

### 3.1 Techniques to Gain a Foothold

#### 3.1.1 *Shellcode*

The wide availability of shellcode, injectable binary code used to perform custom tasks within another process, makes it simple for even novice attackers to create highly reliable payloads for use after exploitation. Initially, shellcode simply spawned a shell from another process; however, it is now used to perform a variety of custom tasks. This section explains what shellcode is, how it works, and the threat it poses.

Malicious code comes in many forms, and ranges from standalone executables to injectable code. Varying in execution requirements, development techniques, and the resulting behavior, malice is the common motive. In this section, we focus on injectable code known as *shellcode* and explains what it is and how it works.

Shellcode is binary code used as the payload in exploitation of software vulnerabilities. The name *shellcode* originates from its initial intentions to spawn a shell within another process but has since evolved to define code that performs any custom tasks within another process.<sup>1</sup> Written in the assembly language, shellcode passed through an assembler creates the binary machine code that the central processing unit (CPU) can execute. When an attacker exploits a vulnerability in a specific process, the shellcode payload is executed seamlessly, acting as if it were part of the original program.

The use of assembly as a low-level programming language instead of a compiled high-level programming language or interpreted scripting language makes shellcode operating system specific. Shellcodes typically do not port well between Linux, UNIX, and Windows platforms due to the differences between system calls and the way these calls use the CPU.

Linux and UNIX operating systems are similar, as they both use system calls to allow processes to interact with the kernel. These system

calls allow shellcode to perform tasks, such as reading or writing to files, binding, and listening on sockets, with the same permissions as the original process into which the shellcode was injected. To perform a system call, the shellcode issues an interrupt (INT 0x80) with a syscall number to specify which function to perform. Syscall numbers are static and do not change between versions of Linux and UNIX kernels. Linux and UNIX use the same interrupt, but Linux puts the system call number and arguments into CPU registers before issuing the interrupt, while UNIX platforms push the system call number and arguments to the stack memory.<sup>2</sup> Due to the differences in register and stack usage during system calls, Linux and UNIX shellcode are not interchangeable; however, a shellcode developer can easily include runtime checks to determine the operating system based on the differences in system call semantics to create shellcode that runs in both Linux and UNIX environments. Windows uses a different set of syscall numbers, which renders Windows shellcode incompatible with Linux and UNIX kernels.

Windows operating systems include system calls, but the limited set of functions and the variance of syscall numbers between versions reduce the effectiveness and reliability of the shellcode. Windows provides an application programming interface (API) through the use of dynamically linked libraries (DLLs) for applications to interact with the kernel. Reliable shellcode typically locates `kernel32.dll` and uses the `LoadLibraryA` and `GetProcAddress` functions to access other DLLs for additional functionality. The functionality of the shellcode and loaded DLLs shares the same permissions of the victim process. Attackers use several different techniques to locate `kernel32.dll` for use within shellcode. The location of `kernel32.dll` is included within the process environment block (PEB), which is a structure that stores process information and can provide the location of `kernel32.dll` since it is always the second module initialized during process startup. Shellcode can search for the location of `kernel32.dll` within the module initialization order stored within the PEB. Another technique called *structured error handling* (SEH) scans through a process' exception handlers to find the default unhandled exception handler as it points to a function within `kernel32.dll`. The shellcode looks for magic bytes, `MZ`, for the beginning of a portable executable and uses this location for `kernel32.dll`. Another method in obtaining `kernel32.dll` includes

walking the thread environment block (TEB) for the magic bytes MZ similar to the SEH approach. After locating kernel32.dll, the shellcode needs to find the location of functions within the DLL. To find these functions, the shellcode queries the export directory and import address table for the virtual memory addresses (VMAs).<sup>3</sup> Once found, these functions can provide the shellcode with access to a wide range of commands comparable to syscalls in Linux and UNIX.

Using system calls and platform APIs, customized shellcode injected into a process can perform virtually any actions on the computer that the author desires. Shellcode development does have limitations, however. Shellcode is dependent on a parent process and usually requires specific encoding to run successfully. In many cases, programming functions copy shellcode into memory as a null-terminated string. These functions copy data into memory until reaching a NULL value and stop. A null-terminated string uses a NULL character to determine the end of the string, which requires the author to remove all NULL bytes from the shellcode to save the entire code to memory. To remove NULL bytes from the shellcode, the instructions that have hexadecimal values of 0x00 need alternative instructions that do not include nulls but perform the same functionality. Exhibit 3-1 shows an assembly instruction to move a value of zero into the CPU's ebx register, which presents four NULL bytes in the byte representation. The NULL free version shows a comparable assembly instruction to set ebx to zero by performing an exclusive or on ebx with itself, and the byte representation shows zero NULL bytes.

Applications can also include input validation to limit copying nonalphanumeric ASCII characters into buffers. Nonalphanumeric characters are common within shellcode due to the hexadecimal values generated when using assembly commands. To circumvent these nonalphanumeric filters, the shellcode can use a limited set of assembly commands that carry hexadecimal values of 0x30–0x39,

|                      | Instruction Contains NULLs |            | NULL Free Version |
|----------------------|----------------------------|------------|-------------------|
| Assembly Instruction | MOV EBX                    | 0x0        | XOR EBX, EBX      |
| Byte Representation  | 0xBB                       | 0x00000000 | 0x31DB            |

**Exhibit 3-1** A NULL free optimized shellcode.

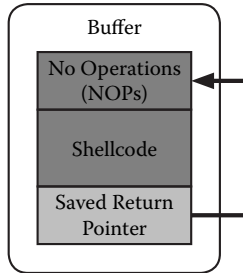
|                      |      |            |         |          |            |
|----------------------|------|------------|---------|----------|------------|
| Assembly Instruction | PUSH | 0x64726f77 | POP EAX | XOR EAX, | 0x64726f77 |
| ASCII Representation | h    | word       | X       | 5        | word       |

**Exhibit 3-2** An alphanumeric shellcode.

0x41–0x5A, and 0x61–0x7A for the alphanumeric characters of 0–9, A–Z, and a–z.<sup>4</sup> An author can develop shellcode using this limited set of commands to perform desired actions and still pass alphanumeric input validation. Exhibit 3-2 shows the assembly instructions to create alphanumeric shellcode to set the CPU’s eax register to zero. The code has a hex value of 68 77 6f72 64 58 35 77 6f72 64, as seen in the byte representation, which translates to the ASCII representation of *hwordX5word* to pass alphanumeric filters.

Typically faced with size limitations, a programmer must develop shellcode that fits in the exploited process’ buffer to inject all of the shellcode into memory. If a shellcode cannot fit within the buffer, a stage-loaded shellcode can incorporate larger pieces of code stored elsewhere. The stub code of stage-loading shellcode, called *stage 1*, uses many different methods to locate the larger piece of code known as *stage 2*. One method reads and executes second-stage shellcode from a remote server by reusing the file descriptor created by the operating system for the inbound connection to inject the initial shellcode. Another method uses an *egg hunt* strategy, which scans memory for a unique piece of the second-stage shellcode, known as an *egg*, and begins execution.

Typically, attackers use shellcode as the payload of an attack on a vulnerability. A successful attack on a vulnerability injects the payload into the targeted process, resulting in code execution. Buffer overflows are a common exploit technique using shellcode as a payload to execute code within the targeted process. A buffer is a segment of memory that can overflow by having more data assigned to the buffer than it can hold. Stack buffer overflows<sup>5</sup> attempt to overwrite the original function return pointer stored in memory to point to the shellcode for the CPU to execute. No-operation-performed (NOP) instructions precede the first instruction in the shellcode to increase the reliability of the return pointer landing on the code. Known as a NOP sled, the NOP instructions tell the CPU to perform no operation and to move



**Exhibit 3-3** Shellcode execution in a stack buffer overflow.

on to the next instruction, eventually executing the shellcode. Exhibit 3-3 shows how a stack buffer overflow results in code execution.

There are many shellcode repositories on the Internet to aid an attacker in developing an attack payload. Sites such as milw0rm.com and metasploit.org contain shellcodes sorted by platform to perform a variety of actions ranging from downloading and executing files to sending a shell to a remote client. In addition to these repositories, there are a number of shellcode generators that convert code written in a high-level programming language into injectable shellcode. For example, ShellForge is a shellcode generator that creates shellcode from code written in C and includes modules to create NULL free and alphanumeric shellcode.<sup>6</sup> Metasploit also offers a shellcode generator that uses an intuitive interface to create customized shellcode for Windows, Linux, and UNIX platforms with multiple encoding modules available.<sup>7</sup>

Shellcode detection technologies include intrusion detection and prevention systems (IDSs/IPSs) and antivirus products. Network-based IDS/IPS appliances use pattern-matching signatures to search packets traveling over the network for signs of shellcode. Most antivirus products offer pattern-matching signatures to detect shellcode within files on the local system. These pattern-matching signatures are prone to false positives and are evaded by encrypted shellcode. Some IDS/IPS appliances minimize evasion by involving x86 emulation technologies, such as libemu, to detect shellcode based on the behavior of executing instructions found in network traffic and local files.

Shellcode development continues to create ways to perform a variety of tasks after vulnerability exploitation. The malicious possibilities provided by shellcode allow attackers to expose information on a victim

computer further. The public availability of shellcode and shellcode-generating tools enables novice hackers with minimal knowledge of shellcode to use highly reliable payloads within attacks.

### 3.1.2 Integer Overflow Vulnerabilities

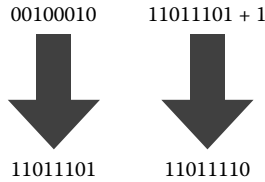
Resulting from insufficient input validation, integer overflows can cause high-severity vulnerabilities. Not to be confused with buffer overflows, integer errors are common and potentially severe.

A computer's central processing unit (CPU) and memory represent integers. Software companies often supply these as input to their programs in binary formats. Integers might represent the size of a packet or length of a string, and applications frequently rely on them when making key decisions about how a program should proceed. If the program does not perform a sanity check on integers, unexpected and potentially dangerous consequences may result. In this section, we explain integers and how errors in integer operations may cause an integer to overflow.

The effect that integer overflows can have varies greatly depending on how the vulnerable application uses the integer. Integer overflows can lead to 100 percent use of CPU resources, denial of service (DoS) conditions, arbitrary code execution, and elevations of privileges. Like many vulnerabilities, integer overflows result when programmers do not consider the possibility of invalid inputs.

CPU registers cannot store integers with infinite values. Their maximum value depends on the width of the register in bits. Additionally, there are two types of integer representations: unsigned and signed. Unsigned integers represent only positive numbers, and signed integers can represent positive and negative numbers. An unsigned integer is positive or zero and includes the positive values 0 through  $2^n - 1$ , where  $n$  represents the number of bits the CPU register can hold. An 8-bit register width, therefore, has a maximum value representation of  $2^8 - 1 = 255$ .

A signed integer, on the other hand, includes the negative values  $-2^{(n-1)}$  through  $2^{(n-1)} - 1$ . For 8-bit signed integers, negative values have a minimum and maximum value range of  $-2^8 = -128$  to  $128 - 1 = 127$ . Most modern computers use *two's complement* to represent signed integers. Converting a number to two's complement is a two-step process.



**Exhibit 3-4** (Left) The decimal number 34 in binary is converted to its one's complement by inverting each of the bits. (Right) 1 is added to the one's complement representation, resulting in the two's complement representation of  $-34$ .

First, the binary representation of the number is negated by inverting each of the bits. This format is known as *one's complement*. The second step is to simply add one to the one's complement, and the resulting value is the two's complement form. Two's complement representation is thus the result of adding one to a one's complement representation of a negative integer.

Consider the transformation of  $-34$  from one's complement to two's complement in Exhibit 3-4. In this example, 34 (00100010) is converted to  $-34$  (11011110). The use of the two's complement form to represent negative integers means that the most significant bit (the one furthest to the left) will always be a 1 if the number is negative. Because of this, that bit is often referred to as the *sign bit*.

An integer overflow occurs when an arithmetic operation produces a result larger than the maximum expected value. An integer that increases beyond its maximum value could result in a potential error condition that attackers may exploit. Integer overflow occurs in signed and unsigned integers. A signed overflow is the result of a value carried over to the sign bit.

Exhibit 3-5 shows an example of an integer overflow in an 8-bit signed integer. The signed integer 126 (01111110) is incremented by 1, producing the resulting value of 127 (01111111). Incrementing 127 overwrites the sign bit, resulting in a negative value of  $-128$  (10000000 in binary) instead of a positive value of 128, as would occur if the integer was unsigned.

|               |      |
|---------------|------|
| 126 01111110  | ↓ +1 |
| 127 01111111  | ↓ +1 |
| -128 10000000 |      |

**Exhibit 3-5** An 8-bit signed integer overflow.

```

254 11111110   ↓ +1
255 11111111   ↓ +1
0 00000000

```

---

**Exhibit 3-6** An 8-bit unsigned integer overflow.

An unsigned overflow is the result of a value no longer representing a certain integer representation because it would require a larger register width. The unsigned integer “wraps around” when an overflow occurs. An unsigned integer wrap-around occurs when a program receives a large input value that wraps the unsigned integer back to zero or a small positive number (see Exhibit 3-6).

Exploits take advantage of integer overflows indirectly. It is impossible to determine if an integer overflow has occurred without understanding the code’s underlying arithmetic context. An example of an integer overflow involves an arithmetic operation error using a C and C++ standard library function called *malloc*. *malloc()* is for allocating a block size of dynamic memory. An integer overflow can cause *malloc()* to allocate less memory than required. *malloc(int size)* takes a single integer as an argument that specifies how much memory the function should allocate for the buffer it creates. The arithmetic operation error typically occurs before the call to *malloc()*, and the result of the operation is used in the call to *malloc()*. Consider this example code in which multiplication and *malloc()* are involved:

```

char *expand_string(char *string, size_t length) {
char *strresult = (char *)malloc(length*2+1);
strcpy(strresult, string);
}

```

This function takes one string and then copies it to a buffer that is larger than the original string to make space for additional characters. In the second line, the code takes the length input, multiplies it by two, and adds one before calling *malloc()* to allocate the specified amount of memory. If an attacker controls the value of the parameter length and sets it to 0x80000000 (2,147,483,648 in decimal), an integer overflow will occur as *malloc(0x80000000\*2+1)* becomes *malloc(1)*. 0x80000000\*2 returns 0 because the leftmost bit of the result shifts out of the value. Only one byte of memory was allocated, but all of the data in the input string will be copied to that buffer by the



strcpy() function. This will result in a buffer overflow that an attacker could use to execute arbitrary code on the system.<sup>8</sup>

The vulnerable code is missing one step that programmers frequently forget: input sanity checks. This function should include one additional step that ensures the input length value is not so large that it will overflow after the arithmetic operation is performed. Integer operation errors also include sign and truncation errors. In sign errors, the sign flaw occurs due to the ignoring of the sign bit in the conversion of an integer. In truncation errors, an integer value truncates while being cast to a data type with fewer bits. Consider the following example:

```
int i = -3;
unsigned short int j;
j = i; // j = 65533
```

First, the code declares *i* as an int (a signed integer type) and assigns it the value of -3. Next, it declares *j* as an unsigned short integer, a variable that can only represent positive integers. When a signed integer of a negative value converts to an unsigned integer of greater value, the most significant bit loses its function as a sign bit. The result is that *j* is not set to -3, but rather to 65,533, the unsigned representation of -3.

Most integer errors are type range errors, and proper type range checking can eliminate most of them. Many mitigation strategies are available that apply boundary checking on integer operations. Some of these include range checking, strong typing, compiler-generated runtime checks, and safe integer operations. Range checking involves validating an integer value to make sure that it is within a proper range before using it. A programmer could add a line in the code to check if the length of an integer is greater than 0 or less than the maximum length before going to the next line of code. This strategy is effective but relies on the programmer to make good choices when writing the potentially vulnerable function.

Strong typing involves using specific types, making it impossible to use a particular variable type improperly. A programmer can declare an integer as an unsigned char to guarantee that a variable does not contain a negative value and that its range is within 1 to 255. This strategy also relies on programmers to make good decisions when

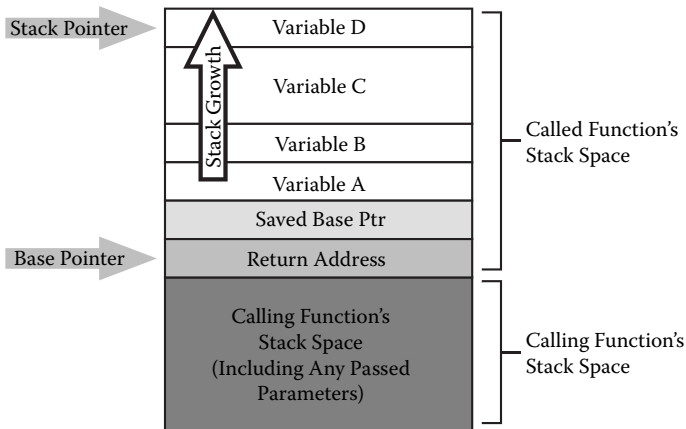
choosing variable types. Compiler-generated runtime checks involve using compiler flags when compiling a program. These flags instruct the compiler to add additional code to the program that checks if an integer is going to overflow and will throw an error when this occurs. Unfortunately, these functions can also contain errors, as is the case with the GCC-`ftrapv` flag.<sup>9</sup>

Safe integer operations involve using safe integer libraries of programming languages for operations where untrusted sources influence the inputs. `SafeInt`<sup>10</sup> and `RCSint` classes are available for C++ and provide useful templates to help programmers avoid integer overflow vulnerabilities. These classes contain automatic checking of common operations that throw exceptions when overflows occur rather than silently failing. Integer overflows are most common in C and C++ programs. They can occur in other languages, but do so much less frequently, since many languages do not allow low-level access to memory like C and C++. Some languages like Java, Lisp, and Ada provide runtime exceptions in cases that would lead to buffer overflows in C and C++. As of May 13, 2009, the National Vulnerability Database<sup>11</sup> reported forty-five CVE matching records for integer errors in the past three months and 333 matching records with high-severity ratings for integer errors in the past three years. While not as common as buffer overflows, it is clear that integer overflows remain a common and severe threat.

### *3.1.3 Stack-Based Buffer Overflows*

In this section, we will explore the concept of stack-based buffer overflows. At the core of any overflow is the lack of boundary consideration. More precisely, overflows are the result of a finite-sized buffer receiving data that are larger than the allocated space. Stack-based buffer overflows are by far the most common type of overflow, as they are generally the easiest to exploit. To understand this, the reader must first understand the mechanics of the stack in Intel-based computers.

*3.1.3.1 Stacks upon Stacks* At a high level, the standard computer stack consists of an array of memory bytes that a programmer can access randomly or through a series of pop-and-push commands. Computer scientists classify stacks as last-in-first-out (LIFO) structures, meaning



**Exhibit 3-7** A typical stack layout.

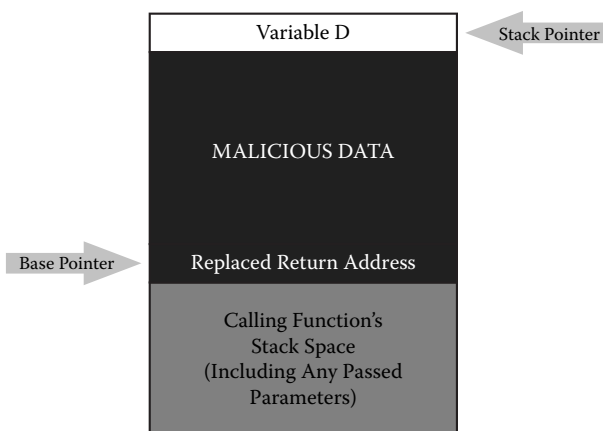
that the last datum added (or “pushed”) to the stack is the first datum pulled (or “popped”) from the stack. In the Intel implementation of a stack and various other processor platforms, local functions use the stack for temporary variable storage, parameter passing, and code flow execution management. Exhibit 3-7 illustrates the typical layout for an Intel stack used to call a function and the called function.

The Intel x86 processor uses two special registers for stack management: the stack pointer (ESP) and the base pointer (EBP). The processor uses the stack pointer to specify the memory address of the last memory pushed onto the stack and the location to use when popping the next datum from the stack. While the size of the datum can be a byte, a word, a 32-bit DWORD, or a 64-bit QWORD value (for 64-bit processors), the most common convention is a DWORD stack access, since pointers and integers are typically 32 bits in size. The called function normally uses the base pointer to point to the original stack pointer location immediately after the function begins. Known as setting up a stack frame, the function uses the EBP to provide a fixed reference point by which it accesses the stack variables. The EBP becomes critical when using the stack frame, as the ESP floats with each push-and-pop off the stack from subsequent function calls (from the perspective of the original called function). After the function assigns the EBP and the value of ESP immediately after the function begins, the function will adjust ESP to accommodate the necessary memory space for local variables used by the function.

This movement of the ESP will place the pointer at the end of the local variable data.

Once the calling function has completed, the function resets the stack pointer to the original location, as specified by the base pointer. The processor, upon seeing the “return” instruction that terminates the function, pops the return address from the stack, as pointed to by the ESP. The processor uses this address as the location of the next instruction to execute. This fact, alone, represents the major target of most stack-based buffer overflows.

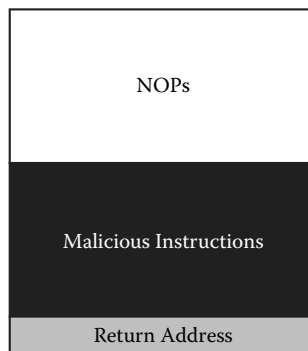
**3.1.3.2 Crossing the Line** Stack-based buffer overflows occur when a function passes more data to a stack variable than the variable can hold. The objective in writing stack-based buffer overflows is to control the flow of code execution and execute potentially malicious code by adding more data than a variable can hold. The most common way this type of attack works is by finding a buffer on the stack that is close enough to the return address and attempting to place enough data in the buffer such that the attacker can overwrite the return address. If successful, the attacker can then influence the next instruction that the processor executes. As seen in Exhibit 3-8, an attacker has managed to fill the buffer (Variable *C*) with a sufficiently large data stream so that the attacker has overwritten Variables *A* and *B*, the saved base pointer, and the return address with the attacker’s values. As a result, the processor uses the values specified in the original location of the



**Exhibit 3-8** Overflowing a stack buffer to control the return address.

return address upon the completion of the function. The processor, unaware of the data's validity, will attempt to execute the instructions at this location. This allows the attacker who filled Variable *C* with the oversized data to directly influence the flow of code execution.

Simply filling a stack buffer with data and overwriting the return address are not enough for attackers to run arbitrary code on victims' computers successfully. Controlling the return address is only one part of a successful stack-based buffer overflow. Attackers still need ways to run their own codes on victims' computers. Many, if not all, attackers approach this problem by sending specially constructed data to the vulnerable buffer. Known as shell code, attackers construct the data sent to the vulnerable buffer such that the first portion of the data contains what attackers call a no-operation-performed (NOP) sled, which is an array of NOP instructions. This portion of the shellcode is optional, but allows attackers to miss their marks when effecting code execution. If attackers can get processors to execute malicious code somewhere within the NOP sled, those processors will not jump into the middle of valid instructions that can cause the flow of execution to deviate drastically from the attackers' original intents. The actual malicious code follows the NOP sleds. Known as the payload of the shellcode, this section of data contains the malicious code that attackers desire to execute. The values for the return addresses follow the payload. Exhibit 3-9 displays the makeup of a typical shellcode data stream. A successful shellcode will allow attackers to control the return addresses and point the return addresses to memory locations that will allow those attackers to begin executing either instructions



**Exhibit 3-9** Anatomy of a typical shellcode structure.

in the NOP sleds (which then run into the first instruction of the payload) or very specific bytes of the payloads. The results are, of course, the execution of arbitrary code on victims' computers.

*3.1.3.3 Protecting against Stack-Based Buffer Overflows* Without exception, the root cause of any stack-based buffer overflow is the lack of bounds checking when accepting input. Buffers allocated on a stack are of finite, predetermined sizes, and as such, it is up to the programmer to ensure that the function copying data into them is within this size constraint. Generally, a program should validate any data taken from an external source (external from the perspective of the application) for both size and content; user-supplied data should never be trusted outright.

Compilers have begun using an additional built-in technique to aid in protecting bad programmers from themselves when it comes to stack-based buffer overflows. Compilers, such as Microsoft's Visual C++ 2005 and 2008, use a technique known as *stack cookies* to ensure that overflowed variables in a stack do not result in a processor using an invalid return address. The technique consists of placing a randomized 32-bit value immediately before the return address on the stack. Before the function terminates, it compares the value on the stack to the original value from the start of the function. If the two values do not match, the program terminates due to a security violation. By preventing the function from returning, the processor never executes the modified return address, and as such, the malicious code does not execute.

Stack cookies do help mitigate the problems associated with stack-based buffer overflows, but they do not provide an excuse for programmers to avoid secure programming techniques.<sup>12</sup> Ultimately, it is the responsibility of the programmer to write secure code to prevent stack-based buffer overflows.

*3.1.3.4 Addendum: Stack-Based Buffer Overflow Mitigation* The previous section described how stack-based buffer overflows work. The mitigation section focused on tips for programmers to avoid writing code that contains these vulnerabilities but did not mention technologies designed to mitigate the threat that have recently been added to operating systems and computer hardware.

Data execution prevention (DEP) is a feature built into modern processors, and which Windows can take advantage of to mark certain areas of memory (like the stack) as nonexecutable, making it much more difficult to exploit a buffer overflow. Certain third-party software, like antivirus programs, also offer buffer overflow prevention (BOP) technology that performs a similar function by monitoring process memory and alerting the user when the processor executes memory that should only contain data. Address space layout randomization (ASLR) is a feature of Windows Vista and the Linux 2.6 kernel that makes the OS load system libraries in random locations each time the computer boots, making it more difficult to write shell code that can do something useful once the attacker has exploited overflow.

#### 3.1.4 *Format String Vulnerabilities*

This section discusses vulnerabilities in the `printf` print formatting and similar functions. These vulnerabilities put the stack, a critical component of program execution, at risk for corruption. Despite this type of vulnerability being first reported in 2000, iDefense rarely sees exploit code using this type of vulnerability to install malicious code or compromise servers. Nevertheless, exploit developers release many format string vulnerabilities each year, and these vulnerabilities continue to present a threat to application security.

The `printf` C function (short for *print-formatted output*) normally prints output to the screen. It accepts a parameter that allows a programmer to specify how the function should attempt to interpret the string. For example, the programmer may want to print the character *A*, the value of *A* as the number 65, or the hex representation of *A*, which is 0x41. For these reasons, `printf()` accepts its first parameter, *format*.

```
int printf(const char *format, ...)
```

The *format* is a string that accepts strings as *%s*, decimal numbers as *%d*, or hex values as *%x*. The *%n* format specifies an important component to exploit format string vulnerabilities because it allows the `printf` function to write to an arbitrary memory location with the number of bytes written so far. The “...” parameter is a way in the

C programming language to accept an unknown number of parameters using a `va_list` structure to store them. Normally, a compiler will compare the number of parameters that a function accepts to the function definition to prevent programming mistakes; however, the `printf` function and other similar functions accept different numbers of parameters depending upon the format string itself. Using only one parameter to `printf` can create a vulnerability if a user can influence that parameter. Again, the format parameter can accept many different formats such as strings (`%s`), decimal numbers (`%d`), or hex values (`%x`). A legitimate call might look like this:

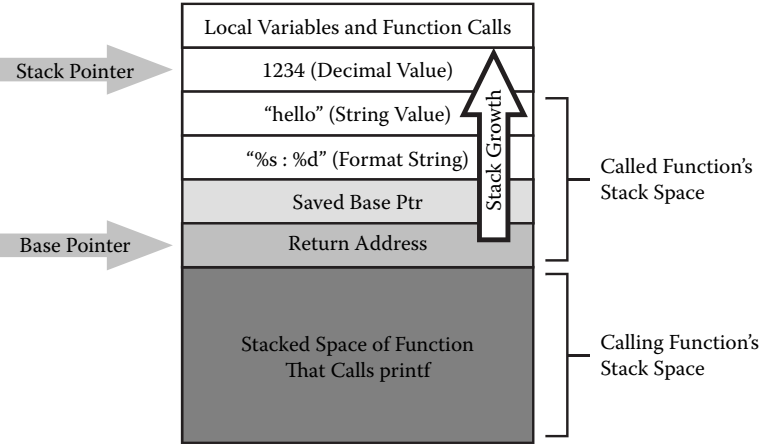
```
string = "hello";  
decimal = 1234;  
printf("%s : %d", string, decimal); // Three  
parameters total
```

This would print out the following:

```
"hello : 1234"
```

To do this, the program pushes the parameters onto the stack before calling the `printf` function (see Exhibit 3-10).

Then, the `printf` function utilizes the `%s : %d` format to determine how many variables it should remove from the stack to populate the values. To execute a simple attack, malicious actors compile a very simple C program, `format_string_vulnerable`, that accepts



**Exhibit 3-10** A stack diagram of a legitimate call to `printf`.



command-line arguments and passes them directly to the format parameter as follows:

```
printf(input);
```

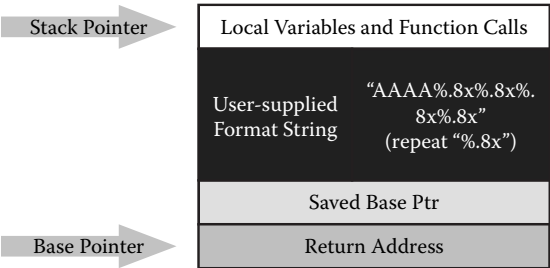
The command below uses Perl to print *AAAA* followed by a format string that instructs printf to show eight digits of precision in hex representation (%x) and to do this repeatedly.

```
$ ./format_string_vulnerable `perl -e 'print "AAAA";
print "%.8x"x 100`
AAAA.00000400.080485d0.bf930e66.
b7fec1b5.00000008.00000088.b7ff2ff4.bf92fe54.b7fec29c.
b7fd85fc.42fd8298.41414141.382e252e.2e252e78.252e7838.
2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.
78382e25.382e252e
```

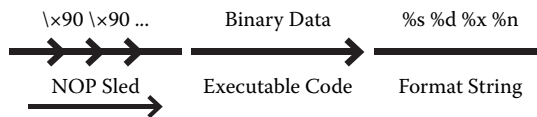
By using the *AAAA* input (0x41414141 in hex above), the attacker can identify where in the stack the string is located and attempt to identify the value that represents the return address. The stack layout for this attack only supplies a format string and no additional parameters (see Exhibit 3-11).

Normally, the stack would contain a variable for each %.8x in the format string; however, in this attack, the format string will cause the function to begin utilizing other stack contents, and if there are enough % symbols in the format string, it will eventually disclose the location of the return address.

When the user can control the format parameter, he or she can specify a format that removes extra data from the stack in this manner. It allows the user to view the contents of memory because he or she can specify any number of variables in the format string. The program may crash because, normally, printf() will push the same



**Exhibit 3-11** A stack layout for a simple attack with a single format parameter to printf.



**Exhibit 3-12** The format of user-supplied data to exploit a format string vulnerability.

number of items onto the stack as the number that it removes from the stack. Any data saved onto the stack no longer contain valid information, including the base pointer and return address of the stack frame. The `%n` is a key component to gaining code execution through format string vulnerabilities because it allows an attacker not only to print data but also to write data to the stack. Using `%n` allows the attacker to take control of the base pointer or return address, thereby allowing him or her to jump to any address chosen. If the attacker specifies binary shellcode within his or her format string, and then overwrites the base pointer or return address to jump to that location, the attacker will gain arbitrary code execution. Exhibit 3-12 displays an example of an attack using a user-supplied format string.

In the attack shown in Exhibit 3-12, the attacker must print an exact number of bytes so that when he or she uses `%n` to overwrite the return address, it overwrites the return address with any address in the range of the NOP sled section. In this way, the entire executable code section of the user's input will execute.

Format string vulnerabilities are as dangerous as buffer overflows or other remote code execution vulnerabilities. Attackers commonly release proof-of-concept code detailing how to exploit the vulnerabilities. Przemysław Frasunek first disclosed this category of vulnerability in June, 2000, on the BugTraq mailing list and the report describing the attacks by Tim Newsham. Since its public disclosure, exploit developers have released many public exploits each year. Despite the availability of exploits, the iDefense Rapid-Response Team rarely encounters format string exploits that install malicious code or attempt to compromise servers. One historical instance is the Ramen Worm of January 2001, which used three vulnerabilities to spread, one of which was a format string vulnerability.<sup>13</sup> Similarly, iDefense Managed Security Services rarely (if ever) encounters network alerts based upon format string vulnerabilities, despite the presence of many vulnerabilities and network-based rules to detect those attacks. Of

those alerts that mention a variation of format string in the messages, most are probes or attempts to gain information about the software rather than exploits that try to execute a malicious payload. While format strings are very dangerous, it is unclear why malicious code authors exploit other vulnerabilities so much more prevalently. One possible explanation is that format string attacks are more prevalent in open source software because attackers can audit the source code. As evidence, most of the historical format string vulnerabilities disclosed on the exploit site milw0rm.com is for open source software.

Fixing calls to `printf()` is easy if programmers can recompile the source code of a program. For example, to fix a dangerous call of the form `printf(input)`, programmers must pass the input variable with a string `(%s)` format instead, such as `printf('%s', input)`. This prevents user input from affecting the format string; therefore, an attacker will be unable to affect the stack or other memory. Most compilers provide warnings to encourage good programming habits and prevent format string vulnerabilities. Additionally, checks at runtime can help prevent the abuse of format string vulnerabilities by verifying that an attacker did not corrupt the stack. Additionally, stack randomization and nonexecutable stacks make it more difficult for attackers to execute code reliably. Other vulnerable functions that utilize the `va_list` structure are more difficult to identify, but are easy for programmers to fix once they identify the vulnerabilities. Replacing the `printf` library and similarly vulnerable libraries to perform more runtime checking of parameters is a necessary step when managing all vulnerabilities of this form. Performing source code analysis can also identify vulnerable calls because the pattern of function calls is easy to identify.

`Printf` vulnerabilities and format string vulnerabilities are a problem of communication between the API programmer who built the C libraries (and any other vulnerable functions) and the programmer who utilizes the library. To solve this problem effectively, either API programmers must sacrifice performance to perform additional runtime checks or programmers must always call functions properly. The stack is a vital component of program execution, and its corruption is at risk. Many functions in the `printf` family are vulnerable to this type of attack. Programmers should also evaluate other functions that depend upon `va_list` structures because they can contain

vulnerabilities similar to format strings, depending upon their purpose and implementation.

### *3.1.5 SQL Injection*

This section looks at structured query language (SQL) injection attacks. These attacks, which result from failing to validate user inputs, have increased in prevalence in the past several years and now often target thousands of sites at a time. Attackers often use search engines to identify vulnerable sites, and then use SQL injection to alter the content of the site to include malicious IFrames, or otherwise download malicious code to Web surfers who visit the compromised sites. SQL injection is simple for attackers to conduct and for developers to protect against, often by using prepared statements or otherwise validating user-submitted strings.

One of the most common and dangerous vulnerabilities in Web applications is structured query language (SQL) injection. SQL injection is fundamentally an input validation error. These vulnerabilities occur when an application that interacts with a database passes data to an SQL query in an unsafe manner. The impact of a successful SQL injection attack can include sensitive data leakage, website defacement, or the destruction of the entire database.

SQL is a computer language that allows programs to interact with relational databases like MySQL and Microsoft SQL Server. To retrieve, insert, or update information in the database, a programmer must craft a query that accesses the correct data to achieve the desired result. The simplest query involves selecting a piece of information from a table. For instance, a database supporting a blog might contain a table that stores data related to the entries in the blog. To retrieve the text of each blog entry, a programmer might execute the following query:

```
SELECT text FROM blog_entries;
```

This query will return the data in the “text” column for each of the rows in the `blog_entries` table. Of course, this query is not very useful because we probably want to display a specific blog or subset of all of the entries rather than every blog in the database.

```
SELECT text,user,timestamp FROM blog_entries WHERE  
user = 'user1';
```

This query is more complex. It still retrieves the text of the blog entry, as well as the name of the user who wrote it and the time of publication. It also includes a `WHERE` clause that specifies that the returned data should only include blogs with the user's name *user1*.

To achieve this, we might create a dynamic query that can accept information from a program, like a Web application. Programmers normally do this through a high-level language such as PHP, Perl, or ASP. The following pseudocode shows how a programmer might accomplish this:

```
#Get Username
username = getInputFromUser()

#Create SQL Query containing username
sql_query = "SELECT text,user,timestamp FROM blog_
entries where user = '" +
username + "'"

#Execute complete query
database.execute(sql_query);
```

First, the program acquires the username that it should search for from the user, commonly through a Web search form or URL variable. Next, the program adds the username to the query by concatenating the two sections of the query with the username. Finally, the program executes the fully formed query in the database to retrieve the results. This type of dynamic query is vulnerable to SQL injection attacks because it does not properly validate the input provided by the user. Exhibit 3-13 shows how this query is constructed, with the query text in light gray and the data supplied by the user in dark gray.

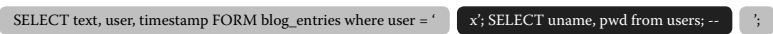
An attacker can exploit this query by providing input that the database does not know it should treat as data and treats as SQL code instead. Exhibit 3-14 shows how the data supplied to the query might completely change its function.

A diagram showing a SQL query string in a light gray box: "SELECT text, user, timestamp FORM blog\_entries where user = ". This is followed by a dark gray rounded rectangle containing the text "user1", which is then followed by a single quote character "'" in a light gray box. The entire sequence represents the query as it would be constructed in memory or in a log.

SELECT text, user, timestamp FORM blog\_entries where user = ' user1 '

---

**Exhibit 3-13** Proper data provided to a structured query language (SQL) query.

A diagram showing a SQL query string in a light gray box: "SELECT text, user, timestamp FORM blog\_entries where user = ". This is followed by a dark gray rounded rectangle containing the text "x'; SELECT uname, pwd from users; --", which is then followed by a single quote character "'" in a light gray box. The entire sequence represents the query as it would be constructed in memory or in a log, showing how the user input has been injected with SQL code.

SELECT text, user, timestamp FORM blog\_entries where user = ' x'; SELECT uname, pwd from users; -- '

---

**Exhibit 3-14** An SQL injection attack.

In this case, the attacker provided the following string in place of the username variable:

```
x'; SELECT uname,pwd FROM users; --
```

This string begins with the character *x* followed by a single quote and a semicolon. The single quote ends the data string, and the semicolon tells the database to execute the query so far. This probably will not return any information unless there is a user named *x*. The database will then process the rest of the query, which selects the columns “uname” and “pwd” from the table named “users.” This query will return a list of all usernames and passwords in the database. Finally, the string terminates with “--”, which declares the rest of the line as a comment that will not be executed. This SQL injection attack forces the database to return sensitive information (usernames and passwords) rather than the expected blog data.

This attack accesses sensitive information but could just as easily modify the database to include malicious content using an UPDATE command, or destroy the entire database using a DROP command. Since early 2007, attackers have launched very widespread SQL injection attacks on websites that update the database to include malicious IFrames in the text. When the server displays this text in a Web page, those IFrames attempt to infect unsuspecting visitors. This attack effectively changes a website the user expects to be safe into a malicious one using SQL injection.<sup>14</sup>

*3.1.5.1 Protecting against SQL Injection* Protecting against SQL injection attacks requires ensuring that data used in an SQL query are valid and will not be executed by the database engine. Programmers normally accomplish this through two possible methods. With the first method, programmers can try to ensure that the data do not include any special characters such as single quotation marks that might cause the database to treat data as code. Many languages provide a function that performs this task. The most commonly known is PHP’s `mysql_real_escape_string`.<sup>15</sup> This function “escapes” special characters by placing a backslash in front of them, causing the database to treat them as data and not execute them. An alternative technique to sanitizing user input is to only allow *good* data into the application rather than escaping bad data. For instance, database fields that should only

contain letters and numbers are validated using regular expressions that match those characters before using the data in the SQL query.

The second method involves the use of parameterized queries. Parameterized queries allow the programmer to define a query and pass data into it without performing dangerous string concatenation. The following pseudocode shows how programmers can use parameterized queries to make the code shown in the previous section safe from SQL injection.

```
#Get Username
username = getInputFromUser()

#Create the Parameterized Query using the %s format
string.
sql_query = "SELECT text,user,timestamp FROM blog_
entries where user = %s;"

#Execute the parameterized query, specifying the data
separately
database.execute(sql_query, (username));
```

With parameterized queries, the database is aware of what is code and what is data and can avoid SQL injection based on that knowledge. Unfortunately, this facility is not available in all programming languages and, as such, cannot protect every application.

When organizations lack the ability or resources to audit and test for SQL injection vulnerabilities in their Web applications, a Web Application Firewall (WAF) may be an alternate solution. ModSecurity is an open source WAF that can act as a reverse proxy, sitting between a Web server and the Internet to filter incoming requests for SQL injection and other types of attacks.<sup>16</sup> A list of commercial and open source WAFs is available from the Open Web Application Security Project (OWASP).<sup>17</sup>

Users of Microsoft's ASP.net Web-programming language should consult the extensive guide produced in 2005 that details techniques and strategies for avoiding SQL injection.<sup>18</sup>

*3.1.5.2 Conclusion* SQL injection attacks have become incredibly common in recent years. Automated attacks launched by botnets continuously scan the Internet for possibly vulnerable Web pages and attempt to compromise them. It is vital to protecting the integrity of

a database that the queries executed on it contain properly validated data that will not be mistakenly treated as code.

The defenses against SQL injection attacks are simple to implement but often overlooked by novice programmers or those looking to stand up a website quickly. When possible, administrators should test both homegrown and commercial off-the-shelf (COTS) Web applications for SQL injection vulnerabilities. OWASP provides a guide to testing applications and other valuable information on SQL injection vulnerabilities through their website.<sup>19</sup> When this testing is not possible, administrators should consider deploying a WAF to provide generic protection against SQL injection and other attacks on Web servers.

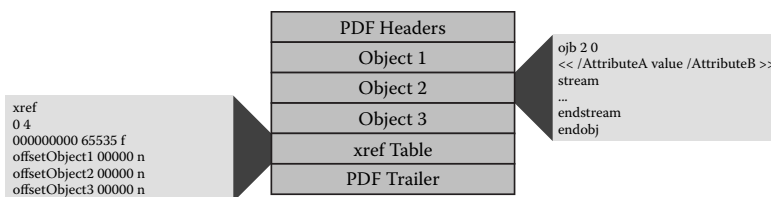
### *3.1.6 Malicious PDF Files*

Portable document format (PDF) files are so common that users often do not realize the potential danger they pose. Adobe Acrobat is a commonly installed application on all Microsoft Windows computers. The PDF file format is risky because many users have vulnerable PDF viewers installed that attackers can exploit to install malicious code. Attackers also commonly use PDF files to launch targeted attacks because they can often convince victims to open a PDF file to install malicious code. Multiple previously unknown or unpatched PDF vulnerabilities have allowed attackers to launch targeted attacks against high-priority victims as of 2009.

To make matters worse, Web browsers load PDF files automatically, so a malicious PDF file can exploit a user's computer without any interaction once the user visits a malicious website. Commercial Web attack toolkits commonly incorporate PDF exploits because PDF viewers are widely installed, and attackers can influence many different browsers, including Internet Explorer and Firefox.

Malicious PDF files usually contain JavaScript, but many exceptions exist that execute arbitrary code without JavaScript. Attackers commonly use JavaScript because the instructions can allocate large blocks of memory (heap spraying), which allows an attacker to reliably jump to certain addresses upon gaining control of execution through a vulnerability. Attackers also rely on JavaScript to hide the intent of their code because they can use `eval()` or similar functions to dynamically execute statements when the JavaScript code runs.





**Exhibit 3-15** Structure of a PDF file.

**3.1.6.1 PDF File Format** The format of a PDF file is largely based on plain text tags, although many stream objects use compression. Opening a PDF file within a hex editor shows objects that are numbered *1 0 obj*, for example. Other sections of the PDF file can reference these objects by number (see Exhibit 3-15).

Each object has attribute tags that describe its purpose. The *obj*, *endobj*, *stream*, and *endstream* tags correspond with the beginning and end of each section, and each attribute starts with */*. The cross-reference (*xref*) table contains entries corresponding to the file offset for each object. One common attribute for data in PDF files is the */FlateDecode* attribute, which the viewer decompresses with the *zlib* library; object 14 is shown here:

```
14 0 obj
<</Length 838 /Filter /FlateDecode>>
stream
... zlib compressed binary data ...
endstream
endobj
```

Malicious PDF files often contain malicious JavaScript code that analysts can inspect after decompressing the *zlib* data. To instruct the PDF viewer to execute the JavaScript code upon opening the file, the author must also assign an action to the object similar to one of the following examples:

```
<</Type/Action/S/JavaScript/JS 14 0 R >>
<</OpenAction <</JS (this.wBx9J6Zzf7\(\))
/S /JavaScript
```

The *Action* attribute specifies that the PDF reader should execute the JavaScript code in object 14. The */OpenAction* attribute can call the JavaScript function *wBx9J6Zzf7()*, which the PDF file defines in a different object. The PDF file format also allows incremental changes

without modifying the original file. For each modification, the application appends a changes section, xref section, and trailer section. As a result, a PDF file may grow large if the author repeatedly updates it. More information on the PDF file format and JavaScript functionality is available from Adobe.<sup>20</sup>

The Adobe JavaScript engine exposes several PDF-specific objects, including app, doc, dbg, console, global, util, dialog, security, SOAP, search, ADBC, and event objects.<sup>21</sup> It also exposes some online collaboration commands for review, markup, and approval. These exposed objects are often the areas of vulnerability that attackers have exploited in the past.

*3.1.6.2 Creating Malicious PDF Files* Many of the most common Web exploit toolkits include one of the following recent PDF exploits:

| CVE ID    | VULNERABLE JAVASCRIPT<br>FUNCTION |
|-----------|-----------------------------------|
| 2007-5659 | collab.collectemailinfo           |
| 2008-2992 | util.printf                       |
| 2009-0927 | collab.geticon                    |
| 2009-1493 | spell.customDictionaryOpen        |
| 2009-1492 | getAnnots                         |

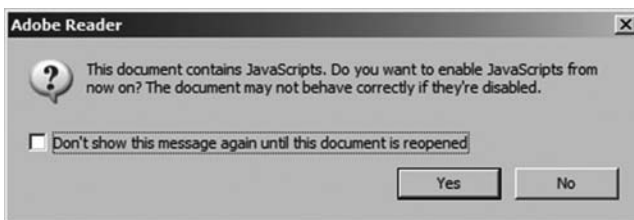
All of these vulnerabilities commonly use JavaScript exploits to execute arbitrary code. Exactly how attackers build these malicious PDF files is unknown; however, there are several public tools available to embed JavaScript within PDF files and decode PDF files. Adobe publishes some advanced commercial tools for modifying and creating PDF files; however, attackers are more likely to use their own tools or freely available tools due to the simplicity of the PDF file format. The origami Ruby framework<sup>22</sup> allows users to generate malicious PDF files by supplying their own malicious JavaScript. The origami framework can modify an existing PDF file by injecting custom JavaScript code that executes when users open the PDF file. PDF tools like make-pdf, pdf-parse, and pdf-id are also available and have similar functionality.<sup>23</sup> JavaScript deobfuscation tools are often necessary to understand the full purpose of JavaScript code contained within PDF files because authors commonly use obfuscation techniques. Tools like jsunpack-n<sup>24</sup> use PDF decoding and JavaScript

interpreters to understand the purpose of JavaScript code contained within PDF files.

Other means of embedding malicious content exist. For example, Adobe added an embedded Flash interpreter as a new feature in Adobe Reader 9.<sup>25</sup> iDefense has analyzed targeted attacks that embed malicious Flash objects within PDF files.<sup>26</sup> The ability to send malicious files embedded in PDF files increases the attack surface of PDF viewers.

The high number of vulnerabilities in Adobe Acrobat and the high number of attacks that use those new vulnerabilities should be of great concern for system administrators. Many actions that administrators should take will reduce the effectiveness of attacks. Administrators should put these measures in place because there are many instances of attackers exploiting these vulnerabilities before Adobe releases updates to vulnerable products.

*3.1.6.3 Reducing the Risks of Malicious PDF Files* Disabling JavaScript in the PDF reader is one way to limit the effectiveness of many exploits (Preferences → JavaScript → Uncheck Enable Acrobat JavaScript). This limits the effectiveness of PDF vulnerabilities that attackers incorporate into Web exploit toolkits. PDF vulnerabilities that do not depend on JavaScript will still be effective. For example, a Flash file in Adobe Reader 9 can still load without JavaScript. Attackers may also be able to trigger the same vulnerabilities that typically use JavaScript without JavaScript. One caveat of disabling JavaScript in this manner is that a user may still execute the malicious JavaScript code. Whenever a PDF contains JavaScript and the viewer has the configuration option to disable Acrobat JavaScript, the message seen in Exhibit 3-16 appears. This prompt gives the user the ability to execute JavaScript code despite disabling it in the configuration, and users must select the correct option to prevent infection



**Exhibit 3-16** Adobe Reader allows users to enable JavaScript, even when they disable it.

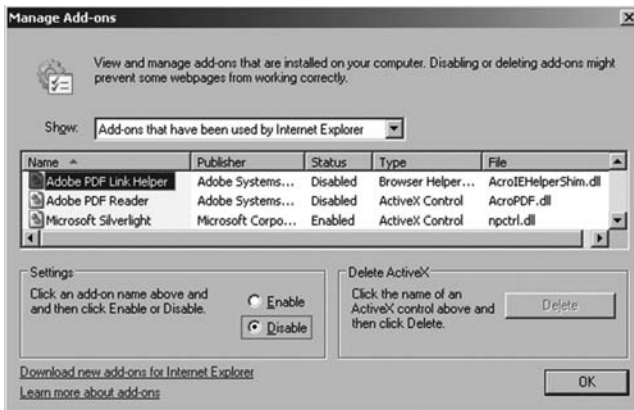
from a malicious PDF file. Normally, selecting “No” has no negative impact on the behavior of the PDF file. Administrators can also configure Adobe Acrobat to disable embedded Flash files and other media files. In the “Preferences → Multimedia Trust (legacy)” menu, uncheck “Allow multimedia operations” for both trusted documents and other documents.

Disabling both JavaScript and multimedia is still ineffective against certain less common vulnerabilities that do not use JavaScript or media files. Examples like the JBIG2 Encoded Stream Heap Overflow Vulnerability (CVE-2009-0928), which attackers used in targeted PDF attacks before Adobe patched it, show that other vulnerabilities exist that administrators must also consider.

Preventing PDF files from loading in the Web browser can reduce the risk of users loading malicious PDF files. Since many of them silently embed PDF files, disabling add-ons that load PDF files will prevent files from automatically loading in the browser. Most of the PDF exploits in malicious tools embed an invisible object or redirect users from another separate page. Victims often do not always purposely open the malicious PDF files since the browser can load them automatically. Administrators should remove the ability of the browser to open embedded PDF files automatically, which forces users to manually open each PDF file. This measure reduces the likelihood that a user will open a malicious PDF file, which attempts to open without user consent. In Firefox, users can disable the Adobe Acrobat add-on. In Internet Explorer, administrators can disable add-ons via Tools → Manage Add-ons → Enable or Disable Add-ons (see Exhibit 3-17).

iDefense tested removal procedures in Internet Explorer 7 and identified that disabling both the browser helper object (AcroIEHelper.dll) and ActiveX control (AcroPDF.dll) did not prevent the PDF file from loading when embedded in a Web page with an IFrame. To fully remove the capability to automatically open PDF files, it was necessary to also remove the PDF file type from the Folder Options menu (Tools → Folder Options → File Types).

PDF conversion tools are another effective way to limit the impact of attacks. Administrators may choose to use tools such as the Linux command-line utility “pdftotext,”<sup>27</sup> which converts a PDF file to plain text. While this conversion tool removes many of the useful features



**Exhibit 3-17** Disabling an Adobe PDF browser helper object and ActiveX control.

(visual components, media files, and interactive features) of PDF files, administrators may prefer other conversion tools that preserve more of the PDF file format.

*3.1.6.4 Concluding Comments* Attackers often use malicious PDF files to target victims, whether in targeted attacks or when attacking the browser. Attackers can modify any existing PDF file to append malicious content; therefore, every PDF file may contain malicious content regardless of how much the user trusts the content. To reduce the risks of malicious PDF files executing arbitrary code, administrators can eliminate some of the common dependencies that malicious PDF files contain, including JavaScript and embedded media content. To mitigate the risk of malicious PDF files more aggressively, administrators should take these actions even though they reduce the functionality of PDF files. While both JavaScript and media content in PDF files are useful for certain documents, they are generally not necessary for every document, and PDF files that function incorrectly without them are rare. This rich functionality is one of the reasons why attackers target PDF files so heavily and is why they will continue to do so in the future.

### *3.1.7 Race Conditions*

Race conditions result when an electronic device or process attempts to perform two or more operations at the same time, producing an

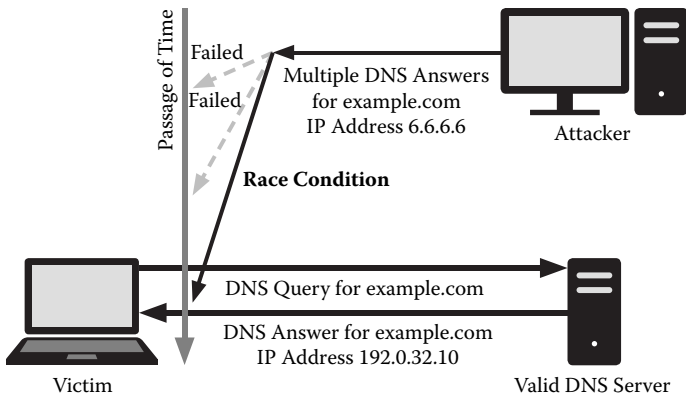
illegal operation. This section explains the concepts behind race conditions with examples and methods for detection and prevention.

Race conditions are a type of vulnerability that an attacker can use to influence shared data, causing a program to use arbitrary data and allowing attackers to bypass access restrictions. Such conditions may cause data corruption, privilege escalation, or code execution when in the appropriate context. Race conditions are also known as *time-of-check and time-of-use* (TOC/TOU) vulnerabilities because they involve changing a shared value immediately after the check phase. When the program uses an unsuspected value, it may execute instructions reserved for a different purpose or allow an attacker to redirect critical information.

Race conditions are limited in where they occur. They require multiple callers accessing shared information. For instance, race conditions can occur if two or more threads influence the same shared memory, files, or other types of data that a program uses. Consider the analogy of a traffic signal that only has two states: green (for go) and red (for stop). A race condition is analogous to two drivers who are both in the intersection at the same time because there is no delay between the switching of states from green to red. Similar to car accidents, race conditions can have catastrophic effects on threads and applications.

Environments favorable to race conditions are becoming more common as more applications use multiple threads and central processing unit (CPU) cores for performance improvements. In the future, tens or hundreds of cores could provide programmers with parallel access to data and make race conditions possible in even more applications. Multiple callers, CPUs, threads, and parallel access systems are all conducive to race conditions. Race conditions over the network are also possible and introduce the most latency; therefore, they could have the largest window of opportunity and thus be easier to exploit. In the time it takes a computer to complete a single network operation (assuming it takes 0.1 seconds), modern CPUs can execute about 7.5 billion instructions. Due to these differences, the window of opportunity for a race condition is typically much smaller for local operations than network-based operations.

*3.1.7.1 Examples of Race Conditions* Network race conditions are common among non-Transmission Control Protocols (non-TCPs) such



**Exhibit 3-18** A domain name service (DNS) answering a race condition over the User Datagram Protocol (UDP).

as User Datagram Protocol (UDP). Many network communications suffer from these problems because the program accepts the first answer to questions it asks. That is, there is a short amount of time between when the user asks the question and when he or she receives a legitimate answer in situations in which an attacker can respond and exploit the race condition. As an example, consider the domain name system (DNS) answering race condition in Exhibit 3-18.

Exploiting the DNS answering race condition and other UDP-based race conditions is difficult because the attacker does not know when the user will ask a question or what the question will be. If the attacker answers the wrong question, the client is likely to ignore it. Therefore, attackers attempting to exploit the DNS race condition flood the victim with answers to questions that he or she has not asked. Making the attack even more difficult, the DNS caching and the DNS time to live (TTL) limit the number of times that the victim needs to ask the question, making the window of opportunity for an attacker small. When attackers are using “blind” attacks, they do not know when the victim is vulnerable, and therefore such attacks are difficult to exploit successfully. Attackers on the local network may have information and feedback mechanisms, making it easier for them to exploit race conditions.

Race conditions can also occur when privileged applications use unprivileged files. As one example affecting the X windowing system, the local service X Font Server (xfs, which runs as root) changes the

permission of a file within “/tmp/” to world writable (meaning anyone can modify the file). If the attacker creates a symbolic link (symlink) within the /tmp/ directory at the right moment, the xfs service will change the permissions of any file on the system because the change in permission affects the target of the symlink instead of the original filename. To exploit this race condition, an attacker repeatedly tries to create the symlink to the “/etc/passwd” file. When the xfs service changes the permissions of the symlink target, it allows the attacker to add a new account to /etc/passwd and therefore gain root privileges.<sup>28</sup> Such an attack requires the attacker to have at least a limited account to create symlinks; therefore, this exploit allows attackers to escalate their privileges to root.

Similar to shared files, race conditions can affect shared process memory. If a user can influence the memory, an attack can manifest in many ways. Suppose a thread performs the following pseudocode to find the function address, which it calls immediately afterward:

```
FunctionAddress = memory[pointerA]
If FunctionAddress within kernel32.dll:      #STATEMENT1
Then:
Parameters = memory[pointerB]
Call FunctionAddress with Parameters          #STATEMENT2
```

In this case, STATEMENT1 represents the TOC, and STATEMENT2 represents the TOU. In the time between STATEMENT1 and STATEMENT2, another thread can alter the values of the memory at pointerA and pointerB, affecting the Call STATEMENT2. An attacker who successfully altered the memory could execute any function with any parameters, even those functions that do not satisfy the restrictions within STATEMENT1. A Linux kernel vulnerability (in version 2.6.29) exists that is similar to this vulnerability because the lock (mutex) between the ptrace\_attach() and execve() functions does not prevent multiple threads from accessing the shared memory. Exploiting the vulnerability allows a local user to escalate his or her privileges.<sup>29</sup>

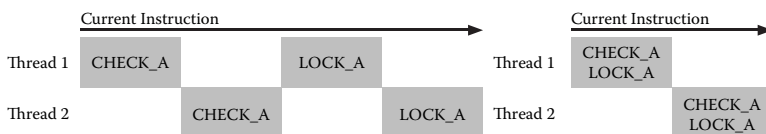
Race conditions are difficult to locate from a testing perspective because instructions can require infinitesimally small amounts of time based upon the high number of instructions that modern CPUs execute per second. The likelihood of two programs requesting the same



shared data at the same time is very unlikely under normal circumstances for a few reasons. First, the operating system scheduler determines how to prioritize multiple applications and threads, not the user mode application. The attacker has little control over these subtle timing constraints and, for this reason, may repeat the same action millions of times before favorable conditions exist. Second, the operating system uses a special signal called an interrupt request (IRQ) to trigger actions. As the name implies, interrupts can change the execution of a program and execute different code. An IRQ changes the execution flow and may either help or hinder exploiting race conditions. The uncontrollable nature of interrupts and scheduling makes it difficult to use timing attacks reliably. Exploiting race conditions requires brute force, especially in high volume when the window of opportunity is very small.

**3.1.7.2 Detecting and Preventing Race Conditions** Since many race conditions require the attacker to use brute force techniques, attackers exploiting race conditions may raise many anomalies and errors on the system. Attempting to exploit a race condition may cause an extended spike in CPU use or a high volume of failed requests. System administrators should look for signs of a high volume of frequently repeated operations.

Race conditions are preventable, provided programmers use the right tools. Semaphores and mutexes (short for *mutual exclusions*) provide instructions that are not vulnerable to race conditions. Mutex instructions succeed when other instructions fail because of their atomic nature. Atomic instructions are single instructions that the CPU can execute in a single clock cycle, unlike nonatomic instructions, which execute in multiple clock cycles. Exhibit 3-19 illustrates a comparison between atomic and nonatomic instructions for checking and locking a resource *A*.



**Exhibit 3-19** A comparison of nonatomic (left) and atomic (right) versions of locking a resource.

In both the nonatomic and atomic cases, the CPU is executing the same instructions; however, in the nonatomic case on the left, both Thread1 and Thread2 obtain the same lock, which should never happen. In the atomic case, the check and lock procedure is part of a single instruction, which prevents the race condition and allows Thread1 to obtain a lock and Thread2 to determine that the lock is unavailable.

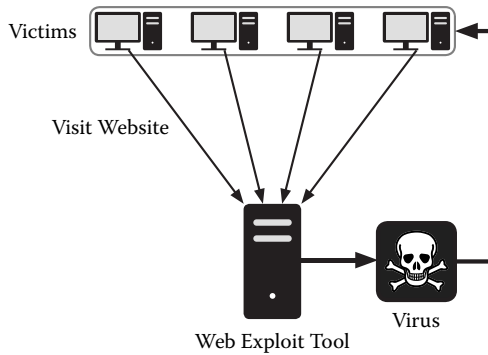
If the operation successfully locks a resource using an atomic (or hardware-based) instruction, the program can proceed knowing that no other thread was able to lock that same resource. Using locks requires that calling programs release allocated locks they already obtained. Many mutex systems also differentiate between read and write locks because, as mitigation for starvation of the writers, giving writers higher priority than readers can benefit the program when there are a high number of readers.

*3.1.7.3 Conclusion* Attackers use race condition vulnerabilities less than buffer overflows and other code execution vulnerabilities. They are less reliable and may require high volumes of traffic or activity, which are undesirable for attackers who do not wish to raise any alarms. Attackers use race conditions most commonly to escalate privileges because attackers cannot often control the desired actions to trigger the race condition based on otherwise untrusted data.

### *3.1.8 Web Exploit Tools*

To accomplish the process of identifying vulnerable targets and delivering appropriate exploits to them, attackers frequently use tools containing exploits for numerous vulnerabilities. Some tools are freely available, though hackers must purchase the most effective ones in hacking forums. After infecting users with the chosen payload, the tools collect statistics that allow attackers to run more targeted attacks and allow criminals to track metrics useful for billing clients for malicious services, such as the number of installations performed.

Web exploit tools (or exploit kits) give attackers the ability to execute arbitrary code using vulnerabilities or social engineering. To attract visitors to the malicious websites, attackers often compromise other servers to append IFrame tags, which direct visitors to attackers' Web exploit tools. Upon visiting a malicious website, the exploit



**Exhibit 3-20** Victims visiting a Web exploit tool, which attempts to install a virus.

tool attempts to launch many different exploits to execute arbitrary malicious code. This section describes Web exploit tools and how they fit into the businesses that have benefited from client-side attacks. Exhibit 3-20 shows how attackers use Web exploit tools to install malicious code on victims' computers.

Most Web exploit tools are so simple that the operator needs only to supply the executable virus for installation. Web exploit tools usually handle hiding, exploitation, and statistics automatically. Other services allow an attacker to make money from running a Web exploit tool and to gain a large number of victims very quickly.

**3.1.8.1 Features for Hiding** Many exploit tools hide exploits through encoding, obfuscation, or redirection. Exploit tools attempt to hide exploits to prevent both detection and analysis, which iDefense discussed in a “State of the Hack” article entitled “iDefense Explains ... Obfuscation.”<sup>30</sup> Exploit tools encode traffic so that the victim decodes it using JavaScript or VBScript. Intrusion detection systems (IDSs) are a defensive measure that is ineffective at detecting encodings because the content transferred over the network is different from the content that the client executes.

Exploit tools also use JavaScript or HTTP headers to profile the client and avoid sending content unless the client is vulnerable. Exploit tools often try to detect multiple vulnerabilities to determine which will be effective and if the tool should attempt multiple attacks against a website visitor. HTTP headers like *user-agent* and browser variables like *navigator* or *app* reveal information that gives attackers

the necessary information. If the exploit tool determines that the client is not vulnerable, it may redirect him or her to a benign URL or display an empty page. Attackers can also configure tools to check the language of the victim's browser and the victim's geographic location, or if the victims arrived at the exploit tool through a valid source. Exploit tools analyze the referrer HTTP header to determine if the victim originates from an infected page. In this way, exploit tools avoid sending malicious content to researchers who may use search engines or other mechanisms to analyze a website.

*3.1.8.2 Commercial Web Exploit Tools and Services* There are a large variety of commercial Web exploit tools and services available to install malicious code on victims' computers. Purchase prices for source code for Web exploit tools vary widely from US\$30 to as much as US\$2,000. Exhibit 3-21 shows thirty different Web exploit tools, including prices, information about the author, and whether iDefense has seen attackers use the tool in the wild.

Other exploit services are market driven by supply and demand. Running an exploit tool on a server requires little computer knowledge to infect victims because the only input is an executable. Market-driven services require even less knowledge or skill. The major markets include selling traffic (IFrames) and selling installs (the customer supplies the executable to run).

Pay-per-install services allow actors to buy and sell installations, which is the easiest way for a customer to install malicious code. Some examples of pay-per-install services include IFramedollars and Loads.cc (see Exhibit 3-22).

Pay-per-traffic services allow attackers to attract a large number of victims to their Web exploit tools. Attackers can then either install their own viruses or sell installs via the pay-per-install model. Examples of pay-per-traffic services include IFrame911.com and RoboTraff.com (see Exhibit 3-22).

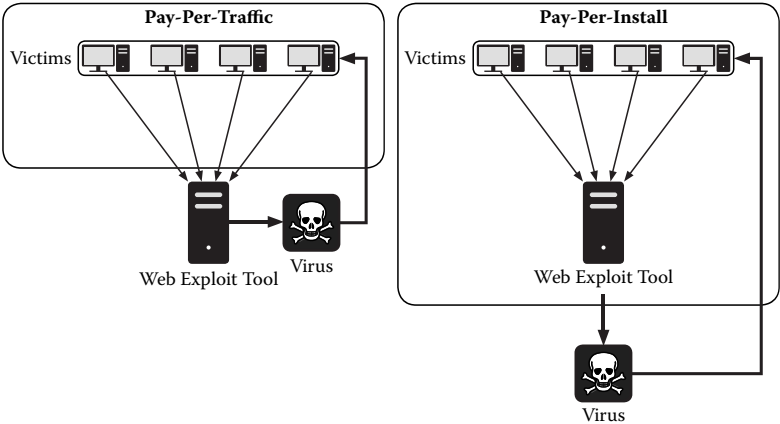
While the examples have public websites and are generally more available to the public, the same pay-per-install and pay-per-traffic services are available privately from individuals on hacking forums. Attackers can also operate each part separately from these services. This has advantages because it limits their interaction and dependence on third-party services. If those services became known or otherwise

| SOLD AS                            | LANGUAGE | PRICE   | AUTHOR                        | SEEN IN WILD |
|------------------------------------|----------|---------|-------------------------------|--------------|
| 0x88                               | PHP      | Unknown | Unknown or no author credited | Yes          |
| AD Pack                            | PHP      | Unknown | Unknown or no author credited | Yes          |
| Armitage                           | PHP      | Unknown | Unknown or no author credited | Yes          |
| Death Pack                         | PHP      | \$90    | Sploit3r                      | No           |
| eCore Exploit Pack                 | PHP      | \$590   | Multiple Aliases              | Unknown      |
| Exploit Multipack                  | PHP      | Unknown | Unknown or no author credited | Yes          |
| Firepack                           | PHP      | Unknown | DIEL                          | Yes          |
| Flo0P Pack                         | PHP      | \$800   | Flo0Py[Error]                 | No           |
| G-Pack                             | PHP      | \$99    | Garris                        | Yes          |
| IcePack Lite                       | PHP      | \$30    | Ni0x / IDT Group              | Yes          |
| IcePack Platinum                   | PHP      | \$400   | Ni0x / IDT Group              | Yes          |
| INFECTOR Professional              | PHP      | Varies  | XOD                           | No           |
| INFECTOR Standart                  | PHP      | \$1,300 | XOD                           | No           |
| Le Fiesta                          | PHP      | € 500   | el                            | Yes          |
| MPack                              | PHP      | \$700   | Dream Coders Team             | Yes          |
| myPOLYSploits                      | PHP      | \$150   | 4epolino                      | Yes          |
| N404-Kit (temporary iDefense name) | PHP      | Unknown | Unknown or no author credited | Yes          |
| Neosploit                          | C (CGI)  | \$1,500 | grabarz                       | Yes          |
| Nuclear's bot                      | PHP      | \$150   | Nuclear                       | Unknown      |
| Nuklear Traffic                    | PHP      | Unknown | Unknown or no author credited | Yes          |
| System                             | PHP      | \$30    | REALiStiC                     | Yes          |
| SmartPack                          | PHP      | \$200   | Unknown or no author credited | Unknown      |

**Exhibit 3-21** Exploit kits commonly used by cyber criminals.*Continued*

| SOLD AS                         | LANGUAGE | PRICE   | AUTHOR                        | SEEN IN WILD |
|---------------------------------|----------|---------|-------------------------------|--------------|
| SPREADER                        | PHP      | \$2,000 | NeRox                         | Yes          |
| Tornado                         | PHP      | \$600   | Expire2012                    | Yes          |
| Ultra Lite Pack                 | PHP      | \$50    | cracklover                    | No           |
| Underwater Exploit Pack         | PHP      | \$500   | Underwater                    | Unknown      |
| Unique Bundle of Exploits       | PHP      | \$400   | system                        | Yes          |
| WebAttacker                     | CGI      | N/A     | Inet-Lux IT Group             | Yes          |
| WebAttacker II                  | PHP      | \$1,000 | Inet-Lux IT Group             | Yes          |
| Z-Kit (temporary iDefense name) | PHP      | Unknown | Unknown or no author credited | Yes          |

**Exhibit 3-21 (Continued)** Exploit kits commonly used by cyber criminals.



**Exhibit 3-22** Pay-per-traffic and pay-per-install commercial markets.

unavailable, investigators could discover their users or otherwise disrupt the business. Attackers may instead generate their own traffic to exploit tools using SQL injection. Attackers can inject IFrames into vulnerable servers. More information on these attacks is available in the “State of the Hack” article “iDefense Explains ... SQL Injection.”<sup>31</sup>

Infected websites often contain multiple levels of redirection; both pay-per-traffic and pay-per-install business models are possible reasons for the multiple redirects. The separation of work includes generating traffic, exploiting systems, and running arbitrary executables

on the victim machine. Attackers can make money in this process by being proficient in any one of the steps.

*3.1.8.3 Updates, Statistics, and Administration* Most Web exploit tools available today target browser vulnerabilities or browser plug-ins. Attackers can develop new exploits or integrate the latest public exploits to improve their likelihood of being successful against a victim. Some attackers advertise that they offer zero-day exploits, which exploit vulnerabilities that the vendor has not released a patch for in its toolkits. Others sell premium versions of their tools that they will integrate with the latest exploits. Despite these advertisements for enhanced services, Web exploit tools often integrate publicly available exploits such as those available from Metasploit or milw0rm.com. The tools sometimes offer statistics on visitors to show how successful their exploits are and what countries, operating systems, and browsers the visitors use. Le Fiesta is a Web exploit tool that tracks many attributes about visitors (see Exhibit 3-23) and builds victim statistics to know which software the operators should target.

Many exploit tools provide administration interfaces protected by usernames and passwords through which an attacker can obtain detailed information, modify the behavior of the exploit tool, or take

| Le FIESTA         |           |      |                 |      |     |
|-------------------|-----------|------|-----------------|------|-----|
| Countries         |           |      |                 |      |     |
|                   | ALL       | LOAD | PER             | HOUR | DAY |
| RU                | 27947     | 3679 | 13.1            | 0    | 118 |
| UA                | 6904      | 843  | 12.2            | 0    | 32  |
| OT                | 5496      | 779  | 14.1            | 0    | 41  |
| DE                | 4488      | 257  | 5.73            | 0    | 32  |
| US                | 2507      | 127  | 5.07            | 1    | 37  |
| GB                | 1265      | 46   | 3.64            | 0    | 17  |
| BY                | 1103      | 105  | 9.52            | 0    | 5   |
| KZ                | 878       | 129  | 14.6            | 0    | 9   |
| JP                | 442       | 22   | 4.98            | 0    | 4   |
| IN                | 378       | 97   | 25.6            | 0    | 1   |
|                   |           |      | * other country |      |     |
| SP1               | 37355     | 3071 | 8.22            | 1    | 254 |
| SP2               | 9295      | 3128 | 33.6            | 1    | 46  |
| VISTA             | 6111      | 138  | 2.26            | 1    | 60  |
| OTHER             | 3070      | 102  | 3.32            | 0    | 27  |
| 2K                | 777       | 190  | 24.4            | 0    | 4   |
| 2K3               | 708       | 37   | 5.23            | 0    | 2   |
| Operating Systems |           |      |                 |      |     |
|                   | 21742     | 3977 | 18.2            | 2    | 179 |
|                   | 20005     | 856  | 4.28            | 0    | 147 |
|                   | 12423     | 1645 | 13.2            | 0    | 41  |
|                   | 2472      | 128  | 5.18            | 0    | 20  |
|                   | 674       | 60   | 8.90            | 1    | 6   |
|                   | 3         | 0    | 0.00            | 0    | 0   |
| Browsers          |           |      |                 |      |     |
|                   | MSIE      |      |                 |      |     |
|                   | OPERA     |      |                 |      |     |
|                   | FFOX      |      |                 |      |     |
|                   | OTHER     |      |                 |      |     |
|                   | CHROME    |      |                 |      |     |
|                   | OPERA 7.0 |      |                 |      |     |

**Exhibit 3-23** Statistics for the Le Fiesta exploit tool.

other actions. Some exploit tools utilize a database such as MySQL to provide permanent storage capabilities.

While update services are only marginally useful because of copying, authors of exploit tools often do lead efforts in hiding and obfuscation efforts. Commercial Web exploit tools have the fastest evolution, making it difficult for researchers to determine what exploits they use.

*3.1.8.4 Proliferation of Web Exploit Tools Despite Protections* Malicious actors have offered many commercial exploit tools free on hacking forums, and it is not clear how profitable exploit tool development is, given the availability of these leaked versions.

Some Web exploit tools contain protections to prevent copying and modification. The protections include the following:

- Source code obfuscation
- Per-domain licenses that check the local system before running
- Network functionality to confirm validity (license checks)
- An end-user license agreement (EULA)

Since authors write many Web exploit tools in PHP, the source code is available. To prevent source code from being available and authors from writing exploits, Web exploit tool authors commonly use commercial PHP obfuscation tools such as NuSphere's NuCoder or Zend Guard when they distribute their source code. Some exploit tools like Neosploit use compiled C code to run as CGI programs, which require much more time to reverse engineer or modify since they do not preserve the original source code.

Despite these protections, copying and proliferation of many exploit tools are still common. Malicious authors write most Web exploit tools in PHP (or a compiled language) because the tools can generate content based upon parameters and use a database; however, authors can copy an existing exploit tool if they are able to observe the exploits it uses and modify them to execute a different executable. Tools like Dezend may also offer attackers and researchers ways to reverse the Zend Guard encoding.<sup>32</sup> Compiled content requires reverse engineering, and although it offers some protection, it is not always capable of preventing modification.

Attackers that use exploit tools do not always purchase tools since some are freely available. The commercial markets that depend on



exploit tools have supply-and-demand components for supporting the Web exploit tool use, including pay-per-traffic and pay-per-install models. The proliferation of Web exploit tools is one indicator that attackers are investing resources into this market. Commercial Web exploit tools often include hiding and obfuscation techniques to evade defensive measures.

Attackers using these tools collect victim trends and statistics that allow them to focus their efforts to be successful in the future. The division of traffic exploits and installs will likely continue as each area improves.

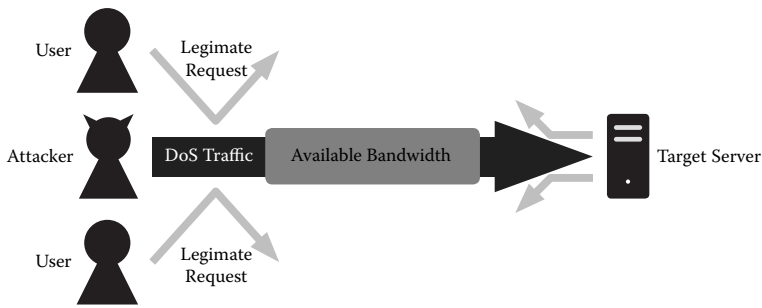
### *3.1.9 DoS Conditions*

Incidents reported daily in the news are a reminder that denial of service (DoS) attacks are a major threat to systems connected to the Internet, especially those of e-commerce, financial services, and government services. This section explains how these attacks work and offers solutions to mitigate the effects of future attacks.

Computer networks are the backbone for telecommunications and consist of a vast array of clients and servers for information exchange. Servers offer services, such as a Web or file transfer protocol (FTP) service, with which clients can interact to share or obtain information. When servers or services cannot respond to client requests, a situation called a DoS condition arises. A DoS occurs when a disruption impairs information exchange, resulting in slow or halted communication. The consequences of a DoS can vary depending on the situation, but they typically incur severe downtime resulting in financial losses. These consequences attract malevolent individuals to perform DoS attacks against targets of interest.

*DoS* is a general term to describe a lack of access to a service. The lack of access can occur for many reasons and at different points between the client and server. Points subjected to a DoS condition are network segments, network devices, the server, and the application hosting the service itself. The conditions necessary to cause a DoS at each of these points differ, but all result in a disruption in activity between the client and server.

A network can only send and receive a certain amount of data at one time. Any data sent that exceeds a network's bandwidth will not



**Exhibit 3-24** A bandwidth-consuming denial of service (DoS) attack.

make it to its destination. One form of DoS occurs when an attacker sends so much traffic to the target that it consumes all of the available bandwidth. In this situation, most requests cannot make it to their intended destinations, which results in a DoS for legitimate clients. Exhibit 3-24 shows an attacker consuming the network bandwidth and denying access to other users.

To consume the available bandwidth of a target, an attacker uses a technique known as *flooding*. Flooding describes the overwhelming traffic used to saturate network communications. Attackers use communication protocols such as User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), and Transmission Control Protocol (TCP) to inundate the target with network traffic.

A similar method to cause a DoS condition starves the resources of network devices. Overutilization of system resources, such as central processing unit (CPU), memory, or data structures stored in memory, can cause system failure. System failure of a network device, such as a firewall, router, or switch, has devastating results because it causes network traffic congestion resulting in performance degradation or an outage.

A particularly effective DoS attack on network device resources is a SYN Flood. A SYN Flood describes the rapid transmission of TCP SYN packets to a target to initiate the three-way TCP handshake to create a session. The attacker initiates the TCP handshake by sending SYN packets to the target but does not acknowledge the target's synchronize and acknowledgment (SYN/ACK) packets. The network device awaits the ACK that the attacker never sends, which creates half-open sessions that the network device stores in a connection table. Eventually, the network device purges the half-open sessions

after a timeout period, but if the attacker can fill the device's connection table, then the device ignores all further legitimate attempts to create a TCP session.

Resource-starving DoS conditions do not just plague network devices. The servers themselves are prone to resource starvation, limiting their ability to process requests. A server using 100 percent of its CPU will have trouble responding to inbound requests. For example, an attacker using the DoS technique called Slowloris sends HTTP GET requests with partial HTTP headers to an Apache Web server. The Apache Web server will wait for the attacker to send the rest of the HTTP header. The attacker never sends the complete HTTP header, just arbitrary header fields to avoid timing out and to keep the Web server waiting. Using this technique, an attacker can consume all available threads on the Apache Web server, resulting in a DoS condition.<sup>33</sup>

To saturate network bandwidth or to starve a system's resources successfully, it is beneficial for an attack to have more resources than the target does. To gain more resources than the target, attackers tend to gather distributed resources and coordinate an attack called a *distributed denial of service* (DDoS). The distributed systems join forces and attack simultaneously to overwhelm the target. The prevalence of botnets across the Internet makes DDoS a reality, as bots within these botnets can focus an attack on a target with devastating results. For example, if a botnet initiates a DDoS attack against a target with 2,000 bots each with 2 Mbps of upstream bandwidth, it can saturate a large network with a combined 4 Gbps of network traffic.

Another option to overwhelm a target's resources becomes available through amplification, which involves using techniques to magnify the effect of an attack beyond the capabilities of the attacker's limited resources. The amplification helps the attacker use more resources than are ordinarily readily available to them. Typically accomplished by using a technique called *reflection*, the attacker spoofs the IP address of requests to the intermediary system, called a *reflector*, which sends responses back to the spoofed IP address. The responses to the spoofed IP address can flood the system at that address, which is the target of the attack.

A domain name system (DNS) amplification attack exemplifies this technique and takes advantage of the size difference between DNS

query packets and DNS answer packets. An attacker spoofs the source IP address within DNS query packets sent to recursive DNS servers. The DNS servers act as a reflector by replying with DNS answer packets to the target. The answer packets generally are many times larger than the initial query packet, resulting in a successful amplification of the attacker's traffic. For more information on this form of attack, refer to the "State of the Hack" article entitled "iDefense Explains Domain Name System (DNS) Amplification Attacks."

In addition to resource starvation, DoS conditions can occur as a by-product of vulnerability exploitation. A server or service susceptible to vulnerabilities, such as buffer overflows and format string overflows, can hang or crash in the event of a successful exploitation. Once crashed or in a hung state, the server or service can no longer carry out its responsibilities, resulting in a DoS condition.

Other DoS conditions stem from the configuration of a service or server. Not only do improper configurations present DoS opportunities, but also even legitimate or proper configurations can block a client from interacting with a service. For example, an account lock-out feature in Microsoft Active Directory could deny a user from accessing the network domain if a password brute force attack exceeds the configured maximum number of allowed logon attempts. The Conficker worm was notorious for locking out user accounts during its propagation when it attempted to brute force the passwords to network shares.

Configuration changes can also be the cause of a DoS. Unintentional DoS can occur from the side effects of network configuration changes. A well-known example of such an erroneous configuration change occurred in 2008 when the popular video website YouTube.com experienced a DoS. The YouTube DoS occurred as the result of Pakistan Telecom attempting to block its customers from visiting the website. Pakistan Telecom broadcasted via the Border Gateway Protocol (BGP), the protocol used to set up routes between routers on the Internet, that it was the destination for YouTube's IP range. This caused routers worldwide to send traffic destined for YouTube to Pakistan Telecom.<sup>34</sup> This resulted in a DoS of YouTube, as visitors around the world were unable to reach the website.

The variety of different types of DoS attacks and the multiple locations where they can occur make a complete DoS prevention package

impractical; however, certain safeguards can reduce the chances of suffering from a DoS condition. Addressing DoS situations requires security safeguards, detection, and adequate response planning.

Security safeguards start with the patching of systems, which reduces the known vulnerabilities that can potentially cause a DoS through exploitation. This does not protect against zero-day and unknown vulnerabilities, but an up-to-date system is less likely to fall victim to a DoS vulnerability.

The common network-level safeguards using security devices can reduce DoS attempts by filtering out erroneous traffic at edge routers and firewalls. Many vendors offer DoS protection features built into their products. Access control lists (ACLs) and rate-limiting rules can immediately address DoS activity by blocking unwanted and flooding traffic. Using a network device with antispoofing functionality, such as Unicast Reverse Path Forwarding (uRPF), can reduce network DoS conditions, as the device verifies the validity of a source IP address and discards the traffic if the source IP address is not valid or is spoofed.

DoS detection requires monitoring of network traffic patterns and the health of devices. Intrusion detection or prevention systems, Netflows, and other network traffic logs can provide an indication of DoS conditions in the event of an increase in network activity or alerts. Monitoring the health of devices can also detect if a DoS is underway. If a system's available resources, whether they are memory, CPU utilization, and/or another resource, reach a critical level of utilization, then the cause of such exhaustion needs mitigation to avoid a DoS condition.

One of the most overlooked DoS prevention requirements is adequate response procedures. Planning response procedures will reduce the impact and outage caused by a DoS. DoS protection services are beginning to surface, including VeriSign's DDoS Monitoring and Mitigation<sup>35</sup> services. VeriSign monitors traffic destined for its customers' networks for DDoS characteristics and provides mitigation by filtering out the DDoS traffic.

Response planning should also include in-house procedures on involving the appropriate resources to mitigate the cause of the issue. Administrators and service providers need to understand that their involvement and processes are required to stop the activity causing a DoS. These processes vary depending on the specific situation or attack,

but they range from the service provider creating upstream ACL rules or black holing source networks via BGP routes, to a systems administrator replacing a server with a hot spare to mitigate a DoS.

Every day, DoS attacks cause outages across the Internet. The increase of botnet prevalence and release of application vulnerabilities make DoS incidents inevitable. DoS conditions spawn from everything from unintentional actions such as configuration changes, to intentional motives, to attacks on opposing political groups or competition, or they can act as a decoy for other malicious intentions. Regardless of the cause of the outage, careful planning can reduce the impact of such an outage and minimize the financial losses involved.

### *3.1.10 Brute Force and Dictionary Attacks*

A password-based authentication system is only as good as its underlying passwords. When attackers use brute force attacks and dictionary attacks against these systems, these passwords may prove to be insufficient. In this section, we will explain these two types of high-level attacks commonly used against password-based authentication systems.

Authentication systems that depend on passwords to determine the authenticity of a user are only as strong as the passwords on which they rely. Most system administrators understand this simple fact and require users to use sufficiently long and complex passwords. While this may reduce the probability of an attacker quickly guessing a user's password, the fact remains that password-only systems are vulnerable to two very well-known attacks: brute force attacks and dictionary attacks.

Attackers can and have used brute force attacks and dictionary attacks against a variety of cryptographic and authentication systems. It is space prohibitive to explain how these attacks work against the wide variety of cryptographic and authentication systems. This article focuses primarily on the attacks as they pertain to the generic model of a password- or passphrase-based authentication system. This generic authentication system uses some form of text-based password or passphrase string in combination with a username or some other form of identifier to authenticate the identity of the user. The use of a

text-based password allows the authentication system to compare the supplied text against the value previously set for the given username.

When dealing with the password component of the authentication system, it is not uncommon for the system to store the password in an encrypted or hashed form. The logic behind this dictates that if an attacker compromises the username and password database, the attacker will be unable to immediately use the passwords without first recovering the passwords using the inverse (if applicable) function of the password encryption system. In the case in which the system stores the passwords as a hash of the text string, no inverse function will exist, restricting the options available to the attacker who has the hash value of the text password. These options, as explained later, are limited to brute force and dictionary attacks.

Users, generally speaking, default to the simplest passwords they can use that still conform to the password standards set by the system administrators. As a result, the majority of users select a password based on a common word or phrase that the user can remember or that has some significant meaning to the user. Attackers capitalize on this fact by using password dictionaries.

Password dictionaries, sometimes referred to as *word lists*,<sup>36</sup> are compilations of known words and known variations on these words that users may use as passwords. These variations can include character substitutions (e.g., changing l's to i's or l's and vice versa), interjecting words from nonnative languages, and combining words into phrases. These password dictionaries are typically flat-text files with one password per line. Attacks can apply these passwords in a systematic way to the authentication system using a known username to find the correct password associated with the username.

When dealing with authentication systems that allow largely language-based passwords (passwords that conform to a given spoken language, such as English, French, or Russian, and do not contain a significantly high number of nonalphanumeric characters), a suitably large password dictionary can have a high degree of success against common passwords.

Dictionary attacks, like most attacks against unknown password-based authentication systems, can be slow when attempting to probe the authentication system one password at a time. When an attacker accesses the authentication system in question over the Internet,

network delays and delays in the authentication system itself can severely reduce the speed at which an attack can occur; however, when an attacker is able to obtain the username–password database, offline dictionary attacks can be highly effective in a short period. Tools such as `pwdump`<sup>37</sup> allow attackers to obtain the hashes for Windows passwords when an account with suitable privileges uses the tool. These hashes (known as NT LAN Manager [NTLM] hashes) are not directly reversible, but the security community knows the algorithm used to generate them. As a result, by using a suitable dictionary and the known algorithm, it is possible for an attacker to quickly iterate through the password dictionary to find the password that matches a given hash.

For any given user, there is no guarantee that the dictionary will contain the password. When the user selects a suitably random password, such as `@fA09wR&§$xZQ`, the probability of a dictionary containing such an entry is exceedingly slim. That said, and as mentioned previously, it is rare for a typical user to use such a random password because such passwords are difficult to remember. When an attacker has compromised the username and password database, and when the attacker has suitable hard drive space, an attacker can make a time–space trade-off using rainbow tables.

While not exactly a password dictionary, a rainbow table contains precomputed hashed passwords that allow an attacker to quickly locate the hash of a user's password in the tables to relate this to the original text password. An explanation of how an attacker generates rainbow tables and how the rainbow tables work is beyond the scope of this book, but readers can generalize the concept as a simple array of passwords addressed by their hash values. Rainbow tables are specific to the algorithm used by the authentication system (such as the NTLM hashes or message-digest algorithm 5 [MD5] hashes) and can be very large. Generally, attackers generate rainbow tables using a brute force approach to password generation, but it is possible for an attacker to generate rainbow tables from a password dictionary. In any case, the generation process is extremely time consuming and computationally expensive. Websites offer a wide variety of rainbow tables for various authentication systems for free<sup>38</sup> or for a small fee.<sup>39</sup>

Brute force attacks work on the principle that, given enough time, an attacker can find any password regardless of its length or complexity.



|    |    |     |             |
|----|----|-----|-------------|
| 1  | 13 | .   | 114         |
| 2  | 14 | .   | 121         |
| 3  | 21 | 44  | 122         |
| 4  | 22 | 111 | .           |
| 11 | 23 | 112 | .           |
| 12 | 24 | 113 | (and so on) |

**Exhibit 3-25** The brute force generation of passwords for the character set 1–4.

Brute force attacks against password-based authentication systems require the attacker to establish the set of letters, numbers, and symbols (known as the *character space* or *key space*) that are permissible for any password. The smaller the valid character set, the less time it will take to complete a brute force attack.

Once the attacker establishes the character set, hereby referred to simply as *the set*, the attacker generates a password by iterating through the set one character at a time. The attack starts by taking the first character in the set and using that as the password. Much the same way that an attacker tests for a valid password using a dictionary attack, the attacker enters the generated password in the authentication system to determine if the password is valid for the given user. If the password is incorrect, the attacker tries the next permutation from the given set and tests the password against the authentication system. The attacker repeats the process until he or she locates a correct password. Exhibit 3-25 explains the output of this process where the character set is only the numbers 1 through 4.

Brute forcing is a time-consuming process for a sufficiently complex password and a large character set. The maximum number of iterations required to find a password depends on two factors: the size of the character set and the maximum size of the password. Mathematically, it is possible to calculate the number of iterations by taking the number of entries in the character set and raising it to the power of the maximum length of the password. Exhibit 3-26 illustrates this as an equation. Furthermore, to determine the maximum amount of time required to find a given password using a brute force attack, it is necessary to multiply the maximum number of iterations against the time required for a single check (assuming that the time required does not vary as the length of the password increases). The

$$f(c, l) = c^l$$

where  $c$  is the character set size and  $l$  is the maximum length of the password

$$g(c, l, t) = t \cdot f(c, l)$$

where  $t$  is the time (in seconds) to perform a single password validity test

**Exhibit 3-26** Formulas to determine the maximum size and time required for a brute force attack.

second part of Exhibit 3-26 depicts the generalized mathematical formula for determining the maximum amount of time required to brute force a password of a given length.

**3.1.10.1 Attack** To clearly illustrate the amount of time required to brute force a password of sufficient complexity, given a character set defined as `abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()`, which has a character set size of forty-six characters and a maximum password length of eight characters, the maximum number of iterations required to fully exhaust the character set for the password length would be roughly 20 quadrillion iterations, or:

$$20,047,612,231,936 = 46^8$$

If the authentication system took one-half of a second per password supplied to indicate the validity of a password and if the password was “))))))”, it would take the attacker roughly 10 quadrillion seconds to find the password. Ten quadrillion seconds is the equivalent of 317,852.8 years. Such a long time renders the password “unbreakable” since the length of the attack outlasts the usefulness of the information under attack.

On the other hand, if the number of iterations per second was significantly higher as would typically be the case in an offline attack, the same password may no longer be unbreakable. For instance, using a tool such as John the Ripper<sup>40</sup> on a 2 GHz Xeon virtual machine image, it is possible to brute force 5,000 passwords per second. Using the same character set and password length as defined previously, it

would take approximately 4 billion seconds, or 127 years. By using a dedicated workstation and optimizing the application to run on multiple processors at the same time, it is possible to increase the passwords-per-second value to four times the speed, giving a rate of 20,000 passwords per second. At this rate, to brute force the entire password space would take approximately 1 billion seconds, or thirty-one years. While still conceivably too long for the password to be useful, the fact remains that given sufficient power processing, an attacker can sharply reduce the time required for brute force password attacks to be successful. Using 100 Amazon EC2 virtual images and splitting the total iteration count across the 100 servers, if each image were capable of performing 5,000 iterations per second, the effective iteration rate would be 500,000. This would reduce the time to find the same “))))))” password to approximately 40 million seconds, or 1.27 years.

A variety of tools are available on the Internet that perform brute force, dictionary, and rainbow table attacks. Exhibit 3-27 identifies some common tools and the types of attacks they perform.

When attempting to brute force a password, it is extremely important to reduce the time required per iteration to make the attack effective. For this reason, offline attacks against username and password

| TOOL             | TARGET  | DICTIONARY<br>ATTACK | RAINBOW<br>TABLE | BRUTE-FORCE<br>ATTACK | OFFLINE OR<br>ONLINE<br>MODE |
|------------------|---|----------------------|------------------|-----------------------|------------------------------|
| John the Ripper  | Password hashes<br>(MD5, SHA1,<br>NTLM, etc)            | Yes                  | No               | Yes                   | Offline                      |
| Ophcrack         | NTLM Hashes   | No                   | Yes              | No                    | Offline                      |
| Cain and Able    | NTLM Hashes,<br>WEP, MD5,<br>SHA1, MySQL,<br>MSSQL, etc | Yes                  | Yes              | Yes                   | Online and<br>Offline        |
| Crowbar          | Web<br>authentication                                   | Yes                  | No               | Yes                   | Online                       |
| L0phtCrack       | NTLM Hashes,<br>UNIX passwd<br>Files                    | Yes                  | No               | Yes                   | Offline                      |
| passcracking.com | MD5 Hashes  | Yes                  | Yes              | No                    | Offline                      |

**Exhibit 3-27** Common password attack tools.

databases are more suitable for brute force attacks than using a live authentication system, since many authentication systems purposely introduce delays.

Brute force attacks against live systems are clumsy and time consuming. Authentication systems can further hamper their effectiveness by introducing delays between authentication attempts. The higher the delay, the more time it will take to find a successful password.

When an attacker performs an attack against a live authentication system, the authentication system should restrict the number of logon attempts by a single user in a short period. Typically, system administrators use a threshold value of six or fewer logon attempts within 10–30 minutes before locking an account. This allows legitimate users the luxury of mistyping their password a few times but prevents dictionary and brute force attacks from rapidly trying passwords to compromise an account.

The system administrators design the complexity requirements for passwords to frustrate both dictionary and brute force attacks. By increasing the character space to include nonalphabetic characters and requiring passwords to be at least six characters long, administrators exponentially increase the amount of time required to brute force a password. Exhibit 3-28 compares the various size and complexity versus the maximum brute force iteration requirements. These same complexities can hamper the effectiveness of a password dictionary by moving the valid password design to a more random or complex word structure that may not typically exist in a password dictionary.

When an attacker obtains a username and password database, the only defense a system administrator has against an offline dictionary or brute-forcing attack is the complexity of the password construction. It is important that system administrators enforce sufficient password construction complexity rules. To prevent successful rainbow table attacks against an offline database, authentication system designers use salt values.

*Salts* are random characters added to the beginning of a supplied text password prior to the hash generation. By using a salt value, authentication systems can store the same password multiple times in a username and password database with different hash values for each user and salt. The salt is typically a multibyte value that is stored in the username and password data in plaintext for the authentication system

| CHARACTER SET   | PASSWORD LENGTH | MAXIMUM BRUTE-FORCE ITERATIONS |
|---|-----------------|--------------------------------|
| abcdefghijklmnopqrstuvwxyz  | 8               | 208,827,064,576                |
| abcdefghijklmnopqrstuvwxyz0123456789                                    | 6               | 2,176,782,336                  |
| abcdefghijklmnopqrstuvwxyz0123456789!@#%^&*()                           | 5               | 205,962,976                    |
| abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#%^&*() | 4               | 26,873,856                     |
| abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#%^&*() | 6               | 139,314,069,504                |
| abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#%^&*() | 8               | 722,204,136,308,736            |

**Exhibit 3-28** Maximum brute force iterations for various character sets and password lengths.

to hash the supplied password properly for comparison. The use of the salt increases the size of the possible password length exponentially, resulting in a significantly higher space requirement for rainbow tables. For instance, using an 8-byte salt would increase the size requirements for a rainbow table by 264 (18,446,744,073,709,551,616) times; however, the salt has little effect on dictionary and brute force attacks since the salt is typically available in the username and password database, therefore allowing the attacker to add the value as a static string.

It is worth noting that it is not necessarily better to include nonalphabetic characters with shorter lengths than it is to require only alphabetic characters with a longer length. For example, and as illustrated in Exhibit 3-28, it takes 100 times more iterations to find an eight-character password made up of a character set containing *a* through *z* than it does to find a six-character password made up of a character set containing *a* through *z* and 0 through 9. Companies can also find the trade-off of using a more complex character set on yellow Post-It notes attached to users' monitors.

3.2 Misdirection, Reconnaissance, and Disruption Methods

3.2.1 Cross-Site Scripting (XSS)

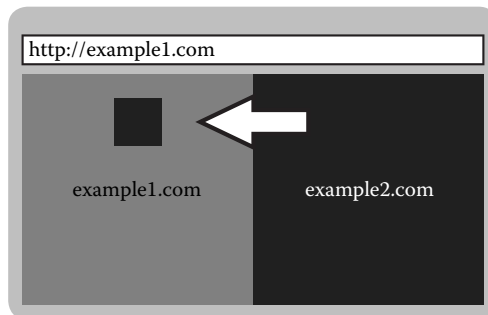
Improper input validation can allow an attacker to execute malicious scripts on Web pages with the same level of access as legitimately

included scripts. Used to access form variables and take actions on behalf of the user, cross-site scripting (XSS) attacks are the most commonly present and widely exploited type of vulnerability.

As the popularity of Web applications has grown in recent years, reports of one Web-based programming error, the XSS vulnerability, has grown in kind. In 2008, the National Vulnerability Database (NVD) recorded 806 XSS flaws, accounting for more than 14 percent of all new vulnerabilities.<sup>41</sup> The website xssed.com tracks thousands of XSS errors, many of which remain unfixed for months or years.<sup>42</sup> Despite its prevalence, many users and programmers do not understand how XSS attacks work or how to defend against them.

Shortly after Netscape developed JavaScript to allow programmers to create more dynamic Web pages, the name cross-site scripting was coined to describe a vulnerability it introduced. At the time, when two pages from different websites loaded next to each other in a browser (in a frame or separate window), JavaScript from one site could “cross” the boundary to read or modify the page from the other site.<sup>43</sup> To resolve this problem, Netscape implemented the same-origin policy with the release of Netscape 2.0 in 1996.<sup>44</sup> As shown in Exhibit 3-29, the same-origin policy prevents documents from one origin (example1.com) from reading or modifying documents from a different origin (example2.com).

All major Web browsers now implement the same-origin policy, and while it is effective, Web programmers can still leave their applications open to this type of attack through sloppy coding. The term XSS now refers to attacks that attempt to sidestep the same-origin



**Exhibit 3-29** Prior to the same-origin policy, one website could alter another when loaded in the same browser.

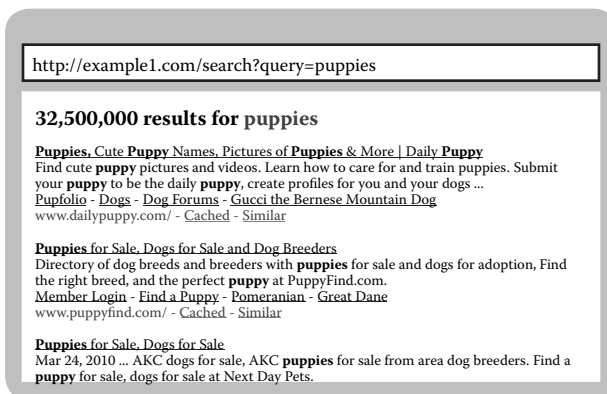
policy by causing malicious script code to run within the context of the original site and therefore with the “same origin.”

Like many common vulnerabilities, XSS flaws are essentially input validation errors. For an XSS attack to occur, a website must accept input from an untrusted source, such as a Web request, and serve the submitted input on a Web page. The infected page may be served to the same user, such as in the case of a search engine, or to any user, such as in the case of a blog’s comment form. If the website fails to sanitize the input, the browser will execute the malicious script.

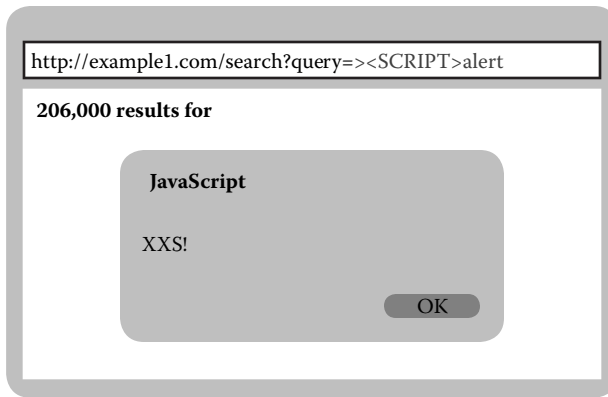
For example, a search engine must accept input from visitors to determine what the user is looking for. This information is often passed to the Web application through a URL parameter. URL parameters are key–value pairs appended to the end of a URL that the Web application can use to generate dynamic content. In the following example, the “query” variable is set to the value puppies.

`http://example.com/search?query=puppies`

The search engine takes this value, searches its database for pages containing the word *puppies*, and returns a result page like that shown in Exhibit 3-30. If the search engine does not properly sanitize the user’s input before displaying it, that input could alter the page in ways the developer did not intend. Rather than inputting a real search query, a user might enter JavaScript code that causes the page to show an alert that the query returns (e.g., `http://example.com/search?query= ><SCRIPT>alert(“XSS!”)</SCRIPT><>`).



**Exhibit 3-30** A search engine displays the result of a user’s query.



**Exhibit 3-31** A vulnerable site executes JavaScript code rather than displaying the query.

The vulnerable search engine queries for the malicious string and then returns it to the user along with the results. Rather than displaying the query text to the user, the browser generates an alert box containing the text “XSS!” demonstrating that code passed by the URL was executed (see Exhibit 3-31).

At first glance, this vulnerability does not appear to be very dangerous, as the user entered the text that caused the code to execute; however, attackers often craft malicious links containing the malicious script code, distribute these malicious links via e-mail, or post them to message boards and simply wait for users to click them.

Using XSS, attackers can steal information from a victim’s browser related to the vulnerable page. If the vulnerability exists in a banking application, the attacker could retrieve account balances and other private information from the page and send it off to his or her own server. XSS also enables attackers to take actions on behalf of victims. For instance, given a vulnerable Web-based e-mail application, the attacker could send e-mails from a victim’s account or forward e-mails containing sensitive information. Attackers can also steal the authentication cookie for a vulnerable site and later use that cookie to log on, thereby negating the need to steal the user’s credentials.

XSS attacks such as the one described above are known as nonpersistent, reflected, or Type-1 attacks because they only alter the page once, when the victim visits the specially crafted URL. Persistent XSS vulnerabilities are less common but much more dangerous. When attackers exploit a persistent XSS vulnerability, they make changes to



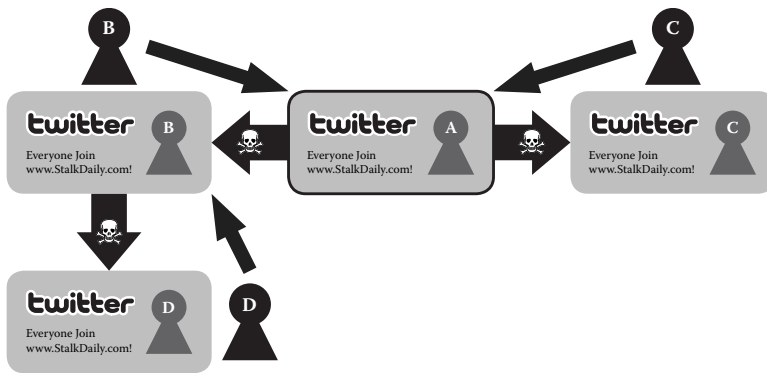
the page content that are stored in a database, which then affects every subsequent user who visits the page.

Many Web applications allow users to submit content that is stored in a database and later displayed to other users. For instance, the comment section of a blog application takes all comments and stores them in a database table associated with the blog entry. The application then selects these comments from the database and displays them when a visitor views the associated blog entry. If the application does not properly sanitize these entries, each subsequent visitor will execute the attacker's code in his or her browser.

Persistent XSS flaws are very similar to another input validation error, SQL injection,<sup>45</sup> in that both vulnerabilities allow the attacker to make persistent changes to a Web application and are predicated on improper input validation. One possible outcome of XSS is a type of Web worm that infects a user's Web pages (such as profiles or social-networking pages) instead of his or her computer.

On April 11, 2009, an XSS-based worm began spreading on the popular microblogging website Twitter.<sup>46</sup> A vulnerability in the Twitter code allowed an attacker to include a `<script>` tag in a profile page that linked to a JavaScript file hosted on a computer controlled by the attacker. When visitors viewed this page, their browser loaded this file and executed it. The content of this script took advantage of the fact that Twitter users visit the page while already signed into their Twitter accounts. The script updated the users' profiles to include the same malicious `<script>` tag, which could then infect visitors to the new user profile. The script also "tweeted" a message advertising [www.StalkDaily.com](http://www.StalkDaily.com), a Twitter competitor. This behavior allowed the messages and malicious code to spread quickly from account to account while advertising the attacker's website. As shown in Exhibit 3-32, when the B and C users visit the A user's infected profile, the malicious code immediately infects their profiles. Later, when the AB user visits the C user's profile, his profile also becomes infected, and so forth.

Users can mitigate the threat from XSS attacks by using modern browsers with XSS filter technologies. Internet Explorer (IE) 8 includes a filter that prevents reflected XSS attacks when it detects them. IE users who cannot upgrade should make use of security zones to prevent scripts from nontrusted websites from running.



**Exhibit 3-32** A cross-site scripting (XSS) worm spreads between Twitter accounts via an XSS attack.

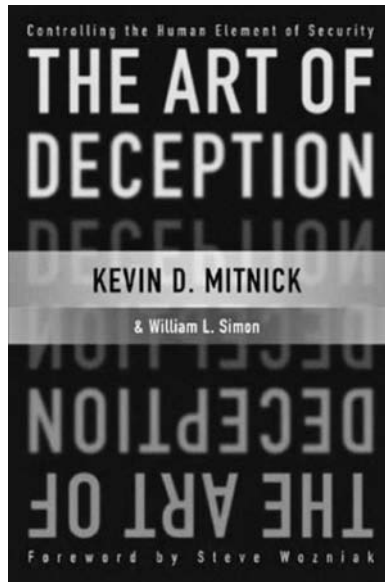
Firefox users can install the no-script plug-in, which also contains an XSS filter. No-script also allows the user to specify which sites they trust to run JavaScript code but denies nontrusted sites by default.

Web application developers should be careful to ensure their programs are not vulnerable to XSS attacks. The key to preventing XSS is treating all users' input as suspicious and sanitizing it before returning it with other dynamic content; however, this task is more difficult than it appears. The Open Web Application Security Project (OWASP) has created a cheat sheet that shows the many ways malicious actors can launch XSS attacks and how to defend against them.<sup>47</sup>

XSS is not only the most common Web vulnerability but also the most common vulnerability class overall.<sup>48</sup> While the impact may not seem as severe as that of the venerable buffer overflow, attackers frequently exploit XSS vulnerabilities, often with severe consequences for the vulnerable site's users.

### 3.2.2 Social Engineering

No matter how quickly an organization patches the latest zero-day vulnerability or how many security products it deploys, one major vulnerability remains in every system: the human being. Social engineering is the art and science of attacking the human element of a system to gain sensitive information or access to restricted areas or systems. In this section, we explain the concepts behind social



**Exhibit 3-33** The *Art of Deception* by Kevin Mitnick. (Mitnick, Kevin D. and William L. Simon; *The Art of Deception*. 2002. © Wiley–VCH Verlag GmbH & Co. KGaA. Reproduced with permission.)

engineering and how modern attackers use the technique to further their criminal operations.

While there are many definitions for social engineering, one of the most succinct and accurate describes the practice as a hacker's clever manipulation of the natural human tendency to trust.<sup>49</sup> Humans must trust each other every day to survive in the modern world. Each time a person drives a car, he or she places a little trust in the people who designed the vehicle and bolted it together and in every other driver he or she encounters. While some people (especially in the security industry) are skeptical of others, or even paranoid, people who have no trust in their fellow human beings could not function in a modern society. A clever attacker can abuse the natural human tendency to trust to convince people to do things that are not in their best interests. While testifying before Congress in 1995, Kevin Mitnick, one of the world's most famous hackers and the author of *The Art of Deception* (see Exhibit 3-33), stated, "I was so successful in [social engineering] that I rarely had to resort to a technical attack."<sup>50</sup>

One way for an attacker to build trust with a target is with information—ideally, nonpublic information. In this context, nonpublic information is anything the target believes the public in general does

not know. Whether or not this belief is true is irrelevant. People often use another person's possession of nonpublic knowledge to make decisions about their trustworthiness. For example, Bob is throwing a twenty-first birthday party for Alice with a large but limited guest list. Bob asks Walter to guard the door and gives him the guest list. Uninvited, Mallory might approach Walter while carrying a wrapped package and say, "Hi, I'm here for Alice's birthday party, I'm not on the list but Bob told me you would let me in." When deciding whether to let Mallory into the party, he will consider the fact that Mallory was carrying a gift, that the party was for Alice's birthday, and that Bob was in charge of who attends the party. One could obtain all of the information Mallory provided by eavesdropping on a casual conversation. While nothing Mallory said gives her authorization to enter the party, Walter is likely to ignore the list and open the door.

Empathy is another weapon in the attacker's trust-building toolkit. Most people feel bad when they see someone in trouble, but more importantly, they identify with them. By appearing in trouble and making it easy for an attacker's target to help, that attacker can use human empathy to get what he or she wants. Consider an elevator controlled by an electronic badge system that prevents unauthorized users from entering the building's secured floors—floors 4–7. Mallory, an unauthorized user, enters the elevator carrying some heavy boxes at the same time that Bob, the authorized user, enters the elevator. Mallory casually says to Bob, "Can you hit 6 for me?" While Bob has not verified that Mallory has access to the sixth floor and has no actual reason to trust her, he may press the button because he identifies with her situation and feels empathy for her.

Appealing to the target's wants or needs is another effective social-engineering technique. People are often willing to take imprudent actions that they think will result in a reward of some kind. In 2008, a survey of 576 British office workers found that almost 20 percent of people are willing to give up their logon and password for a piece of chocolate.<sup>51</sup> Changing the prize to a ticket for a raffle offering a trip to Paris yielded more than 60 percent of the workers' passwords. The same tactic also applies to targets who want to avoid pain or punishment. Offering to carry someone's heavy boxes may get an attacker through a door without authorization, or berating a low-paid help


desk employee over the phone could result in a cornucopia of valuable nonpublic information.

To be successful, attackers often need to use these tactics and others to gain enough information and access to complete their tasks. In Ira Winkler's 1997 book *Corporate Espionage*, he describes a social-engineering penetration test he conducted completely by phone that gave him complete access to a corporation's systems.<sup>52</sup> They completed the task by using pieces of nonpublic information to gain the trust of humans in the company. Each piece of information gave them the ability to get slightly more information. Winkler gains the key pieces in the following order:

1. An executive's name and the company's phone number (from an annual report and local phone book)
2. An executive's employee ID and cost center numbers (from the executive's secretary)
3. A company phone directory (by posing as the executive and using his or her cost center number)
4. A list of fifty-five new employees (from the new-hire administration office by posing as a secretary of an angry executive who needed the list)
5. The types of software and hardware used in the company (from new employees by posing as a security officer giving training to each new employee by phone)
6. The phone numbers and passwords for modems on the company network (from new employees during the same security training call)

While the information in numbers five and six is sensitive on its own, the previous items are small enough that they do not appear to be a major threat to security. By using them together, Winkler was able to gain all of the information he needed to access the computer systems without ever stepping foot in the building.

While the examples above all target a specific individual or group, attackers can also use social engineering against large groups of people. The best example of widespread social engineering is phishing. In a phishing scheme, the attacker sends an e-mail to at least one person (but more likely thousands) that asks the target(s) to reveal private information (passwords, credit card numbers, etc.). The success of a

 Automatic Data Processing, Inc.

---

To: \_\_\_\_\_VAR\_FIELD1\_\_\_\_\_

Your payroll has been rejected because the data at the bottom of the batch is invalid.  
The debit account for \_\_\_\_\_VAR\_FIELD2\_\_\_\_\_ is also invalid.  
Please check it and re-send me the correct batch.  
Please correct the errors and reply me ASAP.

With respect,  
Steve Irwin  
Automatic Data Processing, Inc

**Exhibit 3-34** A Better Business Bureau (BBB) e-mail template using the ADP Employer Services theme.

phishing scheme relies on how cleverly the attacker can manipulate the target into trusting the content of the e-mail.

Using a bank's logo and copying its website comprise one way to establish this trust. The target sees these pieces of information and uses them to judge the trustworthiness of the e-mail. Phishing schemes also often appeal to a user's wants or needs by offering a prize for filling out a form or threatening to lock the target's account if he or she does not comply with the e-mail's instructions.

More targeted phishing schemes, known as *whaling* or *spear-phishing attacks*, allow social engineers with additional knowledge to create convincing e-mails. Consider the e-mail template shown in Exhibit 3-34.

Criminals sent e-mails using this template in April 2008 to thousands of business e-mail accounts. The e-mails included the victim's name in the "To:" field. While a person's name does not seem like a piece of nonpublic knowledge, people are often more likely to trust an e-mail if they feel like it pertains specifically to them. The message implies that the recipient will not receive his or her paycheck, which is likely to invoke a strong emotional reaction. Finally, the attackers sent the e-mails on a Thursday, which is a common day for accounting departments to finalize payroll. All of these items together made this a very successful social-engineering attack.

Attackers commonly use social engineering to install malicious code. A computer system may have the latest patches and be protected against malicious code that spreads through known vulnerabilities, but attackers can still infect a system if they convince the user to download and install their program. One popular way to do this is to entice a user with a video of a hot news topic, such as a celebrity sex tape<sup>53</sup> or death.<sup>54</sup> When the target visits the website, the target



**Exhibit 3-35** Attackers use social engineering and a fake software update to infect a target with malicious code.

receives notification that his or her browser cannot display the video unless he or she downloads new software (see Exhibit 3-35).

Rogue antivirus (AV) applications are a criminal scheme that relies heavily on social engineering. These products appear to be legitimate AV programs but provide no actual protection and incessantly report infections that do not exist. Victims often install rogue AV programs when they visit a website and see a frightening pop-up claiming that a virus or Trojan has infected their computers (see Exhibit 3-36). When the user follows the pop-up's instructions, they unknowingly install the rogue AV program. These schemes prey on users' fears of malicious code and test their patience through annoying alerts. The attackers make money by convincing the users to spend up to US\$89.95 to buy the "full version" of their products. For more information on rogue AV products, see the "State of the Hack" article titled "iDefense Explains ... Rogue Anti-Virus."



**Exhibit 3-36** A rogue antivirus (AV) popup displayed to users to entice them to download malicious code.

These are just a few examples of modern social-engineering attacks. In reality, most attacks involve some level of social engineering, some large and some small. Unfortunately, no technology or simple solution can defend against social-engineering attacks. “There is no patch for human stupidity” is a common but somewhat crude axiom used in the security community to describe this situation. A more accurate version might be “There is no patch for the human tendency to trust.” The best mitigation strategy for an organization against social engineering is to build a culture of security through awareness training and education. Alerting users of widespread attacks as those attacks occur, and giving users regular instruction, will give them the tools to detect social-engineering schemes before the schemes can do any harm.

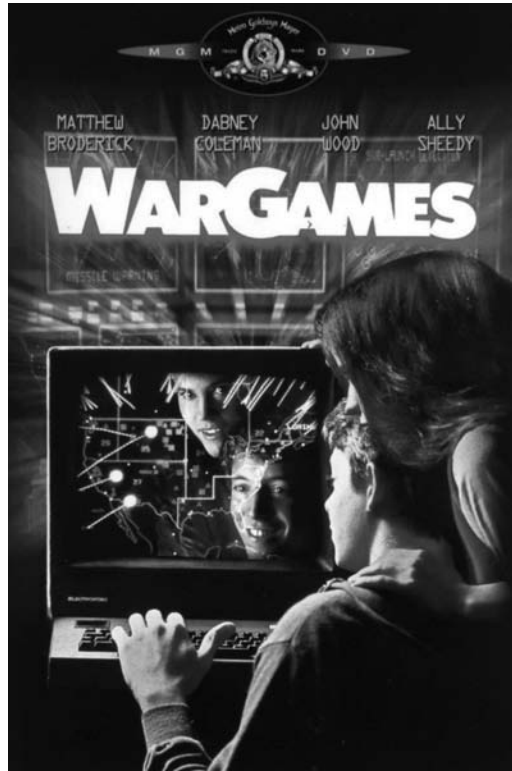
### 3.2.3 *WarXing*

In 1983, the movie *WarGames* caught the imagination of enterprising young hackers everywhere (see Exhibit 3-37). The movie was very influential to the hacker culture, and the act of dialing numbers to discover listening modems became known as *war dialing*. During the past twenty-five years, war dialing, war driving, and war spying developed as reconnaissance techniques used by hackers to discover possible targets and learn more about the networks accessible to them. This section discusses these techniques and the origin of the *WarXing* nomenclature.

Technology professionals love inventing new words, and security researchers are no different. *War dialing*, *war driving*, and *war spying* are names for reconnaissance techniques used by hackers to discover possible targets and learn more about the networks accessible to them.

In *WarGames*, Matthew Broderick portrays a high school student who uses his computer to dial every phone number in Sunnyvale, California automatically to search for modems owned by software companies in hopes of getting copies of the latest games before their release to the public. The movie was very influential to the hacker culture, and the act of dialing numbers to discover listening modems became known as *war dialing*.<sup>55</sup>

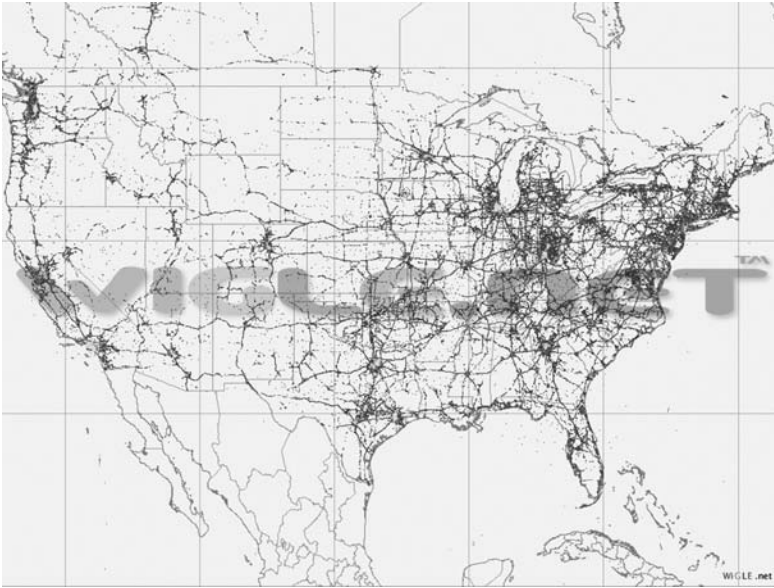




**Exhibit 3-37** *WarGames* movie poster.

War dialing is essentially a reconnaissance technique. It allows an attacker or curious individual to reach out through the phone network and determine what types of other systems might be accessible to him or her. Automated war-dialing programs dial a range of numbers and wait for one or two rings. If a modem answers the call, the program makes a note of it and moves on. If a person or voicemail system answers the phone, the program disconnects. The result is a list of every accessible modem in a particular phone prefix that the war dialer can then target for further attacks.

With the rise of broadband in homes and businesses during the 1990s, fewer and fewer modems were attached to phone lines and war dialing became less popular, but criminals found new uses for the technology. Vishing, or voice-phishing, attacks use the phone system to gather secret information about a target. Attackers can use automated dialers that dial huge volumes of numbers until



**Exhibit 3-38** A WiGLE map of U.S. wireless networks.

a victim answers, at which point a voice-based computerized prompting system filled with social-engineering questions begins asking for (or demanding) personal information of the unsuspecting targets.

In the last decade, war dialing has transformed into other forms of network reconnaissance, carried out with the same hacker spirit that drove the early phone system pioneers. War driving is the act of mapping out the location of Wi-Fi networks. Rather than dialing up a series of telephone numbers, “war drivers” hop in their car with Wi-Fi-enabled laptops or PDAs and drive through neighborhoods recording the location and name of each network they detect. The goal of war driving is not malicious, and its practice does not harm or annoy the networks they detect. The WiGLE Project currently tracks more than 15 million networks worldwide and allows users to search for networks using an interactive map (see Exhibit 3-38).<sup>56</sup>

For those with an interest in mapping Wi-Fi networks but with no car to carry them from place to place, there is war walking, war cycling, and even war flying. The first known instance of war flying occurred in 2002, when members of the WAFreeNet group flew over Perth, Australia, and mapped out that city’s networks from the air.<sup>57</sup>



**Exhibit 3-39** WARSpyLA war-spying equipment.

War spying, another derivative of war driving that goes beyond simple network mapping, targets specific types of wireless-enabled surveillance devices that give a glimpse into areas typically out of view. Low-end security systems, such as those used for surveillance by small businesses, use wireless technology to transmit images to a central computer. This is convenient because it does not require running cables to the cameras but leaves them open to war spying if not properly secured. War spies spend time and money building equipment to seek out and display signals they find, such as the one displayed in Exhibit 3-39, built by WARSpyLA, a Los Angeles-based war-spying group.<sup>58</sup>

While curiosity and not malice drives each of the WarXing activities, the information gathered during WarXing reconnaissance can support destructive and criminal activities. War dialing may lead to attacks on systems connected to the phone system via modems. Mapping wireless networks through war driving could allow an attacker to find open networks that he or she could use to launch criminal activity without being traced, or simply to sniff the traffic of the unencrypted network to steal personal information. War spiers may be watching open security cameras out of curiosity, but malicious actors could also use them to scout a location before a physical break-in. While the threat from WarXing activities may appear

to be trivial, it is important for security professionals to be aware of these techniques and understand their place in hacker culture. There is very little that an organization can do to keep curious war drivers from mapping their networks, but administrators should ensure that they properly secure any wireless devices. For anyone wondering how many Wi-Fi networks are in your neighborhood, head over to the WiGLE Project.<sup>59</sup> The results might be surprising.

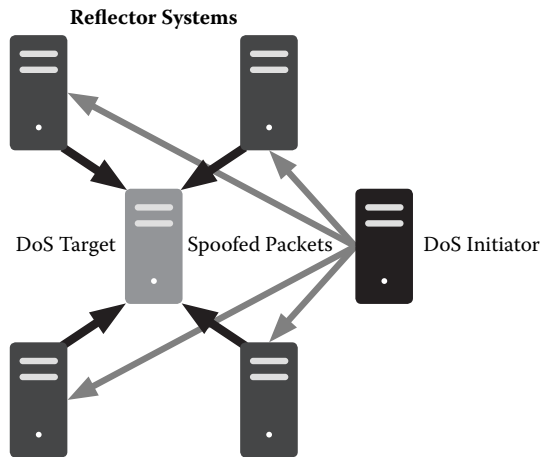
### 3.2.4 DNS Amplification Attacks

This section looks at domain name system (DNS) amplification techniques, which utilize the DNS, or misconfigured DNS servers, to launch denial of service (DoS) attacks while using minimal amounts of bandwidth on the part of the attacker. DNS amplification attacks rely on the ability to spoof the originating Internet Protocol (IP) address in User Datagram Protocol (UDP) packets, thereby instructing a DNS server to reply to a specific address, known as a reflection attack. The amplification portion of the attack relies on the DNS server's willingness to perform a query on behalf of the attacker, which results in a response that is larger than the original query. A recently publicized DNS amplification simply issued queries for the root (".") servers, instead of the more common technique of poisoning a particular DNS server with an abnormally large DNS record. DNS amplification attacks take advantage of the large resources of DNS servers and can be enormously powerful. The article also discusses well-known but rarely implemented techniques to mitigate these attacks.

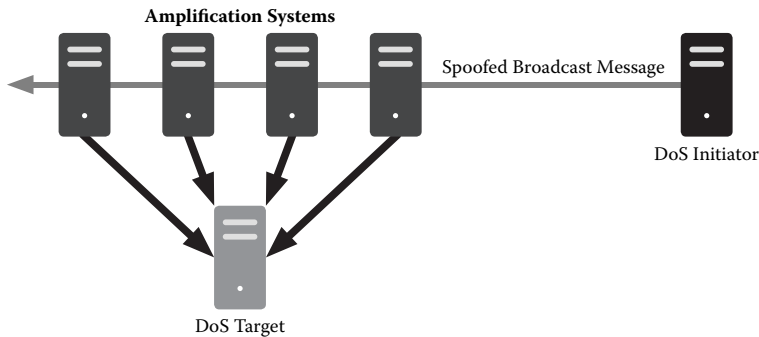
Denial of service (DoS) attacks come in many forms, but successful attacks all result in the same outcome: the targeted service is made unavailable to users. A common form of DoS is known as *network resource exhaustion*. These attacks involve sending more traffic to a server than it can properly handle, effectively blocking communication with legitimate systems. Network exhaustion requires the attacker to generate a very large amount of traffic, frequently using many systems collected in a botnet. This section explains a complex DoS attack known as *domain name system (DNS) amplification*, which takes advantage of features in the Internet architecture to turn a small number of systems into an Internet-based weapon.

To understand DNS amplification, it is first necessary to understand the two concepts that allow the attack to take place. The first is known as *reflection*. Unlike the more commonly used Transmission Control Protocol (TCP), the User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) are not connection based. When two computers communicate using TCP, they first perform a three-way handshake that allows the systems to synchronize settings and ensure that the system that sent the original packet actually intended to establish a connection. UDP packets simply contain a source and destination Internet Protocol (IP) address (as part of the IP packet that encapsulates them), but the recipient of the UDP packet cannot be sure that the source was not falsified, or *spoofed*. A computer that receives a packet with a spoofed IP address will reply to that IP address, rather than the original sender. This is known as a *reflection attack* because the original packet reflects an intermediate system before striking the targeted system. The benefit of a reflection attack is that the victim cannot tell the original source of the traffic without getting information from the intermediate system.

Reflection can also allow an attacker to make traffic appear to come from many sources, making it more difficult for the victim to filter the incoming data. In Exhibit 3-40, the attacker sends ICMP echo requests (*pings*) to multiple intermediate systems while spoofing the IP



**Exhibit 3-40** A reflection attack allows a single attacker to send traffic from many sources.



**Exhibit 3-41** A smurf amplification attack multiplies the total traffic sent by the initiator.

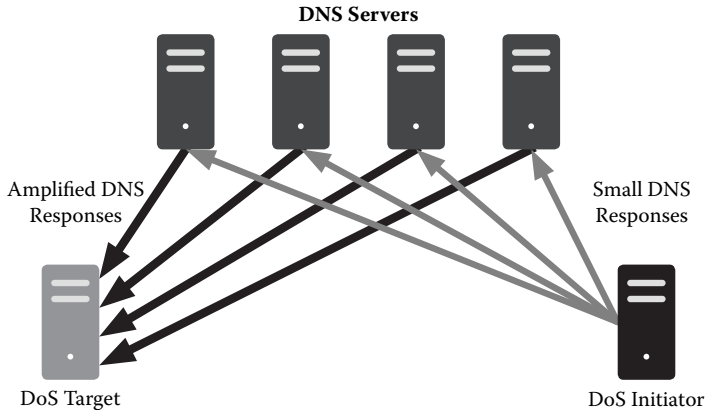
address in the packets to match that of the victim. Each compromised computer dutifully replies to the victim, flooding it with traffic.

While reflection attacks hide the source of a DoS initiator, they still require the initiator to be able to send as much data as he or she wants the victim to see. Amplification attacks take this to the next step, increasing the total amount of traffic received by the target to tens or hundreds of times the original payload size.

One of the oldest amplification attacks is known as the *smurf attack*. In a smurf attack, the initiator sends a ping to the IP broadcast address, spoofing the victim's IP address. Exhibit 3-41 shows how the smurf attack works by sending a single packet that results in an amplification of the total traffic received by the victim. Smurf attacks are no longer a major threat because router configurations no longer pass broadcast packets to other networks, limiting the range of a smurf attack to the local subnet.

A more recently discovered class of DoS attacks takes advantage of the DNS infrastructure to amplify the amount of traffic a single node can send. A DNS request for the A record of a specific domain, such as google.com, returns much more data than is required to make the request. DNS requests commonly used connectionless UDP messages, making them ideal for reflection and amplification attacks.

The key to launching an amplification attack is finding DNS servers that will perform recursive queries and return a larger amount of data than was in the original request. These servers are commonly known as *open resolvers*. For more information about recursive DNS queries and the DNS in general, refer to the “State of the Hack” article entitled “iDefense Explains the Domain Name System (DNS).”<sup>60</sup>

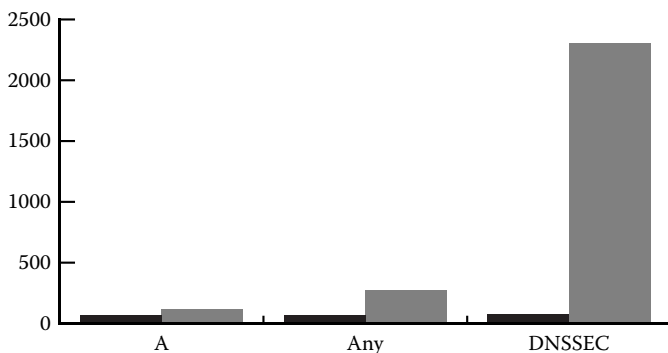


**Exhibit 3-42** DNS servers amplify the amount of traffic generated by the DoS initiator.

A typical DNS A record query requires the client to send at least 49 bytes of data, depending on the length of the domain name requested. In some cases, the response message may be smaller than the original request, but it is very simple to ensure a large response by requesting additional information. Exhibit 3-42 shows how this type of attack exploits DNS servers.

A request for the A record of google.com requires sending 70 bytes of data and will result in a 118-byte response (including the encapsulating Ethernet, IP, and UDP headers). This is only a minor amplification, but by making an ANY query instead of an A record query, the server also returns the values it has stored for MX and NS records, resulting in a 278-byte response, or an almost 4x amplification. To take amplification even further, the attacker can take advantage of the large records provided by DNS Security Extensions (DNSSEC) signed zones. DNSSEC provides cryptographic signature data along with normal DNS record data to allow the recipient to verify that the data are authentic. These records greatly increase the amount of traffic returned by a DNS server. The U.S. government recently signed the .gov domain, and these signature data are available through DNS servers. Making a 74-byte ANY request for the .gov domain, specifying that DNSSEC records be provided, results in a 2,309-byte response, an amplification factor of more than 30. Exhibit 3-43 displays how different DNS requests result in different amplification factors.

These types of attacks rely on open resolvers to carry out malicious deeds, but a new form of DNS amplification recently emerged that



**Exhibit 3-43** The amplification factor depends on the type of DNS query made.

can take advantage of locked-down DNS servers. In January 2009, attackers launched a large-scale distributed denial of service (DDoS) attack against an ISP named ISPrime.<sup>61</sup> To take advantage of DNS servers that are not open resolvers, the attackers made requests for “.”, the designation of the root servers. DNS servers respond to these requests with the IP address of each of the root servers, even when they do not allow recursive queries. This response can include more than 500 bytes of data and only requires a 49-byte request, an amplification factor of more than 10.

Amplifying the amount of bandwidth generated is very beneficial in a DoS attack. An infected computer using a cable modem may only be able to generate 512 Kbps of bandwidth, which is not enough to cause a DoS for most servers. Amplify this value by thirty times, and use this technique on 1,000 nodes, and a small botnet could easily generate more than 14 Gbps of traffic, enough to saturate almost any Internet connection.

**3.2.4.1 Defeating Amplification** DNS amplification attacks rely on two principals without which they could not be effective. First, UDP packets can carry spoofed IP addresses, allowing attackers to reflect traffic off DNS servers. This does not necessarily need to be the case. While UDP does not make it possible to verify the source of a packet, an ISP can inspect packets leaving their network to ensure that the source IP address could actually reside within that network. The Internet Engineering Task Force (IETF) “Best Current Practice (BCP) 38” (BCP38) document suggests this type of filtering.<sup>62</sup> While



many networks have implemented the filtering suggested by BCP38, many large networks have refused to, stating that the filtering requires too much overhead for their equipment or that their customers may need to generate traffic with spoofed-source addresses.

The second necessity for DNS amplification to succeed is the cooperation of one or many DNS servers. In 2006, Dan Kaminsky presented evidence that more than 580,000 open resolvers were active on the Internet.<sup>63</sup> It is unlikely that the operators of these servers intend the public or DoS attackers to use them. Any administrator who operates a DNS server should configure it to either not perform recursive queries or limit it so only specific IP ranges can perform these queries. To combat the latest “root query” form of amplification attack, administrators should configure servers to either not reply to requests for the root domain or limit the systems allowed to request this information to those on a trusted network. More information on how to make these changes in Berkeley Internet Name Daemon (BIND) 9 is available from SecureWorks.<sup>64</sup>

DNS amplification attacks are not a recent discovery, but DDoS networks have not commonly used them. If they began taking advantage of amplification, small botnets could become much more powerful than ever before.

## References

1. Tenouk, “Buffer Overflow 9,” n.d., <http://www.tenouk.com/Bufferoverflowc/Bufferoverflow5.html>.
2. Kernel Panic, “Writing Shellcode for Linux and \*BSD,” n.d., <http://www.kernel-panic.it/security/shellcode/shellcode3.html>.
3. No Login, “Understanding Windows Shellcode,” December 6, 2003, <http://www.hick.org/code/skape/papers/win32-shellcode.pdf>.
4. Phrack, “Phrack Issues,” n.d., <http://www.phrack.com/issues.html?issue=57&id=15>.
5. iDefense explained stack buffer overflows in “State of the Hack,” iDefense Weekly Threat Report, ID# 480099, January 5, 2009.
6. <http://www.secdev.org/projects/shellforge>. Accessed August 11, 2010.
7. Metasploit, “Payloads,” n.d., <http://metasploit.com:55555/PAYLOADS>.
8. For more information on how buffer overflows work, please see the iDefense Weekly Threat Report, ID# 480093, January 5, 2009.
9. U.S. Computer Emergency Readiness Team, “Vulnerability Note VU#540517,” 2004, <http://www.kb.cert.org/vuls/id/540517>.
10. CodePlex, “SafeInt,” 2006–2010, <http://www.codeplex.com/SafeInt>.