Machine-Level Programming IV: Security & Floating Point

Lecture 7

Yeongpil Cho

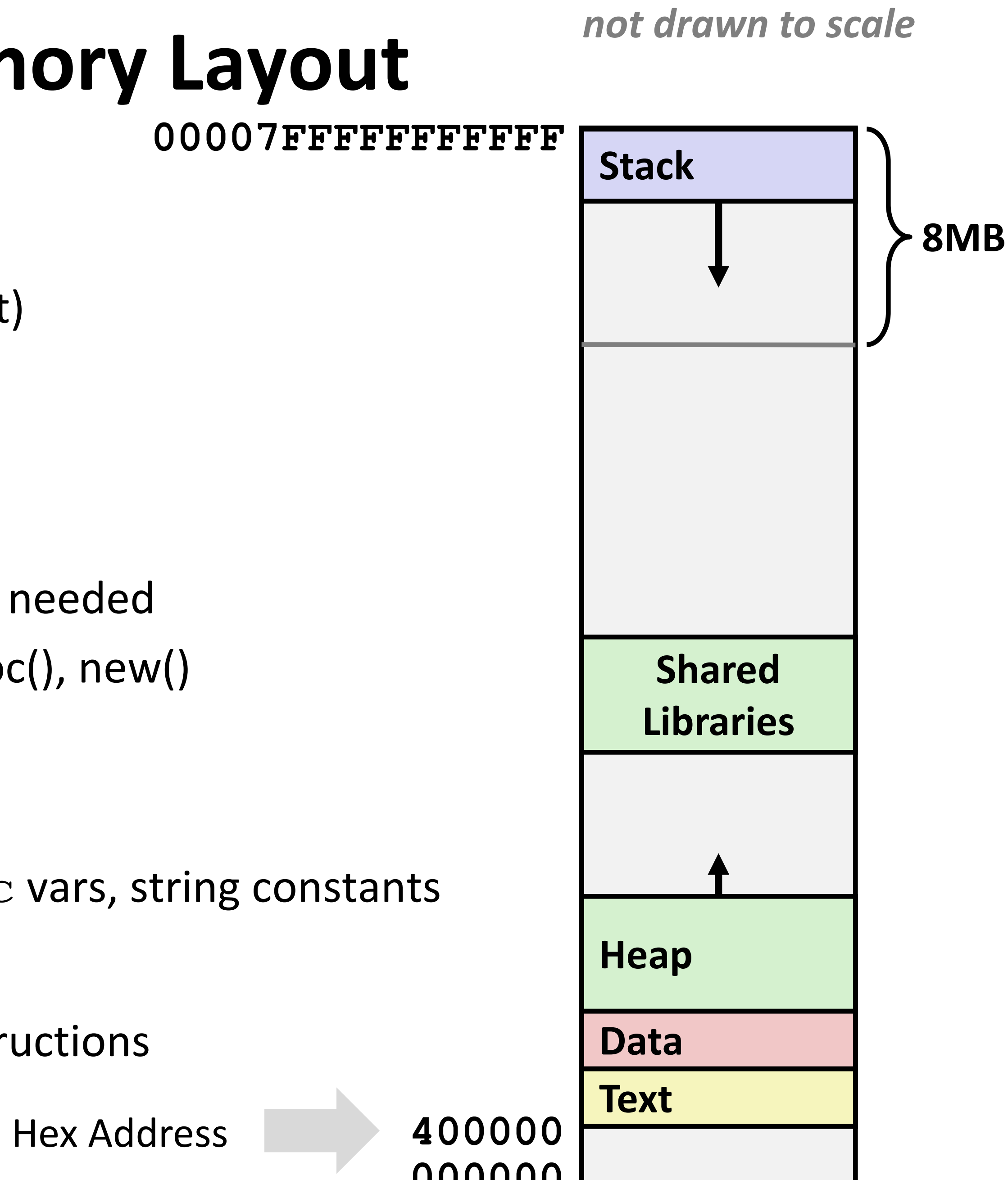
Hanyang University

Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
- **Floating Point**

x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
    - $ ulimited -a
  - 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

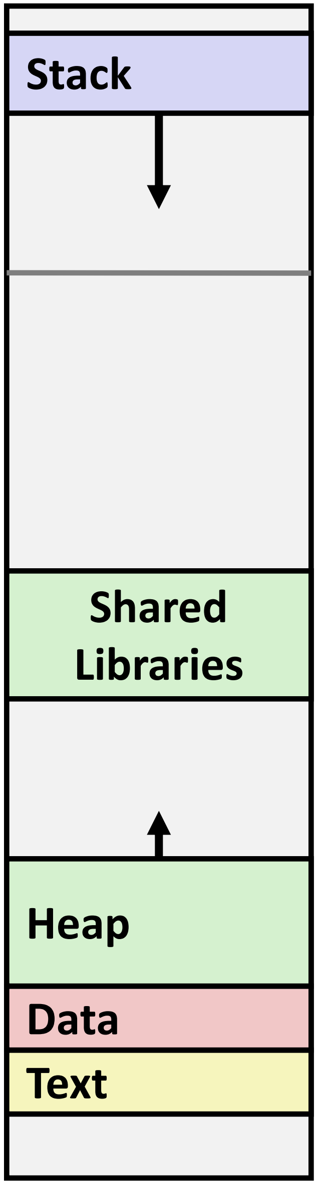
not drawn to scale

```
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 ... */
}
```



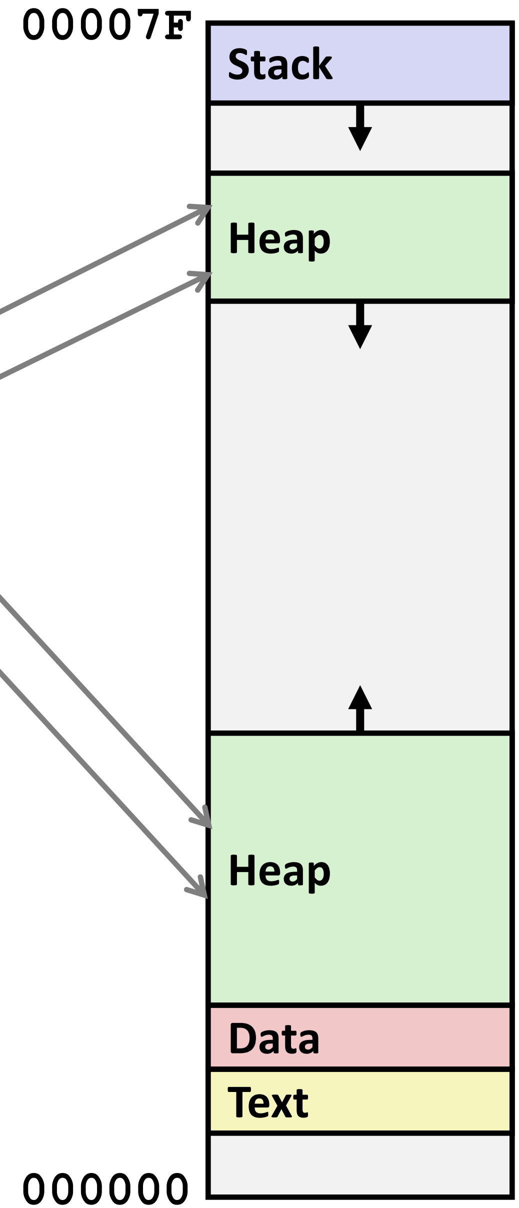
Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`

not drawn to scale



Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
- Floating Point

Recall: Memory Referencing Bug Example

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

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* 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(6)  →   Segmentation fault
```

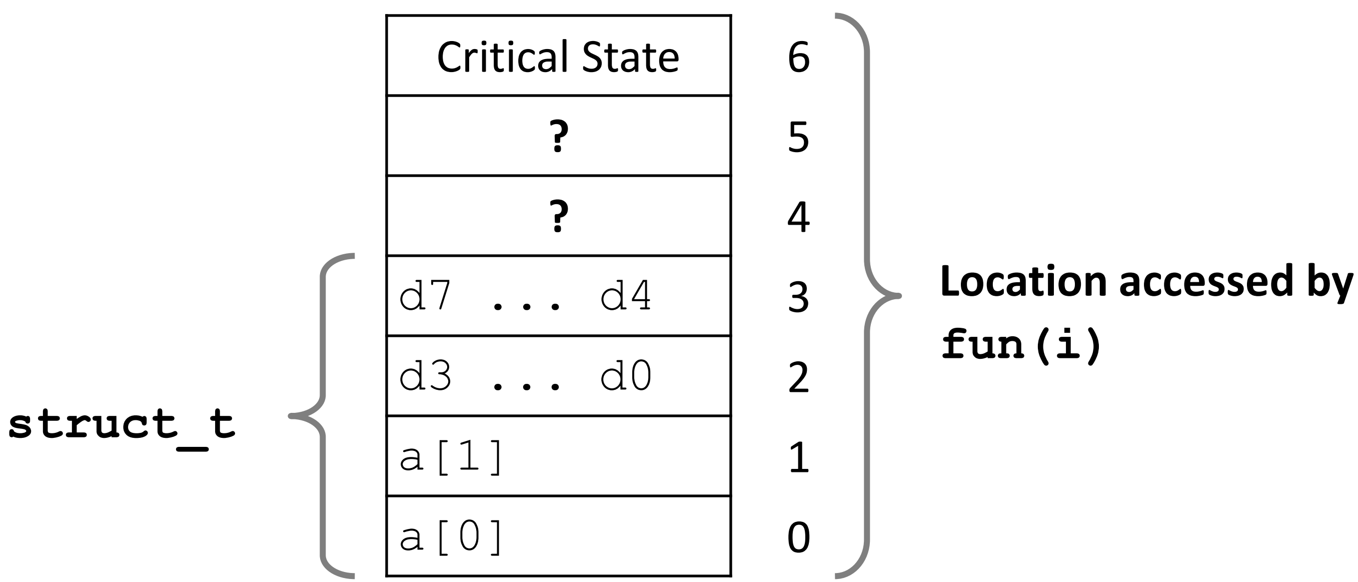
- Result is system specific

Memory Referencing Bug Example

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

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

Explanation:



Such problems are a BIG deal

- **Generally called a “buffer overflow”**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It’s the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

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;
}
```

  - No way to specify limit on number of characters to read

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

Vulnerable Buffer Code

```
/* 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-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

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

Buffer Overflow Disassembly

stack frame size is multiple of 16 B

echo:

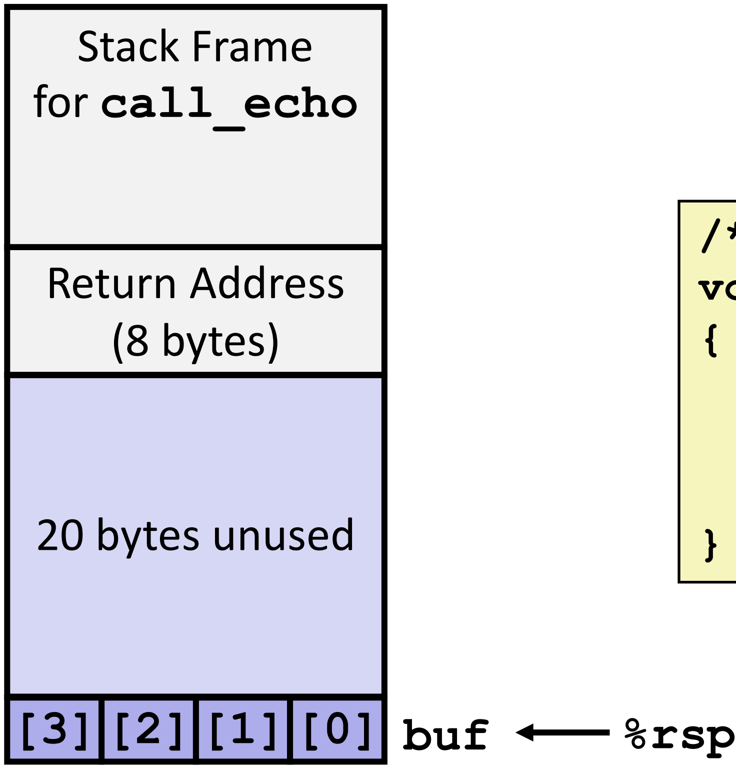
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18          sub    $0x18,%rsp
 4006d3:  48 89 e7             mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff       callq  400680 <gets>
 4006db:  48 89 e7             mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:  48 83 c4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

call_echo:

```
4006e8:   48 83 ec 08          sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00       mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6:  48 83 c4 08          add    $0x8,%rsp
 4006fa:  c3                   retq
```

Buffer Overflow Stack

Before call to gets

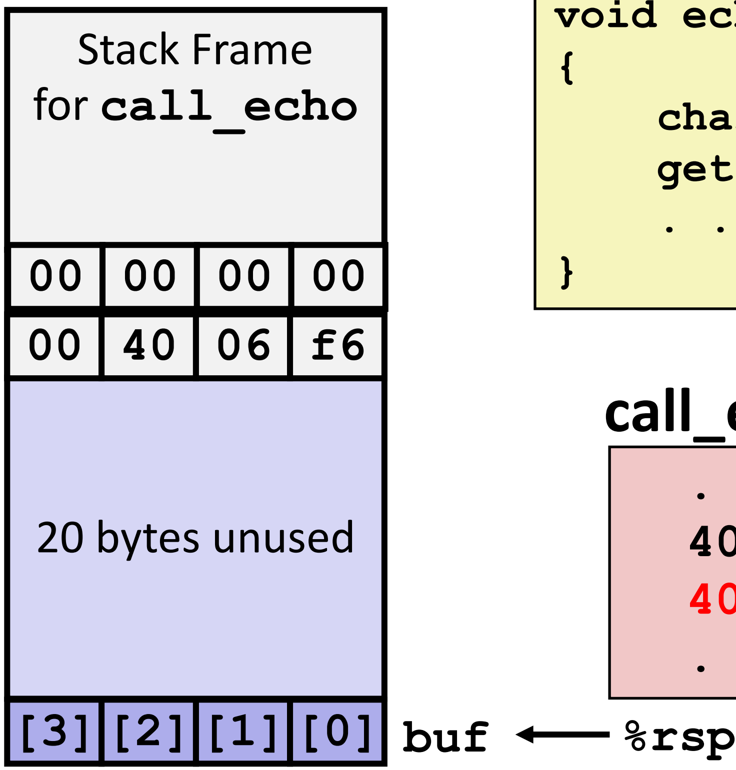



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets




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

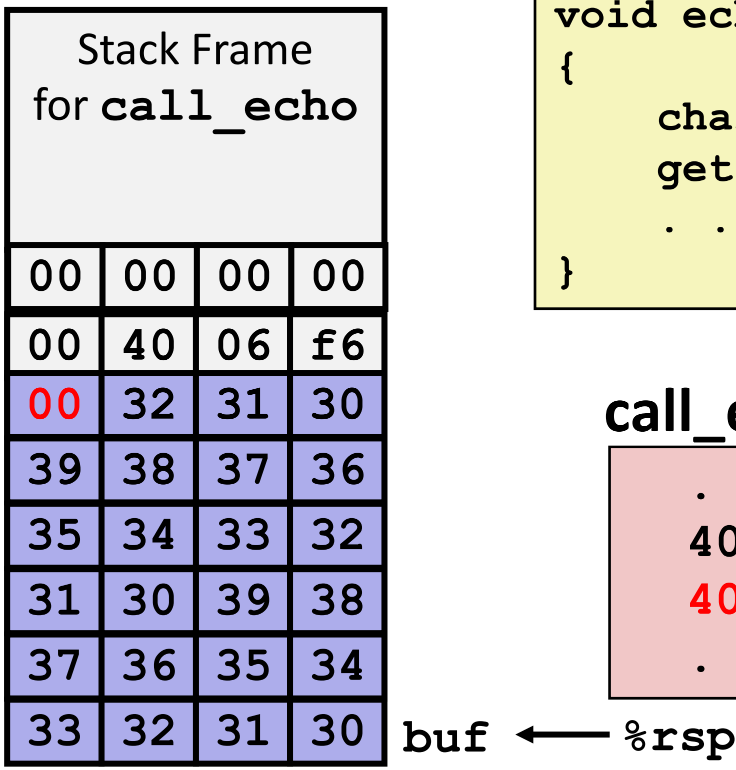
```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
  . . .
  4006f1:   callq   4006cf <echo>
  4006f6:   add     $0x8,%rsp
  . . .
```

Buffer Overflow Stack Example #1

After call to gets




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

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

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

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

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

buf ← %rsp

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

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

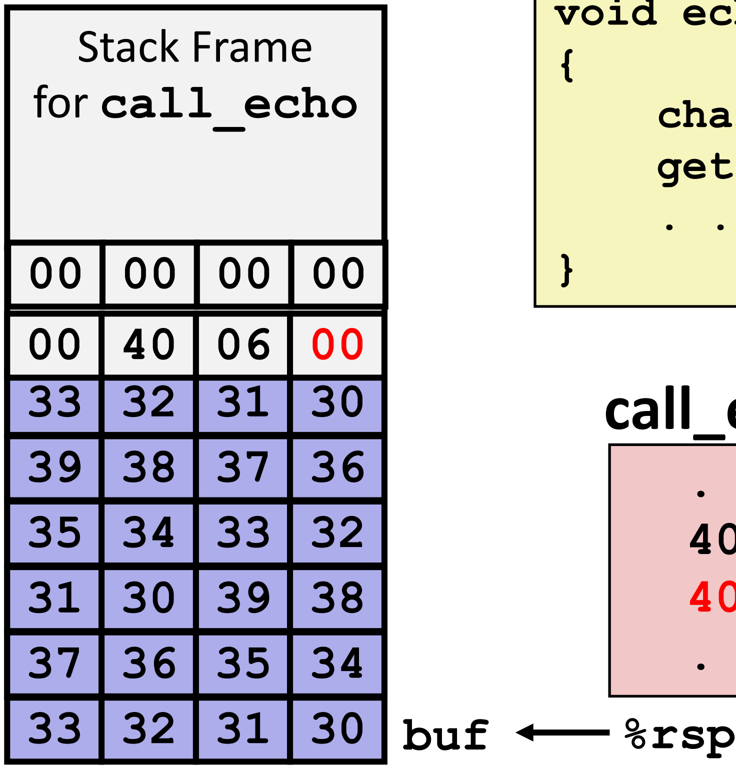
```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

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

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

After call to gets




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

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

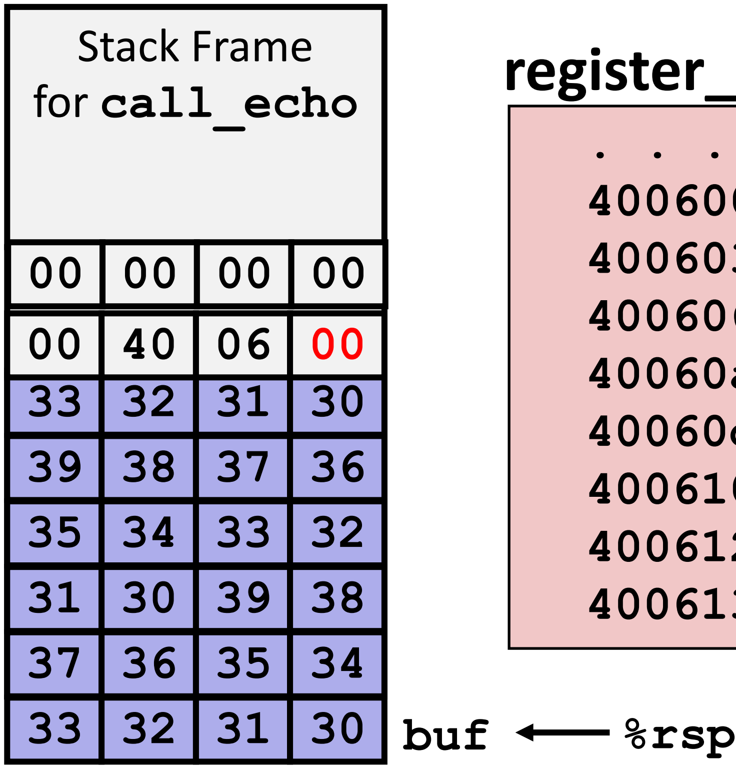
```
   . . .
   4006f1:   callq   4006cf <echo>
   4006f6:   add     $0x8,%rsp
   . . .
```

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

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Stack Example #3 Explained

After call to gets



register_tm_clones:

```
  . . .
  400600:  mov    %rsp,%rbp
  400603:  mov    %rax,%rdx
  400606:  shr    $0x3f,%rdx
  40060a:  add    %rdx,%rax
  40060d:  sar    %rax
  400610:  jne    400614
  400612:  pop    %rbp
  400613:  retq
```

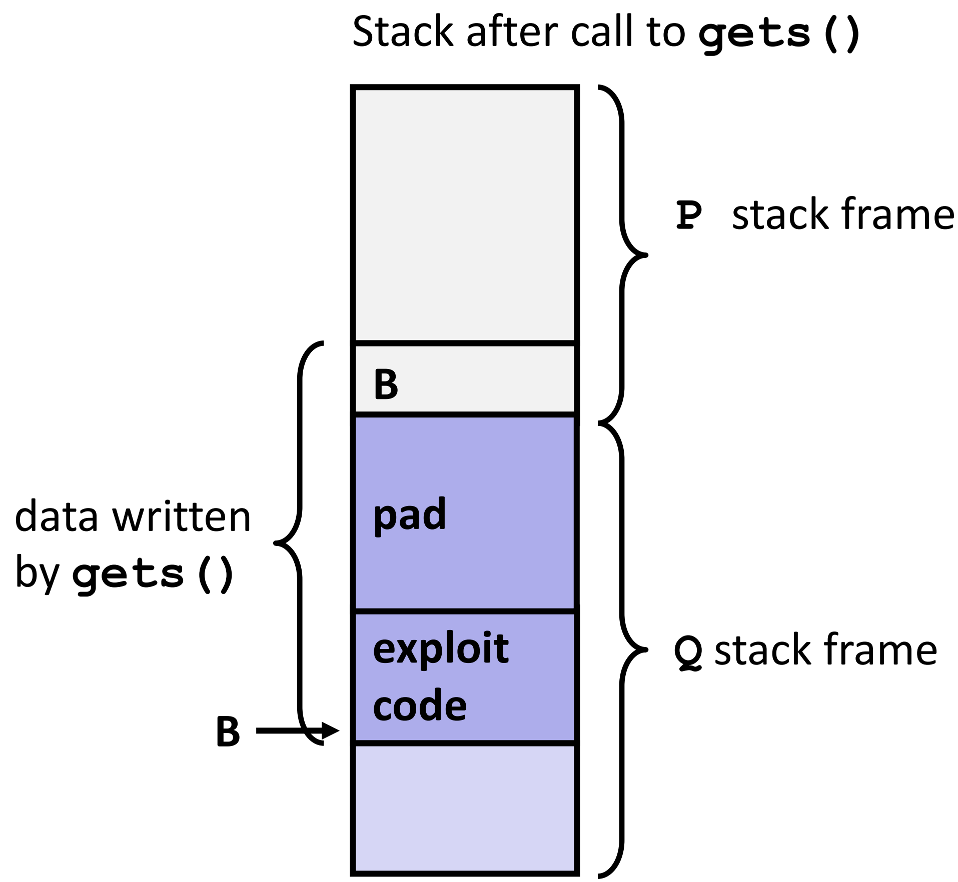
"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Code Injection Attacks

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

return address A

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



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
  - A lot of worms and viruses exploit this vulnerability
    - Worm: A program that
      - Can run by itself
      - Can propagate a fully working version of itself to other computers
    - Virus: Code that
      - Adds itself to other programs
      - Does not run independently
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes ☹
  - Fortunately, Recent measures make these attacks much more difficult

What to do about buffer overflow attacks

- Avoid overflow vulnerabilities

- Employ system-level protections

- Have compiler use “stack canaries”

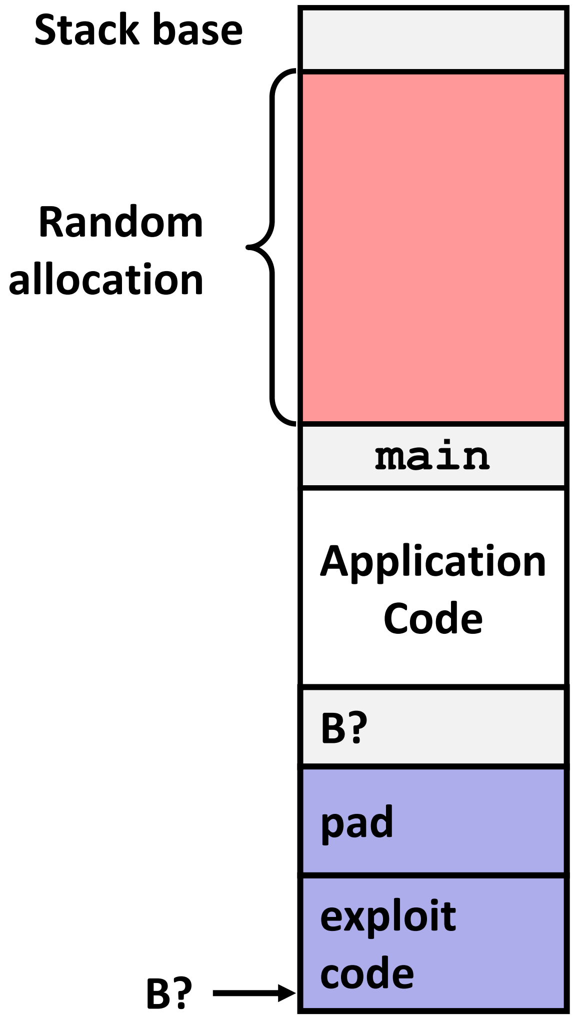
1. Avoid Overflow Vulnerabilities in Code (!)

```
/* 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
      - **gets_s**, **scanf_s** in MS Visual Studio
    - Or use **`%ns`** where **`n`** is a suitable integer
      - e.g., %8s

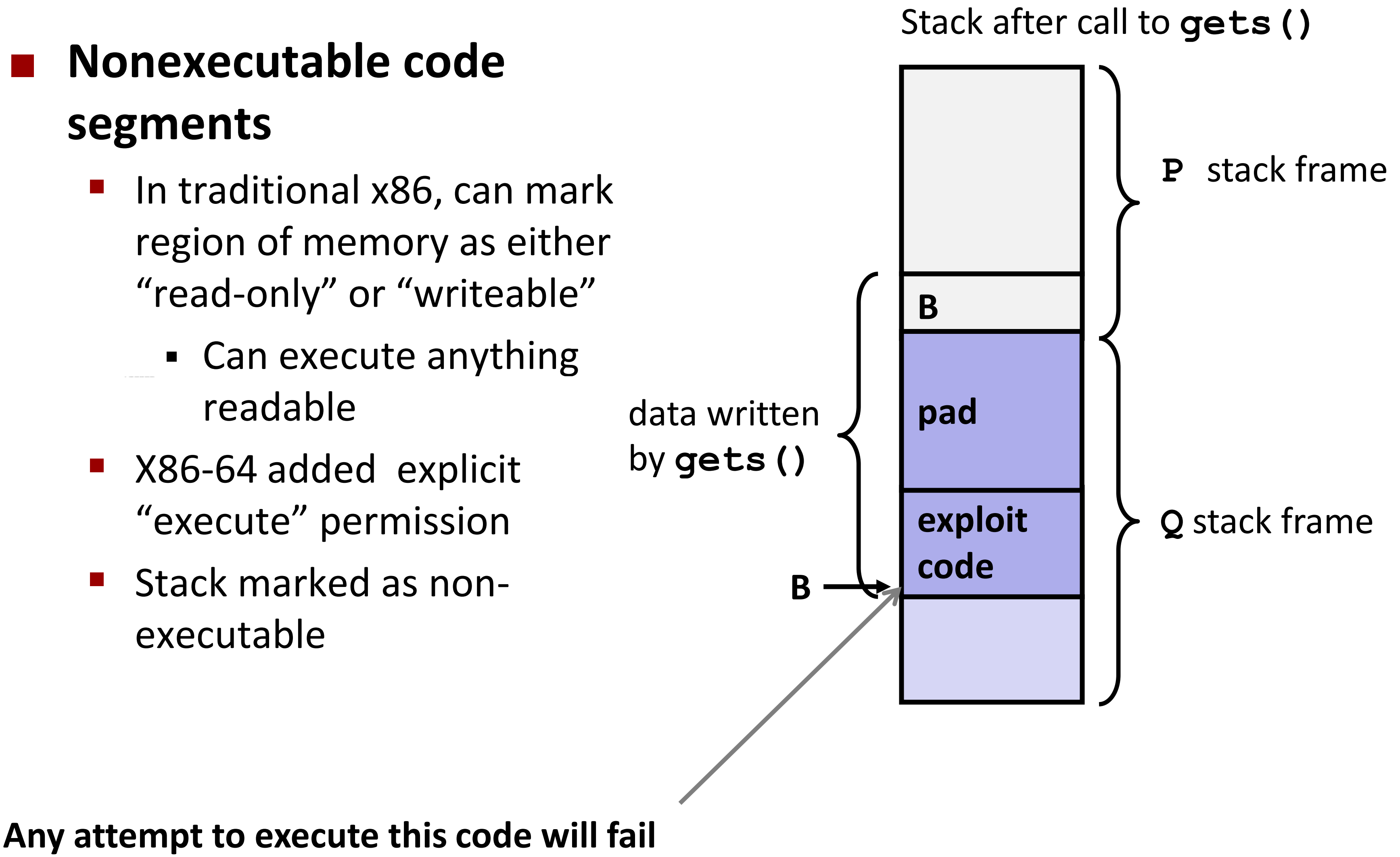
2. System-Level Protections can help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code



2. System-Level Protections can help

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can 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 (disabled earlier)

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

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

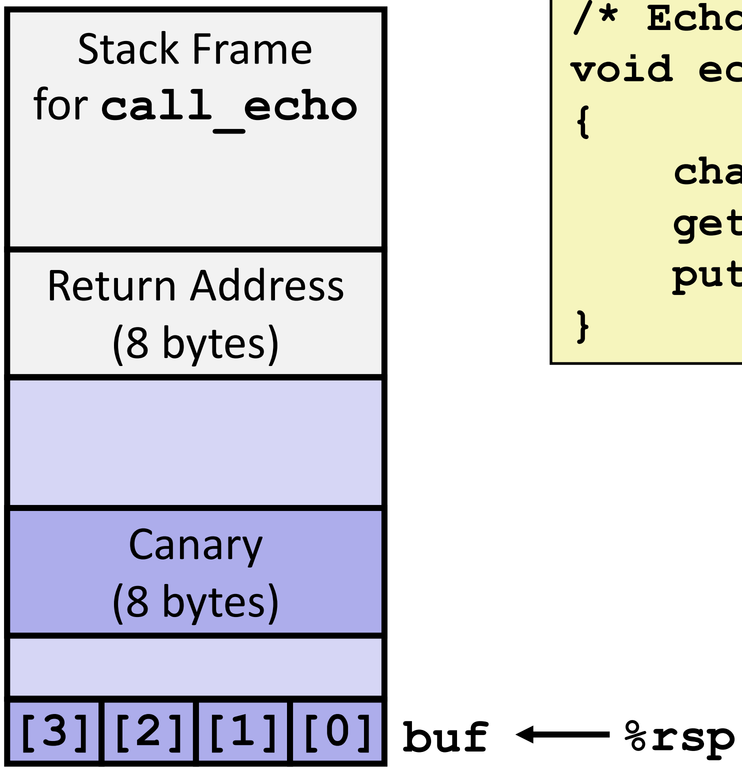
Protected Buffer Disassembly

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
```

Setting Up Canary

Before call to gets

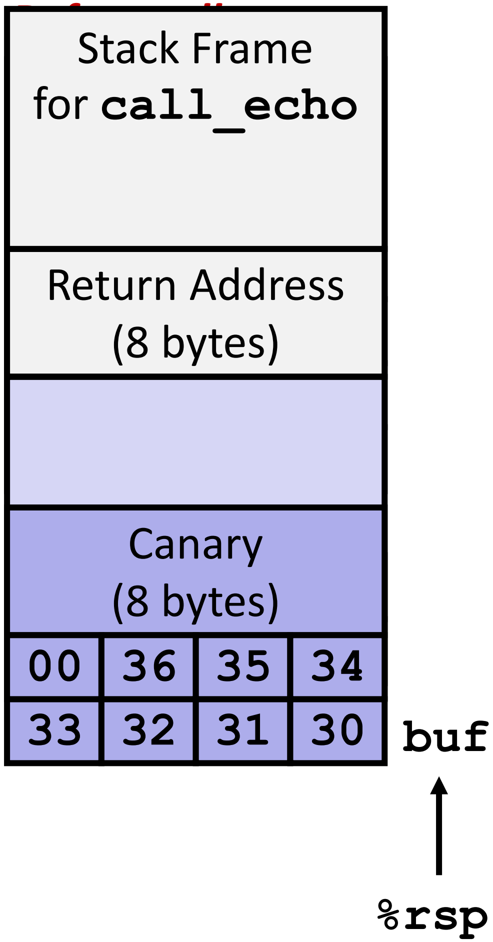



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets




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

Input: ***0123456***

```
echo:
    . . .
    movq     8(%rsp), %rax       # Retrieve from
stack
    xorq     %fs:40, %rax        # Compare to canary
    je       .L6                 # If same, OK
    call     __stack_chk_fail    # FAIL
.L6:         . . .
```

Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*
- **Conditions to construct program from *gadgets***
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable

Gadget Example #1

```
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`

- Use tail end of existing functions

Gadget Example #2

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

Encodes `movq %rax, %rdi`

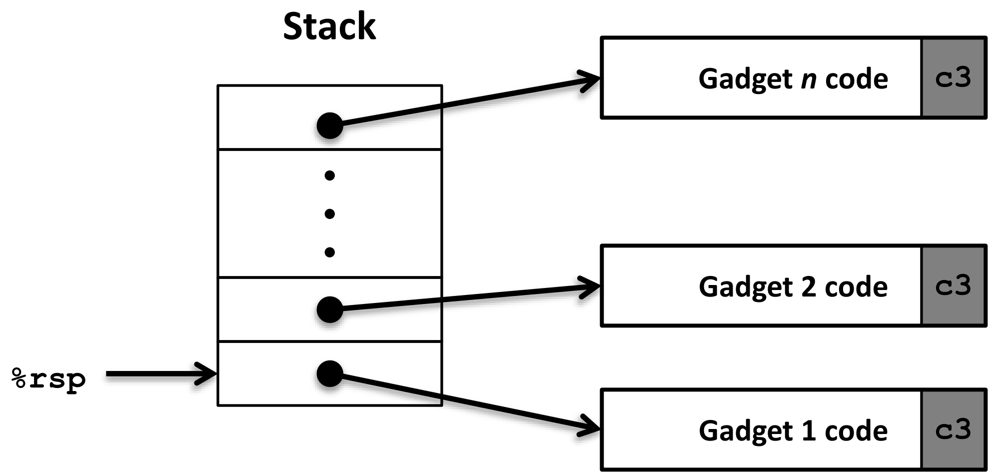
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7   movl   $0xc78948d4,(%rdi)
  4004df:  c3                  retq
```

rdi ← rax

Gadget address = `0x4004dc`

- **Repurpose byte codes**

ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
- **Floating Point**

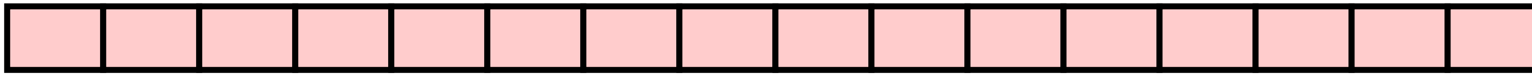
Background

- **Hardware supports for floating point**
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE

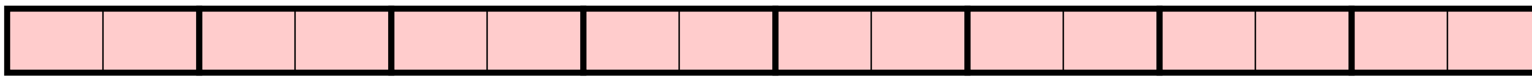
Programming with SSE3

XMM Registers

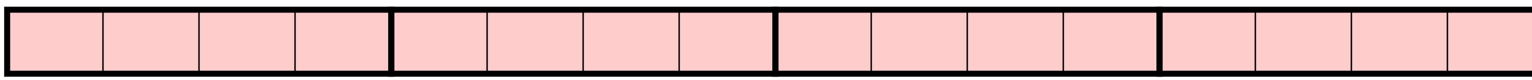
- 16 total, each 16 bytes
- 16 single-byte integers



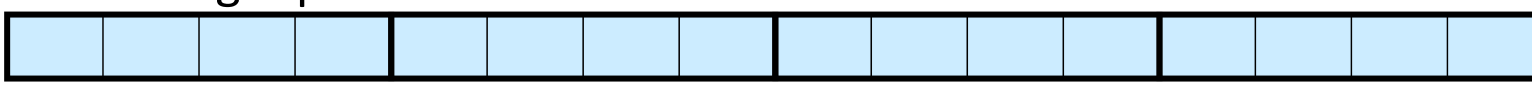
- 8 16-bit integers



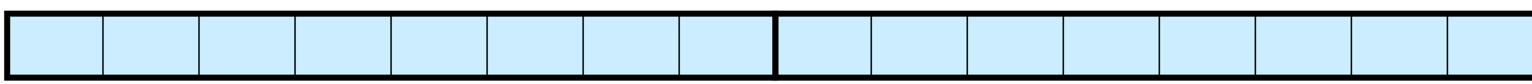
- 4 32-bit integers



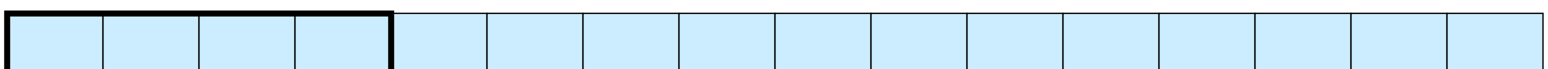
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float

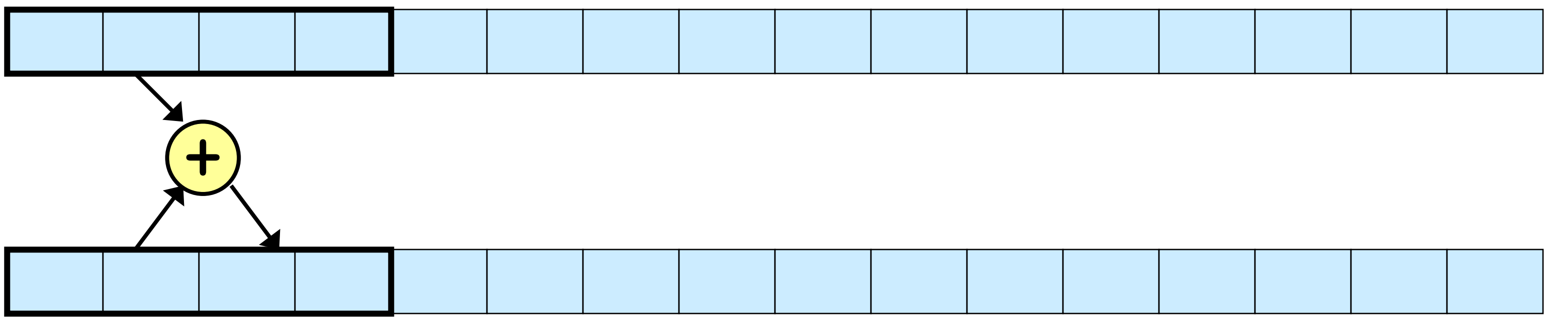


- 1 double-precision float

Scalar & SIMD Operations

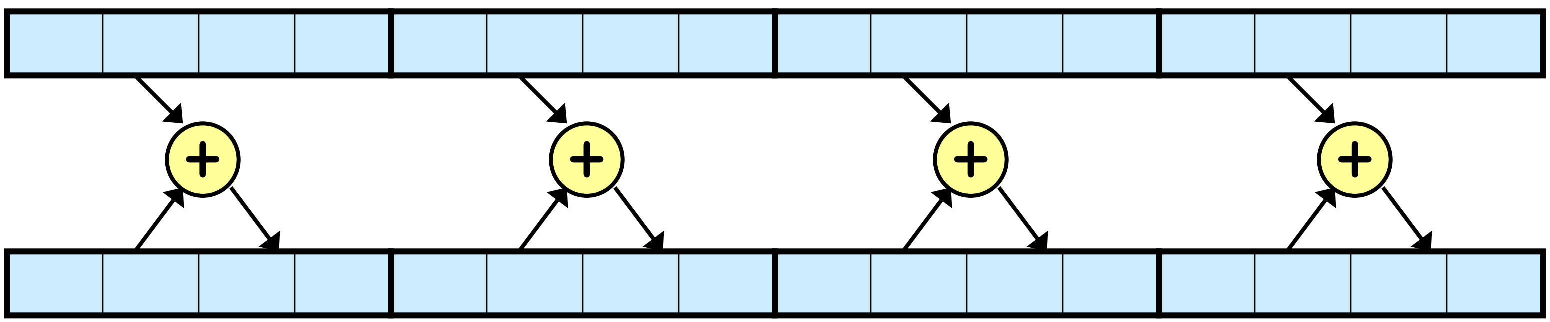
- Scalar Operations: Single Precision

`addss %xmm0,%xmm1`



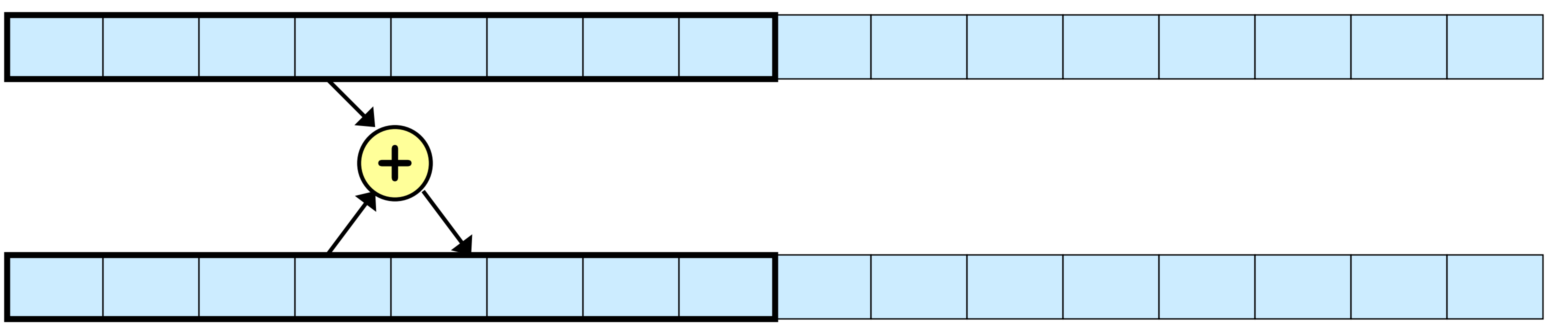
- SIMD Operations: Single Precision

`addps %xmm0,%xmm1`



- Scalar Operations: Double Precision

`addsd %xmm0,%xmm1`



FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, ...
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss    %xmm1, %xmm0
  ret
```

```
  # x in %xmm0, y in %xmm1
  addsd    %xmm1, %xmm0
  ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1   # Copy v
  movsd   (%rdi), %xmm0  # x = *p
  addsd   %xmm0, %xmm1   # t = x + v
  movsd   %xmm1, (%rdi)  # *p = t
  ret
```

Other Aspects of FP Code

- ***Lots* of instructions**
  - Different operations, different formats, ...
- **Floating-point comparisons**
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF
- **Using constant values**
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory