CSCI 2330

When you're learning coding and you finally understand why the website is called stack overflow



TODO

- Assignment 3 assigned due October 28

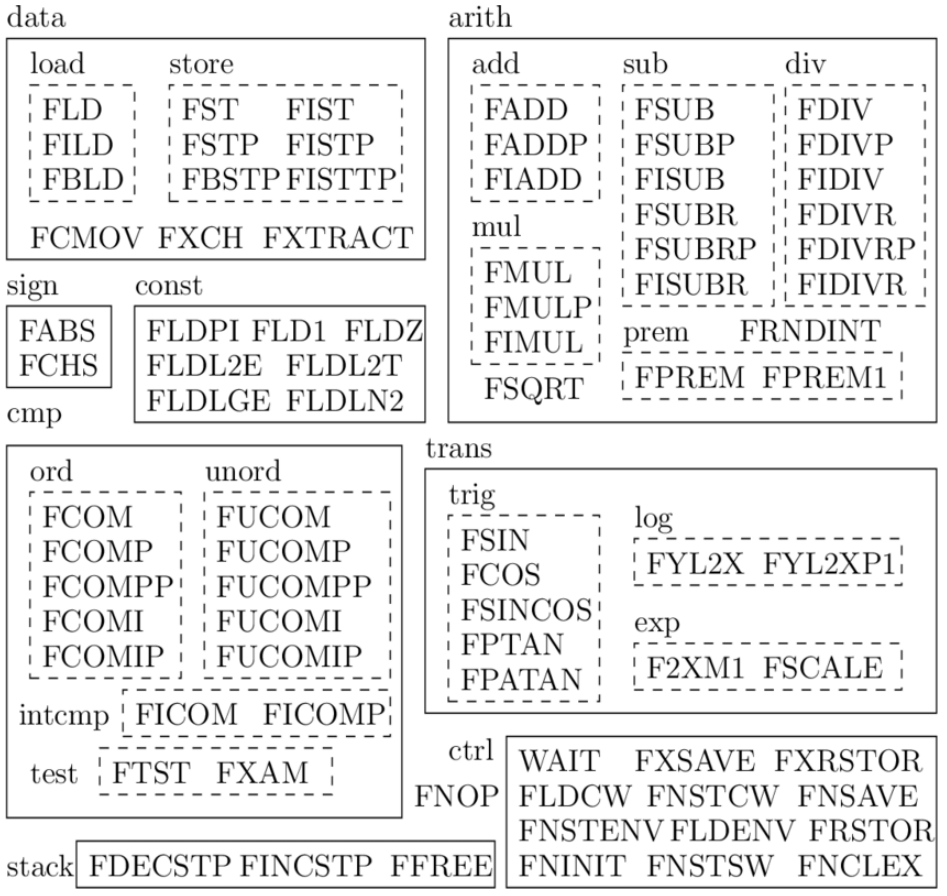
Overview

- Buffer overflows


intel®
i387™SX™
N80387SX
(16-25MHZ)
C2200508 A1
Ⓜ© 1986

Floating point (x87)

- Uses special registers
- And special instructions



What is a security vulnerability?

- A type of bug
- We may have accidentally written some security vulnerabilities in this class!
- But not all bugs are security vulnerabilities…

What is a security vulnerability?

- A type of bug where adversarial input can cause a program to engage in unintended behavior
- The “input” might be input from a Web form or a file
- Or something more subtle, like doing something with a certain timing or in a certain order

What is a security exploit?

- Such an adversarial input that causes a program to engage in unintended behavior

Why learn security exploits?

- You learn how to defend by learning how to attack
  - Learning how to write secure code is important!

Why learn security exploits?

- You are forced to understand how the machine works
  - It forces you to master the material!

Why learn security exploits?

- So that you can use them! :)

!?!?

- Wait, I thought I wasn’t supposed to do that!?
- Well, whoever told you you weren’t supposed to use them was probably someone who didn’t want you to
- It was probably someone in power

Who has power?

- Sometimes those in power shouldn’t be

Inversion of power structures

- Security vulnerabilities present an inversion of traditional power structures
- They bring power to those who exploit them – the clever and the well-motivated
- Instead of some trillion dollar tech company or a government or the head of a bureaucracy

Inversion of power structures

- We need more jailbreaks to take back control of our own devices
- More firewall evasion techniques to evade government censorship across the globe
- More whistleblowing to reveal corporate or government wrongdoing

- Inducing others’ computers to do whatever you want them to do might be the most important thing you learn in this degree program

Rules for hacking

- Only hack in controlled environments

Rules for hacking

- ~~Only hack in controlled environments~~
- Don’t be an asshole

Rules for hacking

- ~~Only hack in controlled environments~~
- Don’t be an asshole
- Don’t hurt people

Rules for hacking

- ~~Only hack in controlled environments~~
- Don’t be an asshole
- Don’t hurt people
- Make the world a better place

Rules for hacking

- ~~Only hack in controlled environments~~
- Don’t be an asshole
- Don’t hurt people
- Make the world a better place
- Try not to get into trouble

Remote Code Execution (RCE)

- We will be learning about a RCE vulnerability
- Remember: a vulnerability allows us to cause a program to engage in unintended behavior
- For an RCE vulnerability, that unintended behavior is running whatever code we want to on someone else's machine
- It is one of the most powerful types of vulnerability

Memory Safety Vulnerability

- When a memory bug in your code can induce vulnerabilities

Buffer overflow vulnerabilities

- A *buffer* is an array of binary data
- A *buffer overflow* is when a program writes past the end of a buffer
- Specifically, we will learn about stack smashing – an overflow of a stack buffer
- A RCE vulnerability that arises when one overflows the buffer of a *stack*

INSECURE.ORG

Nmap.org Npcap.com Seclists.org Sectools.org

Site Search

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

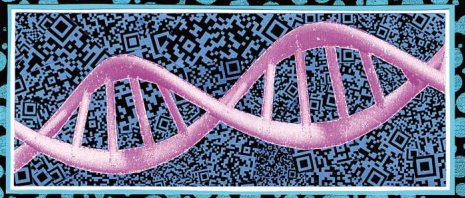
BugTraq, r00t, and Underground.Org
bring you

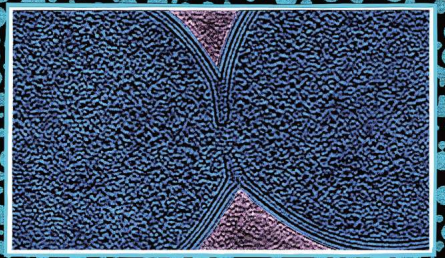
Smashing The Stack For Fun And Profit

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine. Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address. This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.

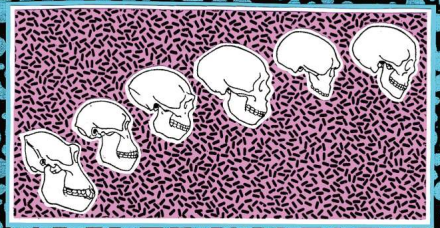
..?RFCK!





ISSUE 72

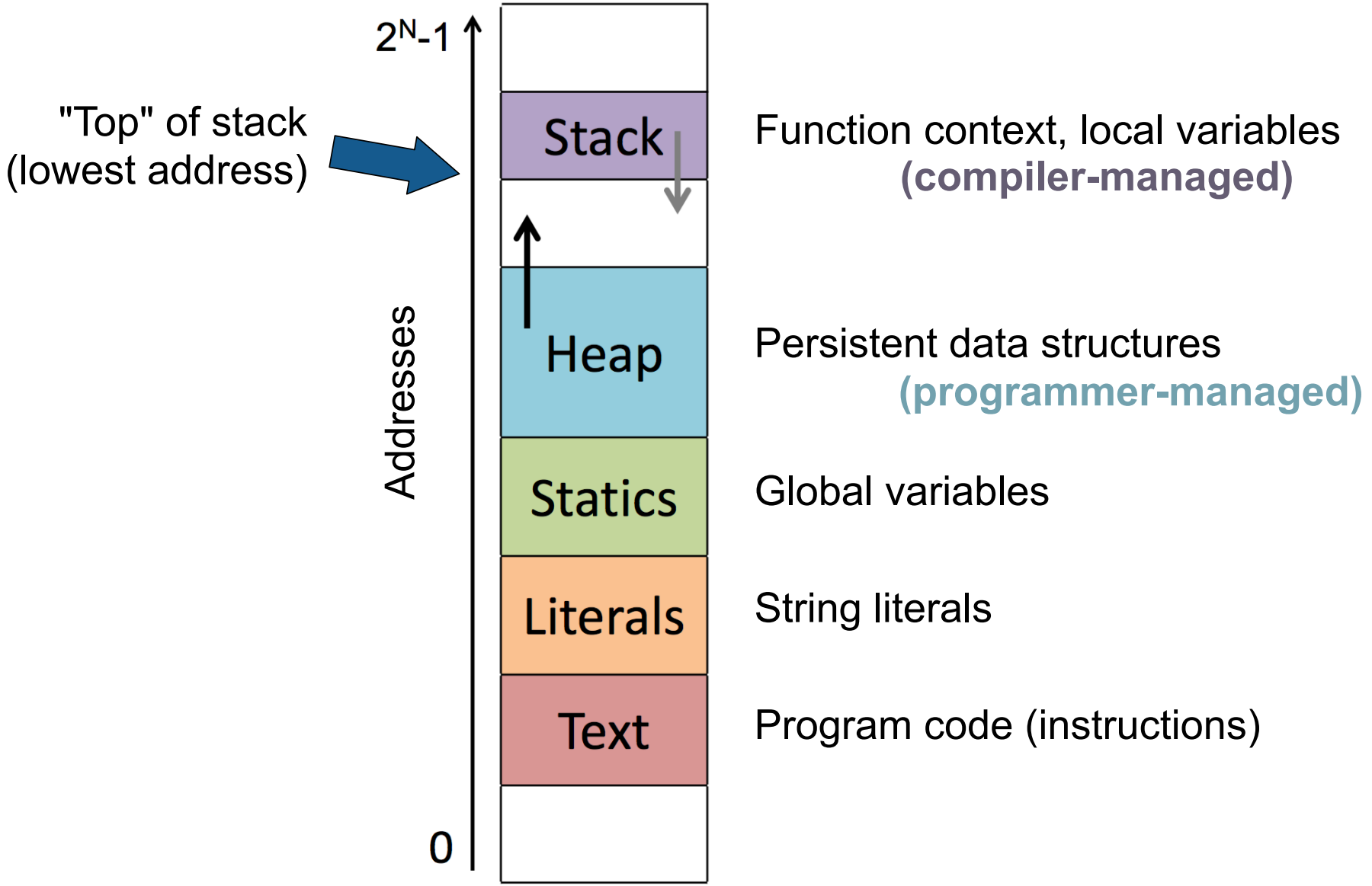
2025



40TH ANNIVERSARY EDITION

Terminology clarification

- Buffer – an array of binary data
- Buffer overflow – when you write past the end of that buffer
- Stack smashing – buffer overflow on the stack
- Stack overflow – **not** the same thing as stack smashing
  - Stack overflow is when the stack grows past its limit (often as a result of infinite recursion)


2N-1
"Top" of stack
(lowest address)
Addresses
Stack
Heap
Statics
Literals
Text
0
Function context, local variables
(compiler-managed)
Persistent data structures
(programmer-managed)
Global variables
String literals
Program code (instructions)

Why do buffer overflows exist?

- `int a[4]; // the array a.k.a. the buffer`
- `a[7] = 5; // C doesn't check bounds`
  `// (unlike Java)`

Overflowing buffers

- `int a[4]; // address 0x1000`
- `a[7] = 5;`
- This is technically undefined behavior
- So *anything* could happen
- Most likely you will write a 5 at address 0x1000 + 7 * sizeof(int)

A vulnerable C function

- Let’s look up the man pages for `gets()`…

Example usage

- `char dest[100]; // buffer of 100 bytes`
- `gets(dest);`

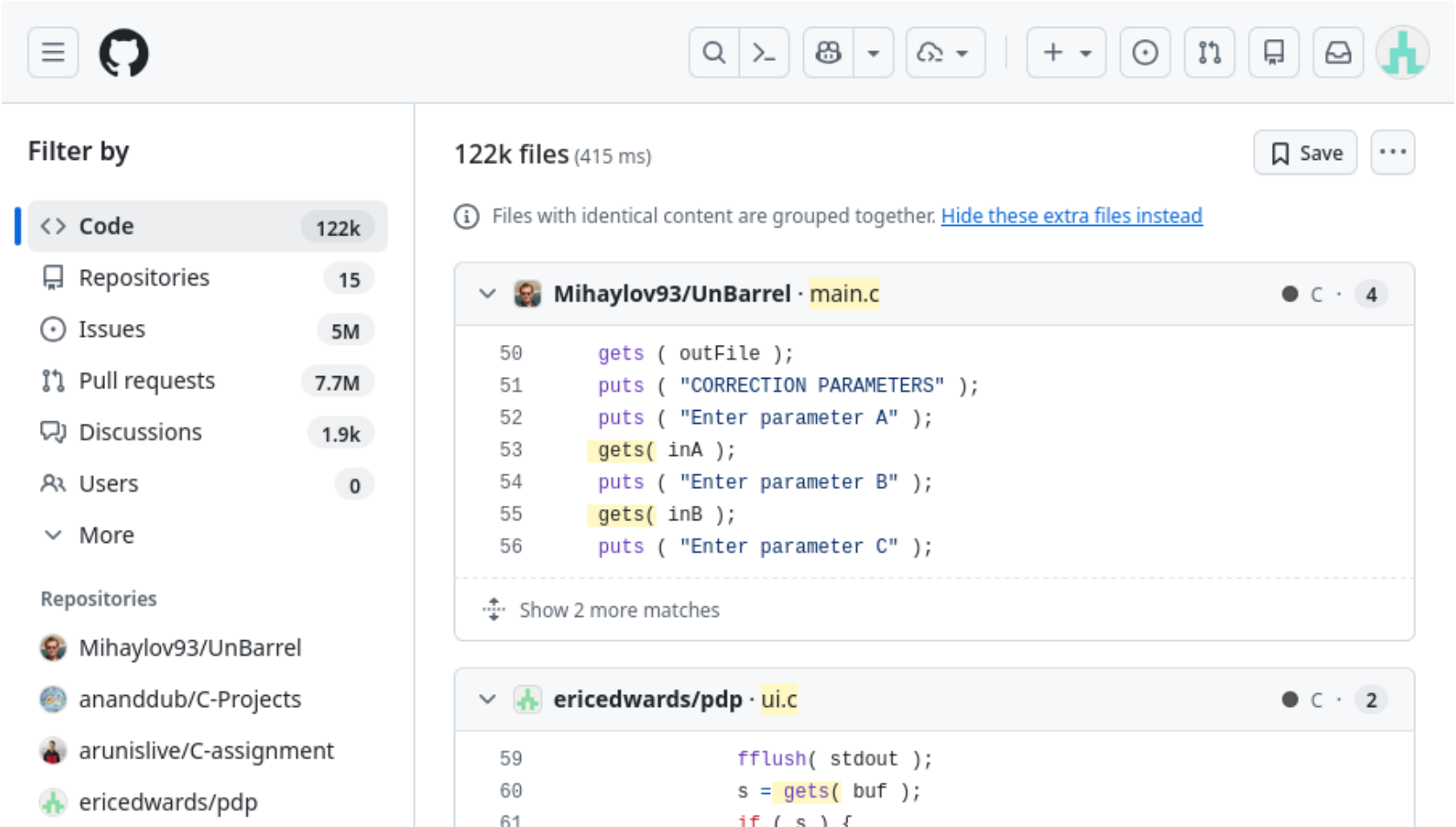
Implementation

```
/* Get string input, store to str */
char *gets(char *str) {
    int c = getchar();
    char *p = str;
    while (c != '\n' && c != EOF) {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return str;
}
```



end-of-file
(input) marker

Lots of vulnerable code



Filter by

- Code 122k
- Repositories 15
- Issues 5M
- Pull requests 7.7M
- Discussions 1.9k
- Users 0
- More

Repositories

- Mihaylov93/UnBarrel
- ananddub/C-Projects
- arunislive/C-assignment
- ericedwards/pdp

122k files (415 ms) Save ...

Files with identical content are grouped together. Hide these extra files instead

Mihaylov93/UnBarrel · main.c — C · 4

```
50      gets ( outFile );
51      puts ( "CORRECTION PARAMETERS" );
52      puts ( "Enter parameter A" );
53      gets( inA );
54      puts ( "Enter parameter B" );
55      gets( inB );
56      puts ( "Enter parameter C" );
```

Show 2 more matches

ericedwards/pdp · ui.c — C · 2

```
59                  fflush( stdout );
60                  s = gets( buf );
61                  if ( s ) {
```

Buffer overflows

- These issues are serious and widespread
- But when do I get to control other people’s machines?

buftest

- What causes this program to segfault?
  - What instruction specifically?
- Let’s get gdb to help!

Calling and returning

- `callq` pushes the return address to the stack
- `ret` pops it and jumps to it

Calling and returning

- `callq`:
  - `subq $8, %rsp`
  - `movq %rip + length_of_callq_instruction, (%rsp)`
  - `jmp target`
- `ret`:
  - `jmp (%rsp)`
  - `addq $8, %rsp`

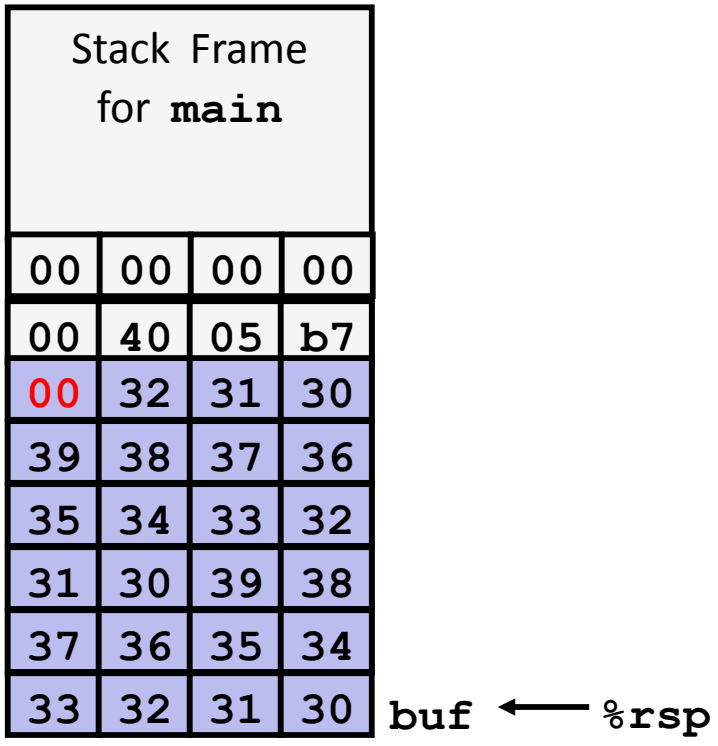
buftest

- What is the shortest input that causes this program to segfault?
  - Let’s count!
  - 0123456789…

```
(gdb) disass echo
Dump of assembler code for function echo:
   0x0000000000400590 <+0>:     sub    $0x18,%rsp
   0x0000000000400594 <+4>:     mov    %rsp,%rdi
   0x0000000000400597 <+7>:     call   0x400480 <gets@plt>
   0x000000000040059c <+12>:    mov    %rsp,%rdi
   0x000000000040059f <+15>:    call   0x400450 <puts@plt>
   0x00000000004005a4 <+20>:    add    $0x18,%rsp
   0x00000000004005a8 <+24>:    ret
End of assembler dump.
```

```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004005a9 <+0>:     sub    $0x8,%rsp
   0x00000000004005ad <+4>:     mov    $0x0,%eax
   0x00000000004005b2 <+9>:     call   0x400590 <echo>
   0x00000000004005b7 <+14>:    mov    $0x40067e,%edi
   0x00000000004005bc <+19>:    call   0x400450 <puts@plt>
   0x00000000004005c1 <+24>:    mov    $0x0,%eax
   0x00000000004005c6 <+29>:    add    $0x8,%rsp
   0x00000000004005ca <+33>:    ret
End of assembler dump
```

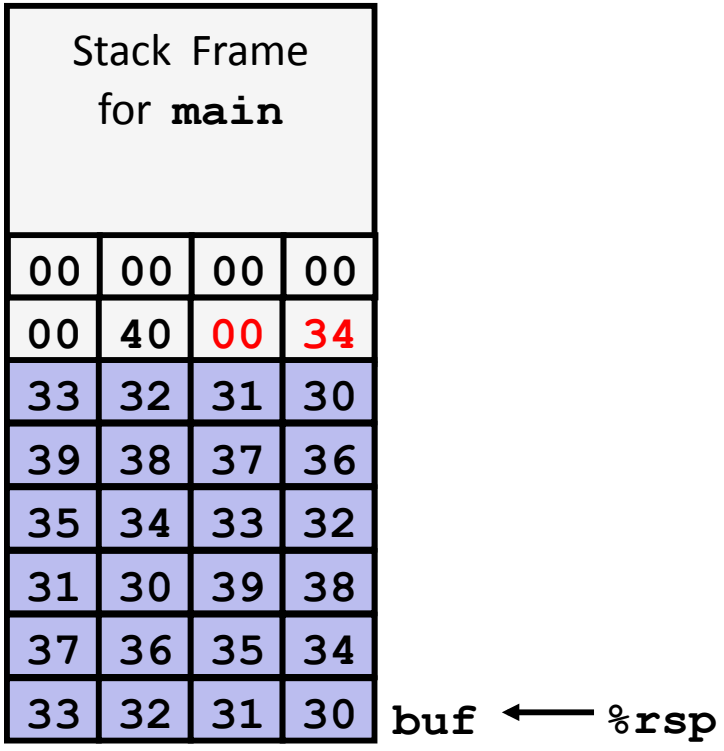
After call to gets




```
unix>./buftest
Type a string:0123456789012345678901
0123456789012345678901
```

Overflowed but did not corrupt state

After call to gets




```
unix>./buftest
Type a string:012345678901234567890123
Segmentation Fault
```

Overflowed and corrupted return pointer

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Recap

- We can overwrite a return address
- The ret instruction segfaults
- We can get another machine to crash!
- But when do I get to control other people’s machines?

Overwriting the return address

- If we overwrite the return address with an *invalid* address of code…
  - Our program crashes
- What else might overwrite the return address with?

Overwriting the return address

- If we overwrite the return address with an *invalid* address of code…
  - Our program crashes
- What else might overwrite the return address with?
  - A *valid* instruction address

Can we call this?

```
// a function that never runs
void dummy() {
    puts("This message will never print");
}
```

```
(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.
```

We need to write an exploit string

- How?
- Padding bytes!
- But how many?
- Let’s try it in gdb…

Problem

- How do we write 0x00 0x00 0x00 0x00 0x00 0x40 0x05 0x7d?

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Problem

- How do we write 0x00 0x00 0x00 0x00 0x00 0x40 0x05 0x7d?
- So '@' → 0x40, '}' → 0x7d
- But how do we input 0x00? 0x05?

Problem

- How do we write 0x00 0x00 0x00 0x00 0x00 0x40 0x05 0x7d?
- So '@' → 0x40, '}' → 0x7d
- But how do we input 0x00? 0x05?
- ASCII control chords 🥲
  - `ctrl+v ctrl+space` → 0x00
  - `ctrl+v ctrl+e` → 0x05

Problem

- ASCII control chords still don’t allow us to type *any* byte
- They are painful to enter
- Difficult to memorize
- Instead, we will use `hex2raw`, a program that helps us output the raw bytes
- Shell redirection will be helpful…

Shell redirection

- `$ somecommand > output.file`
  - Runs somecommand, redirecting the output from stdout that would normally be printed to output.file
- `$ somecommand < input.file`
  - Runs somecommand, redirecting the input from stdin that would normally be inputted using the keyboard
- `$ somecommand < input.file > output.file`
  - Can be combined

Problem

- ASCII control chords still don’t allow us to type *any* byte
- They are painful to enter
- Difficult to memorize
- Instead, we will use `hex2raw`, a program that helps us output the raw bytes
- Let’s try it in gdb…

Problem #2

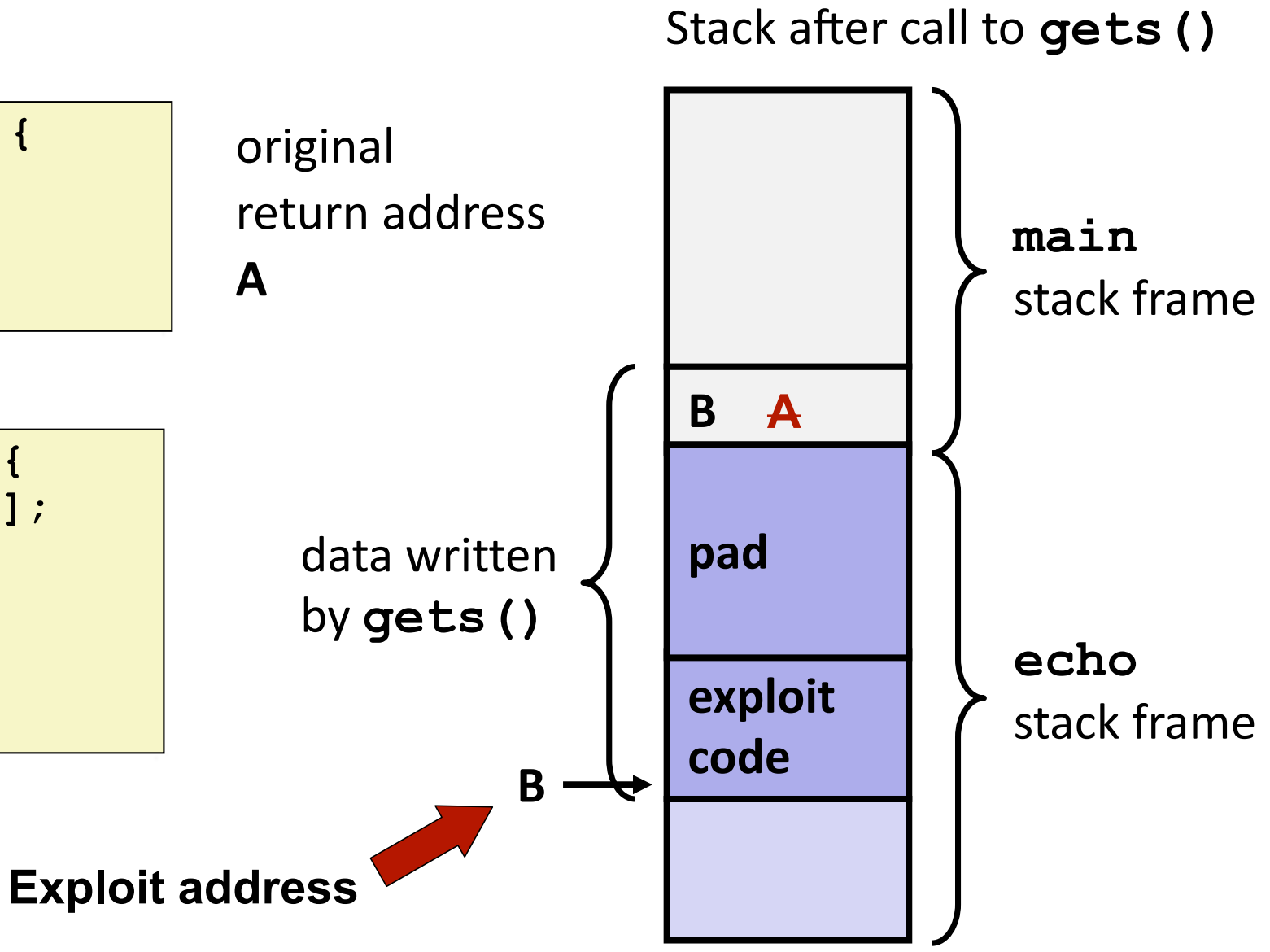
- Endianness 🥲
- What needs byte swapping?
- Do we need to move the padding?

Where else to “return”?

- We can jump to an arbitrary address
- We can jump to an existing function
- What if we want to run arbitrary code?
  - Where could we write this?
  - And where would we jump?

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

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



Exploit code

- Often called *shell code*
  - Due to the idea that attackers wrote just enough to get a remote shell
- Machine code (raw bytes)
  - Not assembly! Not C!

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

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

... etc ...

Don't make any mistakes...

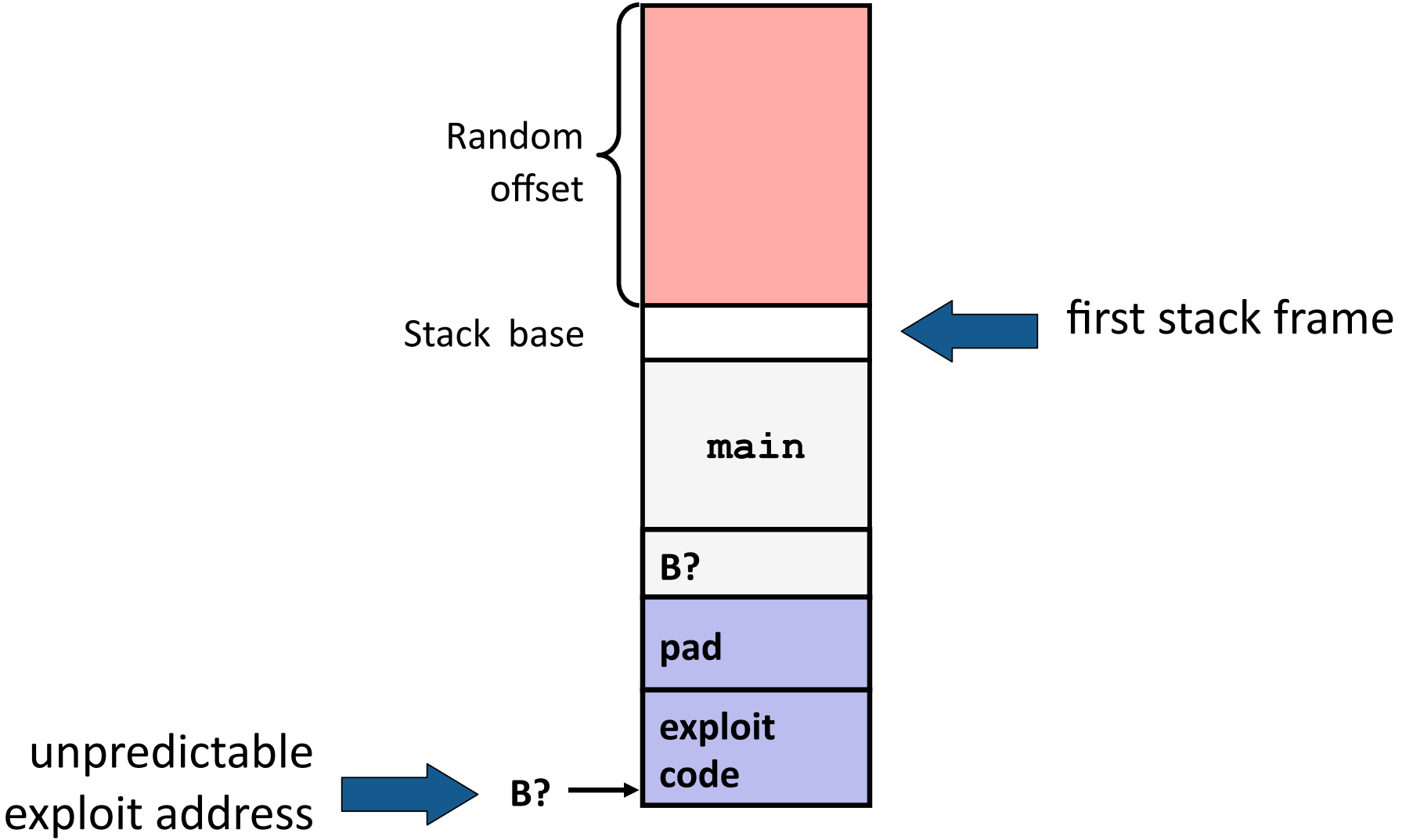
```
/* Echo Line */
void echo() {
    char buf[64];
    fgets(buf, 64, stdin); // safe
    puts(buf);
}
```

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


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

Address space layout randomization

- gdb disables stack randomization for our sanity, but it has been otherwise present this entire time
- Let’s look as `aslrtest`…

TODO

- Assignment 3 assigned due October 28