COMMUNITY FAVORITES Save up to 70% on user favorite books, eBooks, and video training. Learn more.

the trusted technology learning source

Gaining Access to Target Systems Using Application and Operating System Attacks

By Edward Skoudis and Tom Liston

Nov 21, 2007

This chapter is from the book

Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses, 2nd Edition

Learn More Buy

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.

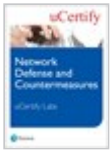
Buffer overflow vulnerabilities are based on an attacker sending more data to a vulnerable program than the original software developer planned for when writing the code for the program. The buffer that is overflowed is really just a variable used by the target program. In essence, these flaws are a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

Related Resources

Store | Articles | Blogs

CISSP Pearson uCertify Course and Labs Access Card, 2nd Edition
By Robin Abernathy, Troy McMillan, uCertify
Book $119.00

CISSP Pearson uCertify Course and Labs and Textbook Bundle, 3rd Edition
By Robin Abernathy, Troy McMillan, uCertify
Book $136.00



Network Defense and Countermeasures uCertify Labs Access Card
By uCertify
Book $119.00

See All Related Store Items

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 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.

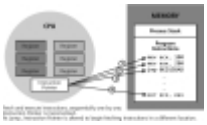


Figure 7.2 How programs run.

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.

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.

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.

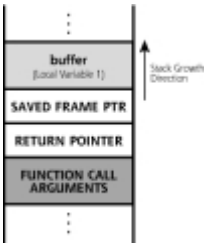


Figure 7.4 A normal stack.

Next, the system pushes the Frame Pointer on the stack. This value helps the system refer to various elements on the stack itself. Finally, space is allocated on the stack for the local variables that the function will use. In our example, we've got one local variable called `buffer` to be placed on the stack. These local variables are 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.

Figure 7.5 Some very vulnerable C code.

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`.

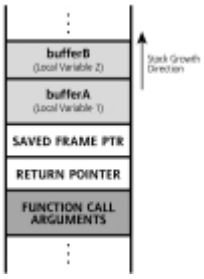


Figure 7.6 A view of the stack of the vulnerable program.

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 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!

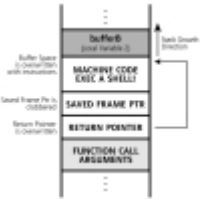


Figure 7.7 A smashed stack.

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 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 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.

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 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 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 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.

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. 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*.

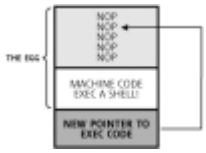


Figure 7.8 The structure of an exploit (also known as a sploit) for a buffer overflow vulnerability.

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.

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.

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

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.

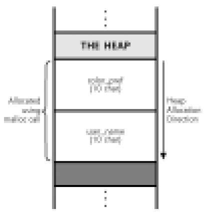


Figure 7.10 The heap holds the memory we malloc'ed.

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.

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 blueblueblueblueroot. 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 blueblueblueblue) 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.

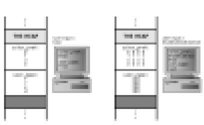


Figure 7.11 Running the vulnerable program with two different inputs.

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. 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 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 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.

Figure 7.12 The components of Metasploit.

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.
- *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 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.

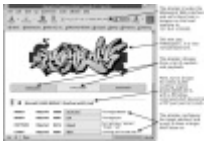


Figure 7.13 Metasploit's Web-based interface.

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 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 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 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 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 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 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:

```
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 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.

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

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 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). 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.

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.

+ Share This Save To Your Account

< Back **Page 3** of 8 Next >