Security: Buffer Overflow

CS 105: Computer Systems

Guest Lecturer: Beth Trushkowsky

February 22, 2022

Learning Goals

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

Exercise: memory layout of a `double`

s	Exponent encoding	Fractional part of Mantissa
1	11 bits	52 bits

Recall the data type `double` uses 8 bytes, as shown above.
Suppose we have: `double pi = 3.14;`
In hex, the value of the variable `pi` is `0x40091eb851eb851f`

1. **Underline which hex digits encode the fractional part of the mantissa.**

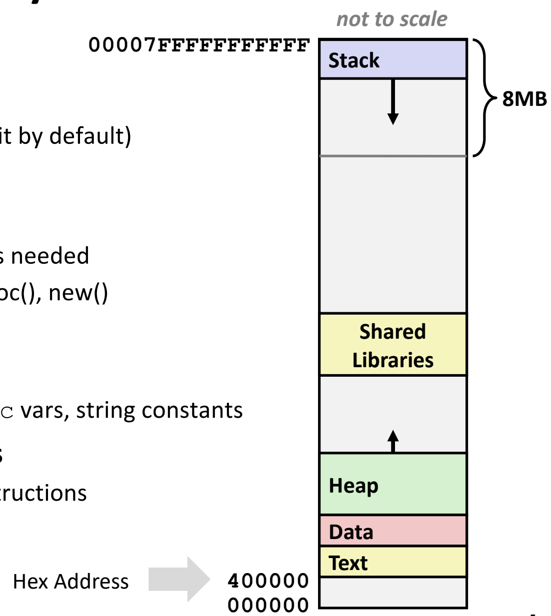
   `0x40 09 1e b8 51 eb 85 1f`

2. **If `&pi` is `0x100`, what should be the one-byte content (in hex) at memory address `0x102` on a *little endian* machine?**

   `0xeb`

x86-64 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

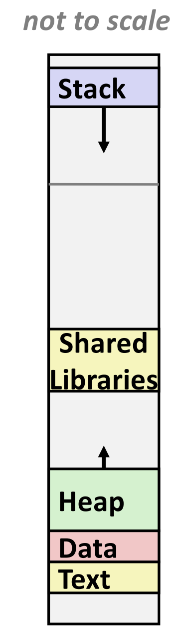
```
char big_array[1L << 24];  /* 16 MB */
char huge_array[1L << 31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

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

Where does everything go?



Memory Referencing Bug 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;
}
```

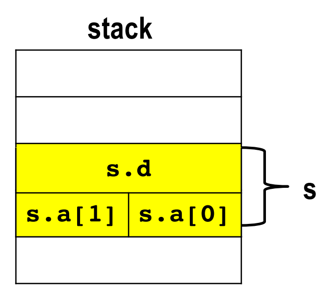
```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(5)  →  3.14
fun(6)  →  Segmentation fault
```


Exercise: Memory Referencing Bug Example

- **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
- **Note that s requires 16 bytes, as shown. Indicate where in the diagram `s.a[0]`, `s.a[1]`, and `s.d` are located.**
  - Recall an `int` is 4 bytes and a `double` is 8 bytes

```
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;
}
```



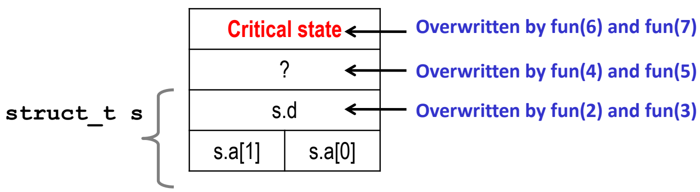
Memory Referencing Bug: Explanation

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

double fun(int i) {
  struct_t s;
  s.d = 3.14; /* 0x40091eb851eb851f */
  s.a[i] = 0x40000000;
  return s.d;
}
```

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

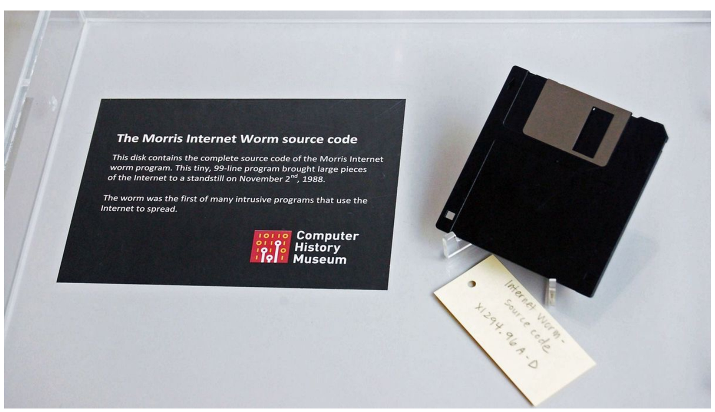
What sort of critical state could be here?



Buffer Overflow

- **Exceeding memory size allocated for an array**
  - Generally called a "buffer overflow" aka "stack smashing"
- **Why is it a big deal? Causes a lot of security vulnerabilities!**

Morris Worm



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, 400 hours community service, and 3 years probation
- **Robert Morris now a professor at MIT…**
- **Part of his approach was a buffer overflow attack!**

String Library Code

- **Implementation of Unix function `gets()`**

```
/* 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;
}
```

How large is the destination buffer?

What's the limit on characters that are read?

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

Running example using `gets`

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

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

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

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

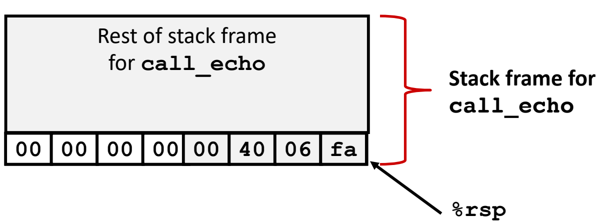

Example: calling echo

```
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
```



What's the return address for `call_echo`?
Note: return address in *little endian*

Example: instruction `sub` in echo

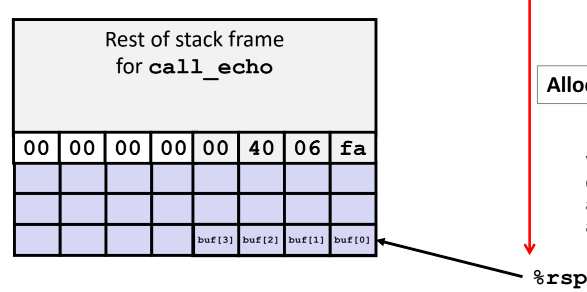
```
void call_echo() {
    echo();
}
```

```
/* 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
```

Allocate space on stack for `buf`



Why 24 bytes?
Often more space is allocated than is actually needed because of data alignment requirements.

Example: preparing to call gets (in echo)

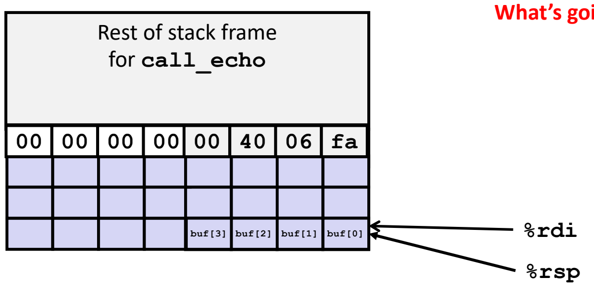
```
void call_echo() {
    echo();
}
```

```
/* 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
```

What's going into `%rdi`? Why?



Example: Calling gets (in echo)

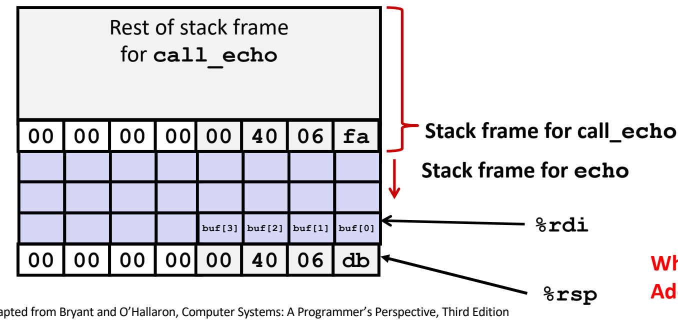
```
void call_echo() {
    echo();
}
```

```
/* 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
```

```
gets
400680:   ...
```



What is the Return Address for echo?

Example: in gets, reading first character

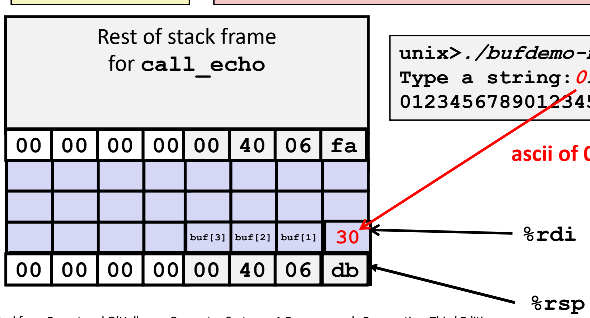
```
void call_echo() {
    echo();
}
```

```
/* 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
```

```
gets
400680:   ...
```




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

ascii of 0 is 0x30

Example: in gets, read string length 23

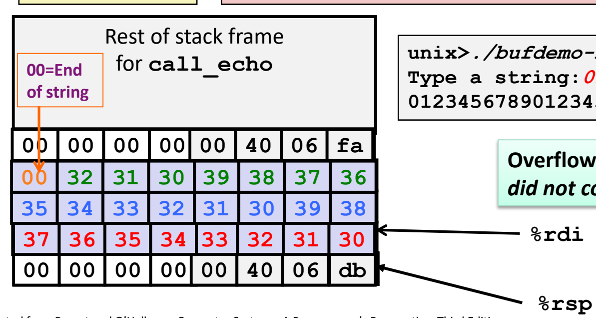
```
void call_echo() {
    echo();
}
```

```
/* 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
```

```
gets
400680:   ...
```




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

Overflowed 4 byte buffer, but ***did not corrupt return address***

Example: in gets, read string length 25

```
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
```

gets

```
400680:        …
```

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
00	00	00	00	00	40	06	db

%rdi

%rsp

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

Overflowed 4 byte buffer, and corrupted return address!!

Example: In echo after gets read 25 and puts returns

```
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
```

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

%rsp

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

Overflowed 4 byte buffer, and corrupted return address!!

Where will %rsp point after add instruction?
What will happen when retq?

Example: Returning (from echo, gets read 25)

```
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
```

```
400034:        ???
```

What is here?

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

%rsp

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


Example: What instruction gets executed?

```
400034:        ???
```

What is here?

Valid machine instruction → **Continue execution**

AND THEN …

Not a valid machine instruction → **Segmentation fault**

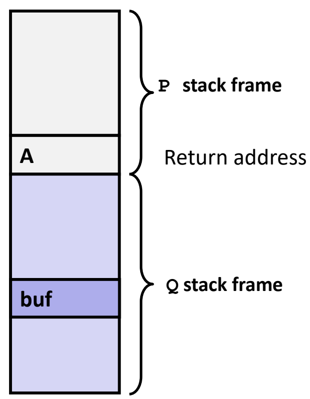
Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

Return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack *before* call to `gets()`



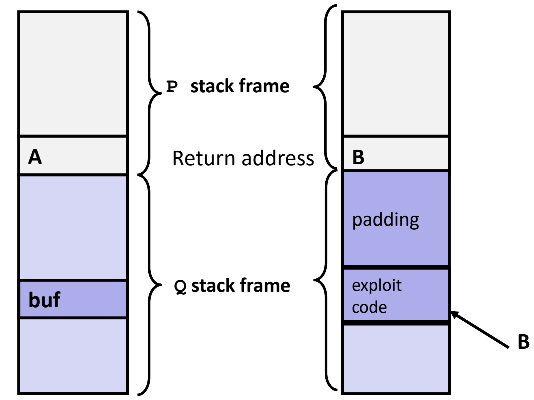
Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

Return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack *before* call to `gets()`   Stack after call to `gets()`



- **Input string contains byte representation of executable code**
- **Overwrite return address A with address of `buf` array**

What happens when Q returns?

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - 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**
  1. Avoid overflow vulnerabilities
  2. Employ system-level protections
  3. Have compiler use "stack canaries"

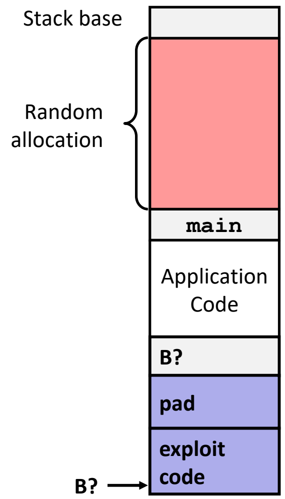
1. Avoid Overflow Vulnerabilities in Code (!)

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

- **For example, use library routines that limit string lengths**
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer

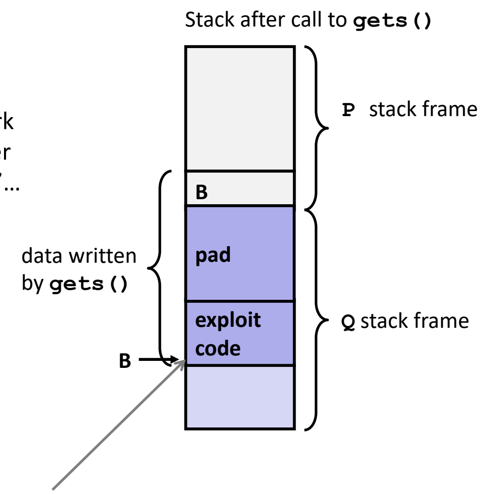
2. System-Level Protections can help

- **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



Any attempt to execute this code will fail

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 (disabled earlier)

```
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 (400733–40073c)

Check canary on stack (400753–400758)

Detect buffer overflow (400763)

Exercise

Assume your computer uses ASCII encoding for strings and that 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. You can assume the system knows how many bytes the “BANG” instruction is after it reads the first byte corresponding to “B”.

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?@

Before call to gets

Rest of stack frame for `call_echo`

Return address	00	00	00	00	00	40	06	fa
					[3]	[2]	[1]	[0]

&buf = 0x403f30

After call to gets

Rest of stack Frame for `call_echo`

				00	40	3f	30
39	38	37	36	35	34	33	32
31	30	39	38	37	36	35	34
33	32	31	30	47	4E	41	42

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	&#127;	DEL

Source: www.LookupTables.com