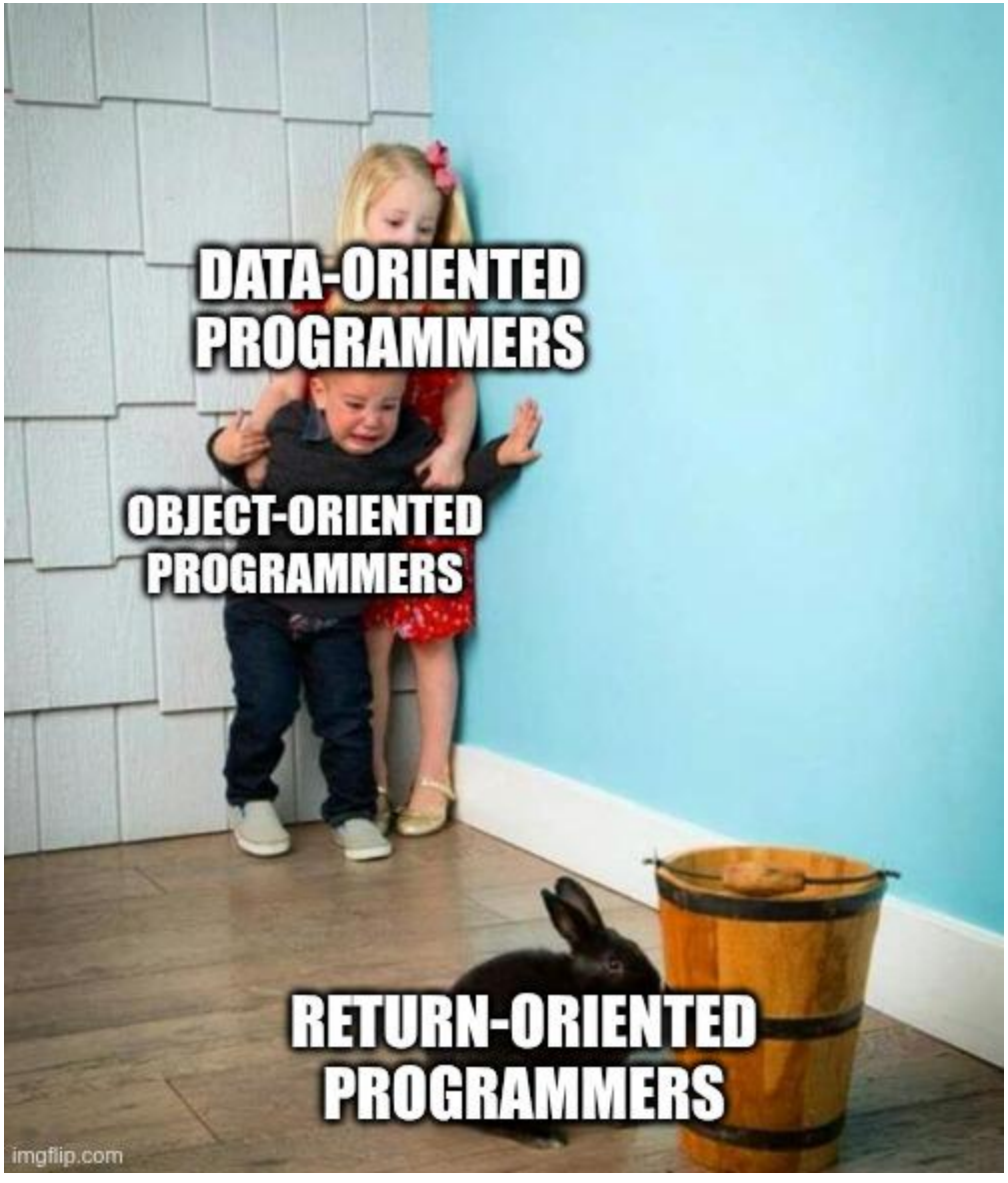

DATA-ORIENTED
PROGRAMMERS
OBJECT-ORIENTED
PROGRAMMERS
RETURN-ORIENTED
PROGRAMMERS
imgflip.com

TODO

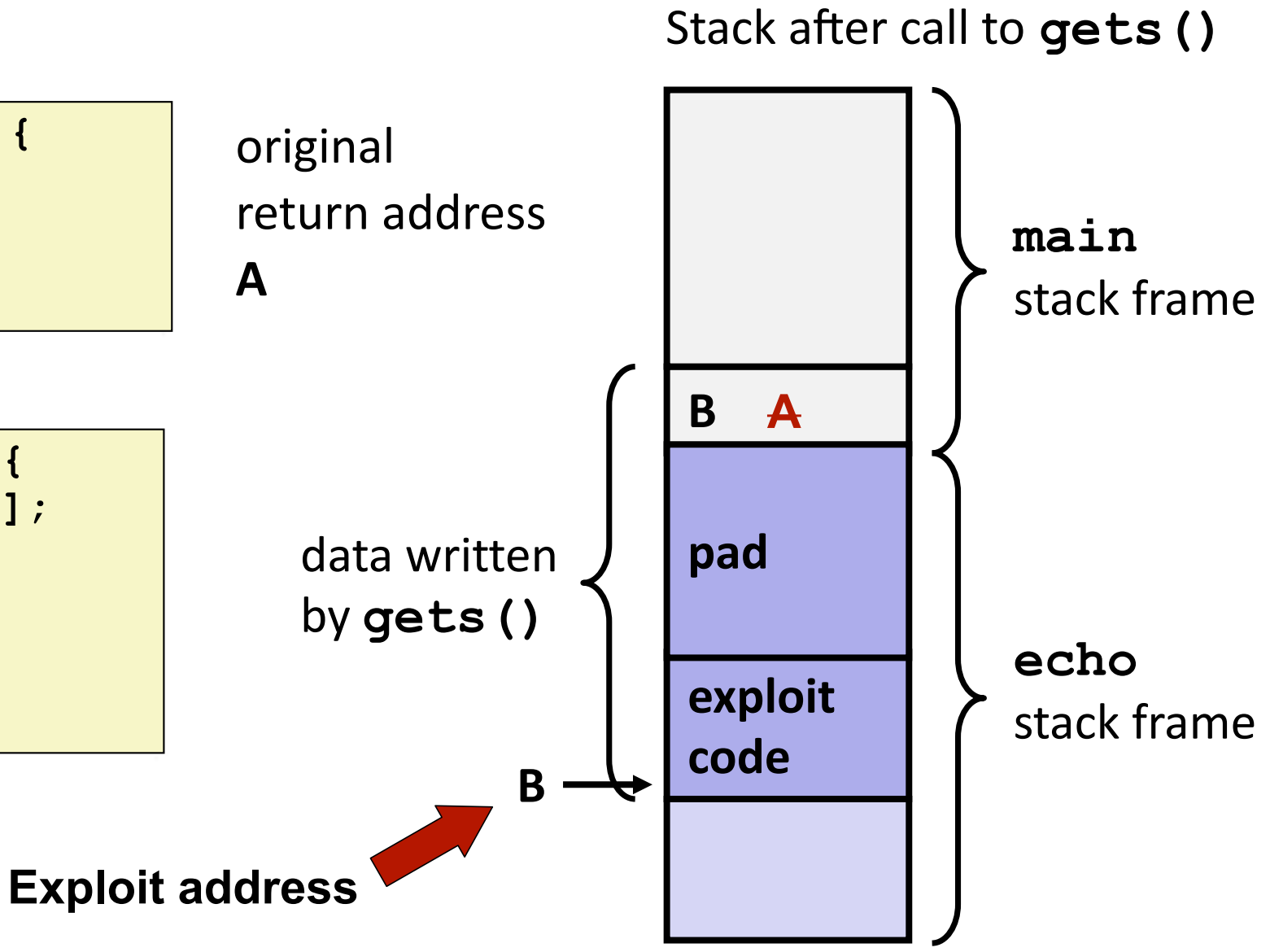
- Assignment 4 assigned due November 10
- Study for Exam 2 November 6

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;
}
```

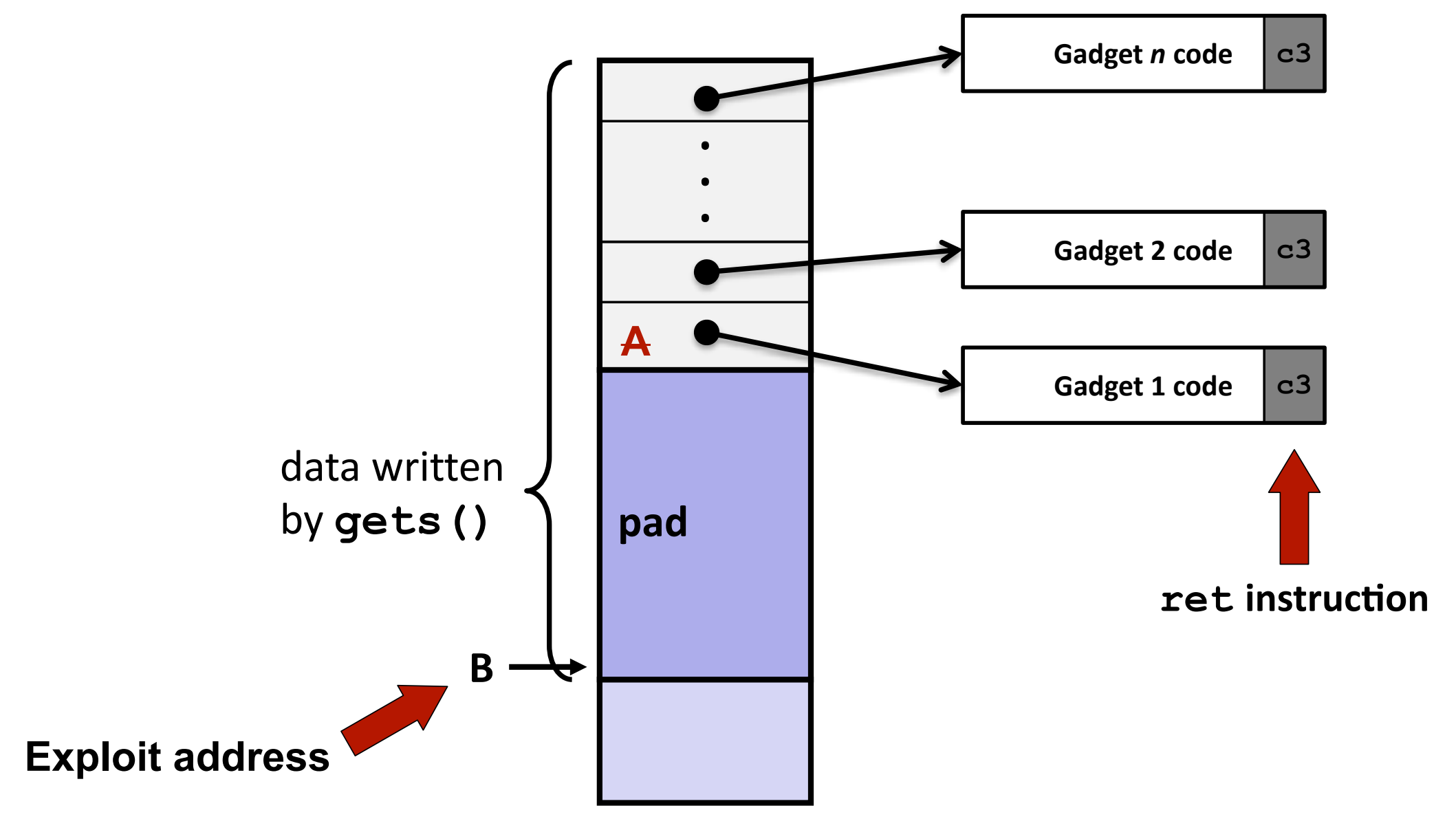


Goal

- Print “never print”

New threat model

- Now 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?


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

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*

Return-to-function attack

- Why does “returning” to the address of a function move the stack pointer up by 8 bytes each “call”?
- Why not 16? Why does `%rsp` change at all?

Return-oriented programming

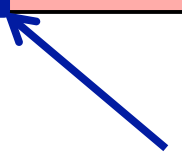
- OK so we can call a bunch of functions, but can we do better than calling *entire* functions?

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
```

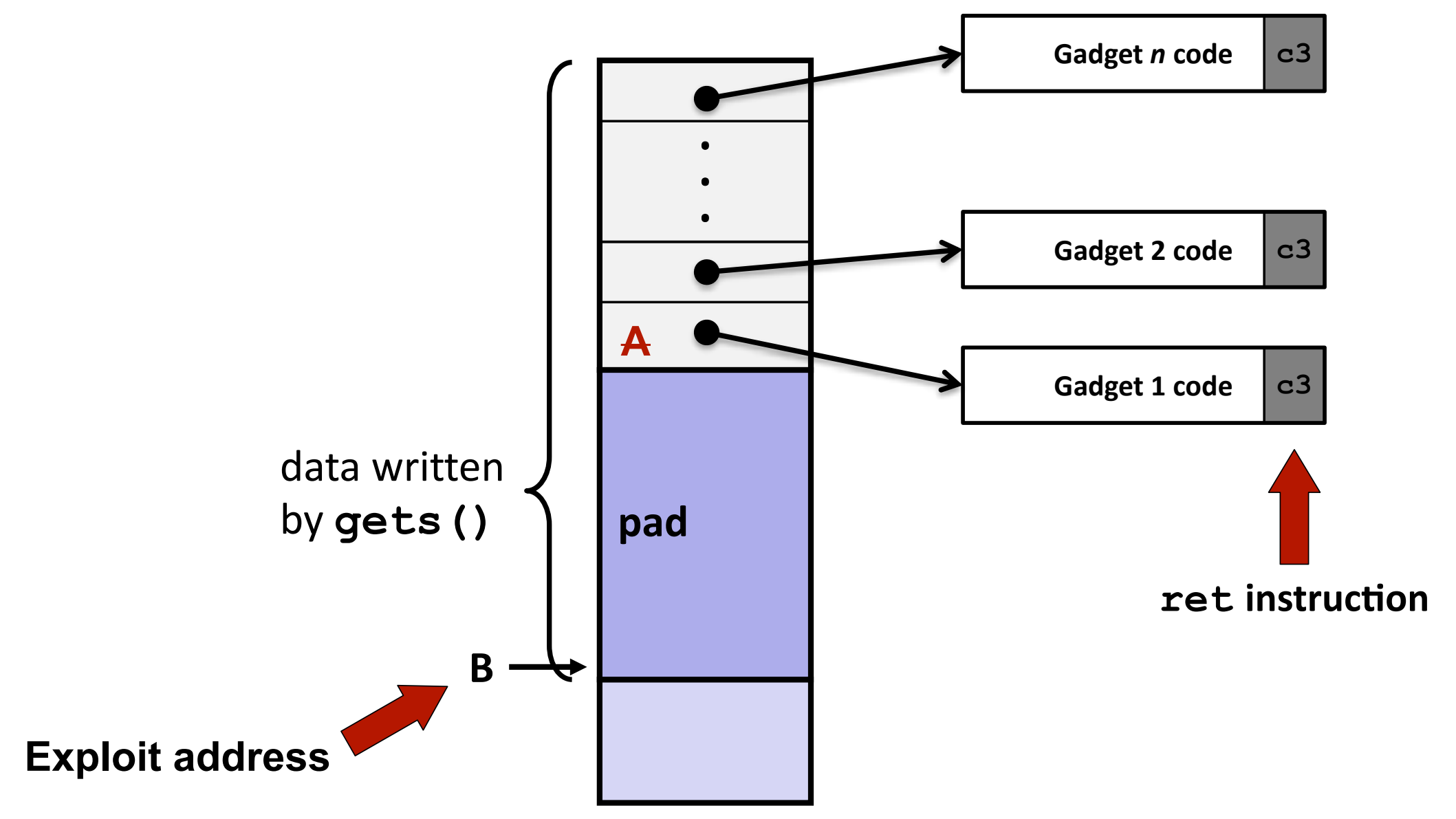


rax ← rdi +rdx

Gadget address = 0x4004d4

Observation #1

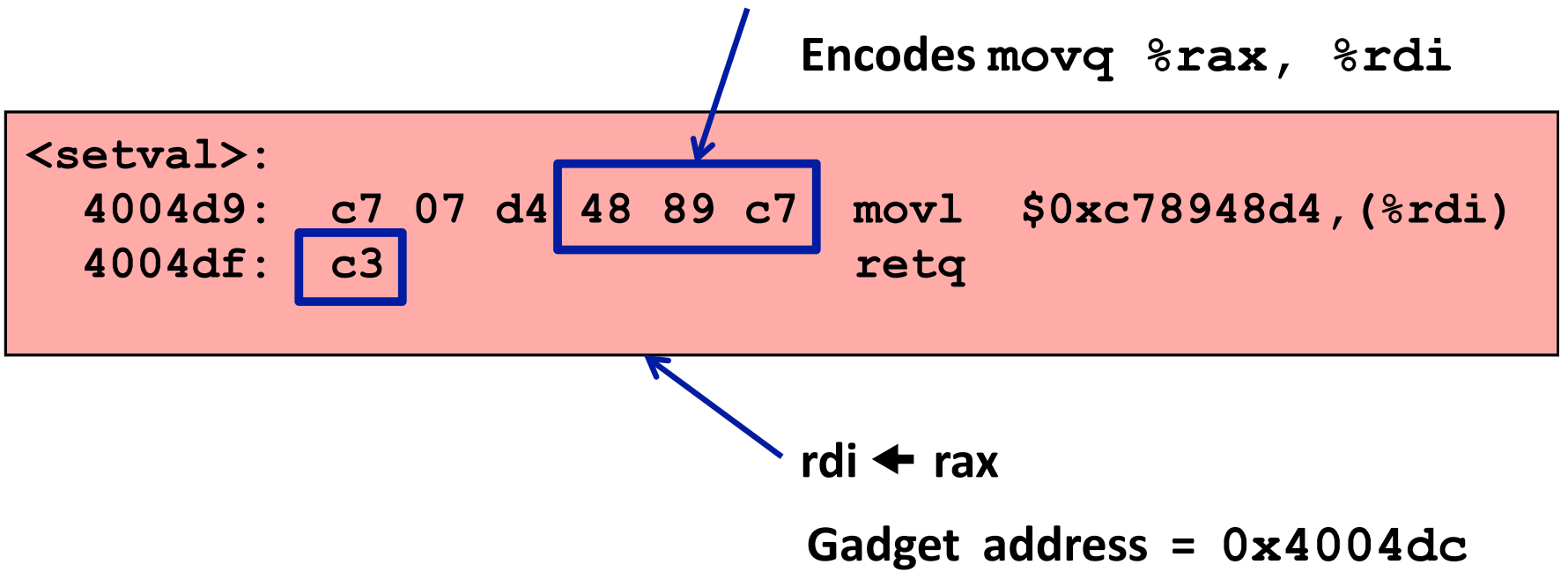
- We don’t actually need to jump to the beginning of a function
- We just need to eventually hit a `ret` instruction to keep going along the chain


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;
}
```



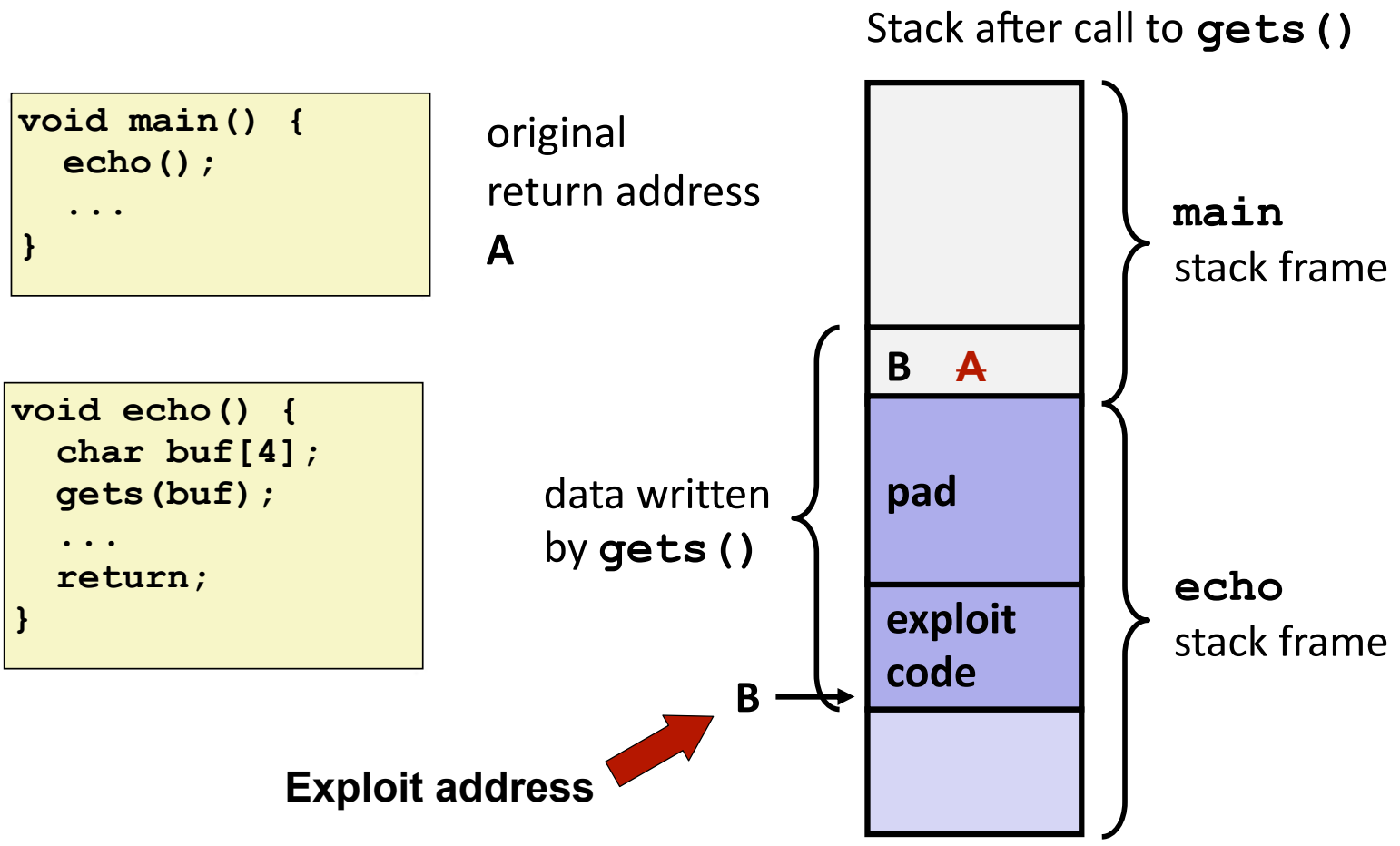
Return-oriented programming

- 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

Overflow exercises

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

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



Real world example

- Buffers are all around us!
- Let’s look at an example of arbitrary code execution from
  Animal Crossing
- Ran on a Nintendo GameCube (PowerPC)


Your Funds
4,774 Bells
Tom Nook
.....Now, this IS regrettable!
Based on the notification I've
received directly from


entry 15
October 06, 2026
End
ink
Caps
space
letters
punct
icons
cursor
ok
backspace

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	¡	¿	Ä	À	Á	Â	Ã	Ȧ	Ç	È	É	Ê	Ë	Ì	Í	Î
1	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Ø	Ù	Ú	Û	Ü	ß	Þ	à
2		!	"	á	â	%	&	'	(	)	~	♥	,	-	.	♪
3	0	1	2	3	4	5	6	7	8	9	:	💧	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	ă	⁘	ä	ȧ	_
6	ç	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	è	é	ê	ë	
8		ì	í	î	ï	·	ð	ñ	ò	ó	ô	ő	ö	ø	ù	ú
9	‾	û	ü	ý	ÿ	þ	Ý	¡	§	ª	º	‖	µ	³	²	¹
A	‾	¬	Æ	æ	„	》	《	☼	☁	☂	☾	☃	≥	≤	/	∞
B	○	✕	□	△	+	⚡	☃	♀	☘	★	💀	😮	😂	😢	😠	😈
C	x	÷														
D	;	#				“	„	‘	,	Œ	œ	e	er	re	\	
E																
F																

Outcome

- Run arbitrary code in game
- **Jailbreak for the console**


STOP HITTING YOURSELF

- Buffer are all around us
- Every time you enter input into something, it’s going into a buffer
- And probably copied between a few different buffers until its final destination

- If the machine we are attacking is remote
  - E.g., a web site
- We are running arbitrary code on that machine
- Hence remote code execution

Buffer overflow conclusion

- Other RCE-exploitable memory bugs:
  - Double free:
    - `free(p); ...; free(p);`
  - Format string:
    - `printf(user_input);`
  - Integer overflows/underflows:
    - `malloc(nearing_overflow * sizeof(int));`
  - Dangling pointers:
    - `free(p); ...; p[...] = ...;`

- You can use these attacks to cause harm
- But there will always be people who want to restrict what we can do, what we can read, what can say
  - Device jails also have buffers
  - So does DRM
  - So do firewalls



Structure of course

- First third (Exam 1):
  - Data Representation
  - Memory
- Second third (Exam 2):
  - Machine Code
  - Memory exploits
- Final third (Final exam):
  - Memory hierarchy ← here
  - Processes

Looking ahead to Exam 2

- We have now covered all material on it
- Mostly on x86-64
- Light on buffer overflow material (this week)
  - But be prepared to answer some true/false questions on this topic
- No GDB commands
- Can bring a 3x5 index card

TODO

- Assignment 4 assigned due November 10
- Study for Exam 2 November 6