of false positives from our vulnerability-scanning tools. Chief Information Security Officers (CISOs) and auditors often lament the fact that many of the high-risk findings discovered by a vulnerability scanner turn out to be mere fantasies, an error in the tool that thinks a system is vulnerable when it really isn't. Such false positives sometimes comprise

skoudis.book Page 369 Wednesday, November 30, 2005 10:04 PM

30 to 50 percent or more of the findings of an assessment. When a CISO turns such an erroneous report over to an operations team of system administrators to fix the nonexistent problems, not only does the operations team waste valuable resources, but the CISO could lose face in light of these false reports. Getting the ops team to do the right thing in tightening and patching systems is difficult enough, and it only gets harder if you are wrong about half of the vulnerability information you send them in this boy-who-cried-wolf situation.

Metasploit can help alleviate this concern. The assessment team first runs a vulnerability scanner and generates a report. Then, for each of the vulnerabilities identified, the team runs an exploit framework like Metasploit to verify the presence of the flaw. The Metasploit framework can give a really high degree of certainty that the vulnerability is present, because it lets the tester gain access to the target machine. Real problems can then be given high priority for fixing. Although this high degree of certainty is invaluable, it's important to note that some exploits inside of the frameworks still could cause a target system or service to crash. Therefore, be careful when running such tools, and make sure the operations team is on standby to restart a service if the exploit does indeed crash it.

In addition to improving the accuracy of security assessments, exploit frameworks can help us check our IDS and IPS tools' functionality. Occasionally, an IDS or IPS might seem especially quiet. Although a given sensor might normally generate a dozen alerts or more per day, sometimes you might have an extremely quiet day, with no alerts coming in over a long span of time. When this happens, many IDS and IPS analysts start to get a little nervous, worrying that their monitoring devices are dead, misconfigured, or simply not accessible on the network. Compounding the concern, we might soon face attacks involving more sophisticated bad guys launching exploits that actually bring down our IDS and IPS tools, in effect rendering our sensor capabilities blind. The most insidious exploits would disable the IDS and IPS detection functionality while putting the system in an endless loop, making them appear to be just fine, yet blind to any actual attacks. To help make sure your IDS and IPS tools are running properly, consider using an exploit framework to fire some sploits at them on a periodic basis, such as once per day. Sure, you could run a vulnerability-scanning tool against a target network to test your detection capabilities, but that would trigger

skoudis.book Page 370 Wednesday, November 30, 2005 10:04 PM

an avalanche of alerts. A single sploit will tell you if your detector is still running properly without driving your analysis team batty.

One of the most common and obvious ways the good guys use exploit frameworks is to enhance their penetration testing activities. With a comprehensive and constantly updated set of exploits and payloads, a penetration tester can focus more on the overall orchestration of an attack and analyzing results instead of spending exorbitant amounts of time researching, reviewing, and tweaking individual exploits. Furthermore, for those penetration testers who devise their own exploit code and payloads, the frameworks offer an excellent development environment. Exploit frameworks don't completely automate penetration test exercises, though. An experienced hand still needs to plan the test, launch various tools including the exploit framework, correlate tool output, analyze results, and iterate to go deeper into the targets. Still, if you perform penetration testing in-house, your team could significantly benefit from these tools, performing more comprehensive tests in less time. If you rely on an external penetration testing company, ask them which of the various exploit frameworks they use, and how they apply them in their testing regimen to improve their attacks and lower costs.

One final benefit offered by exploit frameworks should not be overlooked—improving management awareness of the importance of good security practices. Most security pros have to work really hard to make sure management understands the security risks our organizations face, emphasizing the need for system hardening, thorough patching, and solid incident response capabilities. Sometimes, management's eyes glaze over hearing for the umpteenth time the importance of these practices. Yet, a single sploit is often worth more than a thousand words. Set up a laboratory demo of one of the exploit frameworks, such as Metasploit. Build a target system that lacks a crucial patch for a given exploit in the framework, and load a sample text file on the target machine with the contents "Please don't steal this important file!" Pick a very reliable exploit to demonstrate. Then, after you've tested your demo to make sure it works, invite management to watch how easy it is for an attacker to use the point-and-click Web interface of Metasploit to compromise the target. Snag a copy of the sensitive file and display it to your observers. When first exposed to these tools, some managers' jaws drop at their power and simplicity. As the scales fall from their

skoudis.book Page 371 Wednesday, November 30, 2005 10:04 PM

eyes, your plea for adequate security resources might now reach a far more receptive audience, thanks to your trusty exploit framework.

BUFFER OVERFLOW ATTACK DEFENSES

There are a variety of ways to protect your systems from buffer overflow attacks and related exploits. 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

So what can a system administrator or security professional do to prevent 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 researchers are constantly discovering new vulnerabilities. Vendors are scrambling to create fixes for these holes. You must have a regular program that monitors various mailing lists, such as the Bugtraq, US-CERT, and the SANS mailing lists we discuss in more detail in Chapter 13, The Future, References, and Conclusions. 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 program for testing newly patched systems and rolling them into

skoudis.book Page 372 Wednesday, November 30, 2005 10:04 PM

production. You cannot just apply a vendor's security fix to a production system without trying it 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.

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 e-mail. If attackers discover a vulnerability that they can exploit over this incoming path, they might be able to use it to send an outgoing connection that gives them even greater access, the so-called shell shoveling technique we briefly discussed with Metasploit in this chapter and go into more detail when we discuss Netcat in the next chapter. To avoid this problem of reverse shells, you need to apply strict filters to allow outgoing traffic only for services with a defined business need. Sure, your users might require outgoing HTTP or FTP, but do they really need outgoing X Window System access? Probably not. You should block unneeded services at external firewalls and routers.

A final defense against 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, including heap-based overflows and return-to-libc attacks, but the vast majority of stack-based buffer overflows fail if they cannot execute instructions from the stack. Solaris and HP-UX 11i have built-in nonexecutable system stack functionality, but the system has to be configured to use this capability. 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:

skoudis.book Page 373 Wednesday, November 30, 2005 10:04 PM


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

Similarly, in HP-UX 11i, an administrator must set the kernel tunable parameter `executable_stack` to zero.

The mainstream Linux kernel does not have built-in nonexecutable system stack functionality, but separate tools can be downloaded to give a Linux machine such functionality. To configure a Linux system with a nonexecutable stack, you'll have to apply a kernel patch. Solar Designer, a brilliant individual we 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*. Other tweaks of the Linux kernel, including PaX (*http://pax.grsecurity.net*), also alter the way the stack functions to minimize the chance of successful buffer overflow exploitation.

Unfortunately, Windows 2000 does not currently support nonexecutable stack or heap capabilities. Currently, Microsoft has added this functionality to Windows XP Service Pack 2 and Windows 2003 Service Pack 1, a feature they call Data Execution Prevention (DEP). This capability marks certain pages in memory, such as the stack and heap, as nonexecutable.

There are two kinds of DEP supported in Windows XP Service Pack 2 and Windows 2003 Service Pack 1: hardware-based DEP and software-based DEP. The hardware-based DEP feature works only on machines with processors that support execution protection technology (a feature advertised as NX capability, for nonexecution), a special setting in the CPU that refuses to execute memory segments that are only supposed to hold data, such as the stack and heap. Some of the more recent CPU products include NX functionality.

The software-based DEP, on the other hand, works on any kind of processor Windows runs on. It is activated by default in Windows XP Service Pack 2 and Windows 2003 Service Pack 1 for essential Windows programs and services, those elements of the operating system itself that so often come under attack. An administrator can increase this level of security to protect all programs and services on the machine, but this might impact backward compatibility with some

skoudis.book Page 374 Wednesday, November 30, 2005 10:04 PM

specific programs that do attempt to run code from the stack or heap, an unusual occurrence for most programs. If you do have a few of these strange beasts, you could even set up DEP for all programs except a list of specific programs that you expect to run data from the stack or heap, such as unusual debuggers and programs that automatically alter their own code. You can look at your DEP settings on Windows XP Service Pack 2 and Windows 2003 Service Pack 1 by going to Start ➠ Settings ➠ Control Panel ➠ System ➠ Advanced. Then, under Performance, click Settings and go to Data Execution Prevention to see the user interface shown in Figure 7.14.

This software-based DEP is currently an active area of research within the computer underground, as it has not been thoroughly documented by Microsoft. Attackers are trying to reverse engineer it to see if it can be foiled. Interestingly, a group of security researchers out of Russia released a white paper describing how to attack the software-based DEP function using a heap overflow by carefully

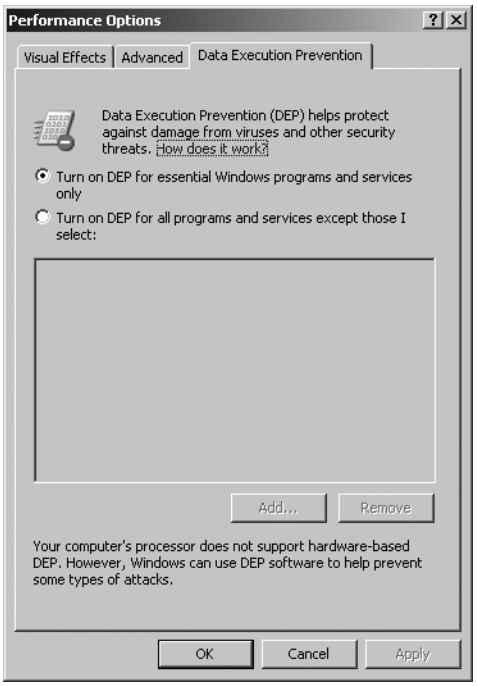


Figure 7.14 Windows XP Service Pack 2 and Windows 2003 Service Pack 1 Data Execution Prevention.

skoudis.book Page 375 Wednesday, November 30, 2005 10:04 PM

re-creating the data structures that DEP employs within the heap to protect it. The white paper is an amazing read, and can be found at *www.maxpatrol.com/defeating-xpsp2-heap-protection.htm*.

Buffer Overflow Defenses for Software Developers

Although system administrators and security personnel can certainly do a lot to prevent 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, 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 the `gets` function we discussed earlier. The following free tools accept regular C and C++ source code as input, to which they apply heuristic searches looking for common security flaws including buffer overflows:

- ITS4 (which stands for It's the Software, Stupid—Security Scanner), available at *www.cigital.com/its4/*
- RATS (Rough Auditing Tool for Security), available at *www.securesw.com/rats/*
- Flawfinder, available at *www.dwheeler.com/flawfinder*

Additionally, help educate your software developers by encouraging them to read about secure programming. Some of my favorite resources for secure coding on a Windows platform include the book *Writing Secure Code 2* by Howard and Leblanc (Microsoft Press, 2002). For those who develop on a Linux and UNIX platform, you can get a great, free white paper on developing secure code on Linux and UNIX from Dave Wheeler's Web site (*www.dwheeler.com/secure-programs*).

skoudis.book Page 376 Wednesday, November 30, 2005 10:04 PM

Download this and give it to your software development team. Print it out, put a big red bow on it, and you've got a free gift for someone!

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 *http://immunix.org,* and 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 is essentially a hash of the current return pointer and a secret known by the system. The canary operates much like its namesakes, which were used by coal miners in the past. In a coalmine, 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 stops execution of the program, thereby foiling a buffer overflow attack. When a function call finishes, the operating system first rehashes the return pointer with its special secret. If the hashed return pointer and secret match the canary value, the program returns from the function call normally. If they do not match, the canary, return pointer, or both have been altered. The program then crashes gracefully. In most circumstances, it is far better to crash gracefully than to execute code of an attacker's choosing on the machine.

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

skoudis.book Page 377 Wednesday, November 30, 2005 10:04 PM

Microsoft also added canary functionality to prevent the alteration of return pointers in the Windows 2003 stack. This feature, which is built in and turned on by default, does not require any activation or configuration by a system administrator. That's the good news. Unfortunately, security researchers have discovered techniques for thwarting this canary. In particular, researcher David Litchfield has developed some techniques for inserting code that makes it look like the canary is intact, even though it has been altered, in effect tricking the system into running the attacker's code. This technique is described in detail at *www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf.*

Although none of the techniques discussed in this section for preventing buffer overflows is completely foolproof, the techniques 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 health care 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 hundreds or thousands of passwords, 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 many users have dozens of passwords. Users are forced to remember and maintain passwords for logging into the network, signing on to numerous applications, accessing frequently used external Web sites, logging into voice mail, and even making long-distance phone 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 foothold

skoudis.book Page 378 Wednesday, November 30, 2005 10:04 PM

on a system. Most users manually synchronize their passwords for every password-protected system they access. Sadly, therefore, a user who has a password in your high-security environment might be using the same password for that external e-commerce application over which your organization has no control. After guessing one weak password in the low-security environment, the attacker can take over an account on the supposedly higher security system. Indeed, the plague of passwords is quite widespread.

Why, then, do we continue to rely on them so much? We do so because password-authentication mechanisms are really cheap. Most operating systems and applications have built-in password authentication, so their users and administrators have simply applied the least expensive (and often least secure) tool in place.

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

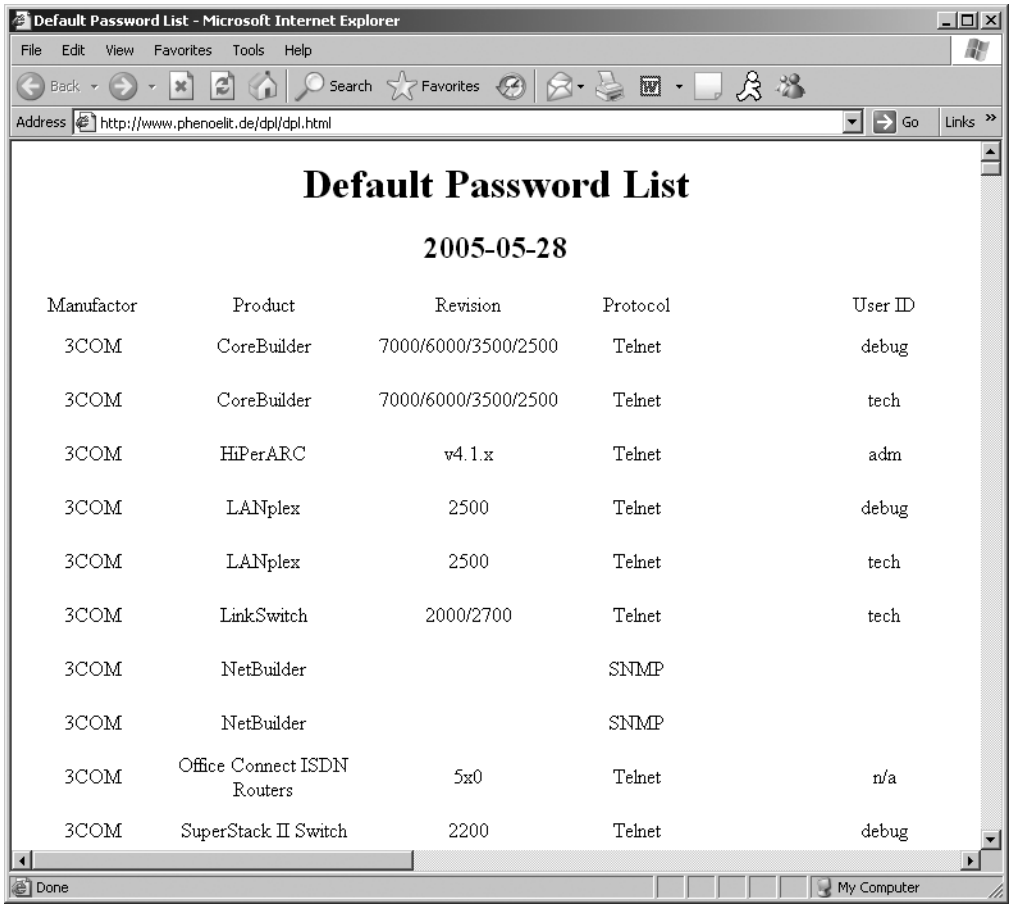
GUESSING DEFAULT PASSWORDS

Many applications and operating systems include built-in default passwords established by the vendor. Often, overworked, uninformed, or lazy administrators fail to remove default passwords from systems. Attackers can quickly and easily guess these default passwords to gain access to the target. The Phenoelit hacking group out of Germany maintains a huge database of default passwords for a variety of platforms, available at *www.phenoelit.de/dpl/dpl.html*. This Web site, shown in Figure 7.15, includes default passwords for systems ranging from 3COM switches to Zyxel's modem routers, and everything in between.

Password Guessing Through Login Attacks

What if none of the default passwords works? Another technique for guessing weak passwords is to run a tool that repeatedly tries to log in to the target system across the network, guessing password after password. The attacker configures a password-guessing tool with a common or known user ID on the target system. The password-guessing tool then guesses a password, perhaps using a wordlist

skoudis.book Page 379 Wednesday, November 30, 2005 10:04 PM



Default Password List

2005-05-28

Manufactor	Product	Revision	Protocol	User ID
3COM	CoreBuilder	7000/6000/3500/2500	Telnet	debug
3COM	CoreBuilder	7000/6000/3500/2500	Telnet	tech
3COM	HiPerARC	v4.1.x	Telnet	adm
3COM	LANplex	2500	Telnet	debug
3COM	LANplex	2500	Telnet	tech
3COM	LinkSwitch	2000/2700	Telnet	tech
3COM	NetBuilder		SNMP	
3COM	NetBuilder		SNMP	
3COM	Office Connect ISDN Routers	5x0	Telnet	n/a
3COM	SuperStack II Switch	2200	Telnet	debug

Figure 7.15 An online database of default passwords.

from a dictionary. The attacker points the tool at the target machine, which might have a command-line login prompt, Web front-end login dialog box, or other method of requesting a password. The attacker's tool transmits its user ID and password guess to the target, trying to log in, and then automatically determines if the guess was successful. If not, another guess is tried. Guess after guess is launched until the tool discovers a valid password.

One of the most fully functional and easy-to-use tools for automating this password-guessing attack is Brutus, available for free at *www.hoobie.net/brutus.* It runs on

skoudis.book Page 380 Wednesday, November 30, 2005 10:04 PM

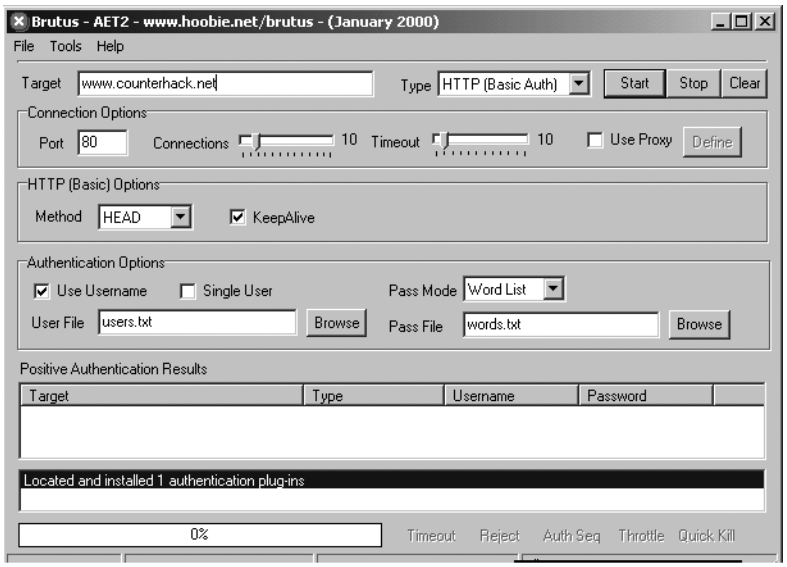


Figure 7.16 Brutus in action.

Windows, has a point-and-click GUI, shown in Figure 7.16, and is remarkably effective.

The attacker configures Brutus with the following information:

- The target system address or domain name
- The source of password guesses, which can be a file of words or a brute-force selection of all possible character combinations
- The protocol to use when interacting with the target, which could be HTTP with Basic Authentication, HTTP with an HTML form, Post Office Protocol 3 (POP3) e-mail, FTP, Windows authentication and file sharing with Server Message Block (SMB) protocol, and Telnet
- The text that Brutus will receive if authentication is successful
- The text the application generates when authentication fails

Then, the attacker simply clicks the Start button. Brutus grinds away for between minutes and weeks, and starts popping back with answers.

skoudis.book Page 381 Wednesday, November 30, 2005 10:04 PM

It's important to note that Brutus often yields many false positives due to bugs in the code, not problems with this overall type of attack. Keep that in mind if you ever run Brutus: Not all of your discovered accounts will be accurate!

If you want a more UNIX/Linux-friendly password-guessing tool with better accuracy, you should check out THC Hydra, available for free at *http://thc.org/thc-hydra*. This fine tool, written by van Hauser, includes a command-line interface and a GUI option if you really want it. Hydra runs on Linux and many flavors of UNIX, and even works on Windows, provided that you've installed the free Cygwin environment, an amazing UNIX-like world that runs on top of Windows. You can get the Cygwin environment for free at *www.cygwin.com*.

The nicest part about Hydra is its generous protocol support. It can guess passwords for more than a dozen different application-level protocols, including Telnet, FTP, HTTP, HTTPS, HTTP-PROXY, LDAP, SMB, SMBNT, MS-SQL, MYSQL, REXEC, SOCKS5, VNC, POP3, IMAP, NNTP, PCNFS, ICQ, SAP/R3, Cisco auth, Cisco enable, and Cisco AAA. Whew! That's a lot of different applications, making it highly useful in password-guessing attacks. Also, Hydra doesn't suffer from the false positive problems of Brutus, making it my personal favorite for password guessing.

Password guessing can be a slow process. Each login attempt could take a few seconds. To go through an entire 40,000-word dictionary could take 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, this password-guessing technique has additional limitations. The constant attempts to log in to the target generate a significant amount of regular network traffic and log activity, which could easily be noticed by a diligent system administrator or an IDS. An additional challenge an attacker faces when trying to guess a password 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

skoudis.book Page 382 Wednesday, November 30, 2005 10:04 PM

at least slowed down significantly. Account lockout features are a good idea in preventing password-guessing attacks. However, with account lockout in place, an attacker could conduct a DoS attack by purposely locking out all of your accounts using a script, so be careful to fine-tune your account lockout policies based on the threats you face.

THE ART AND SCIENCE OF PASSWORD CRACKING

Guessing default passwords usually doesn't work, because many administrators change the defaults. Password guessing with an automated tool could take a very long time, and, at its worst, it could get an attacker detected or lock out accounts. A much more sophisticated approach to determining passwords that avoids these problems is password cracking, an approach totally separate from password guessing. However, to analyze how password cracking works, you first need to understand how passwords are stored on most systems.

When you log in to most machines, whether they are Linux systems, Windows boxes, Novell servers, Cisco routers, or any other type of machines, you typically provide a user ID 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. The computer could base this decision on the contents of a local file of the passwords for all users, comparing the password you just typed in with your password in the file. Unfortunately, a file with every user's password in clear text would be an incredible security liability, a sitting duck waiting for the bad guys to harvest it. An attacker gaining access to such a password file would be able to log in as any user of the system.

System designers, realizing this dilemma of requiring a list of passwords to compare to for user login without having a huge security hole, decided to solve the problem by applying cryptographic techniques to protect each password in the password file. Thus, for most systems, the password file contains a list of user IDs 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

skoudis.book Page 383 Wednesday, November 30, 2005 10:04 PM

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 log in to the system, the machine gathers the password from the user, applies the same cryptographic transformation used to generate the password file, and compares the results. If the encrypted or hashed value of your password matches the encrypted or hashed value in the file, you are allowed to log in. Otherwise, you are denied access. The process works beautifully, allowing you to log in successfully, turning away attackers, and never keeping a clear text file of password.

LET'S CRACK THOSE PASSWORDS!

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

Most systems include a password file that contains encrypted or hashed representations of the passwords. Password cracking involves stealing the encrypted password representations and trying to recover the original clear text password using an automated tool. A password-cracking tool operates by setting up a simple loop, as shown in Figure 7.17.

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

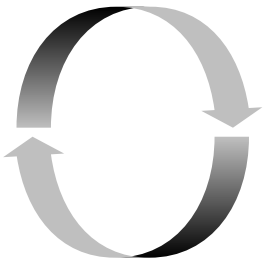


Figure 7.17 Password cracking is really just a loop.

skoudis.book Page 384 Wednesday, November 30, 2005 10:04 PM

available online, in many languages, including English, Russian, Japanese, French, and, for you *Star Trek* fans, even Klingon! Most password-cracking tools come with a small but effective wordlist. For example, John the Ripper's list includes approximately 2,000 words, whereas the Cain wordlist includes a whopping 306,000 entries!

For other wordlists that are quite effective, check out two sources: the CERIAS wordlist collection (*http://ftp.cerias.purdue.edu/pub/dict/dictionaries/*), and the Moby wordlist (*www.dcs.shef.ac.uk/research/ilash/Moby/*). Both lists are free, and include hundreds of thousands of words from a variety of languages. 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 guesses every possible combination of characters to determine the password. The tool might start with alphanumeric characters (a–z and 0–9), and then progress to special characters (!@#$, and so on). Even for a fast password-cracking tool, this brute-force guessing process can take an enormous amount of time, ranging from hours to centuries. It all depends on the strength of the password crypto algorithm and how difficult the user's password is to guess.

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 starts guessing passwords using a dictionary term. Then, it creates 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. The best hybrid generators even start to shave characters off of dictionary terms in their guess-creating algorithms.

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 or hashed password file, they can run the password-cracking tool on their own systems, in the comfort of their own homes or on any other machine that suits their fancy. This makes password cracking much faster than password guessing

skoudis.book Page 385 Wednesday, November 30, 2005 10:04 PM

through trying to log in to the target machine. Although using a password-guessing tool to log in across the network requires many valuable seconds to evaluate each guess, a password-cracking tool can guess thousands or tens of thousands of passwords per 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 few minutes.

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 password-cracking virtual supercomputer. Or, if an attacker has compromised 100,000 machines using a bot for remote control of these victims, the attacker can harvest the processing power of a 100,000-node network to make the password cracking operation really fly! We discuss the nefarious bots that can support such a feat in more detail in Chapter 10, Phase 4: Maintaining Access.

Password-cracking tools have been around for a couple of decades, and an enormous number of them are available. Some of the most notable password-cracking tools in widespread use today include the following:

- Cain, a fantastic free tool available from Massimiliano Montoro at *www.oxid.it/cain.html*
- John the Ripper, a powerful free password cracker for UNIX/Linux and some Windows passwords, written by Solar Designer, available at *www.openwall.com/john*
- Pandora, a tool for testing Novell Netware, including password cracking, written by Simple Nomad, and available at *www.nmrc.org/project/pandora*
- LC5, the latest incarnation of the venerable L0phtCrack password cracker, an easy-to-use but rather expensive commercial password cracker at *www.atstake.com/products/lc/purchase.html*

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

skoudis.book Page 386 Wednesday, November 30, 2005 10:04 PM

Cain and Abel: Cracking Windows (and Other) Passwords with a Beautiful GUI

Cain and Abel are a dynamic duo of security tools that can be used for either attacking systems or administering them. Their name is a nod to the biblical brothers who didn't get along all that well. The Cain and Abel tools, happily, work together far better than those ancient brothers ever did. Typically, a user relies on Cain to gather information about systems and to manipulate them directly, while Abel usually runs as a background process a user can access remotely to dump information about a target environment. In other words, Cain is highly interactive, with a fancy GUI offering all kinds of interesting attack functionality. Abel runs in the background, and can be remotely accessed to dump data from its host system.

Frankly, the Cain and Abel pair of tools is hard to categorize. This amazing software contraption, created by Massimiliano Montoro, includes more than a dozen different useful capabilities that we discuss throughout this book. Although we're covering Cain and Abel here in the section on password cracking, Cain and Abel are not designed just for cracking passwords. They are extremely feature rich, including just about everything and the kitchen sink, as a final touch! Montoro constantly scours the Internet for useful ideas included in white papers and other tools, and then adds such capabilities to Cain and Abel, making the duo a powerful collection of various computer attack widgets. Cain includes the following functionalities:

- Automated WLAN discovery, in essence a war-driving tool that looks quite similar to NetStumbler, the tool we discussed in Chapter 6.
- A GUI-based traceroute tool, using the same traceroute techniques we discussed in Chapter 6 in the context of the traceroute, tracert, and Cheops-ng tools.
- A sniffer for capturing interesting packets from a LAN, including a variety of user IDs and passwords for several protocols. We discuss sniffers in more detail in Chapter 8.
- A hash calculator, which takes input text and calculates its MD2, MD4, MD5, SHA-1, SHA-2, and RIPEMD-160 hashes, as well as the Microsoft LM, Windows NT, MySQL, and PIX password representation of that text. That way, an attacker can quickly verify assumptions associated with specific information discovered on a target system.

skoudis.book Page 387 Wednesday, November 30, 2005 10:04 PM

- A network neighborhood exploration tool to scan for and find interesting Windows servers available on the network.
- A tool to dump and reveal all encrypted or hashed passwords cached on the local machine, including the standard Windows LM and NT password representations, as well as the application-specific passwords for Microsoft Outlook, Outlook Express, Outlook Express Identities, Outlook 2002, Microsoft Internet Explorer, and MSN Explorer.
- An ARP cache poisoning tool, which can be used to redirect traffic on a LAN so that an attacker can more easily sniff in a switched environment, a technique we discuss in more detail in Chapter 8.
- A remote promiscuous mode checker, to try to test whether a given target machine is running a sniffer that places the network interface in promiscuous mode.
- Numerous other features, with new functionality added on a fairly regular basis.

Cain integrates each of these functions into a nice GUI, which, although complex, sorts out the individual features quite nicely. The Abel tool, on the other hand, has no GUI. Instead, it runs as a service in the background, giving remote access capabilities to a lot of functionality, including the following:

- A remote command shell, rather like the backdoor command shells we discuss in Chapter 10.
- A remote route table manager, so an administrator can tweak the packet routing rules on a Windows machine.
- A remote TCP/UDP port viewer that lists local ports listening on the system running Abel, rather like the Active Ports and TCPView tools we discussed in Chapter 6.
- A remote Windows password hash dumper, which an attacker can use to retrieve the encrypted and hashed Windows password representations from the Security Accounts Manager (SAM) database, suitable for cracking by … you guessed it … the Cain tool.

In this section, however, we're going to focus on one of the most useful capabilities of Cain, namely its extremely functional password cracker. Cain is able to

skoudis.book Page 388 Wednesday, November 30, 2005 10:04 PM

crack passwords for more than a dozen different operating system and protocol types. Just for the Windows operating system, Cain can crack the following password representations:

- Microsoft LM, the really weak Windows password authentication also known as LanMan, still included by default in all Windows NT, 2000, XP, and 2003 systems in the local SAM database
- The LM challenge passed across the network, which is a challenge–response authentication protocol based on the underlying LM hash, but includes special features for network authentication to a Windows domain or a file server
- Windows NT hash, a form of Windows password storage stronger than LM, supported in Windows NT, 2000, XP, and 2003 machines and stored in the local SAM database, as we discussed in Chapter 4, Windows NT/2000/XP/2003 Overview.
- NTLMv1, a challenge–response protocol passed across the network, offering slightly better security than the LM challenge passed across the network
- NTLMv2, an even stronger form of challenge–response authentication across a Windows network
- MS-Kerberos5 Pre-Auth, the Microsoft Kerberos authentication deployed in some Windows environments

RETRIEVING THE PASSWORD REPRESENTATIONS FROM WINDOWS To use Cain to crack Windows operating system passwords, the attacker usually first grabs a copy of the password representations stored in the SAM database of the target machine. To accomplish this, Cain includes a built-in feature to dump 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. These administrator rights are required because the password dump function must attach to the running Windows authentication processes to extract the SAM database right from their memory space, a process that requires administrative privileges. It's interesting to note that dumping the SAM database from memory allows Cain to bypass Windows Syskey protection, which adds an extra 128 bits of cryptographic protection around the SAM database while it resides on the hard drive only. When in the memory of running authentication processes, Cain can easily grab it with

skoudis.book Page 389 Wednesday, November 30, 2005 10:04 PM

administrative rights. Besides Cain, an alternative program for getting these password representations using the same memory-dumping technique is the free Pwdump3 program, available at *www.openwall.com/passwords/nt.shtml.* As with Cain, to use Pwdump3 to extract password representations, the attacker must have administrative privileges on the target system.

Besides dumping the SAM, 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 an administrator backs up a system using the built-in Windows tool Ntbackup.exe, by default, a copy of the SAM database with the password representations is usually placed in the `%systemroot%\repair\sam._` file.

Another option for getting the password representations is to steal the administrator recovery floppy disks. When a Windows 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 CD-ROM and retrieve the SAM database by dumping it from the local registry image on the hard drive. A handy tool for retrieving and altering Windows passwords using a Linux boot disk can be found at *http://home.eunet.no/~pnordahl/ntpasswd/bootdisk.html.* This tool can be used to change the administrator or other user's password, even if Syskey is installed. It's important to note, however, that changing a user's password by booting to a Linux CD-ROM causes the system to lose access to the EFS keys for that user on Windows XP and 2003. Thus, on those versions of Windows, if you use the password-changing boot disk, you'll lose all EFS-protected data in the accounts for which you change passwords. On Windows 2000, the EFS keys are stored differently, letting this Linux boot disk change the passwords without losing EFS-encrypted files.

Cain offers one final option for getting password representations: sniffing them off of the network. Cain includes a very powerful integrated network capture tool that monitors the LAN looking for Windows challenge–response authentication packets, which Windows will send in a variety of different formats, depending on

skoudis.book Page 390 Wednesday, November 30, 2005 10:04 PM

its configuration, including LM Challenge–Response, NTLMv1, NTLMv2, and Microsoft Kerberos. Whenever users try to authenticate to a domain or mount a remote file share or print server, their Windows machine authenticates to the server using one of these protocols. Taken together, the challenge and response associated with each protocol are based cryptographically on the user's password. After grabbing the challenge and response from the network using its integrated sniffing tool, Cain can crack them to determine the user's password. We discuss sniffers and how they manipulate LAN traffic in more detail in Chapter 8. But for now, keep in mind that Cain can sniff a variety of Windows authentication protocols and crack the passwords associated with them.

CONFIGURING CAIN Cain is very easy to configure, as shown in Figure 7.18. The attacker can set up the tool to do dictionary attacks (using any wordlist of the attacker's choosing as a dictionary, or the integrated 306,000-word dictionary Cain includes). Cain also supports hybrid attacks that reverse dictionary guesses, apply mixed case to guesses, and even append the numbers 00 through 99 to dictionary

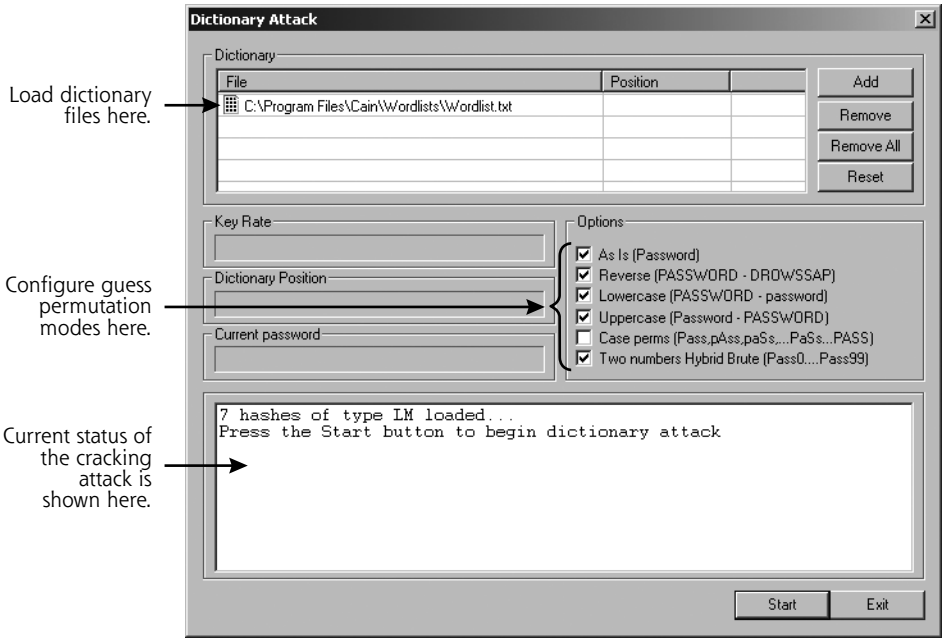


Figure 7.18 Configuration options for Cain.

skoudis.book Page 391 Wednesday, November 30, 2005 10:04 PM

words. It also offers complete brute-force password-cracking attacks, attempting all possible character combinations to form password guesses.

Finally, instead of forming, encrypting, and comparing the password guesses in real time, Cain supports a password-cracking concept sometimes called Rainbow tables, in honor of the first tool that implemented this attack, RainbowCrack, by Zhu Shuanglei. With a Rainbow-like attack, the bad guy computes an encrypted dictionary in advance, storing each password along with its encrypted form in memory or in a file on the hard drive. This table is typically indexed for fast searching based on the encrypted password representation. Then, when mounting a password-cracking attack, the bad guy bypasses the guess–encrypt–compare cycle, instead just grabbing the cryptographic password representation from the victim machine and looking it up in the Rainbow table. After spending the initial time and energy to create the Rainbow tables, all subsequent cracking is much quicker, because the tool simply has to look up the password representations in the table. In effect, we preload most of the password-cracking work. For Cain, the attacker can generate the Rainbow tables using a separate tool called Winrtgen.exe, available at the Cain Web site (*www.oxid.it*). Then, once the encrypted wordlist is developed, the attacker can point Cain to it to perform the comparisons to determine the passwords.

CRACKING PASSWORDS WITH CAIN After loading the password representations, selecting a dictionary, and configuring the options, the attacker can run Cain by clicking the Start button. Cain generates and tests guesses for passwords very quickly. Table 7.1 depicts the amount of time necessary to crack the very weak LM hashes using a quad-processor 2.4-GHz machine, a pricy machine, but not out of range for some attackers. Of course, with Moore's law resulting in faster computers every other year, these numbers are plummeting. Keep in mind, however, that Table 7.1 illustrates the times for LM hash cracking. NT hashes are several orders of magnitude stronger than the incredibly weak LM hashes, for reasons described in Chapter 4.

That's pretty impressive performance! A full brute-force attack (every possible keystroke character) against the weak LM representations takes less than 120 hours, or 5 days, to recover any password, regardless of its value of normal alphanumeric and special characters (those that you can form using the SHIFT key).

skoudis.book Page 392 Wednesday, November 30, 2005 10:04 PM

Table 7.1 Approximate LM Cracking Times with Cain, Using a Quad-Processor Machine

Character Set	Time
Alphanumeric	< 2 hours
Alphanumeric, some special characters	< 10 hours
Alphanumeric, all special characters (except high-end ASCII typed with the ALT key)	< 120 hours (5 days!)

And, if the attacker has more processing horsepower, the attack requires even less time. It's important to note, though, that Windows allows users to choose passwords that include the upper-end ASCII characters by holding down the ALT key and typing numbers to represent such characters. Although these ALT characters significantly drive up password cracking times, most users don't rely on them, instead favoring the easier-to-type alphanumeric and special characters.

The main Cain screen, illustrated in Figure 7.19, shows the information dumped from the target's SAM database (including User Name, LM representation, and NT Hash). As Cain runs, each successfully cracked password is highlighted in the display. There is one especially interesting column in Figure 7.19: the "<8" notation. This column is checked for each password with an LM representation that ends in AAD3B43… That's because, as we discussed in Chapter 4, the original password was seven characters or less, padded to be exactly 14 characters by the LM algorithm. When LM splits the resulting string into two seven-character pieces, the high end will always be entirely padding. Encrypted padding, with no salts, always has the same value, AAD3B43 and so on. Salts, those little random numbers used to boost the difficulty of cracking passwords, are described in more detail in Chapter 4. Of course, if Windows used salts to force some nonpredictability into the password crypto scheme, the same encrypted padding would indeed have different results. So, the presence of this "<8" column illustrates two things: that the passwords are split into two seven-character pieces by LM, and that no salts are used in Windows.

USING CAIN'S INTEGRATED SNIFFER As we discussed earlier, Cain allows an attacker to sniff challenge–response information off of the network for cracking.

skoudis.book Page 393 Wednesday, November 30, 2005 10:04 PM

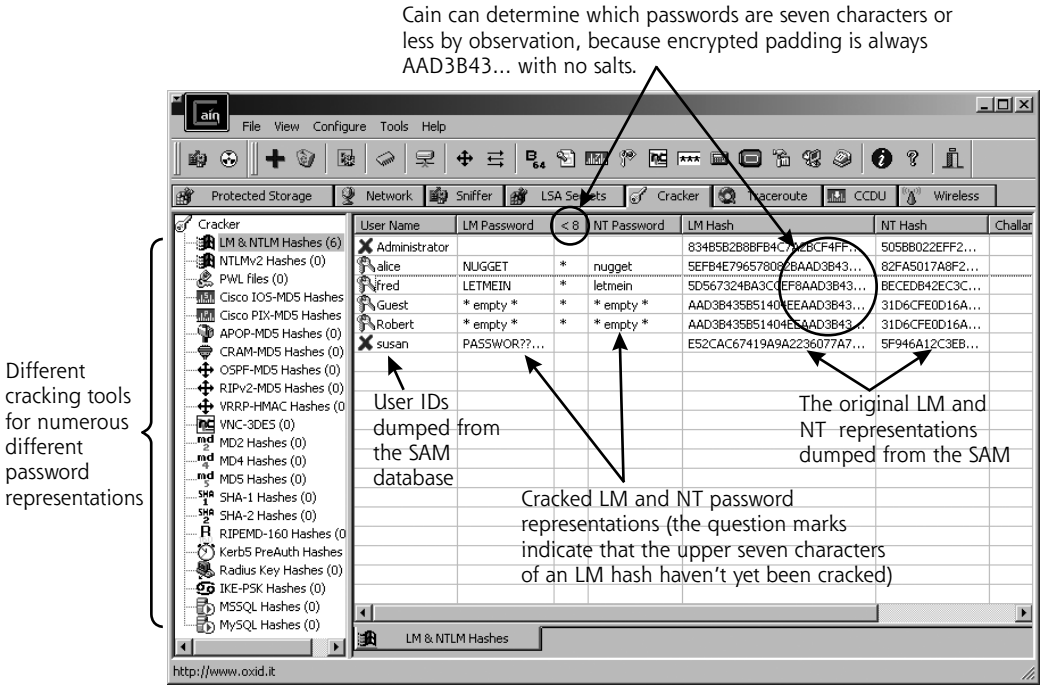


Figure 7.19 Successful crack using Cain.

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 Cain in sniffing mode to snag user authentication information from the network.

Of course, it might be very difficult for attackers to insert themselves in such a sensitive location. To get around this difficulty, an attacker can trick a user via e-mail into revealing his or her password hashes. Consider the e-mail shown in Figure 7.20, 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, in the form of file://SOMESERVER. On this SOMESERVER machine, the attacker has installed Cain and is running the integrated sniffing tool.

skoudis.book Page 394 Wednesday, November 30, 2005 10:04 PM

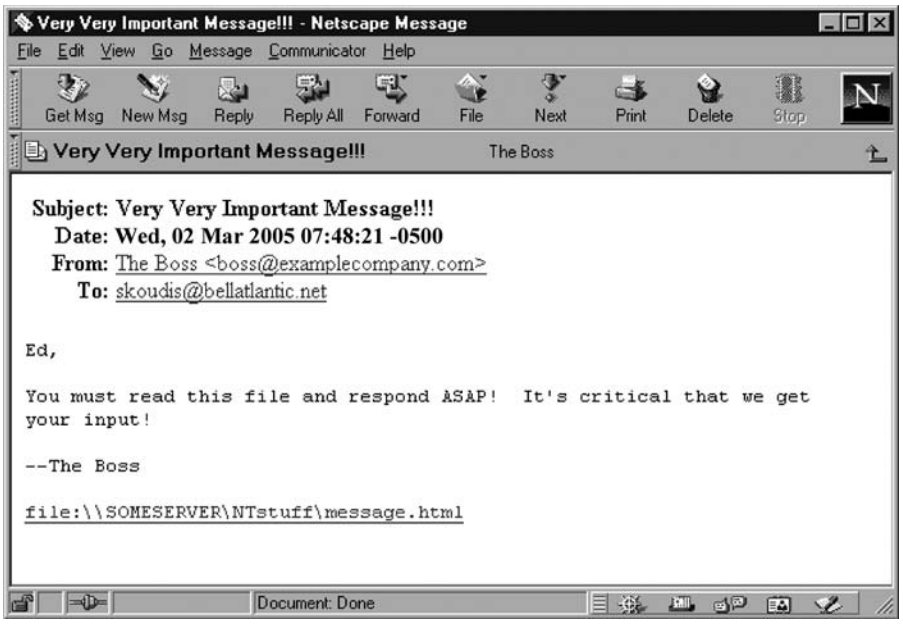


Figure 7.20 Would you trust this e-mail?

When the victim clicks the file:\\ link, the victim's machine attempts to mount the share on the attacker's server, interacting with the server using a Windows challenge–response protocol such as LM Challenge, NTLMv1, NTLMv2, or Kerberos, depending on the system's configuration. Once the victim clicks the link, the attacker's sniffer displays the gathered challenge and response, as shown in Figure 7.21.

To complete the attack, the attacker can save this captured data and feed it into Cain to retrieve the user's password, as shown in Figure 7.22. This technique, which combines social engineering via e-mail, sniffing data from the network, and password cracking, really demonstrates the power of several aspects of Cain.

CAIN DOESN'T DO JUST WINDOWS Beyond these Windows operating system password-cracking capabilities, Cain can also crack Cisco-IOS Type-5 enable passwords, Cisco PIX enable passwords, APOP-MD5 hashes, CRAM-MD5 hashes, RIPv2-MD5 hashes, OSPF-MD5 hashes, VRRP-HMAC-96 hashes, VNC's 3DES passwords, RADIUS Shared Secrets, Password List (PWL) files from

skoudis.book Page 395 Wednesday, November 30, 2005 10:04 PM

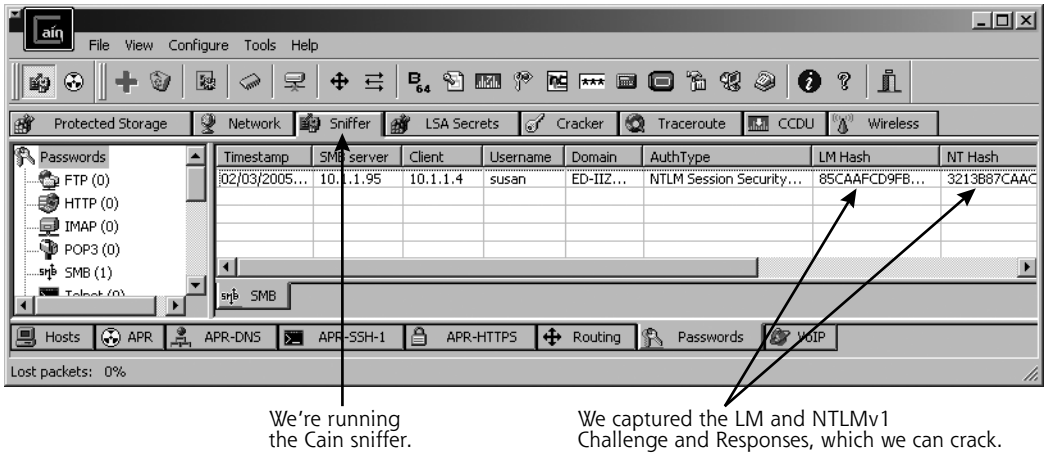


Figure 7.21 Cain's integrated sniffer captures the challenge–response from the network for cracking.

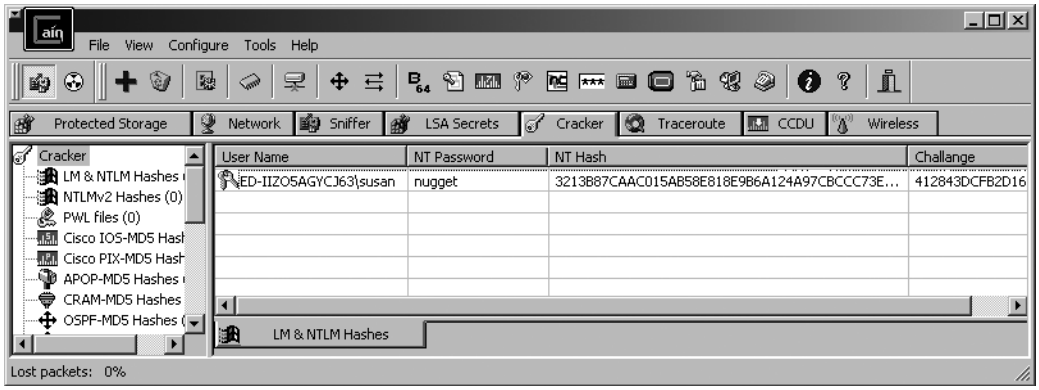


Figure 7.22 A sniffed Windows challenge–response successfully cracked.

Windows 95 and Windows 98, Microsoft SQL Server 2000 passwords, MySQL323 passwords, MySQLSHA1 hashes, and even IKE preshared keys. Whew! That's quite an exhaustive list. That last item in the list, associated with the IKE protocol, is especially useful for the bad guys in a VPN environment. Many IPSec implementations use IKE to exchange and update their crypto keys. Most systems and VPN gateways, by default, use IKE in a manner called aggressive mode, designed to exchange new keys quickly across the network. Many organizations have deployed their IPSec products using a preshared key as an

skoudis.book Page 396 Wednesday, November 30, 2005 10:04 PM

initial secret to exchange the first set of session keys via aggressive mode IKE. This preshared key is usually just a password typed by an administrator into the IPSec clients and VPN gateway. Unfortunately, if an attacker sniffs the aggressive mode IKE exchange using Cain's built-in sniffer, the bad guy can crack this preshared key. Using this information, the attacker can then load the preshared key into the attacker's own IPSec client, and ride in through the VPN gateway, impersonating the original user. This preshared key IKE cracking capability originated in a tool called IKE Crack, but the functionality has been nicely imported into both Cain's sniffer and password-cracking features.

Cracking UNIX (and Other) Passwords Using John the Ripper

Despite its ability to attack other operating systems, Cain still runs just on Windows. Another free, high-quality password cracker that can run on more environments is John the Ripper, one of the best tools today focused only on password cracking. John the Ripper (called John for short) is a free tool developed by Solar Designer, the gentleman we discussed earlier in this chapter who wrote the nonexecutable kernel patch for Linux to defend against stack-based buffer overflows. Although John is focused on cracking UNIX and Linux passwords, it has some extended modules that can crack other password types, including Windows LM representations and NT hashes.

John runs on a huge variety of platforms, including Linux, UNIX, Windows of all kinds, and even the ancient DOS platform. Yes, you can dust off that old DOS system and use it to crack passwords. To boost its speed, John even includes optimized code to take advantage of various specific CPU capabilities, such as Intel's MMX technology.

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 LM hashes from Windows machines. Also, Dug Song, the author of the FragRouter IDS and IPS evasion tool that we discussed in Chapter 6, has written modular extensions for John that crack files associated with the S/Key one-time-password system and AFS/Kerberos Ticket Granting Tickets, which are used for cryptographic authentication. Finally, a developer

skoudis.book Page 397 Wednesday, November 30, 2005 10:04 PM

named Olle Segerdahl has written an NT hash-cracking module for John, freely available at *www.openwall.com//john/contrib/john-ntlm-v03.diff.gz*.

RETRIEVING THE ENCRYPTED PASSWORDS As described in Chapter 3, Linux and UNIX Overview, 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`.

Most modern UNIX variants include an option for using shadow passwords. In such systems, the `/etc/passwd` file still contains general user account information, but all encrypted passwords are moved into another file, usually named `/etc/shadow` or `/etc/secure`. Figures 7.23 and 7.24 show the `/etc/passwd` and `/etc/shadow` files, respectively, from a system configured to use shadow passwords. 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 buffer overflow of program that runs as root or a related technique, to gain root access. After achieving root-level access, the attacker makes a copy of the shadow password file to crack.

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 that might have been a victim of a buffer overflow that simply crashed the target process. The core file is generated 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, an attacker can comb through it to look for the encrypted passwords. This technique of 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 from the target machine. They'll then pore through the core file looking for passwords to crack to gain access to the FTP server.

CONFIGURING JOHN THE RIPPER Although it doesn't have a fancy GUI like Cain, the command-line John tool is still trivially easy to configure. The attacker must

skoudis.book Page 398 Wednesday, November 30, 2005 10:04 PM

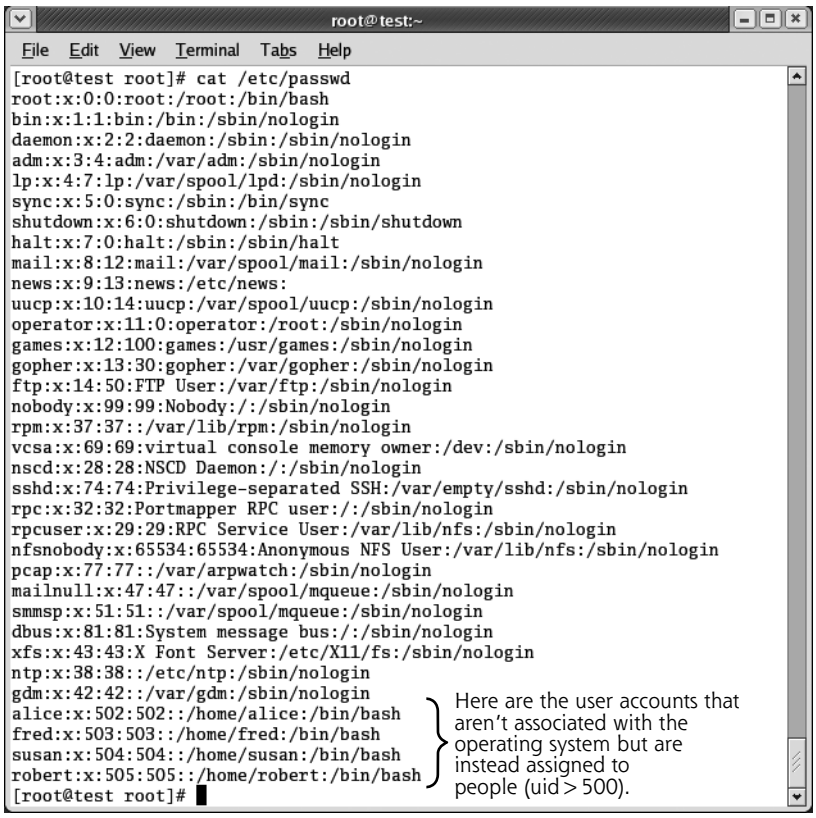


Figure 7.23 When password shadowing is used on a system, the `/etc/passwd` file contains user information, but no passwords.

feed John 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, suitably enough, unshadow, which is shown in Figure 7.25.

Another very nice feature of John is its ability to detect automatically the particular encryption algorithm to use during a cracking exercise, differentiating various UNIX and Linux password encryption techniques from each other, as well as

skoudis.book Page 399 Wednesday, November 30, 2005 10:04 PM

Figure 7.24 The corresponding `/etc/passwd` file contains the encrypted passwords.

the Windows LM representation. This autodetect capability is based on the character set, length, and format of the given file containing the passwords. In this way, John practically configures itself automatically. Although the autodetect function is nifty, the absolute greatest strength of John is its ability to create many permutations quickly for password guesses based on a single wordlist. Using a wordlist in a hybrid-style attack, John appends and prepends characters, and attempts dictionary words forward, backward, and typed in twice. It even truncates dictionary terms and appends and prepends 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

skoudis.book Page 400 Wednesday, November 30, 2005 10:04 PM

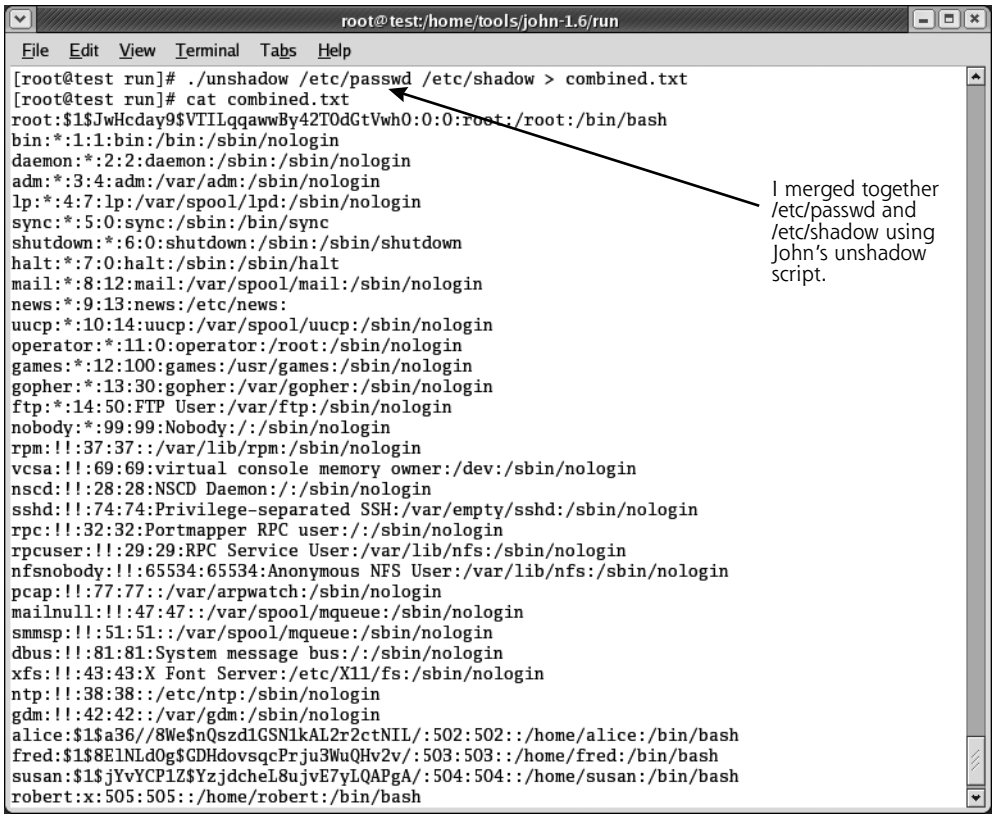


Figure 7.25 Running the unshadow script from John the Ripper.

modifying dictionary terms. Quite simply, John has the best hybrid guessing engine available publicly today.

With all of this slicing and dicing of words to create password guesses, John acts like a dictionary food processor. 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 them.

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

skoudis.book Page 401 Wednesday, November 30, 2005 10:04 PM

- *"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 account name and General Electric Computer Operating System (GECOS) field, a block of arbitrary text associated with each account.
- *Wordlist mode.* As its name implies, this mode guesses passwords based on a dictionary, creating numerous permutations of the words using the rule set.
- *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 7.26.

While John is running, it displays successfully cracked passwords on the screen, and stores them in a local file called john.pot. If you ever run John, make sure you clean up after yourself by removing john.pot! Whenever I'm doing a security assessment, I always look for leftover john.pot files that a lazy system administrator or auditor forgot to destroy. Using a remnant john.pot, I can rely on the password-cracking work having been done by another user, making my attack go much more quickly. 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 percentage of the current mode that is completed, as well as the current password guess John has just created.

DEFENSES AGAINST PASSWORD-CRACKING ATTACKS

Cain and John the Ripper represent the best of breed password-cracking tools, and can quickly determine passwords in most environments. In my experience at

skoudis.book Page 402 Wednesday, November 30, 2005 10:04 PM

Figure 7.26 Running John the Ripper to crack passwords.

numerous organizations, Cain or John often return dozens of passwords after running for a couple of minutes. Given the obvious power of these cracking 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. Carefully apply several defensive techniques that work together to help eliminate weak passwords, starting with 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. In fact, I prefer having a minimum password length of at least 15 or even more characters. I know what you are thinking: "There'd be riots in the cubicles if I configured a minimum password length of 15 characters!" However, we need to get our users out of the mindset of having passwords, and move them into the notion of passphrases. For example, a password of "Gee, I think I'll buy another copy of Counter Hack!" is a lot

skoudis.book Page 403 Wednesday, November 30, 2005 10:04 PM

harder to crack than a password of #dx92!$XA, and the former is a lot easier to type as well. Also, I didn't arbitrarily choose that 15-character minimum. As it turns out, on Windows 2000 and later, if you set a password to 15 characters or more, the system will not store a LM hash at all for that password, instead relying solely on the stronger NT hash in the SAM database. That automatically gets rid of the scourge of LM hashes for such accounts, significantly improving your password security in a Windows environment. We look at an additional LM purging capability shortly.

Furthermore, passwords should have a defined maximum lifetime of 90, 60, or 30 days, depending on the particular security sensitivity and culture of your organization. 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 sticky notes. Of course, your culture might vary. Finally, make sure that your password policy is readily accessible by employees on your internal network 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 be 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 workplace 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. If you don't opt for passphrases, you should alternatively recommend that users rely on 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 tale of a fateful trip." A password derived from this phrase would be Jsrb,Ayhat,atoaft. As you might recall, there were seven stars in the TV program, so, we can add that information to the password, coming up with Jsrb,Ayhat,atoaft7*, which would be reasonably difficult to guess, as it contains alphabetic and numeric characters, mixed

skoudis.book Page 404 Wednesday, November 30, 2005 10:04 PM

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* in your own awareness initiatives, make sure to warn your users not to set their password to the example Jsrb,Ayhat,atoaft7*, because 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 his or her 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 far less able to create passwords that are too easily guessed. 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. However, 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 Domain Controllers, and other systems. Many modern variants of UNIX include password-filtering software. For those that do not, you can use a variety of third-party tools to add this capability, including a pluggable authentication module (PAM) tool written by Solar Designer, the author of John the Ripper. This module is available for Linux, Solaris, and FreeBSD systems for free at *www.openwall.com/passwdqc.*

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

- Password Guardian, a commercial tool available for sale at *www.georgiasoftworks.com*
- Strongpass, a free tool available at *http://ntsecurity.nu/toolbox*

skoudis.book Page 405 Wednesday, November 30, 2005 10:04 PM

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 or authentication across untrusted networks, you should avoid using traditional password authentication. Instead, consider one-time password tokens or smart cards for access. Or, utilize biometric authentication to augment passwords, such as handprint, fingerprint, or retina scanners.

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 Cain or John the Ripper, check for crackable passwords every month or every quarter. As always, avoid using programs from untrusted sources.

Before conducting this type of assessment, make sure you have explicit permission from management. Otherwise, you could 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 in place for interacting with users whose passwords can be easily guessed. Don't e-mail or call them on the phone to tell such users to change their passwords, because you'd then make them more subject to social engineering attacks. Instead, configure their accounts to require a password change the next time they log in.

Protect Your Encrypted or Hashed Password Files

A final very important technique for defending against password-cracking tools is to protect your encrypted or 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.

skoudis.book Page 406 Wednesday, November 30, 2005 10:04 PM

On all of your UNIX systems that support it, make sure that you activate password shadowing, which stores the password representations in the `/etc/shadow` file, readable only by root. On Windows machines, if you do not have to support backward compatibility for Windows for Workgroups or Windows 95 or 98 clients, disable the incredibly weak LM authentication. In an environment that includes only Windows NT and later machines, you can get rid of the weak LM representations by defining the registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\NoLMHash on all systems. This registry key tells the system not to store the LM representation when each user next changes his or her password. Thus, with this key defined, your LM hashes will gradually disappear as each user's password expires over the next 90, 60, or 30 days. Furthermore, the registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\LMCompatibilityLevel can be set to a value of three on Windows NT and later clients to force them to send the more difficult-to-crack NTLMv2 representations across the network. This same registry key can be set on servers to a value of five to force them to accept only NTLMv2 authentication, again breaking backward compatibility with Windows for Workgroups, Windows 95, and Windows 98, but significantly improving your security.

Finally, whenever you make a backup using the Ntbackup.exe program, remember to 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 Web has proven to be a particularly fruitful area for attackers to exploit.

skoudis.book Page 407 Wednesday, November 30, 2005 10:04 PM

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 injection. 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 these attacks and defend against them.

All of the Web attack techniques described in this section can be conducted even if the Web server uses the 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. In other words, SSL supports authentication, and protects data in transit. 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. When the data is located in the browser, SSL doesn't prevent changes to that data by the person sitting at the browser. If an attacker is browsing your Web application, he or she might just change some crucial data in the browser. If your Web application trusts whatever comes back, the bad guy might be able to undermine your Web application completely. Remember, the browser is potentially enemy territory, with an attacker sitting at its controls, so you can't trust what comes back from it unless you explicitly validate that data. Let's look at such attacks in more detail, starting with account harvesting.

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 user IDs and even passwords of a vulnerable application. Account harvesting is really a simple concept, targeting the authentication process when an application requests a user ID and password. The technique works against applications that have a different error message for users who type in an incorrect user ID than for users who type a correct user ID with an incorrect password.

skoudis.book Page 408 Wednesday, November 30, 2005 10:04 PM

Bad User ID, with Bad Password

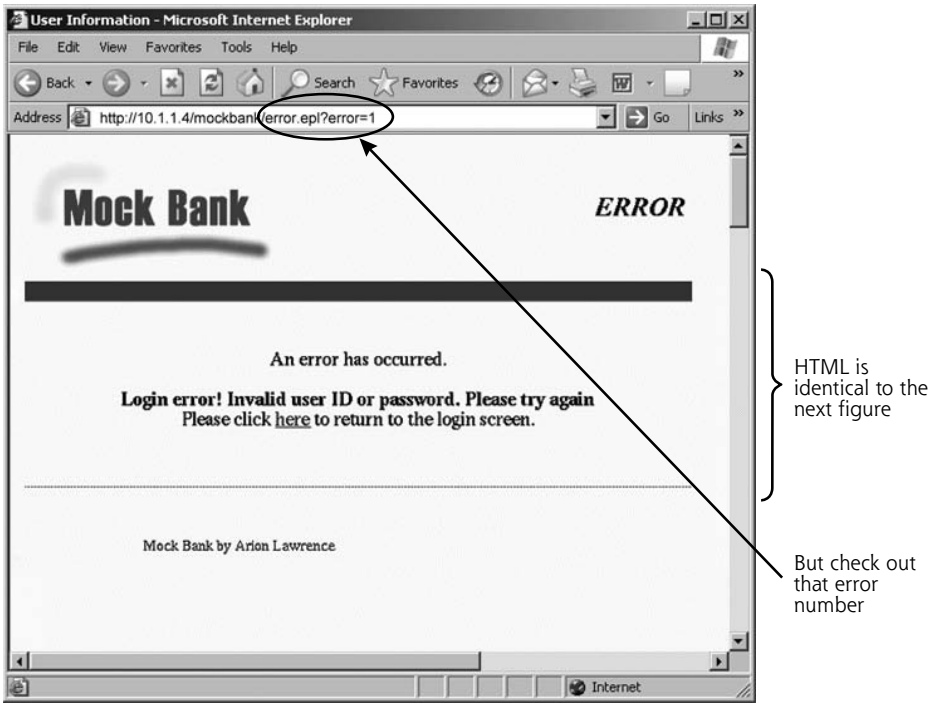


Figure 7.27 Mock Bank's error message when a nonvalid (i.e., bad) user ID is entered.

Consider the error message screens for the application shown in Figure 7.27 and Figure 7.28. These screens are from a proprietary Web application called Mock Bank, written by Arion Lawrence, a brilliant colleague of mine. We use Mock Bank internally to show common real-world problems with Web applications to our clients, as well as to train new employees in the methods of ethical hacking. The first screen shows what happens when a user types in a wrong user ID, and the second shows the output from a correct user ID and an incorrect password. The actual HTML and appearance of the browser in both pages are identical. However, look at the location line in the browser of each figure a bit more closely. Notice that when the user ID is incorrect, error number 1 is returned, as in Figure 7.27. When the user ID is valid and the password is wrong, error number 2 is returned, as in Figure 7.28. This discrepancy is exactly what an attacker looks for when harvesting accounts.

skoudis.book Page 409 Wednesday, November 30, 2005 10:04 PM

Good User ID, with Bad Password

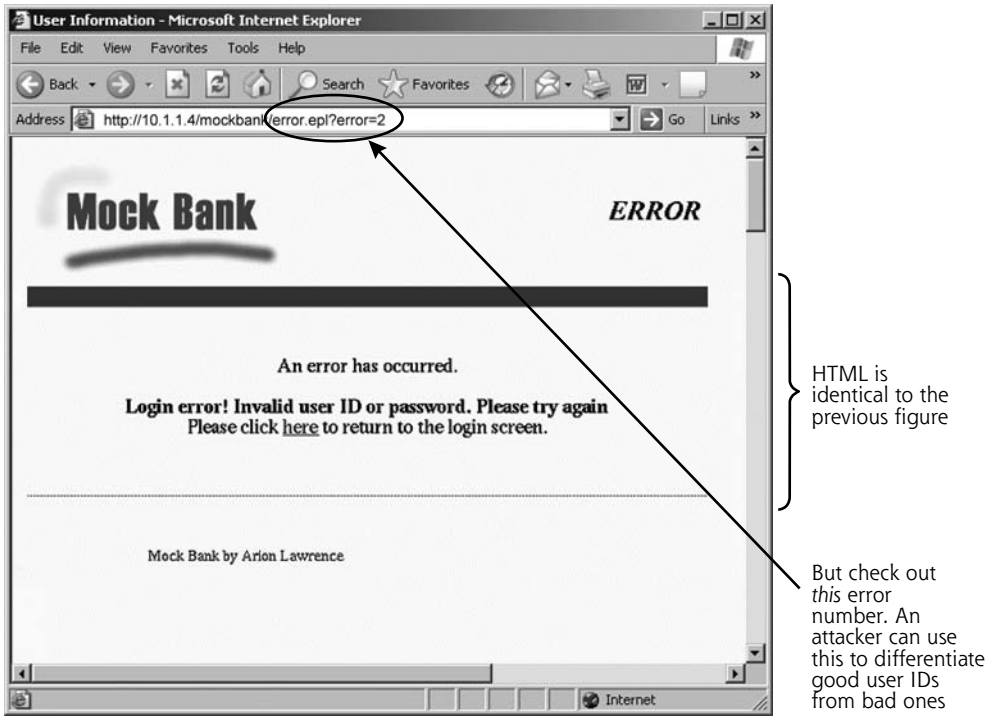


Figure 7.28 Mock Bank's error message when a valid (i.e., good) user ID is entered with a bad password. Note the change in the URL error number parameter.

Based on this difference in error messages in the URL, an attacker can write a custom script to interact with the Web application, conducting a dictionary or brute-force attack guessing all possible user IDs, and using an obviously false password (such as zzzzz). The script will try each possible user ID. If an error message is returned indicating that the user ID is valid, the attacker's script writes the user ID to a file. Otherwise, the next guess is tested. This is pure user ID guessing through scripting, adding a bit of intelligence to discriminate between invalid and valid user IDs. In this way, an attacker can harvest a large number of valid user IDs from the target application. In this Mock Bank example, the parameter called error is the differentiating point between the two conditions. Of course, any element of the returned Web page, including the HTML itself, comments in the HTML, hidden form elements, cookies, or anything else, could be

skoudis.book Page 410 Wednesday, November 30, 2005 10:04 PM

the differentiator between the bad user ID and good user ID conditions. The attacker will choose a suitable differentiating point to include in the logic check of the login attack script.

After running a script to harvest good user IDs, 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 or use the Brutus or Hydra tools we discussed earlier in this chapter to try password guessing across the network. The attacker takes the user IDs previously harvested and tries guessing all passwords for that account using login scripting. If the target application does lock out accounts, the attacker can easily conduct a DoS attack using the harvested user ID 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 user ID or password. Rather than telling the user, "Your user ID 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 user ID or password was 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. Even a single space or period that is different between the two authentication error conditions could tip off an attacker's script.

UNDERMINING WEB APPLICATION SESSION TRACKING AND OTHER VARIABLES

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 user ID and password, or through a client-side certificate on an HTTPS session), most Web applications generate a session ID to track the user's actions for the rest of the browsing session of that site. The Web application generates a session ID and passes it to the

skoudis.book Page 411 Wednesday, November 30, 2005 10:04 PM

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 or the development environment used to create the Web application assigned to it. It does not have to be called sessionID, 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 in the vast majority of applications. The session ID is application-level data, generated by the application and exchanged by the Web browser and Web server. Although it is encrypted by SSL as it moves across the network, the session ID can be altered at the browser by the browser user without impacting the underlying SSL connection.

Implementing Session IDs in Web Applications

So 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 7.29, and passed as a parameter in an HTTP GET request. For all subsequent Web requests, the URL is passed

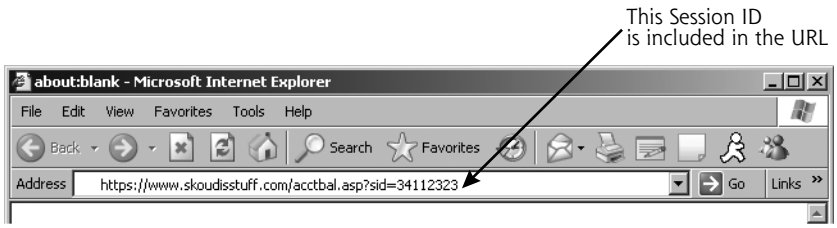


Figure 7.29 Session tracking using the URL.

skoudis.book Page 412 Wednesday, November 30, 2005 10:04 PM

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. With 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 does not show the user these hidden elements, but the user can readily see them simply by invoking the browser's view source function for the page. In the raw HTML, a hidden form element will have the following appearance:

```
<INPUT TYPE="HIDDEN" NAME="Session" VALUE="34112323">
```

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: persistent cookies and nonpersistent cookies. A persistent cookie 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. A nonpersistent cookie, on the other hand, 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.

ATTACKING SESSION TRACKING MECHANISMS

Many Web-based applications have vulnerabilities in properly allocating and controlling these session IDs. An attacker might 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, a process sometimes referred to as *session cloning*. 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

skoudis.book Page 413 Wednesday, November 30, 2005 10:04 PM

time. Still, many Web-based applications won't even notice this problem, accepting and processing transactions from both the attacker and the legitimate user.

In fact, it's pretty hard for an application to even figure out that this has happened. Suppose the application associates a session ID number with the IP address of the user. Well, there's a problem in that many users might be surfing from behind a single proxy or a many-to-one dynamic NAT device, so all such users will have the same apparent IP address. One user on the other side of the proxy could still clone the session of another user of the proxy. Furthermore, trying to nail the session ID to the IP address is bad because sometimes a user who surfs through a large ISP will have a changed apparent source IP address, right in the middle of a surfing session! Because of some complex routing and proxying that some ISPs perform, a completely legitimate user might get a different IP address in real time. Web applications that check session credentials against the IP addresses would think that such users are really being attacked, when they aren't. They were just given a different IP address.

An application with predictable session credentials allows 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 stock trading, the attacker could make trades on behalf of the user. For an online health care application … well, you get the idea.

To perform this kind of attack, the bad guy first needs to determine another user's session ID. To accomplish this, the attacker logs in to the application using a legitimate account assigned to the attacker, and observes the session ID assigned to that session. The attacker looks at how long the session ID is and the types of characters (numeric, alphabetic, or others) that make it up. The attacker then writes a script to log in again and again, gathering hundreds of session IDs to determine how they change over time or to see if they are related in any way to the user ID. Then, applying some statistical analysis to the sampled session IDs, the attacker attempts to predict session IDs that belong to other users.

So how does an attacker actually manipulate the session ID? First, the attacker logs in to the application using his or her own account to be assigned a session ID. Then, the attacker attempts to modify this session ID to clone the session of

skoudis.book Page 414 Wednesday, November 30, 2005 10:04 PM

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 can save the Web page sent by the server to the local file system. The attacker then edits the session ID in the hidden form elements of the local copy of the Web page, and reloads the local page into the browser. By simply submitting the form back to the server, the attacker can send the new session ID and could clone another user's session.

If sessions are tracked using persistent cookies, the attacker can simply edit the local cookie file. In Mozilla Firefox and Netscape browsers, all persistent cookies are stored in a single file called `cookies.txt`. For Internet Explorer, cookies from different servers are stored in their own individual files in the Temporary Internet Files directory for each user. An attacker can edit these persistent cookies using any text editor, as shown in Figure 7.30. To exploit a session ID based on a persistent cookie, the attacker can log in to the application to get a session ID, close the browser to write the cookie file, edit the cookies using his or her favorite

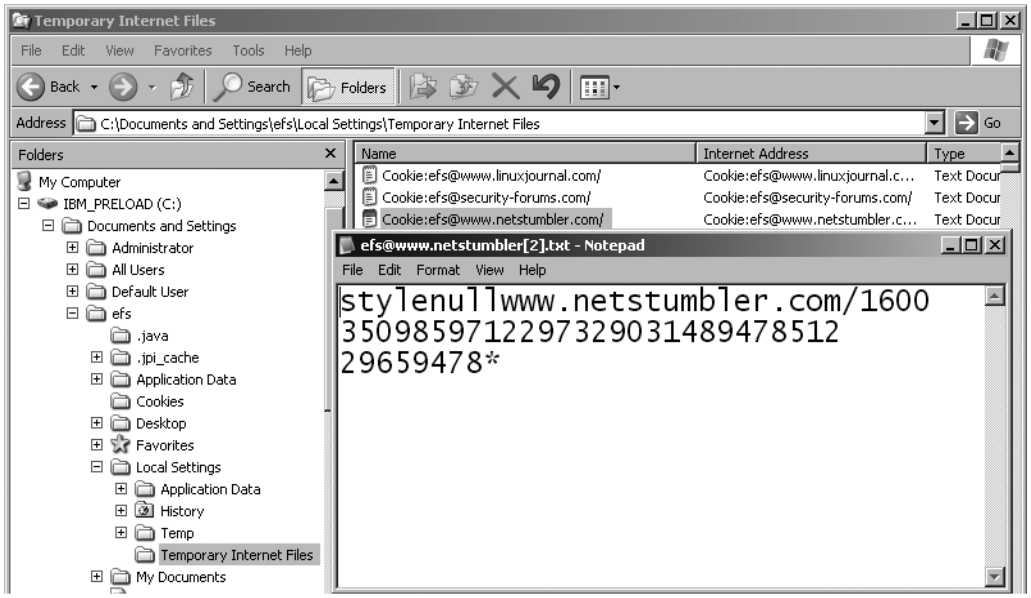


Figure 7.30 Editing nonpersistent cookies using Notepad.

skoudis.book Page 415 Wednesday, November 30, 2005 10:04 PM

text editor, and relaunch the browser, now 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 nonpersistent 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 nonpersistent cookies, especially those passed via SSL, so they don't bother worrying about protecting the information stored in such cookies. Unfortunately, bad guys use very powerful techniques for altering nonpersistent cookies.

To accomplish this feat, Web application attackers most often rely on a specialized Web proxy tool designed to manipulate Web applications. A Web application manipulation proxy sits between the browser and the server, as shown in Figure 7.31. All HTTP and HTTPS gets channeled through the proxy, which gives the attacker a window to view and alter all of the information passed in the browsing session, including nonpersistent cookies. Thus, the bad guy has a very fine-grained level at which to modify any cookies that are passing by. What's more, these specialized proxies let the attacker edit any raw HTTP/HTTPS fields and HTML information including cookies, hidden form elements, URLs, frame definitions, and so on.

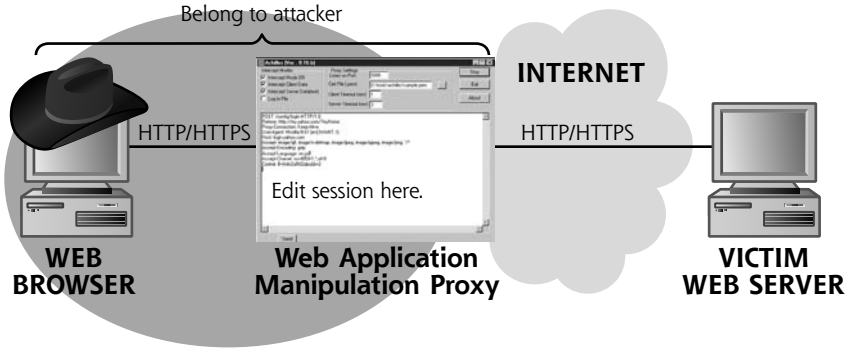


Figure 7.31 A Web application manipulation proxy lets the attacker alter the HTTP and HTTPS elements passing through it, including nonpersistent cookies.

skoudis.book Page 416 Wednesday, November 30, 2005 10:04 PM

It is crucial to note that these Web application manipulation attacks are not person-in-the-middle attacks where a bad guy changes another user's data going to the application. In these Web application manipulation attacks, the bad guy controls both the browser and the proxy. Attackers use the proxy to alter their own data going to and from the Web application, including session ID numbers and other variables. That way, any victim Web server that trusts the information that comes from the browser will be tricked. The attacker applies the browser and Web application manipulation proxy in tandem: The browser browses, while the proxy lets the attacker change the elements inside the HTTP and HTML itself.

Because this proxy concept is so powerful in attacking Web applications, various security developers have released a large number of these Web application manipulation proxies, both on a free and a commercial basis. Table 7.2 shows some of the most useful Web application manipulation proxies, as well as their claims to fame.

Table 7.2 Web Application Manipulation Proxies

Tool Name	Licensing Terms	Platform	Claim to Fame	Location
Achilles	Free	Windows	First to be released and easiest to use	www.mavensecurity.com/achilles
Paros Proxy	Free	Java	Incredibly feature rich; my favorite among the free tools	www.parosproxy.org
Interactive TCP Relay	Free	Windows	Supports HTTP/HTTPS and any other TCP protocol	www.imperva.com/application_defense_center/tools.asp
WebScarab	Free	Java	Free, open source, and actively updated, with a modular interface for adding new tools and features	www.owasp.org
SPI Dynamics SPIProxy/WebInspect	Commercial	Windows	Records browsing and then automates attacks, integrates with other SPI Dynamics tools	www.spidynamics.com
Web Sleuth	Commercial	Windows	Excellent filtering capabilities	www.sandsprite.com/Sleuth/

skoudis.book Page 417 Wednesday, November 30, 2005 10:04 PM

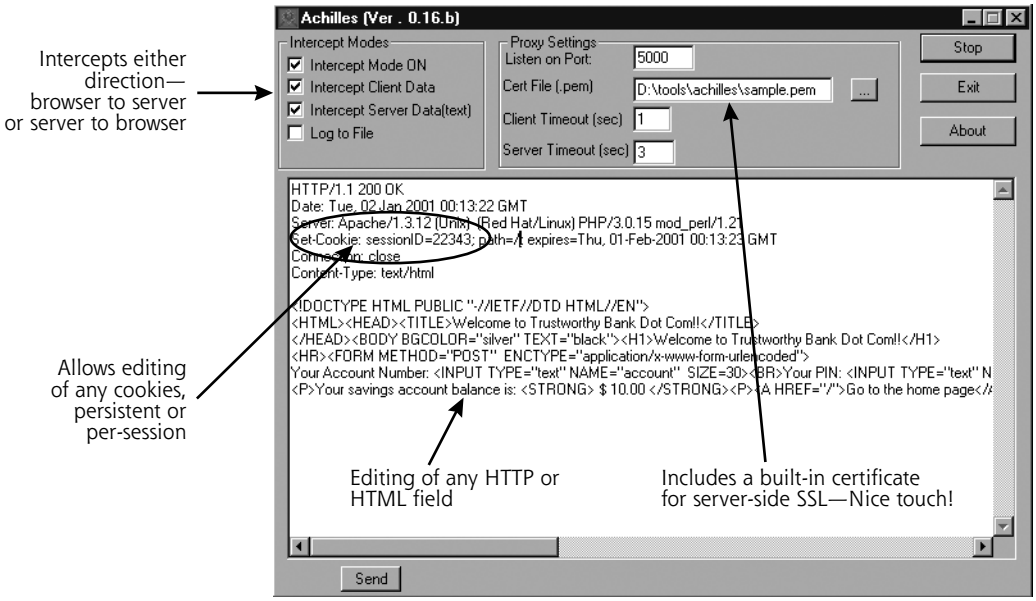


Figure 7.32 The Achilles screen, one of the easiest to use Web application manipulation proxies.

To launch this kind of attack, the bad guy runs the browser and the Web application manipulation proxy, either on separate systems or on a single machine. To get a feel for how these tools work, let's look at the one with the simplest user interface, Achilles, which is shown in Figure 7.32. In the main Achilles window, all information from the HTTP or HTTPS session is displayed for the attacker to view. 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 any information in this session in the main window and type right over it. The attacker then clicks the Send button, which transfers the data from Achilles to the server or browser.

Most Web application manipulation proxies support HTTPS connections, which are really just HTTP connections protected using SSL. To accomplish this, as displayed in Figure 7.33, the proxy sets up two SSL connections: one session between the browser and the proxy, and the other between the proxy and the Web server. All data is encrypted at the browser and sent to the proxy. At the

skoudis.book Page 418 Wednesday, November 30, 2005 10:04 PM

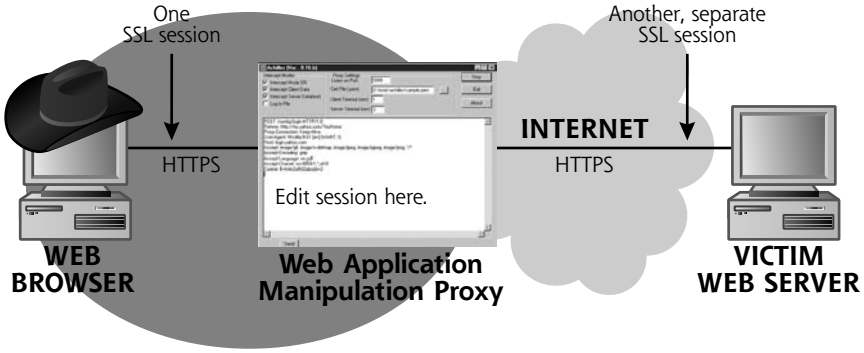


Figure 7.33 Handling HTTPS (that is, HTTP over SSL) with a Web application manipulation proxy.

proxy, the data is decrypted and displayed to the attacker, letting the bad guy alter it. Then, the data is encrypted across another, separate SSL session and sent to the victim Web server. When a response is served up by the server, the same process is applied in the opposite direction. Most of the proxies even come with a built-in digital certificate for server-side SSL 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 might display a warning message saying that the certificate from the server isn't signed by a trusted certificate authority, because the proxy inserts its own certificate in place of the Web server's certificate. However, the attacker is running both the browser and the proxy, so the warning message can be ignored by the attacker.

Although Achilles is the easiest to use of the Web application manipulation proxies, it isn't the most powerful. My current favorite Web application manipulation proxy is Paros Proxy, shown in Figure 7.34. Originally developed by the fine folks at ProofSecure, the Paros proxy maintains an excellent history of all HTTP requests and responses as the attacker surfs a given site through the proxy. Later, the attacker can review all of the action, with every page, variable, and other element recorded. Further, in addition to supporting server-side SSL, like most of the Web application manipulation proxies already do, Paros also allows its user to import a client-side SSL certificate that can be used to authenticate to a Web site that requires a client certificate. This client-side support is a strong differentiator among the free tools. Paros also features a built-in automated Paroweb spider

skoudis.book Page 419 Wednesday, November 30, 2005 10:04 PM

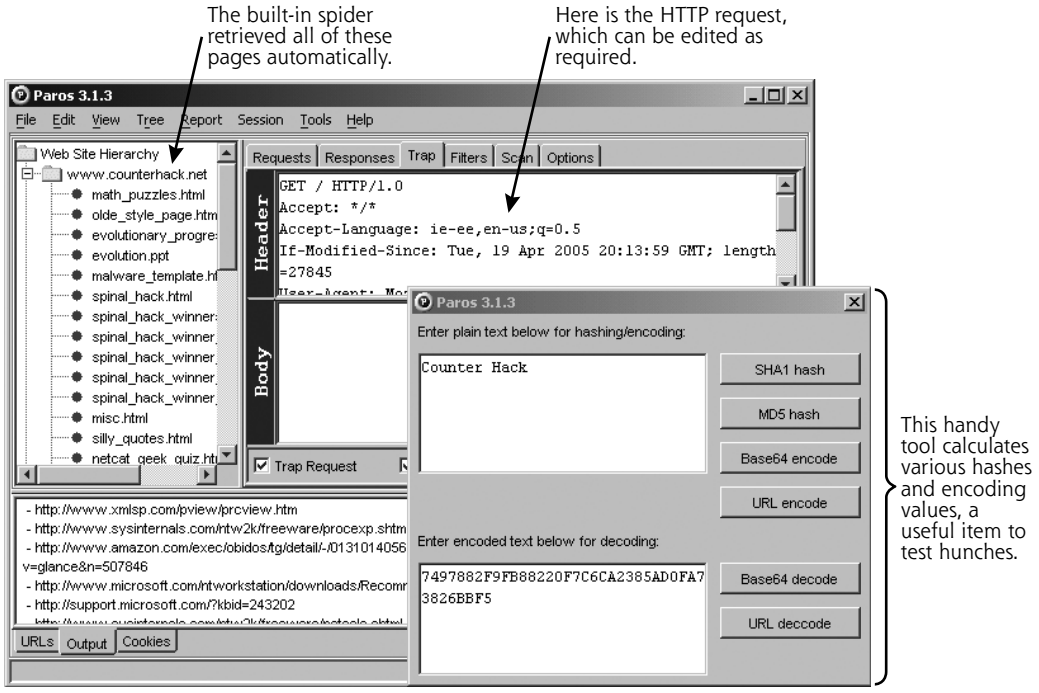


Figure 7.34 The Paros Proxy is one of the best freely available Web application manipulation proxies.

that can surf to every linked page on a target Web site, storing its HTML locally for later inspection, all the while harvesting URLs, cookies, and hidden form elements for later attack.

Another nice touch in Paros is a built-in point-and-click tool for calculating the SHA1, MD5, and Base64 value of any arbitrary text typed in by its user or pasted in from the application. When attacking Web applications, the attacker sometimes has a hunch about the encoding or hashing of a specific data element that is returned. Using this calculator, the attacker can quickly and easily test such hunches. The tool also includes automated vulnerability scanning and detection capabilities for some of the most common Web application attacks, including SQL injection, an issue we discuss later in this chapter. Finally, the Paros find and filter features let an attacker focus on specific aspects of the target Web application, such as certain cookie names, HTTP request types, or other features. What a great tool!

skoudis.book Page 420 Wednesday, November 30, 2005 10:04 PM

As we've seen, an attacker can modify session credentials using these Web application manipulation proxies, but session credentials only scratch the surface. Many Web applications send a great deal of additional variables to the browser for temporary or permanent storage in cookies or hidden form elements. Using a Web application manipulation proxy, the attacker can also view or edit any of these very enticing items passed to the browser. Some applications pass back account numbers, balances, or other critical information in cookies, expecting that they will remain unchanged and trusting them when they return from the browser.

Of particular interest are Web applications that pass back a price to the browser, such as an e-commerce shopping cart. Of course, an e-commerce application has to pass back a price so that customers can see on the screen how much they are spending, but that price should only be displayed on the screen. In addition to displaying the price on the screen, some applications use a cookie or a hidden form element to pass a price back to the browser for a shopping cart.

In such applications, the server sends the price to the browser in the form of a cookie or hidden form element, and the browser sends the price back to the server for each subsequent interaction to maintain the shopping cart or add to it. There is nothing to say that the user can't edit the price in the cookie or hidden form element while it's at the browser or in a Web application manipulation proxy. An attacker can watch the price go through a Web application manipulation proxy, edit it at the proxy, and pass it back to the server. The question here is this: Does the server trust that modified price? I've seen several e-commerce applications that trust the price that comes back from the user in the cookie or hidden form element.

For example, consider a Web application that sells shirts on the Internet. Suppose for this company, shirts should be priced at $50.00. This price is displayed on the screen in HTML, but is also passed in a cookie in a shopping cart. The attacker can use a Web application manipulation proxy to edit that cookie to say, "The $50.00 shirt is now changed to ten cents," or even zero. The price will be sent to the Web application, and if the Web application is vulnerable, the attacker will get a shirt for ten cents, or even for free. The attacker might even lower the price to a negative number, and perhaps the shirt will arrive in the mail with a check for the attacker's troubles! Quite frankly, the Web application doesn't need to

skoudis.book Page 421 Wednesday, November 30, 2005 10:04 PM

send the price in the cookie. It should only send a product stock-keeping unit (SKU) number or some other reference to the product, but not its price. Furthermore, it shouldn't trust the integrity of data received from the browser, as an attacker can alter any data using a Web application manipulation proxy.

DEFENDING AGAINST WEB APPLICATION SESSION TRACKING AND VARIABLE ALTERATION ATTACKS

To defend your Web applications from this type of attack, you must ensure the integrity of all session tracking elements and other sensitive variables stored at the browser, whether they are implemented using URLs, hidden form elements, or cookies. To accomplish this, use the following techniques for protecting variables sent to the browser:

- Digitally sign or hash the variables using a cryptographic algorithm, such as a Hash-Based Message Authentication Code (HMAC), as shown in Figure 7.35. When the application needs to pass a variable back to the browser, it creates a hash of the variable using a secure hash algorithm with a secret key known only to the Web application on the Web server. The variable and this hash are sent to the user. Evil users who try to change the data (and even the hash itself) will not be able to create a matching hash of their changed data, because they don't know the secret key. Thus, the application can perform an integrity check of all returned values to make sure their data and hashes match, using that secret key.

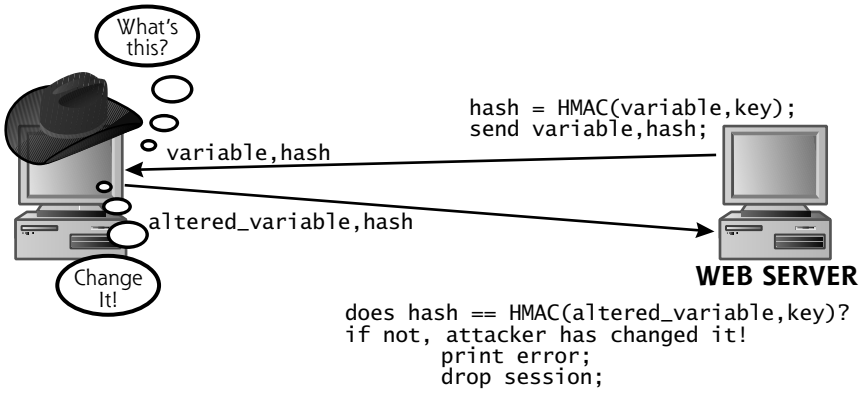


Figure 7.35 Applying an integrity check to a variable passed to a browser using the HMAC algorithm.

skoudis.book Page 422 Wednesday, November 30, 2005 10:04 PM

- If you are passing multiple variables in a single cookie, be careful when concatenating all of them together and loading them into a single cookie. Suppose you want to pass one variable that has a value of *dogfood* and another variable that has the value *court*. If you just concatenate these before hashing, the value *dogfood* and *court* will have the same hash as *dog* and *foodcourt* (as well as *dogfoo* and *dcourt*, I suppose). That gives the attacker a slightly better chance at figuring out what you are mixing together in your hashing algorithm. To minimize this chance, you should separate the values in the cookie with a delimiter character that won't be included in the variable values themselves. For example, include a separation character when concatenating, such as "&", as in *dogfood&court*.
- Encrypt the information in the URL, hidden form element, or cookie. Don't just rely on SSL, which protects data in transit. In addition to SSL, use some form of encryption of sensitive variables.
- Make sure your session IDs are long enough to prevent accidental collision. I recommend that session credentials be at least 20 characters (that's 160 bits) or longer.
- Consider making your session IDs dynamic, changing from page to page throughout your Web application. That way, an attacker will have a harder time crafting specific session ID numbers for specific users.

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 percent 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 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, clone 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 a user clicks the

skoudis.book Page 423 Wednesday, November 30, 2005 10:04 PM

Logout button, his or her 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 (e.g., for 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.

Additionally, defenders can use specialized Web proxy tools to help defend against these attacks. The commercial products AppShield from Watchfire and InterDo by Kavado sit in front of a Web application and look for incoming requests in which an attacker manipulated a cookie or other state element that is supposed to remain static for a given browsing session. They also look for other suspicious behavior.

SQL INJECTION

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 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.

To accomplish these so-called SQL injection attacks, the bad guys first 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 then experiments 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 string values entered into SQL statements.

skoudis.book Page 424 Wednesday, November 30, 2005 10:04 PM

Additionally, semicolons often act as separating points between multiple SQL statements. Using a considerable amount of trial and error, the attacker attempts 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. Interestingly, the Paros Web application manipulation proxy tool we discussed in the previous section has an automated SQL injection flaw detection capability, based on fuzzing user input. In the section on buffer overflows at the beginning of this chapter, we discussed fuzzing input for size by continually varying the amount of data sent until the application behaves in a strange fashion. Paros fuzzes user input not based on size, but instead focuses on altering all variables passed to a Web application, including information sent in the URL, elements of forms (both displayed and hidden form elements), and cookies. Paros looks for SQL injection flaws by sending quotes, semicolons, and other meaningful elements of SQL to the target application to make it generate a strange error message that could be a sign of an SQL injection flaw.

To get a feel for how SQL injection works, let's look at a specific example from a tool called WebGoat, a free Web application available for download from *www.owasp.org*. WebGoat implements a simulated e-commerce application, where users can pretend to buy HDTV equipment and other items. However, like the Mock Bank application we looked at earlier in this chapter, WebGoat is full of various Web vulnerabilities. By downloading WebGoat and experimenting with it in your lab on a Windows or Linux machine, you can improve your Web application assessment skills in a mock environment. If you can learn to find the flaws of WebGoat, you can apply the same skills in other applications and help make the world a more secure place.

WebGoat is an ideal tool for learning, as shown in Figure 7.36. It includes complete lesson plans, a report card on the users' progress so far, and almost two dozen different common Web application flaws (including SQL injection issues, as well as authentication and session tracking flaws similar to those we discussed earlier). Along the way, the tool offers hints for conquering each individual vulnerability, ranging from very ambiguous guidance to explicit directions for attacking a specific flaw. The Web-based user interface can be tweaked to make the Web application display all HTTP parameters, HTML, cookies, and even Javascript in-line for convenient analysis by the would-be attacker. Finally, to help apprentices make sure

skoudis.book Page 425 Wednesday, November 30, 2005 10:04 PM

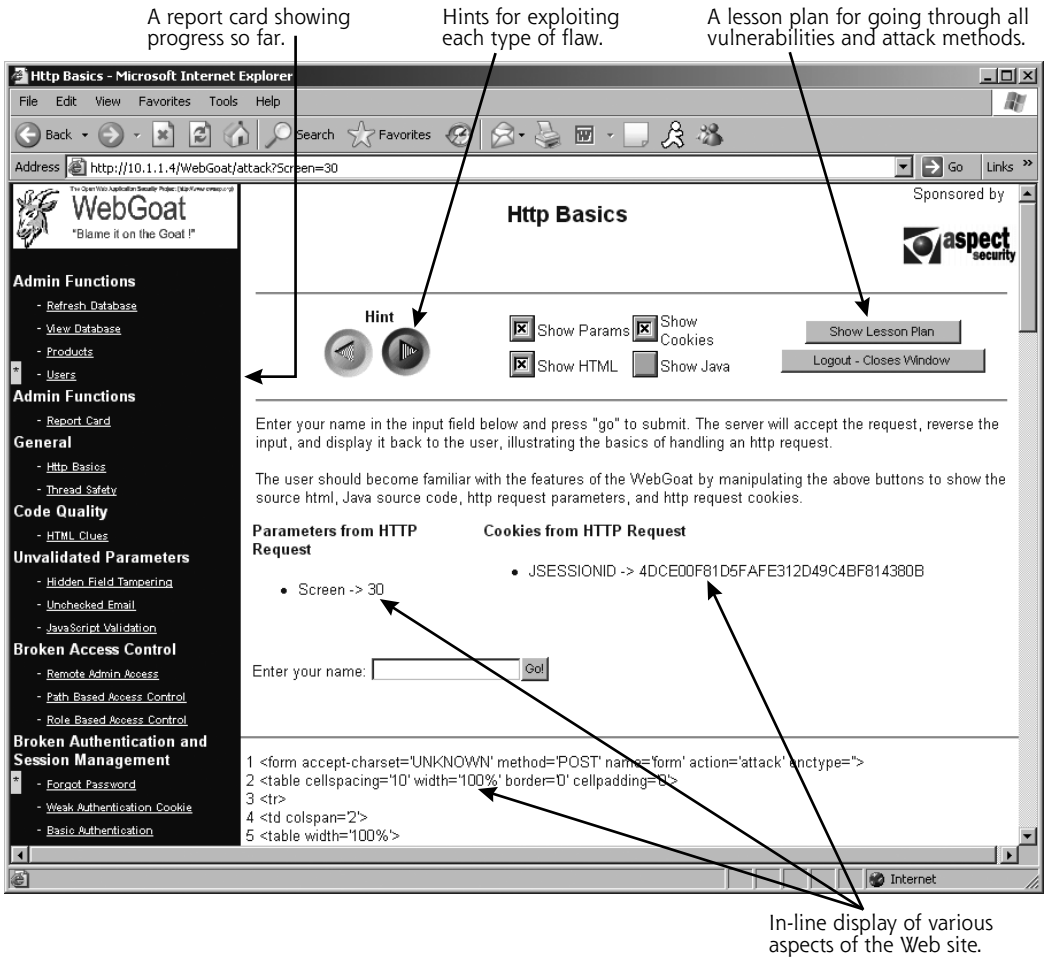


Figure 7.36 WebGoat is a great environment for learning Web application security assessment techniques.

that they are absorbing the material, there's even a final challenge, a hintless component of the application the users must master on their own.

One of the flaws designed into WebGoat involves SQL injection. The application lets users review their credit card numbers stored in the application, based on their account numbers. As illustrated in Figure 7.37, the user Joe Snow has two credit card numbers entered into the application.

skoudis.book Page 426 Wednesday, November 30, 2005 10:04 PM

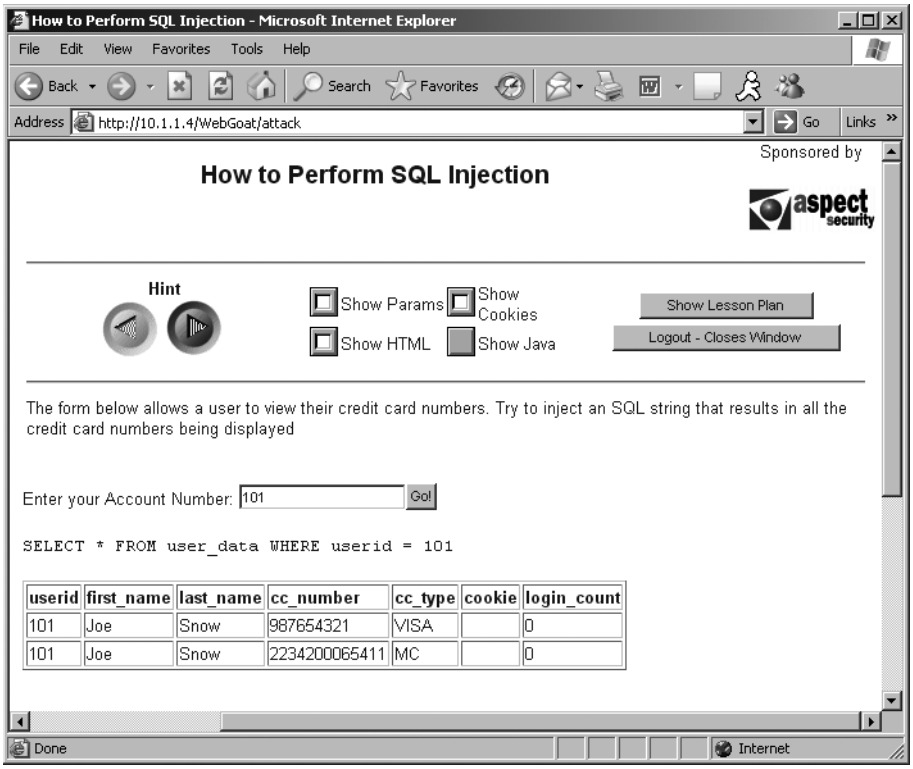


Figure 7.37 In WebGoat, user Joe Snow reviews his credit card numbers via his account number.

Now, suppose this Joe Snow user is evil. For SQL injection attacks, this bad guy might start out by entering quotation characters into the application as part of an account number. Remember, many SQL environments treat quotation characters as important terminators of strings. By sending an additional quotation character, the attacker might be able to generate an error message from the back-end database.

In Figure 7.38, the evil Joe Snow has submitted an account number of 101". Those closed quotes at the end are going to cause problems in the application. As a helpful hint about what's going on, WebGoat displays the SQL statement that will be used to query the back-end WebGoat database:

```
SELECT * FROM user_data WHERE userid = 101
```

skoudis.book Page 427 Wednesday, November 30, 2005 10:04 PM

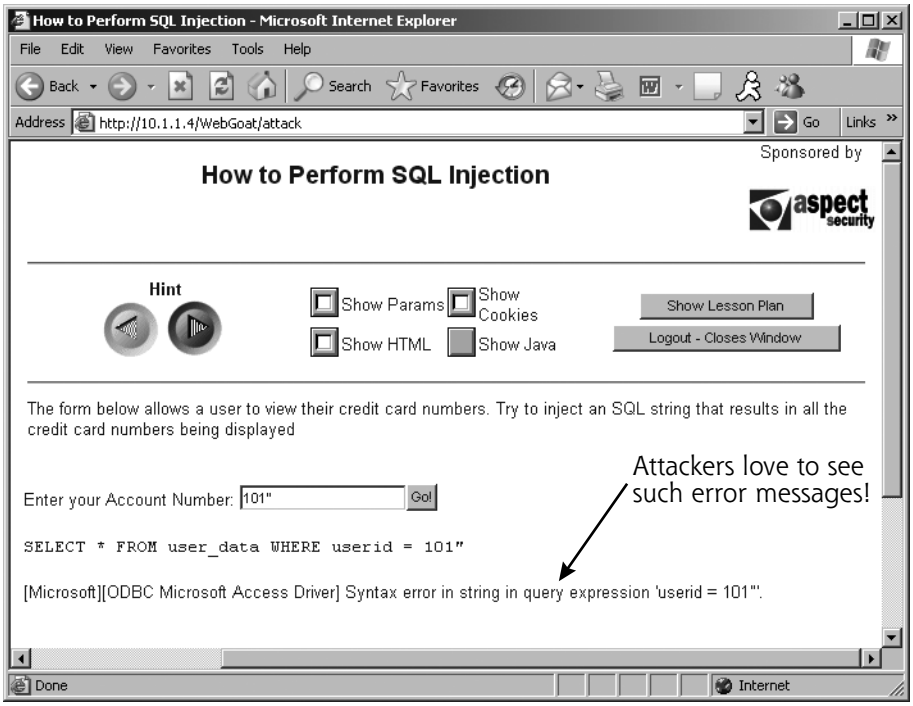


Figure 7.38 The evil user types in an account number of 101″ and gets an error message.

Of course, real-world applications wouldn't display the SQL itself, but WebGoat does for training purposes. Unfortunately, the application blindly takes anything entered by the attacker in the HTML form and puts it after the `userid =` portion of the SQL statement. If Joe Snow just enters a number, the application performs as expected, looking up the account information for that account number. However, if the attacker enters quotation marks, the resulting SQL becomes:

```
SELECT * FROM user_data WHERE userid = 101"
```

Those quotation marks at the end are the problem. Databases don't like to see such things, because they are syntax errors in SQL. Thus, the application indicates this error to Joe Snow by printing out that ugly ODBC Microsoft Access Driver message. Although that error might be ugly to most users, for evil Joe Snow, it's like gold. Any time an application responds with a syntax, SQL, SQL

skoudis.book Page 428 Wednesday, November 30, 2005 10:04 PM

Syntax, ODBC, or related error message, we've got a major sign the application is vulnerable to SQL injection.

Now, to really attack this application, the bad guy injects a little SQL logic into the target application. This time, the bad guy types an account number of `101 or 'TRUE'`. The resulting SQL created by the application will be:

```
SELECT * FROM user_data WHERE userid = 101 or 'TRUE'
```

Let's consider that `WHERE` clause in the SQL `SELECT` statement. We're looking for data where the `userid` has the value `101` or `'TRUE'`. Based on the rudimentary logical operator OR, anything OR `'TRUE'` is true. "The sky is purple" or `'TRUE'` is a true statement, based on the nature of OR. So, this `WHERE` clause is true for everything in the `user_data` table. Thus, the application looks up all data in that table and displays it to the attacker. As shown in Figure 7.39, the bad guy now has a list of credit card numbers for other users, obtained via SQL injection.

Our example from WebGoat showed injection techniques for SQL query statements (a `SELECT` command in particular). Injected `UPDATE` commands can allow an attacker to modify data in the database. Ultimately, if attackers carefully construct commands within SQL, they can get raw access to the back-end database.

DEFENSES AGAINST SQL INJECTION

To defend against SQL injection and related attacks through user input, your Web application must be developed to filter user-supplied data carefully. Remember, the application should never trust raw user input. It could contain injected commands and all kinds of general nastiness. Wherever a user enters data into the application, the application must strongly enforce the content type of data entered. A numerical user input should really only be an integer; all non-numerical characters must be filtered. Furthermore, the application must remove unneeded special characters before further processing of 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

skoudis.book Page 429 Wednesday, November 30, 2005 10:04 PM

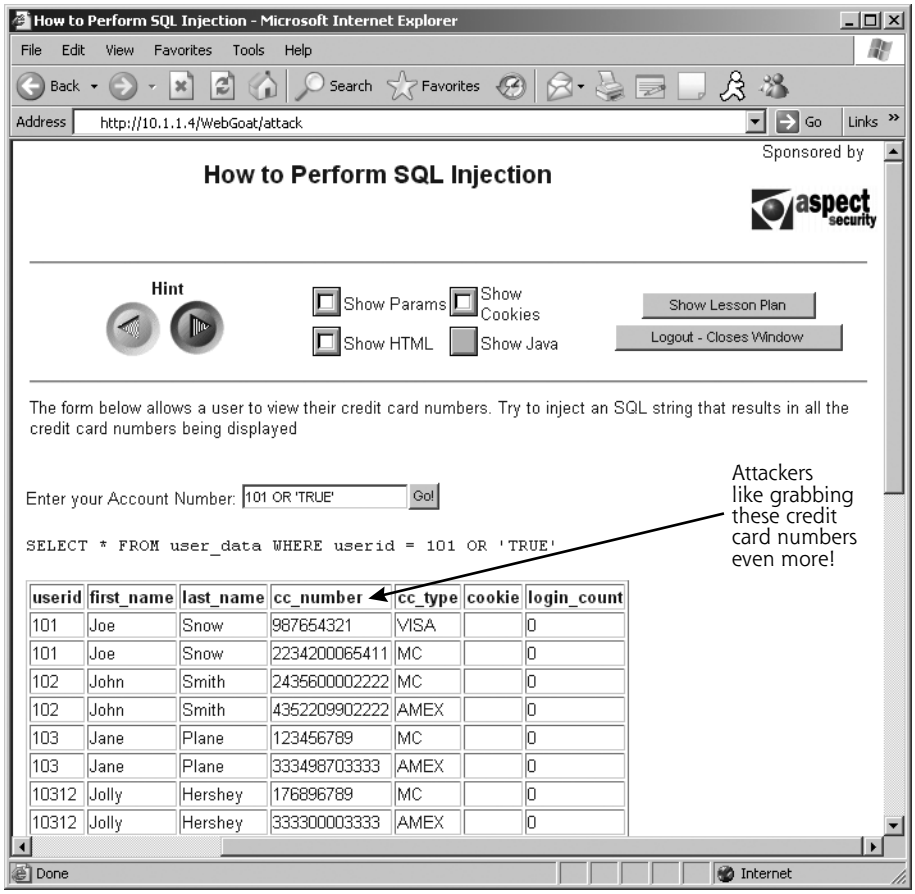


Figure 7.39 The evil user enters an account number of `101` or `'TRUE'` to get all account information via SQL injection.

- Asterisks (`*`)—Wildcard selectors
- Percent signs (`%`)—Matches for substrings
- Underscore (`_`)—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

skoudis.book Page 430 Wednesday, November 30, 2005 10:04 PM

Your best bet is to define which characters your application requires (usually just alphanumeric) and filter out the rest of the riff-raff users send you.

For those characters that might be dangerous but really necessary, introduce an escape sequence or substitute. One popular method of substituting innocuous replacements involves using an `&` and two letters to represent an otherwise scary character. For example, an apostrophe (`'`) can be changed to `&ap`, less than (`<`) can become `&lt`, and so on.

Furthermore, your input filtering code in the Web application can look for and remove potentially damaging SQL statements, including such SQL-relevant words as `SELECT`, `INNER`, `JOIN`, `UNION`, `UPDATE`, and `TRUE`.

These potentially damaging characters and statements should be filtered out or substituted on the server side of the Web application. Many Web application developers filter input on the client side, using Javascript or other techniques, mistakenly thinking that will stop SQL injection and related attacks. Yet, an attacker can bypass any client-side filtering using a Web application manipulation proxy like Achilles or Paros to inject arbitrary data into the HTTP or HTTPS connection. Remember, the browser is potentially enemy territory, so any filtering that occurs there can be subverted by the attacker. Even pull-down menus can be subverted using a proxy, as an attacker adds further options to the menu via a proxy that can include SQL injection and related attacks.

Another level of defense against SQL injection involves limiting the permissions of the Web application when accessing the database. Don't let your Web application have database administrator capabilities on your database! That's incredibly dangerous. Build the Web application and configure the database so that the Web application logs in with a very limited permission account, with the ability to view and update only those fields of those tables that are absolutely required. Clamping down on these permissions won't eliminate SQL injection attacks, but it can really limit the attacker's ability to explore the database fully.

Finally, Web application developers should consider the use of parameterized stored procedures in their applications. In the examples we've discussed here, the Web application gathers user input and uses it to compose database query strings, which

skoudis.book Page 431 Wednesday, November 30, 2005 10:04 PM

it then forwards to the database for execution. Composing these queries on the fly at the Web application results in SQL injection when attackers provide SQL-relevant commands or operators in user input. A Web architecture that uses parameterized stored procedures, on the other hand, doesn't feed raw SQL statements generated by the Web application into the database. Instead, this architecture relies on stored procedures, code that runs on the database server itself, to interact with the database. By moving the logic for interacting with the database to the database server, the Web application can provide the stored procedure a set of discrete parameters drawn from user input that are used in queries defined within the stored procedure itself. The stored procedure breaks down the user input into the individual parameters that need to be fed into the database search. Because the query logic isn't created on the fly, but is instead coded into the stored procedure relying on user input merely as a set of parameters, stored procedures help minimize the chance of SQL injection.

In this section, we've looked at three of the most common attacks against Web applications, namely account harvesting, state manipulation, and SQL injection. These are three of the biggest Web application attacks, but there are many other vulnerabilities that Web applications could face, including cross-site scripting (which involves bouncing a malicious browser script off of a Web site) and command injection (which lets an attacker inject operating system commands in user input), among many others. To learn more about such flaws, the single best source freely available on the Internet is the Open Web Application Security Project (OWASP) at *www.owasp.org*. Everything created by the team at OWASP is free and open source. They are the people behind WebGoat, as well as numerous other tools for testing and securing Web applications.

Their *Guide to Building Secure Web Applications and Web Services* is quite comprehensive, including details associated with design, architecture, implementation, event logging, and more! It really is a must-read for any Web developer today.

EXPLOITING BROWSER FLAWS

Thus far, we've focused on attacking Web applications involving bad guys undermining the logic that lives on Web servers for nefarious purposes. However, a significant and scary trend involves attackers coopting e-commerce sites and using them as a delivery mechanism for malicious code to vulnerable Web browsers.

skoudis.book Page 432 Wednesday, November 30, 2005 10:04 PM

Numerous browser vulnerabilities are discovered on a regular basis, especially (but not exclusively) in Internet Explorer. There are several types of browser holes, including buffer overflows, flaws that let an attacker escape the security restrictions on scripts or other active Web content (such as the Java runtime environment), exploits that let malicious code bypass cryptographic signature checks, and problems that let malicious code execute in a different security zone than it should. All of these problems could be triggered if the victim surfs to the wrong Web site with a vulnerable browser.

Microsoft, as well as other vendors, has historically not rated such browser flaws as critical, because they say that the victim user must be tricked into surfing to the attacker's Web site. If users surf only to trusted sites, they should be unaffected by such problems, or so the thinking goes.

However, this assumption is false, as we saw in several major attacks, with many more likely in the future. In these attacks, the bad guys first undermined trusted midsized e-commerce sites. The attackers installed code on these Web sites that would exploit browser vulnerabilities when an unsuspecting, but trusting, user surfed to these e-commerce sites. Later, when users surfed to the e-commerce sites, their browsers were exploited, and malicious code was inserted on their machines.

In June 2004, this attack was pulled off using the Download.Ject flaw in Internet Explorer that let a Javascript run arbitrary code on a vulnerable browser that surfed to a site hosting Download.Ject exploitation software. Attackers took over a dozen e-commerce sites using various buffer overflow attacks, and installed browser-exploiting code there. When a user surfed to one of the infected Web sites, the Download.Ject flaw in the user's browser was triggered, causing the victims to download a keystroke logger program called berbew from a Russian Web site. This keystroke logger grabbed financial information from the browser, including account numbers and passwords for e-commerce sites and banks, as illustrated in Figure 7.40. Here is the flow of these increasingly common types of attacks:

1. The attacker takes over some e-commerce or other trusted site on the Internet. The attacker installs code on this site that can exploit browser vulnerabilities.
2. An innocent victim surfs to the infected Web site.

skoudis.book Page 433 Wednesday, November 30, 2005 10:04 PM

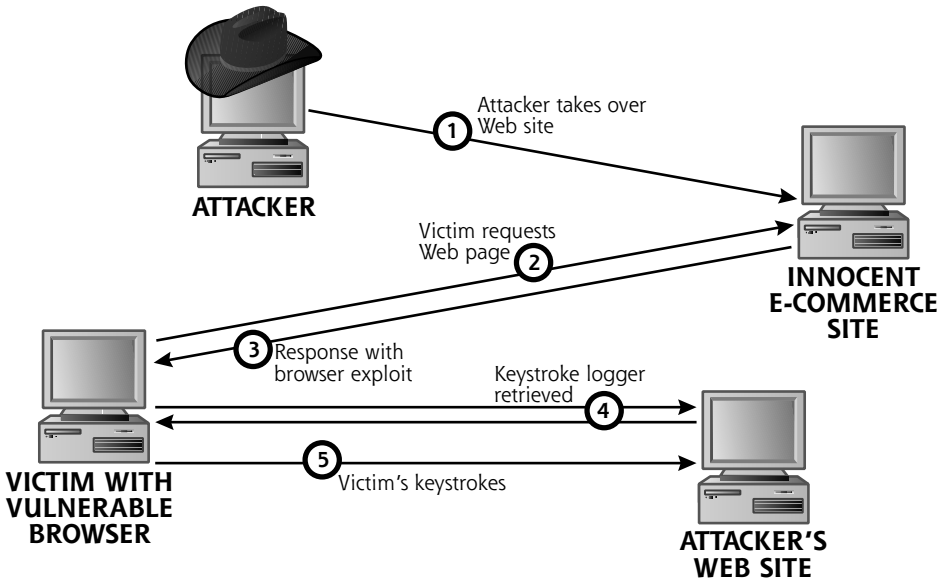


Figure 7.40 Compromising an e-commerce site and using it to deliver keystroke loggers to victims with vulnerable browsers.

3. The infected Web site responds with a Web page that exploits the browser.
4. Based on the exploitation of Step 3, the browser connects to the attacker's site and grabs some malicious code from it, such as a keystroke logger, a bot, or a worm.
5. The evil code on the victim's machine now runs, doing nasty stuff to the user, such as stealing his or her keystrokes.

In November 2004, we saw a similar attack, this time exploiting an at-that-time-unpatched buffer overflow in Internet Explorer called the IFRAME flaw. This time, the attackers took over some advertising sites that posted banner ads on a variety of other news and e-commerce Web sites. If you viewed any of these ads at any of these sites with a vulnerable browser, you'd get a worm called Bofra installed on your machine. Bofra would steal sensitive information and try to take over other nearby systems.

skoudis.book Page 434 Wednesday, November 30, 2005 10:04 PM

As users increasingly deploy personal firewalls to block the automated propagation of malicious code to their machines, such browser-based attacks will likely grow in prominence. By riding through a user's normal Web surfing and exploiting browser holes, the attacker's actions bypass the personal firewall on a machine. The vast majority of personal firewalls are configured to allow one or more Web browsers to access the Internet, thus poking a significant hole in the protection offered by the firewall if the browser itself is vulnerable.

DEFENDING AGAINST BROWSER EXPLOITS

These browser-based exploits are an increasing threat, but how do you defend against such attacks?

First, keep your browsers patched. If there's a new hole reported in a browser, make sure to patch it immediately. Unfortunately, both the June and November 2004 attacks exploited holes for which there was no patch yet released. Still, it's a good idea to keep your systems patched.

Next, utilize an up-to-date antivirus tool on all systems, especially those machines that browse the Internet. Happily, the code used in most of these attacks so far was detectable with antivirus tools by the time the attack was widespread, which prevented many users from being compromised.

Furthermore, you might want to consider using a browser other than Internet Explorer. I don't want to start a product war here. However, Internet Explorer is a major target for these types of attacks, given its market dominance. Other browsers have holes, too, but they are less likely to be targeted by attackers, simply because fewer people use them. The attackers are looking for lots of easy prey, and Internet Explorer users sure are a large population. However, please do not underestimate the amount of work needed to transition to another browser. For personal users, learning a new browser might take some time. In enterprise environments, a different browser might break some of your critical applications. Recoding those applications could take significant resources, thus making a transition to another browser financially impossible.

skoudis.book Page 435 Wednesday, November 30, 2005 10:04 PM

CONCLUSION

Throughout this chapter, we've seen powerful techniques 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. This chapter explores techniques for gaining access by manipulating applications and operating systems.

Buffer overflows are among the most common and damaging attacks today. They exploit software that is poorly written, allowing an attacker to enter input 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 or heap, a buffer overflow could result. Without this proper bounds checking, an attacker can send 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. For heap-based buffer overflows, an attacker can manipulate other variables in the heap, and possibly execute malicious code.

Exploitation frameworks like Metasploit help automate the production and use of exploits, such as stack-based and heap-based buffer overflows. These tools

skoudis.book Page 436 Wednesday, November 30, 2005 10:04 PM

let attackers write modular exploits and payloads, tying the two together in an easy-to-use interface.

Defenses against buffer overflow attacks include applying security patches in a timely manner, filtering incoming and outgoing traffic, and configuring systems so that their stacks cannot be used to store executable code. Software developers can also help stop 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 or hashed passwords from a system and using an automated tool to determine the original passwords. Password-cracking tools create password guesses, encrypt or hash the guesses, and compare the result with the encrypted or hashed password. The password guesses can come from a dictionary, brute-force routine, or a hybrid technique. Cain is one of the best tools for cracking passwords on Windows machines. On UNIX systems (as well as Windows), 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 (with appropriate permission from management) to enforce the policy. You might 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 variables, and SQL injection. 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 user IDs and passwords are consistent.

Attackers can undermine Web application session tracking by manipulating URL parameters, hidden form elements, and cookies to try to clone another user's session. To defend against this technique, make sure your applications use strong

skoudis.book Page 437 Wednesday, November 30, 2005 10:04 PM

session tracking information that cannot easily be determined by an attacker and protect all variables passed to a browser.

SQL injection 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 and make sure your Web application logs in to a database with minimal privileges.

Numerous browser-based vulnerabilities let an attacker take over a browsing machine that surfs to an infected Web server. By compromising trusted Web servers, attackers can spread their browser exploits to a large population. To defend against such attacks, keep your browsers patched, and utilize up-to-date antivirus tools.

skoudis.book Page 438 Wednesday, November 30, 2005 10:04 PM