# CSCI 2330



# TODO

- Assignment 4 assigned due November 10

# Lab 4

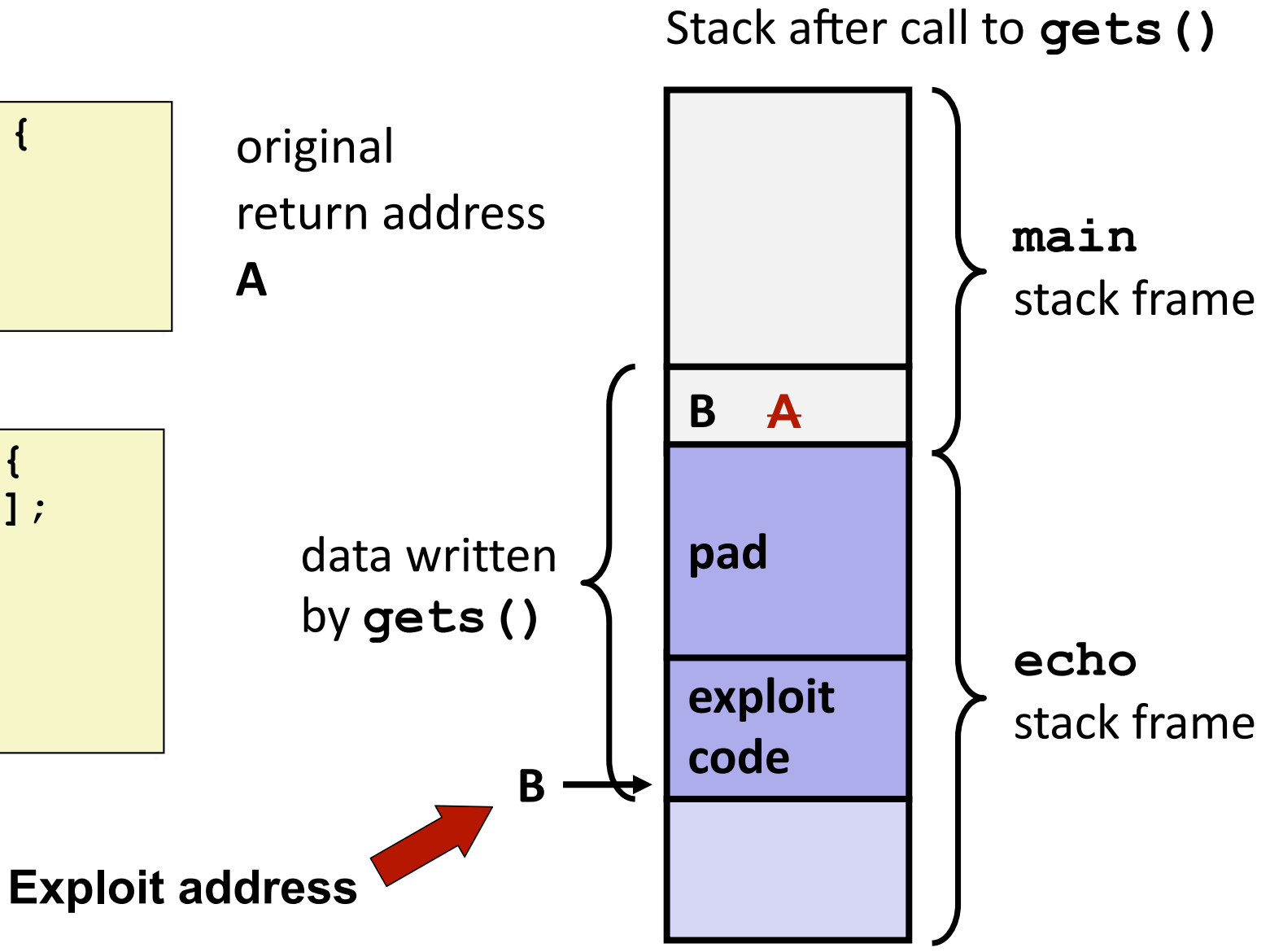
- Five buffer overflow exploits against a vulnerable program
- #1 is similar to the `buftest` return-to-function example
- #2 and #3 are return-to-buffer attacks (stack randomization is disabled)
- #4 and #5 are return-oriented programming

# Example return-to-buffer attack

- We won’t be doing anything as intricate as generating shell code
- But let’s show how we can execute arbitrary code

```
void main() {
  echo();
  ...
}
```

```
void echo() {
  char buf[4];
  gets(buf);
  ...
  return;
}
```



# Goal

- Print “never print”

# Dump of dummy

```
(gdb) disass dummy
Dump of assembler code for function dummy:
   0x000000000040057d <+0>:     sub    $0x8,%rsp
   0x0000000000400581 <+4>:     mov    $0x400660,%edi
   0x0000000000400586 <+9>:     call   0x400450 <puts@plt>
   0x000000000040058b <+14>:    add    $0x8,%rsp
   0x000000000040058f <+18>:    ret
End of assembler dump.
```

Use length-limited string routines (**`fgets`**, **`strncpy`**, ...)

No **`%s`** in **`scanf`**

... etc ...

**Don't make any mistakes...**

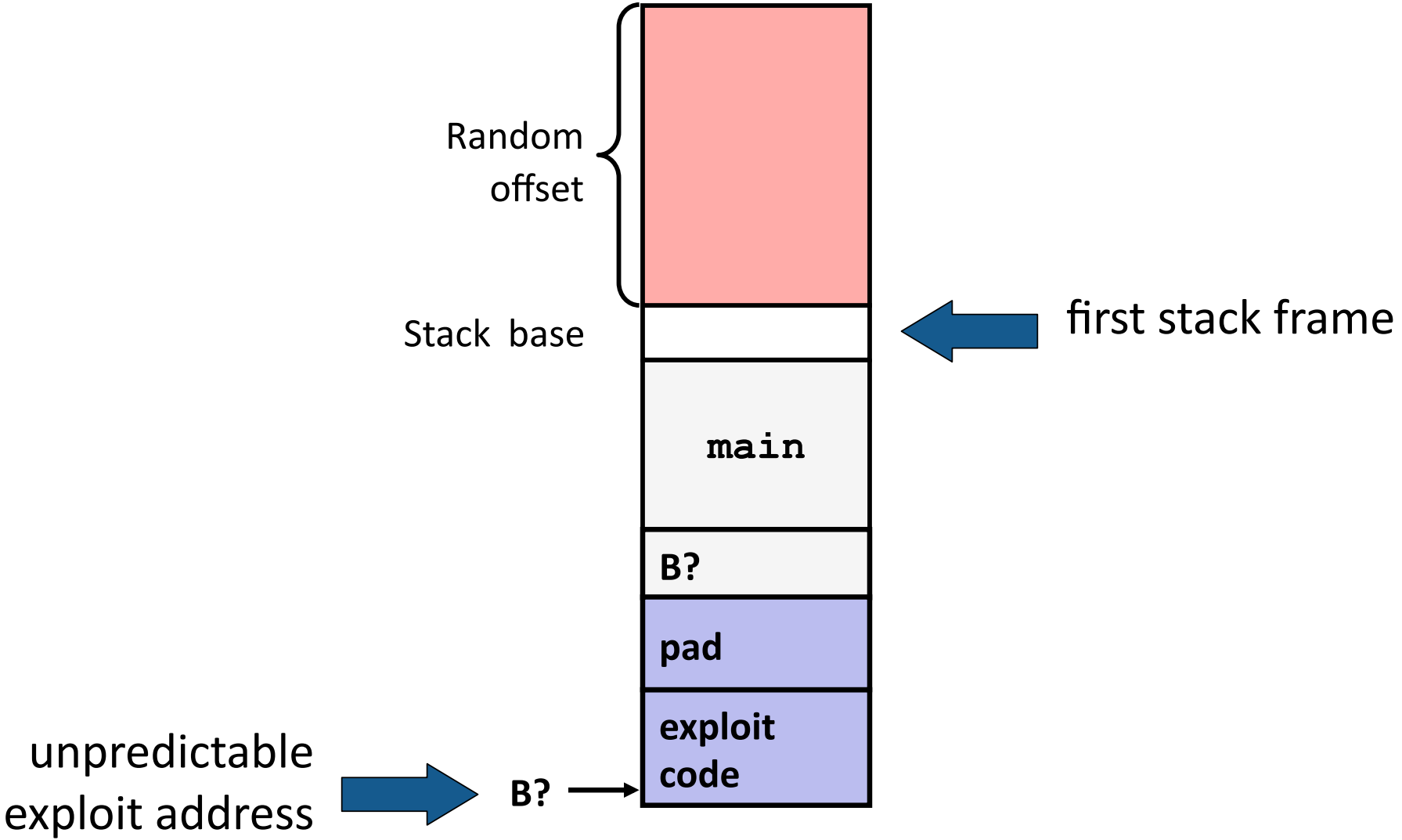
```
/* Echo Line */
void echo() {
    char buf[64];
    fgets(buf, 64, stdin); // safe
    puts(buf);
}
```

# Other protections

- Since fixing all code is hard, we might want other protections
- Operating systems provide multiple protections to help mitigate these attacks
  - None are foolproof, but they make attacks more difficult

# Address space layout randomization

- a.k.a. ASLR
- Randomizes the location of some memory segments including the stack and the heap
- Makes return-to-buffer attack more difficult
- Doesn’t affect return-to-function attack


Random
offset
Stack base
first stack frame
main
B?
pad
exploit
code
unpredictable
exploit address
B?

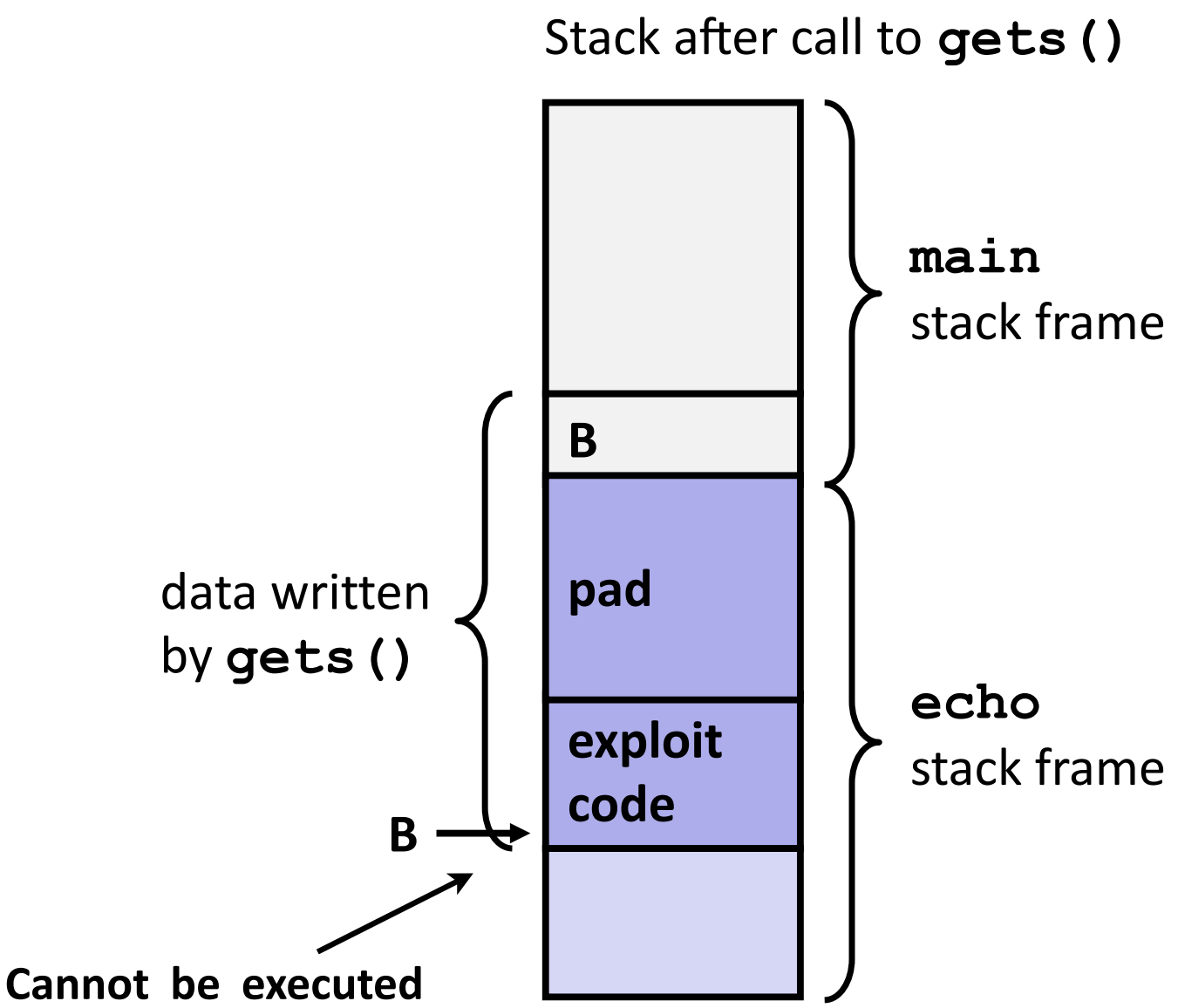
# Nonexecutable stack

- Von Neumann architecture:
  - Is it code?  Is it data?
  - It’s a matter of interpretation…
  - It all shares the same memory



# Nonexecutable stack

- Maybe we shouldn’t be able to execute code on the stack?
- On x86, called “NX bit”
- Set on a memory segment to indicate that the stack segment should not be executed


Stack after call to gets()
main
stack frame
B
pad
exploit
code
echo
stack frame
data written
by gets()
B
Cannot be executed

# Nonexecutable stack

- Which attack we've seen would this make more difficult?
- Which would be unaffected?

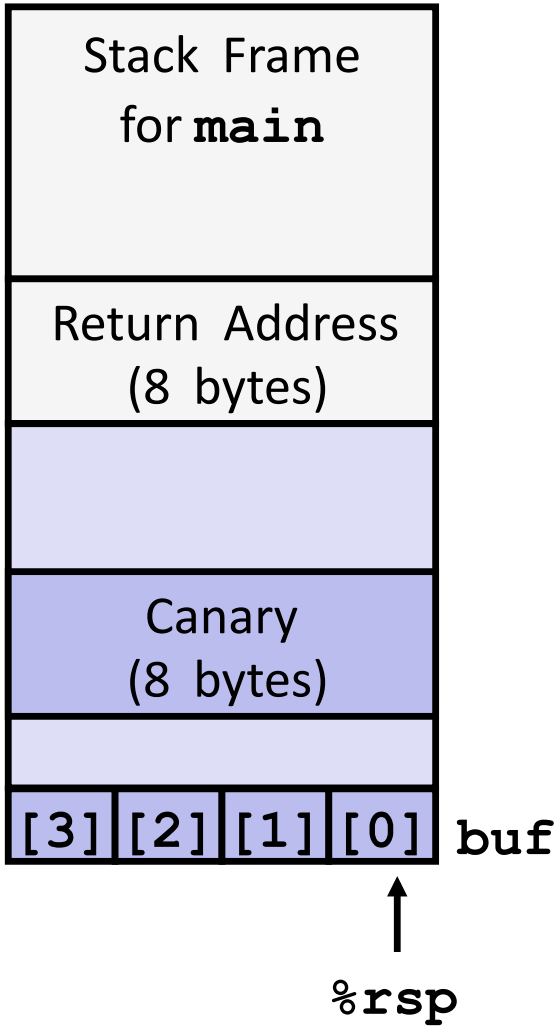
# Nonexecutable stack

- Which attack we've seen would this make more difficult?
- Which would be unaffected?
- In fact, just like with ASLR, `buftest` has also had the NX bit set this entire time!

# Canaries



*Before call to gets*



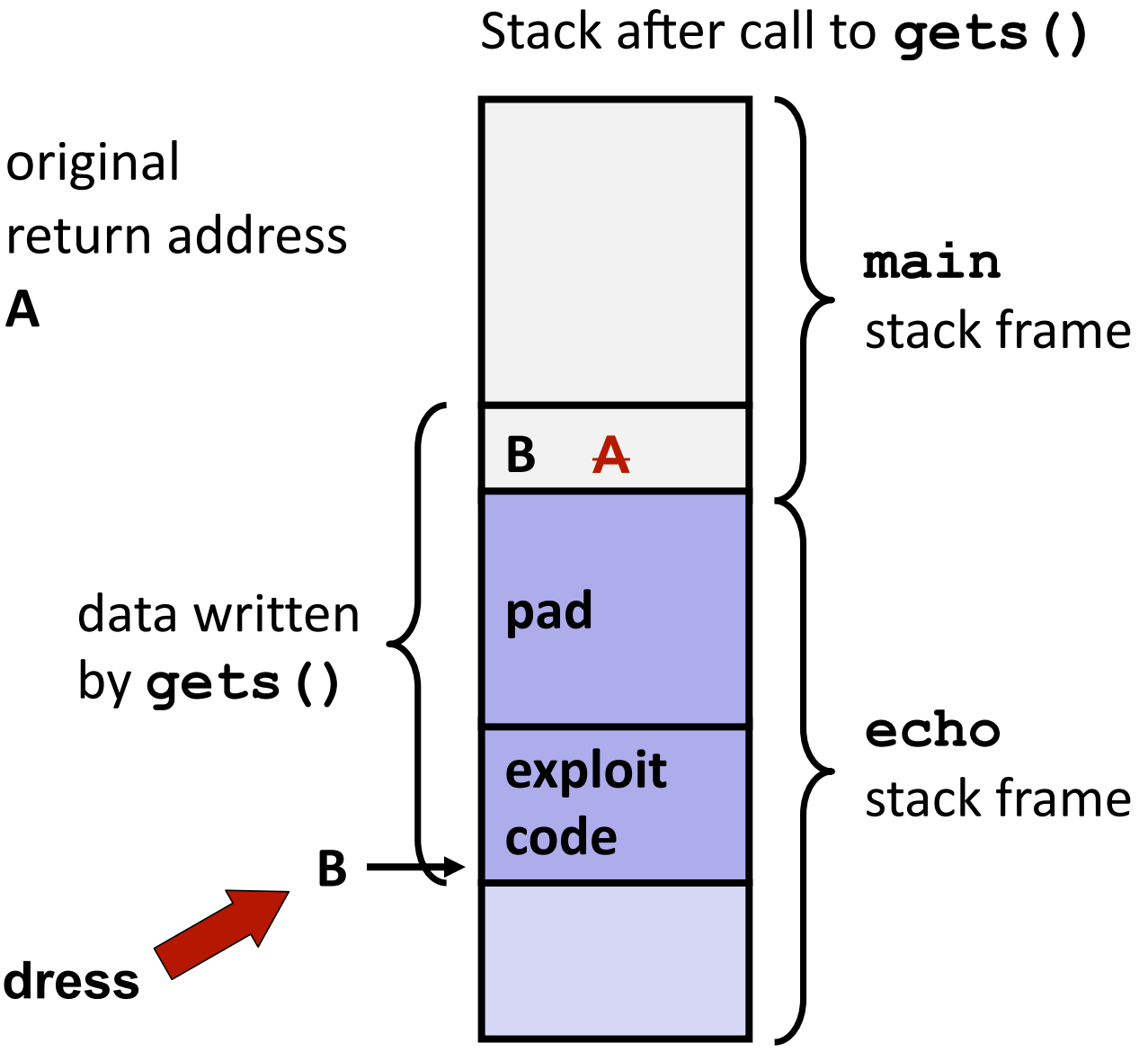

```
/* Echo Line */
void echo() {
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**echo:**

```
  40072f:  sub    $0x18,%rsp
  400733:  mov    %fs:0x28,%rax   # Get canary
  40073c:  mov    %rax,0x8(%rsp)  # Put on stack
  400741:  xor    %eax,%eax
  400743:  mov    %rsp,%rdi
  400746:  callq  4006e0 <gets>
  40074b:  mov    %rsp,%rdi
  40074e:  callq  400570 <puts@plt>
  400753:  mov    0x8(%rsp),%rax  # Retrieve canary
  400758:  xor    %fs:0x28,%rax   # Check canary
  400761:  je     400768 <echo+0x39>
  400763:  callq  400580 <__stack_chk_fail@plt>
  400768:  add    $0x18,%rsp
  40076c:  retq
```

```
void main() {
   echo();
   ...
}
```

```
void echo() {
   char buf[64];
   gets(buf);
   ...
   return;
}
```



# Canaries

- The canary is randomly chosen at runtime (why?)
- The upside is more difficult attacks
- Is there a downside?
  - They are often not enabled by default
- Which attacks would this make more difficult?
- Let’s look at `buftest-canary`…

# Threat model

- Let’s assume ASLR and NX bit
- But no canaries
- We can call `dummy()`
- But this is not arbitrary code
  - This is existing code!
- Can we execute *arbitrary* code?

# Return-oriented programming (ROP)

- What if there is no function existing in the program that does *exactly* what we want?
- ROP is a method to execute arbitrary code by utilizing the existing code in the program
- ROP builds an exploit by chaining together chunks of existing code

# Return-to-function attack

- What happened after we jumped to dummy()?

# Return-to-function attack

- What happened after we jumped to dummy()?
- Segfault!
- But why?  Did we do something wrong?
- Where is it trying to return to?

# Return-to-function attack

- What if we duplicate the return address?

# Return-to-function attack

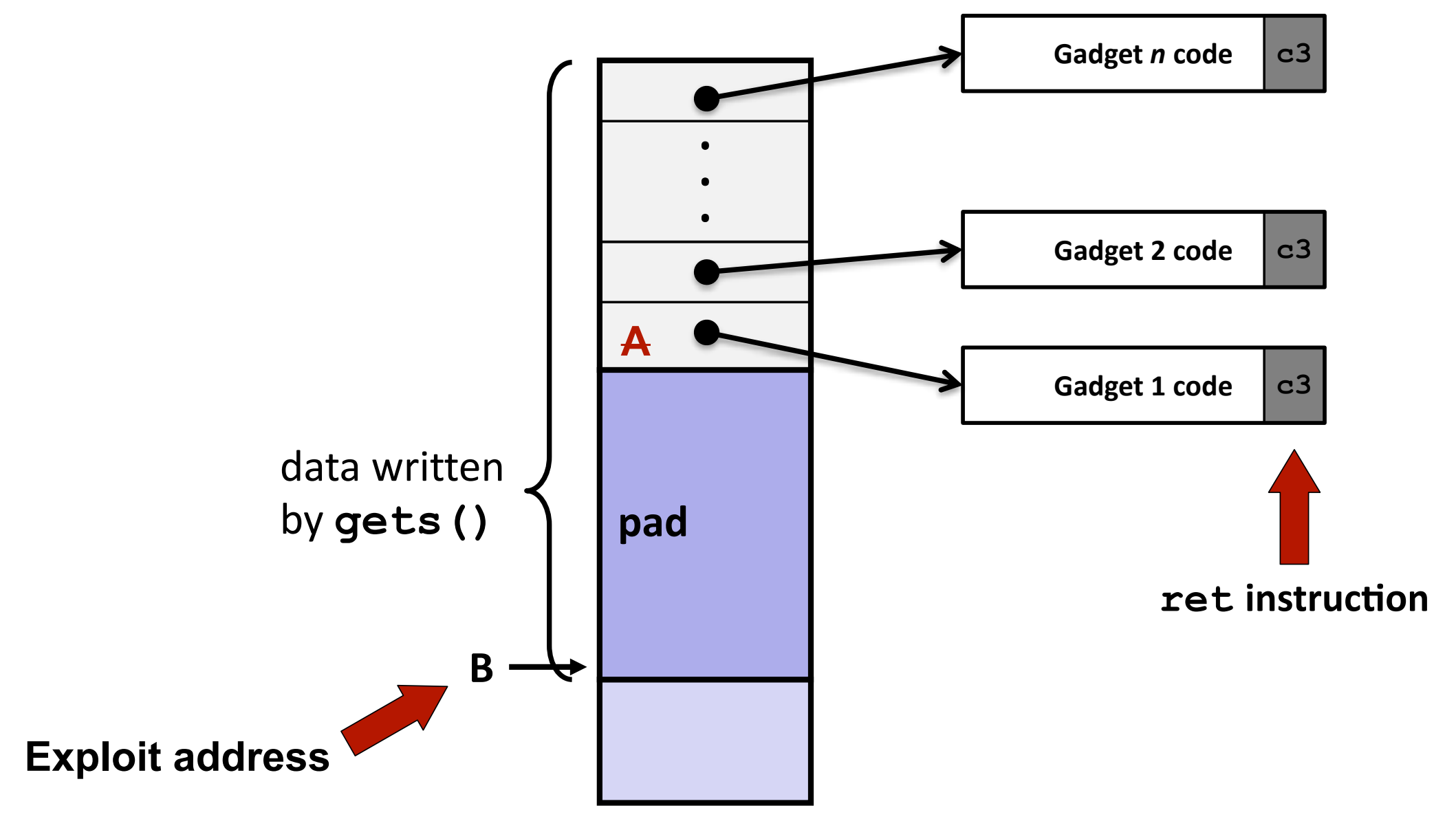
- What if we duplicate the return address?
- Calls the dummy twice!
- What if we triplicate the return address?

# Return-to-function attack

- What if we duplicate the return address?
- Calls the dummy twice!
- What if we triplicate the return address?
- Calls the dummy thrice!

# Return-to-function attack

- We now have a method of
  - calling whatever functions we want
  - as many times as we want
  - in whatever order we want
- Let’s call any of these chunks of code we jump to a *gadget*


Gadget n code
c3
Gadget 2 code
c3
Gadget 1 code
c3
A
pad
data written
by gets()
B
Exploit address
ret instruction

# Return-oriented programming

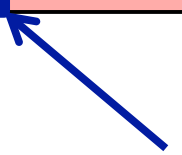
- But can we do better than calling entire functions?

# Observation #1

- We don’t actually need to jump to the beginning of a function
- We just need to eventually hit a `ret` instruction to keep going along the chain

```
long ab_plus_c(long a, long b, long c) {
   return a * b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

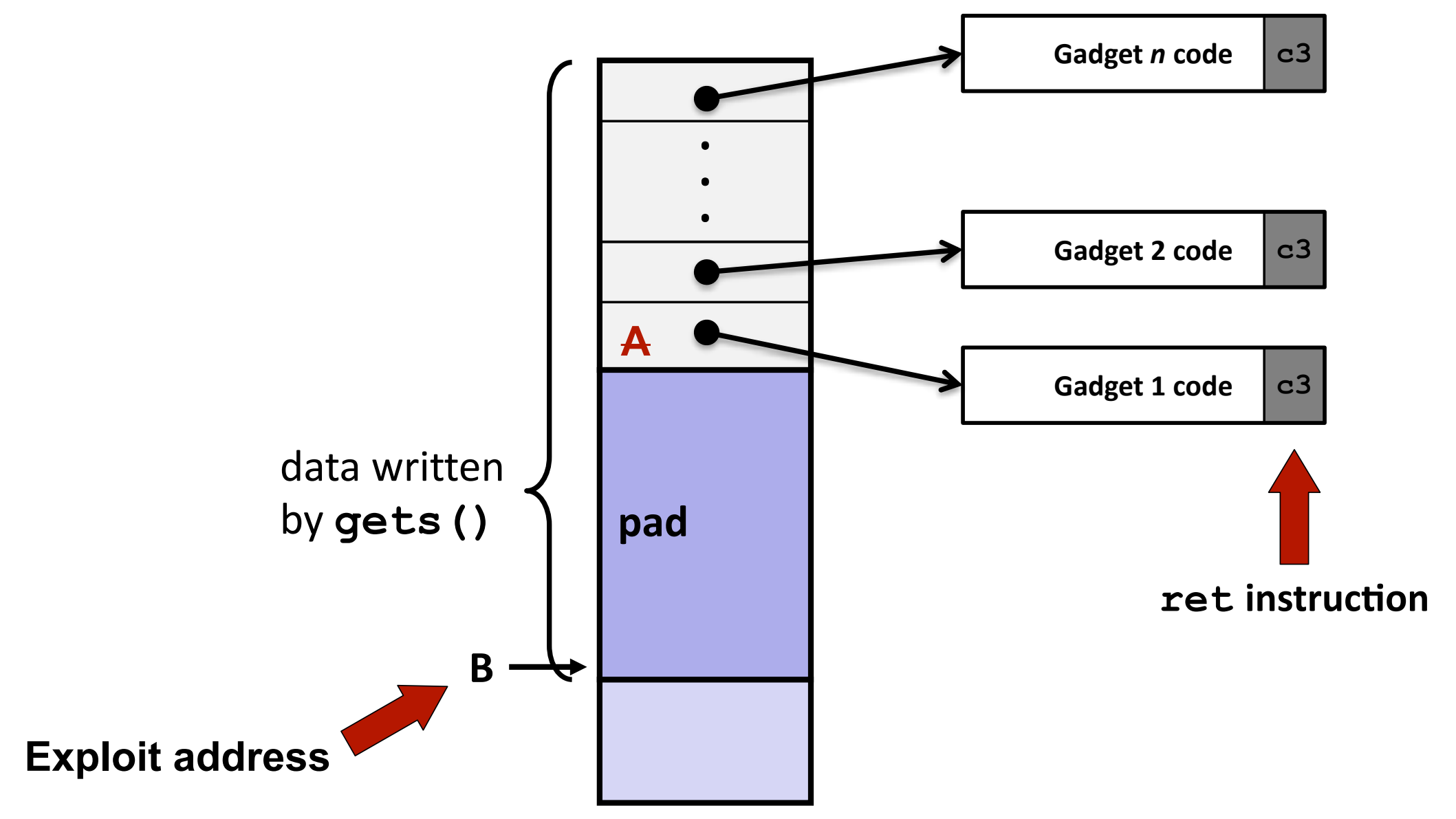


**rax ← rdi +rdx**

**Gadget address = 0x4004d4**

# Observation #1

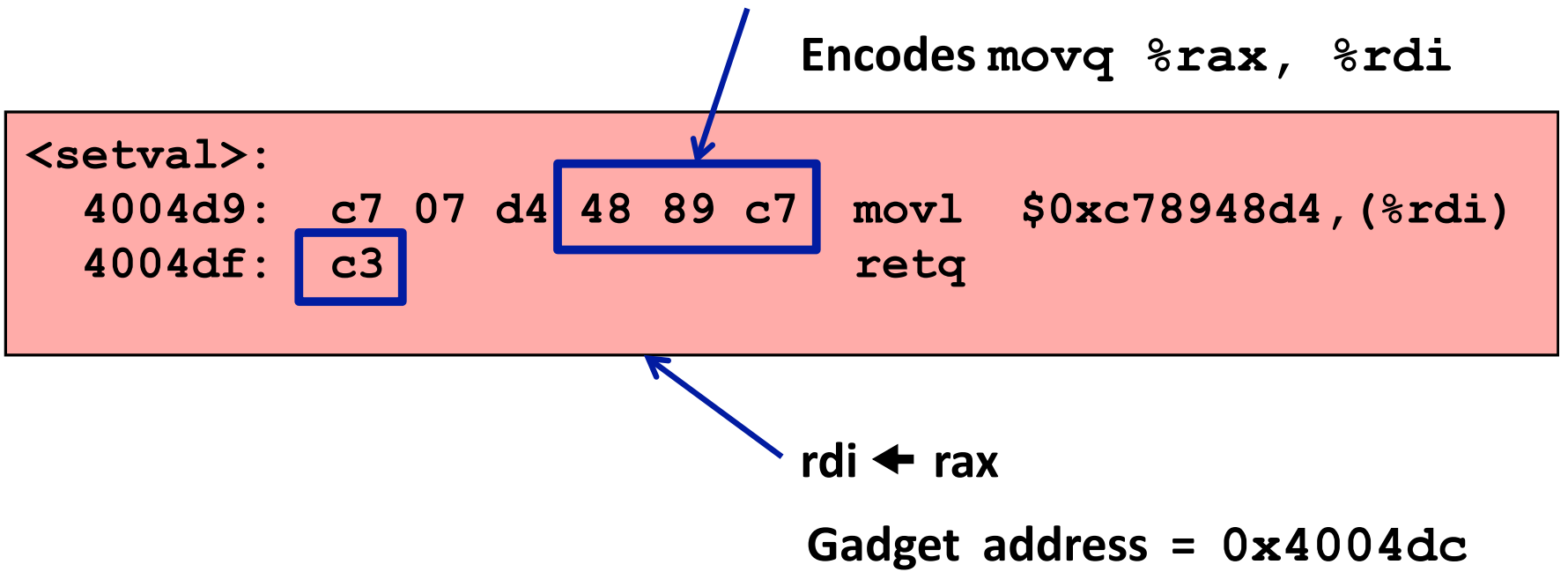
- We don’t actually need to jump to the beginning of a function
- We just need to eventually hit a `ret` instruction to keep going along the chain


Gadget n code
c3
Gadget 2 code
c3
Gadget 1 code
c3
A
pad
data written
by gets()
B
Exploit address
ret instruction

# Observation #2

- We don’t actually need to jump to the beginning of an instruction (!?)
- At least, an *originally intended* instruction!
- We can do this because x86 instructions are *variable length*

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



# Return-oriented programming

- How would we know that “48 89 c7” is an instruction?
- Lookup tables
- Lab 4 exploits #4 and #5 are ROP exploits
- Will require using an (included) lookup table

# TODO

- Assignment 4 assigned due November 10