Lecture 11: Security

January 25, 2020 Chris Stone

Lab 3 (Bomb) Due 1:15pm Friday

Lab 4 (Attack) Starts 1:15pm Friday

Take-Home Midterm available by 5pm Friday Afternoon
(75-minute exam due 5pm the following Friday)

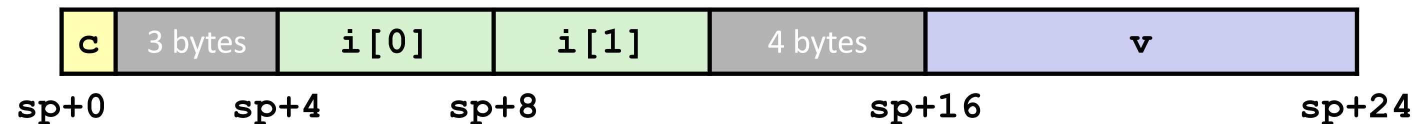
Recommended textbook reading (ASAP): 3.9, 3.10
Good review problems: 3.46-3.48
Textbook preview for Thursday: Review through 3.10

But first: Unions!

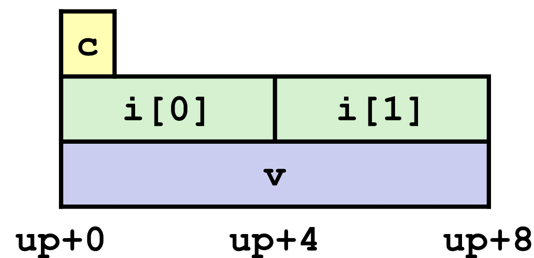
Unions

Like structs, but all the fields all overlap.

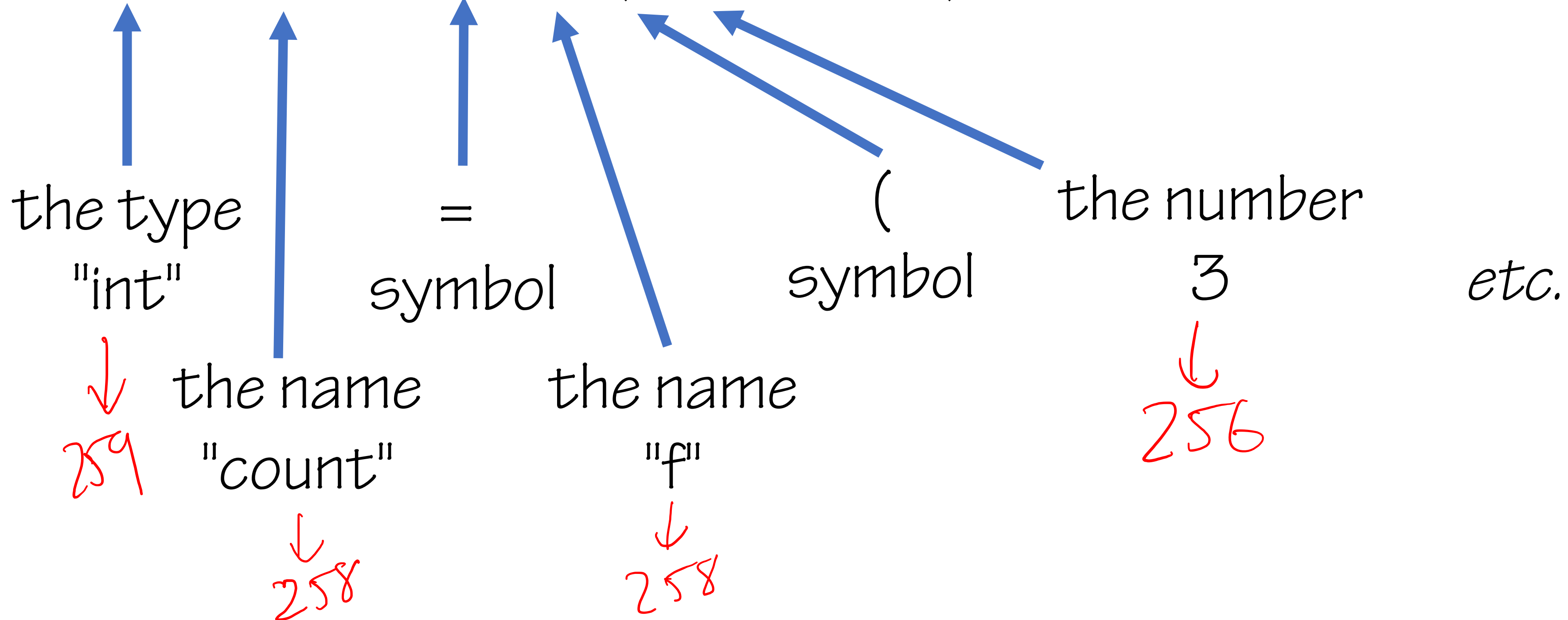
```
struct {
  char c;
  int i[2];
  double v;
} *sp;
```




```
union {
  char c;
  int i[2];
  double v;
} *up;
```



Example: Lexing/Tokenizing Strings



Application: Classic Lexing/Tokenizing in C

```
union {
   int n;
   double d;
   char* sp;
} yylval;
```

```
int lex(inputString) {
  if (...inputString starts with an integer...) {
      yylval.n = ...the value of that integer...;
      return 256;
  } else if (...inputString starts with the = operator...) {
      return '=';
  } else if (...inputString starts with the ++ operator...) {
      return 257;
  } else if (...inputString starts with a variable name...) {
      yylval.sp = ...pointer to that variable name...;
      return 258;
  } else if (...inputString starts with a type name...) {
      yylval.sp = ...pointer to the type name...;
      return 259;
  } else ...
```

We'll see a more convincing application when we reach networking!

Using Unions To Peek at Byte Ordering

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

larger addresses⟶

Endianness Revealed!

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

dw.c[0] … dw.c[7]

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

larger addresses⟶

Sun/Internet (big-endian):

```
i[0] = 0xf0f1f2f3
l[0] = 0xf0f1f2f3f4f4f6f7
```

Intel/ARM (little-endian):

```
i[0] = 0xf3f2f1f0
l[0] = 0xf7f6f5f4f3f2f1f0
```

Memory and Memory Safety

Learning Goals

- Understand what a buffer overflow is and how it can happen
- See how the stack can be exploited to run malicious code
- Practice writing an exploit
- Discuss techniques to address buffer overflow attacks

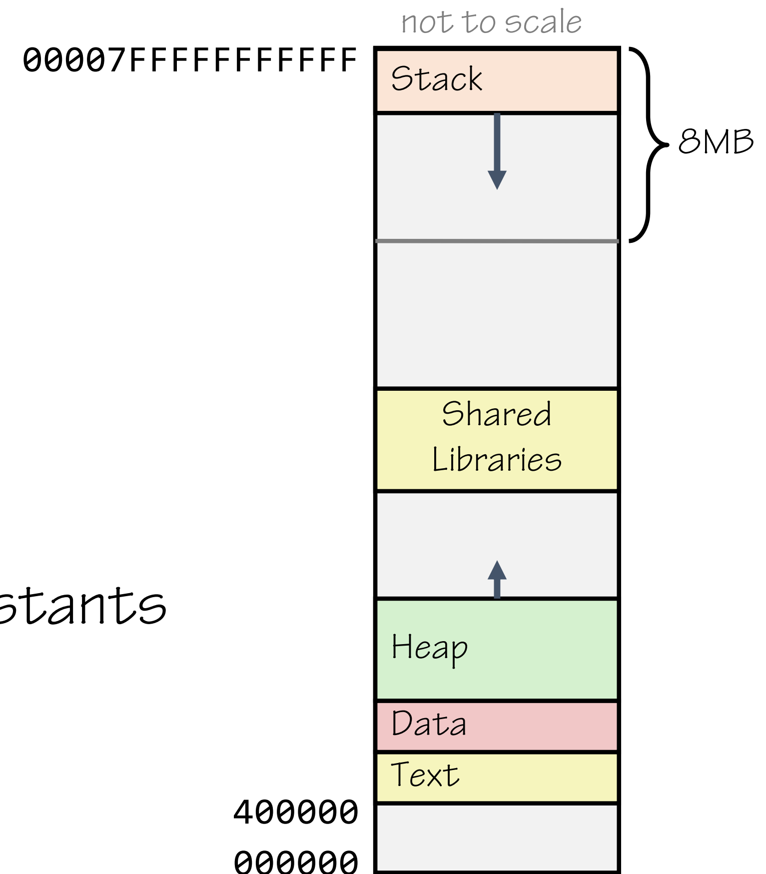
x86-64 Linux Memory Layout

Stack
- Runtime stack (8MB limit by default)
- E. g., local variables

Heap
- Dynamically allocated as needed
- When call **`malloc()`**, **`calloc()`**, **`new`**

Data
- Statically allocated data
- E.g., global vars, **`static`** vars, string constants

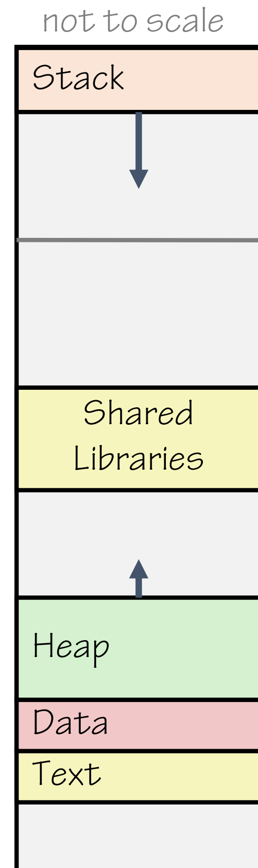
Text / Shared Libraries
- Executable machine instructions
- Read-only



Memory Allocation Example

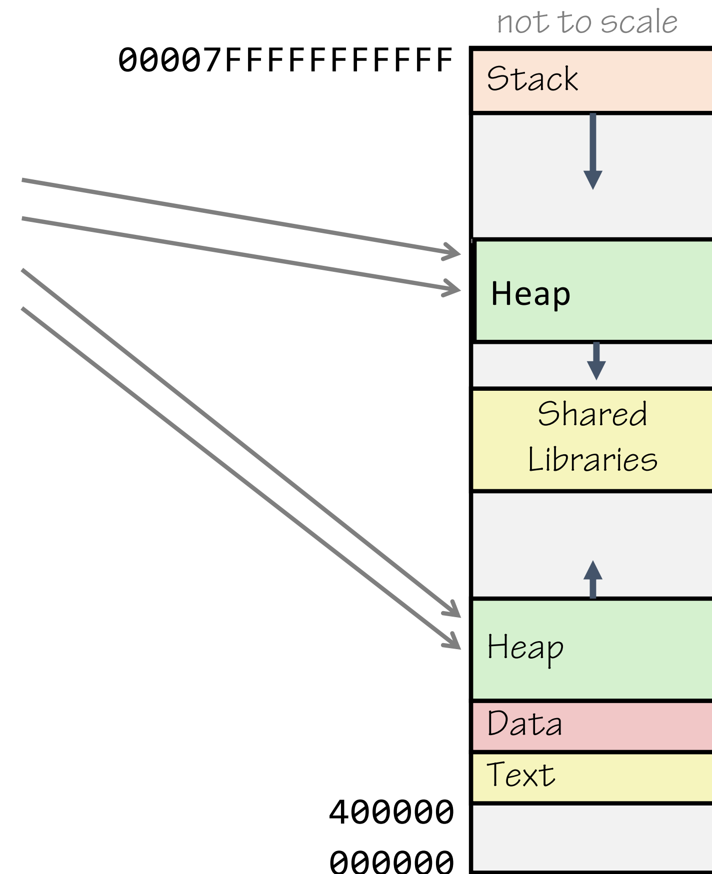
```
char big_array[1L<<24];    /* 16 MB */
char bigger_array[1L<<30]; /*  1 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    void* p1 = malloc(1L << 28); /* 256 MB */
    void* p2 = malloc(1L << 8);  /* 256  B */
    void* p3 = malloc(1L << 32); /*   4 GB */
    void* p4 = malloc(1L << 8);  /* 256  B */
    /* Some printf statements ... */
}
```

Where does everything go?



x86-64 Example Addresses

&local	0x00007ffffffe504
p1	0x00007fffe7e26010
p3	0x00007ffee7e25010
p4	0x00000000414053b0
p2	0x00000000414052a0
&big_array	0x0000000040404060
&bigger_array	0x0000000000404060
&global	0x0000000000404044
&main()	0x0000000000401125
&useless()	0x000000000040111a



Note: very much not to scale!

Memory Corruption Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d = 3.14;
  s.a[i] = 0x40000000 ; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  →   3.14                  0x40091eb851eb851f
fun(1)  →   3.14                  0x40091eb851eb851f
fun(2)  →   3.1399998664856       0x40091eb840000000
fun(3)  →   2.00000061035156      0x4000000051eb851f
fun(4)  →   3.14                  0x40091eb851eb851f
fun(5)  →   3.14                  0x40091eb851eb851f
fun(6)  →   Segmentation fault
```

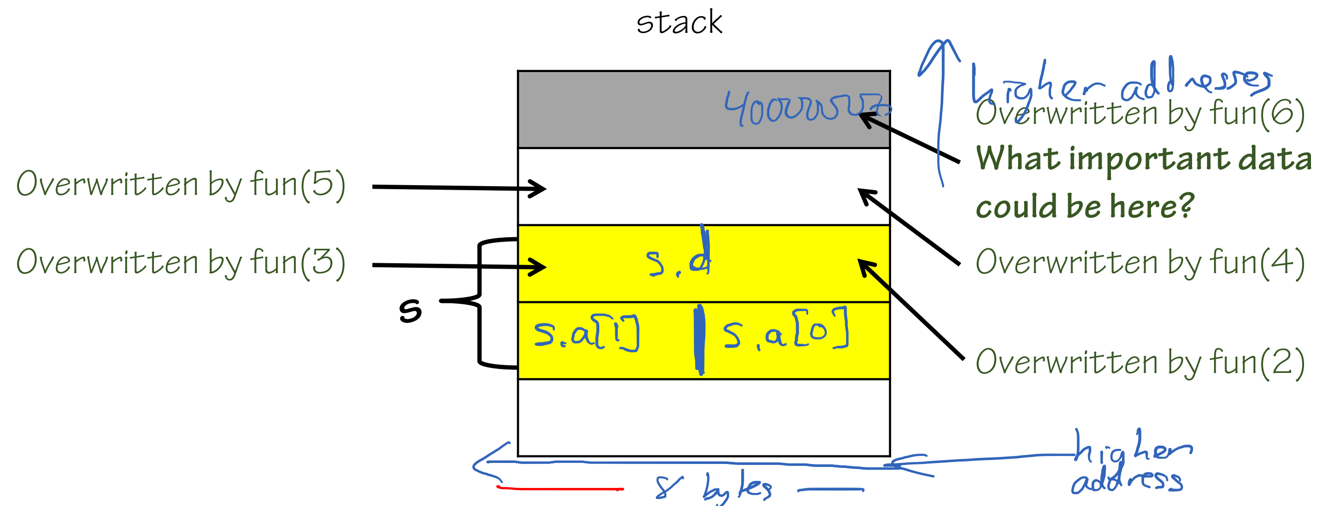
Thinking About the Crash

Assume each row in the stack diagram is 8 bytes

- Addresses increase from *bottom to top*
- Addresses increase from *right to left* within a row (little endian)

If s is located as shown, where are s.a[0], s.a[1], and s.d ?

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d = 3.14;
  s.a[i] = 0x40000000;
  return s.d;
}
```



Buffer Overflow

Exceeding memory size allocated for an array

- Generally called a "buffer overflow"
- If the array is on the stack, "stack smashing"

Why is it a big deal? Causes a lot of security vulnerabilities!

Morris Worm

- Nov. 2, 1988 -- Cornell grad student Robert Morris (somewhat unintentionally) creates first internet worm
  - Affected about a tenth of computers on the Internet at the time
  - Morris fined $ 10,050,
    given 400 hours community service, and 3 years probation
- Robert Morris now a professor at MIT…
- Part of his approach was a buffer overflow attack!

Morris Worm

The implementation of Unix **gets()** function

What's the problem here?

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other library functions
  - strcpy, strcat: Copy strings of arbitrary length
  - scanf, fscanf, sscanf, when given %s conversion specification

Vulnerable Code (Running Example)

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

← BTW, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix> ./bufdemo
Type a string:012345678901234567890123
012345678901234567890123
```

```
unix> ./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault
```

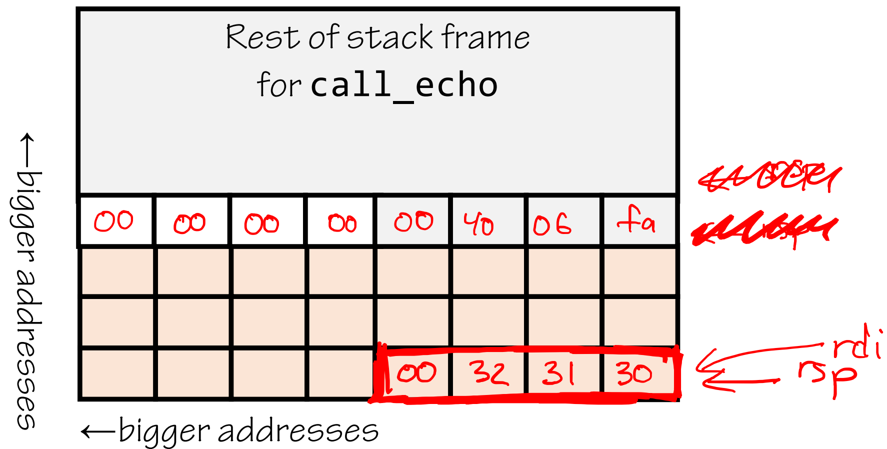
Normal Run: 3 characters

```
void call_echo() {
    echo();
}
```

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

```
4006f1:       e8 d9 ff ff ff             callq  4006cf <echo>
4006fa:       c3                         retq
```

```
00000000004006cf <echo>:
  4006cf:     48 83 ec 18             sub    $0x18,%rsp
  4006d3:     48 89 e7                mov    %rsp,%rdi
  4006d6:     e8 a5 ff ff ff          callq  400680 <gets>
  4006db:     48 89 e7                mov    %rsp,%rdi
  4006de:     e8 3d fe ff ff          callq  400520 <puts@plt>
  4006e3:     48 83 c4 18             add    $0x18,%rsp
  4006e7:     c3                      retq
```




```
unix> ./bufdemo-nsp
Type a string:012
012
```

ascii of 0 is 0x30

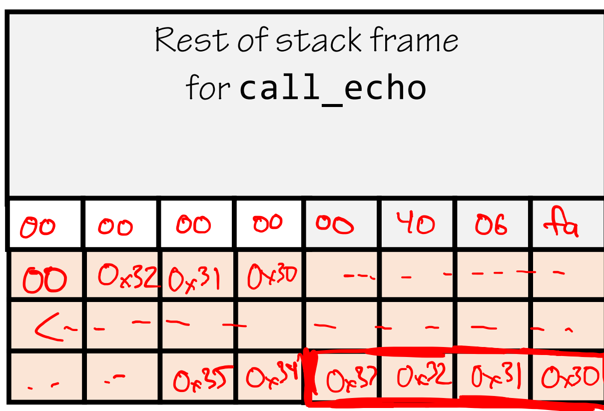
Risky Run: 23 characters

```
void call_echo() {
    echo();
}
```

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

```
4006f1:     e8 d9 ff ff ff          callq  4006cf <echo>
4006fa:     c3                      retq
```

```
00000000004006cf <echo>:
 4006cf:    48 83 ec 18          sub    $0x18,%rsp
 4006d3:    48 89 e7             mov    %rsp,%rdi
 4006d6:    e8 a5 ff ff ff       callq  400680 <gets>
 4006db:    48 89 e7             mov    %rsp,%rdi
 4006de:    e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:    48 83 c4 18          add    $0x18,%rsp
 4006e7:    c3                   retq
```




```
unix> ./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

ascii of 0 is 0x30

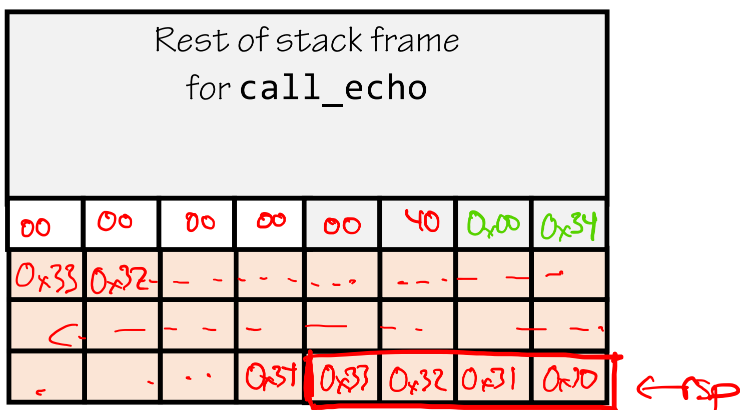
Buggy Run: 25 characters

```
void call_echo() {
    echo();
}
```

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

```
4006f1:    e8 d9 ff ff ff        callq  4006cf <echo>
4006fa:    c3                    retq
```

```
00000000004006cf <echo>:
 4006cf:    48 83 ec 18          sub    $0x18,%rsp
 4006d3:    48 89 e7             mov    %rsp,%rdi
 4006d6:    e8 a5 ff ff ff       callq  400680 <gets>
 4006db:    48 89 e7             mov    %rsp,%rdi
 4006de:    e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:    48 83 c4 18          add    $0x18,%rsp
 4006e7:    c3                   retq
```




```
unix> ./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

ascii of 0 is 0x30

Observation

Rest of stack frame
for `call_echo`

00	00	00	00	00	40	00	34
33	32	31	30	39	38	37	36
35	34	33	32	31	30	39	38
37	36	35	34	33	32	31	30

```
unix> ./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

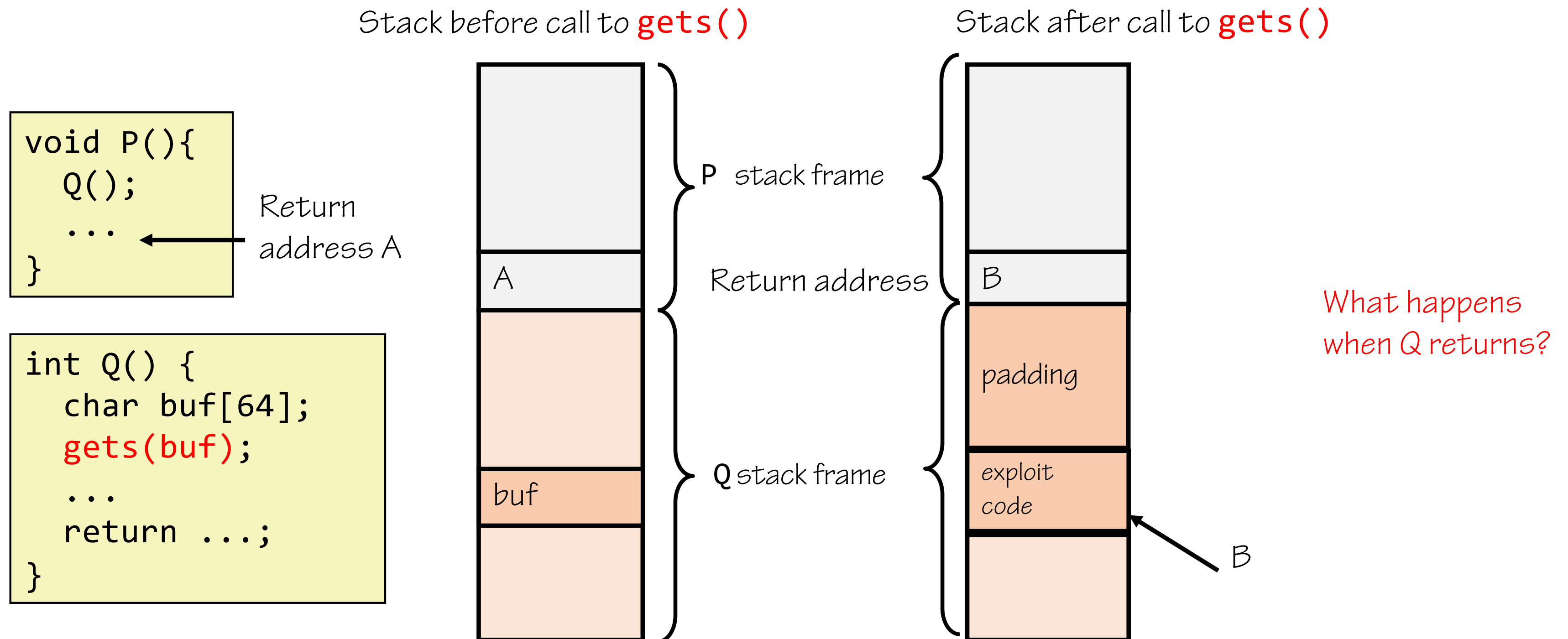
The program crashed because the code "returned" (jumped) to address 0x400034, which didn't contain valid machine code.

And by typing in a carefully-chosen 32-character string, we can make echo() "return" (jump) to any address we want!

Code Injection Attacks

Input string includes bytes encoding machine code

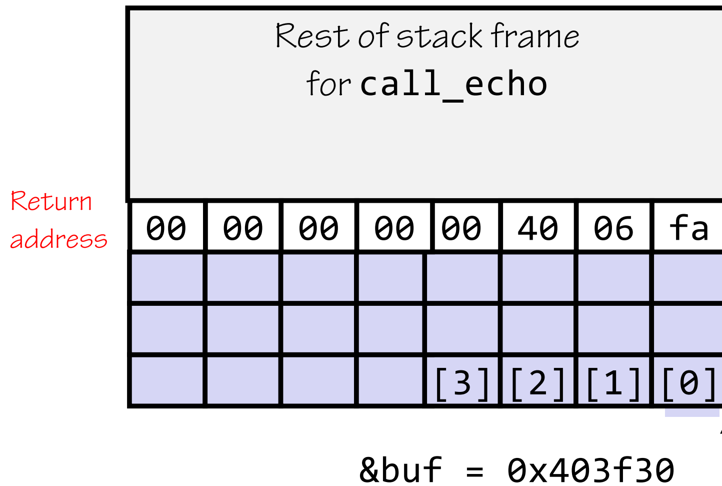
Overwrite return address A with address of that code!

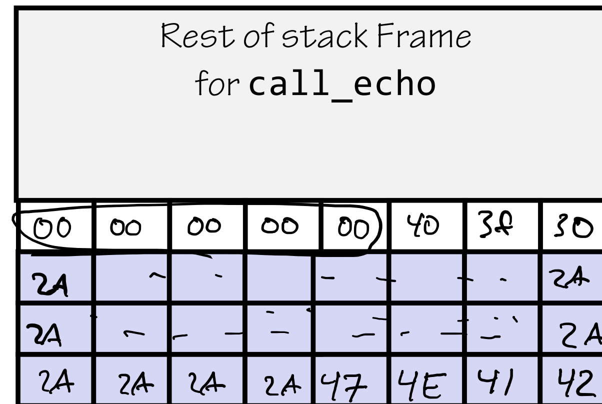


Exercise

Assume the ASCII for the string “BANG” is also a machine instruction that makes your computer explode. Come up with an input to **echo** that makes your computer explode.

1. Show the stack (use hex values) after the call to **gets**. An ASCII table is below.
2. Write the text input string here: "BANG**********----*Ø?@" (20)





Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

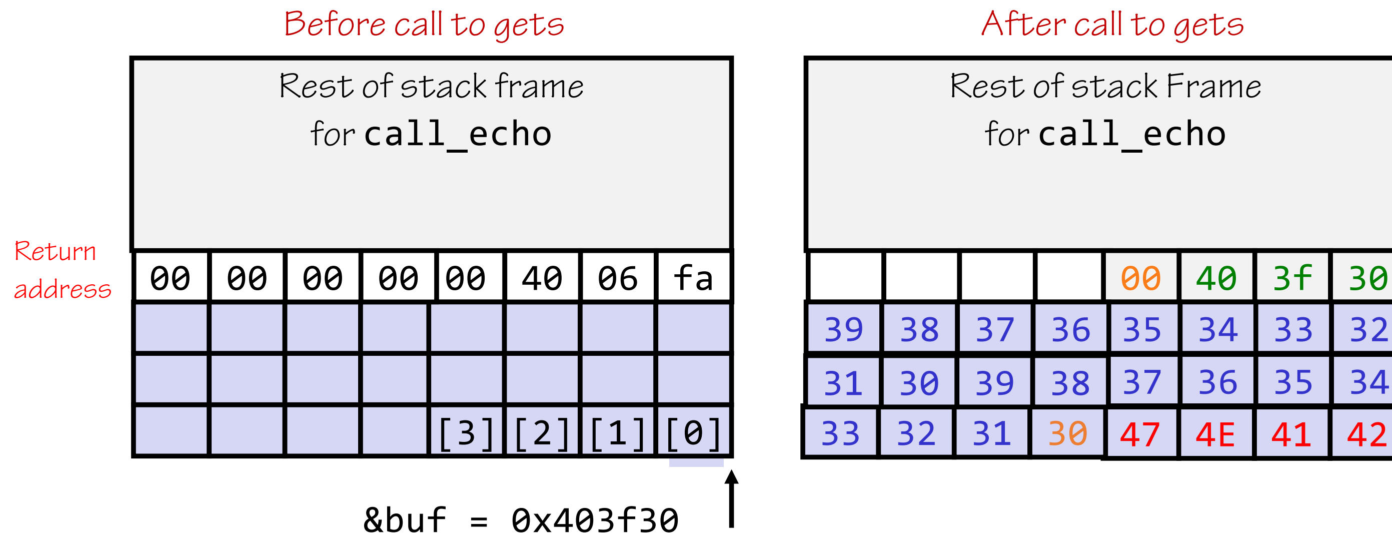
Source: www.LookupTables.com

Exercise

Assume the ASCII for the string "BANG" is also a machine instruction that makes your computer explode. Come up with an input to **echo** that makes your computer explode.

1. Show the stack (use hex values) after the call to **gets**. An ASCII table is below.

2. Write the text input string here: BANG01234567890123456789 0?@



Exploits Based on Buffer Overflows

- Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- You will learn some of the tricks in Attack Lab
  - Hopefully to convince you to never leave such holes in your programs!!
- Prevention techniques
  - Avoid overflow vulnerabilities
  - Employ system-level protections
  - Have compiler use "stack canaries"

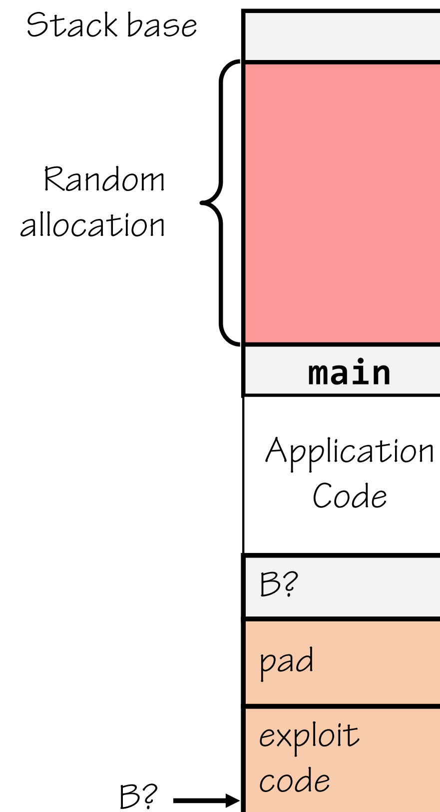
1. Avoid Overflow Vulnerabilities in Code (!)

- For example, use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - E.g., see "*Secure Programming in C and C++*" (linked on Piazza)

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

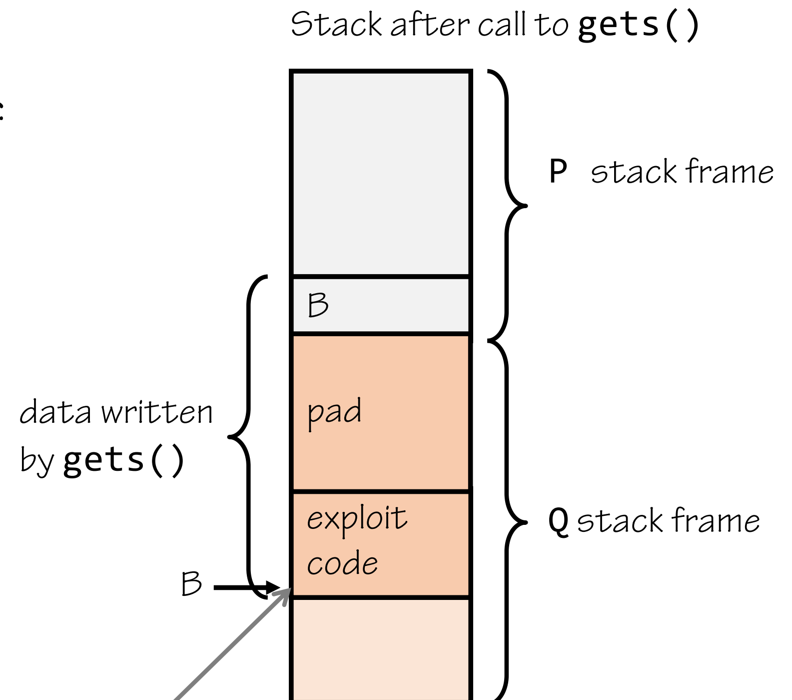
2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program so address of buffer is not known
  - Makes it difficult for hacker to determine address of inserted code



2. System-Level Protections can help

- Non-executable code segments
  - In previous x86, could mark region of memory as either "read-only" or "writeable"… could execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable



3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - -fstack-protector
  - Now the default,
    e.g., in your bomb machine code.

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Canary-Protected Buffer Disassembly

echo:

```
40072f:   sub    $0x18,%rsp
400733:   mov    %fs:0x28,%rax
40073c:   mov    %rax,0x8(%rsp)
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
400758:   xor    %fs:0x28,%rax
400761:   je     400768 <echo+0x39>
400763:   callq  400580 <__stack_chk_fail@plt>
400768:   add    $0x18,%rsp
40076c:   retq
```

Put canary on stack

Check canary on stack

Detect buffer overflow