Breakout

- Pointers in C

```
long i, *pi, **ppi;
i = 4; pi = &i; ppi = & pi;
```

Assume i is located at 0x8000; pi is located at 0x8008; and ppi is located at 0x8010

What value is contained in *a* for the following statements –

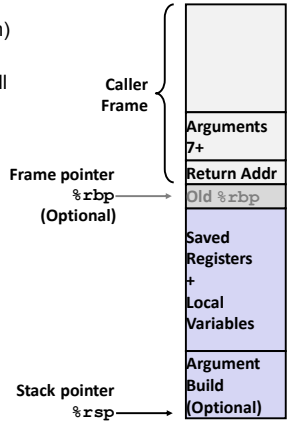
a = ppi+1; 0x8010
a = &ppi; 0x8010
*a = **ppi; 0x4*
*a = *(pi+1); 0x8000*
*a = *(*ppi+1); 0x8000*

193

193

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



194

Today

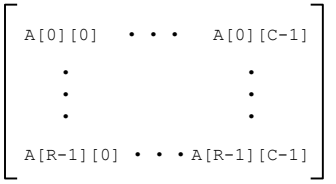
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

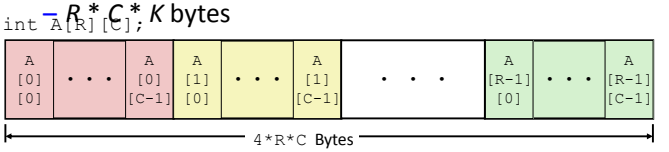
195

Multidimensional (Nested) Arrays

- Declaration

  T **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes

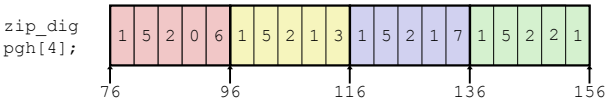




196

Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

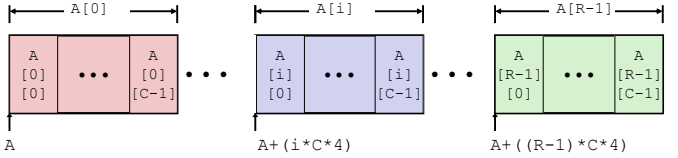


- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously

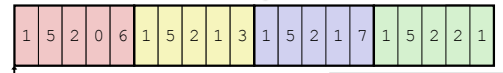
Nested Array Row Access

- Row Vectors
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A`** + *i* * (*C* * *K*)

```
int A[R][C];
```



Nested Array Row Access Code




```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

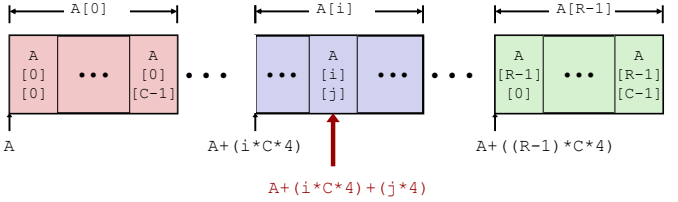
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- Machine Code
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

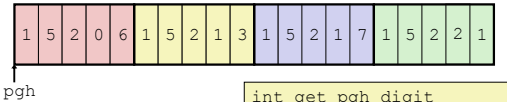
Nested Array Element Access

- Array Elements
  - **`A[i][j]`** is element of type *T,* which requires *K* bytes
  - Address **`A`** + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

```
int A[R][C];
```



Nested Array Element Access Code




```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

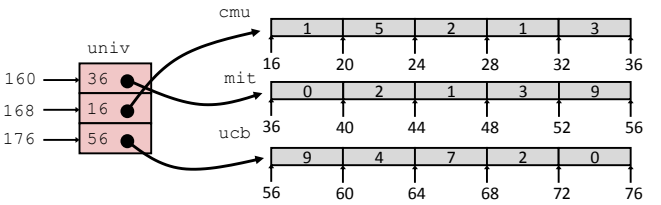
201

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

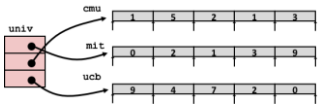
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



202

Element Access in Multi-Level Array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```




```
salq    $2, %rsi              # 4*digit
addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- Computation
  - Element access **`Mem[Mem[univ+8*index]+4*digit]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

203

Array Element Accesses

Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```

Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

204

N X N Matrix Code

- Fixed dimensions
  - Know value of N at compile time
- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays
- Variable dimensions, implicit indexing
  - Now supported by gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

205

16 X 16 Matrix Access

- **Array Elements**
  - Address **A** + i * (C * K) + j * K
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
  # a in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi              # 64*i
  addq    %rsi, %rdi            # a + 64*i
  movl    (%rdi,%rdx,4), %eax   # M[a + 64*i + 4*j]
  ret
```

206

n X n Matrix Access

- **Array Elements**
  - Address **A** + i * (C * K) + j * K
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi            # n*i
  leaq    (%rsi,%rdi,4), %rax   # a + 4*n*i
  movl    (%rax,%rcx,4), %eax   # a + 4*n*i + 4*j
  ret
```

207

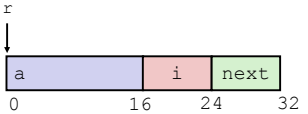
Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Unions
- Floating Point

208

Structure Representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

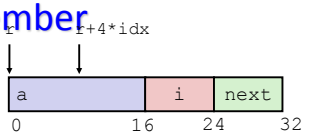


- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

209

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

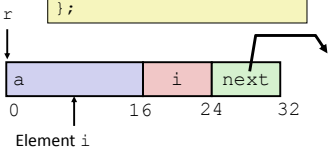
210

Following Linked List

- C Code

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```



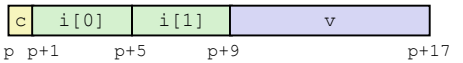
Register	Value
%rdi	r
%rsi	val

```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

211

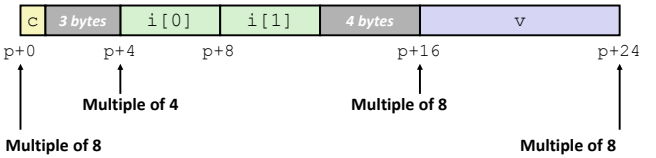
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



212

Alignment Principles

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

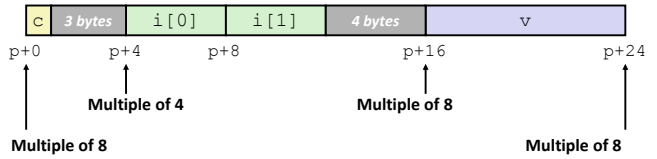
Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, `long`, **`char *`**, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- Example:
  - K = 8, due to **`double`** element

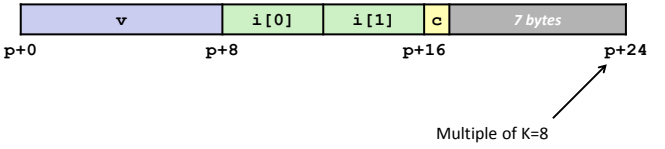
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

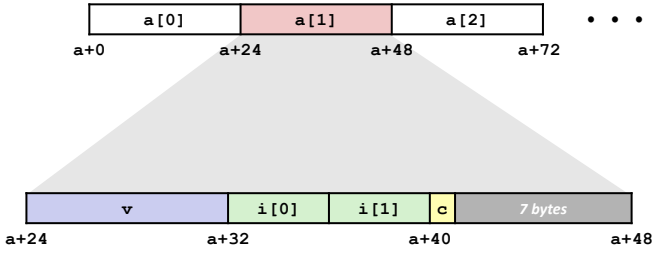
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

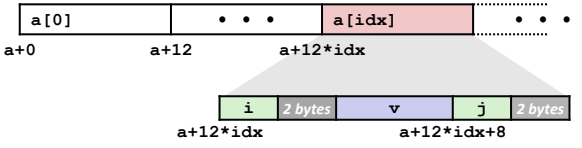


217

Accessing Array Elem

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- Compute array offset 12*idx
  - **`sizeof(S3)`**, including alignment spacers
- Element **`j`** is at offset 8 within structure
- Assembler gives offset **`a+8`**
  - Resolved during linking




```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

218

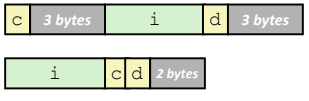
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)



219

Today

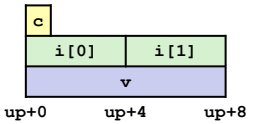
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Unions
- Floating Point

220

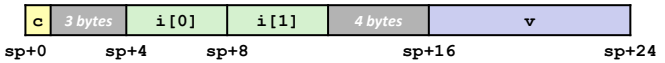
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



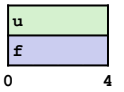

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



221

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as `(float) u`?

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(unsigned) f`?

222

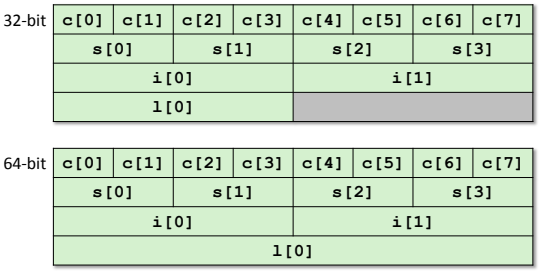
Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - Sparc
- Little Endian
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- Bi Endian
  - Can be configured either way
  - ARM

223

Byte Ordering Example

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



224

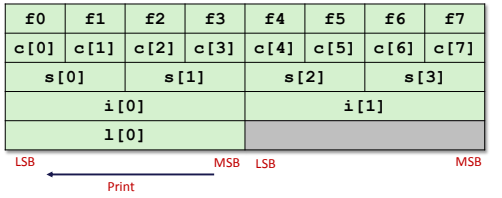
Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

225

Byte Ordering on IA32

Little Endian



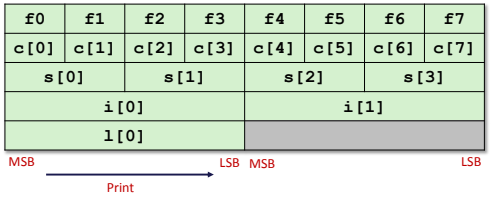
Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

226

Byte Ordering on Sun

Big Endian



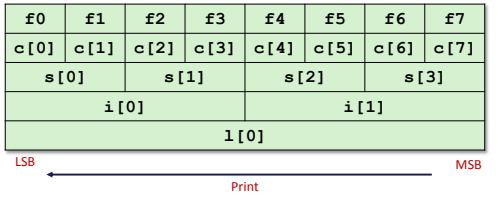
Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

227

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

228

Summary of Compound Types in C

- Arrays
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
  - Use index arithmetic to locate individual elements
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
  - Elements packed into single region of memory
  - Accessed using offsets determined by compiler
- Combinations
  - Can nest structure and array code arbitrarily
- Unions
  - Overlay declarations
  - Way to circumvent type system

Potential Exploits

- Buffer Overflow
  - Vulnerability
  - Protection

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
```

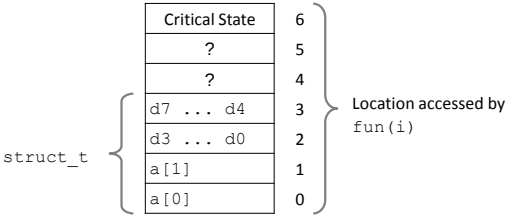
- Result is system specific

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
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

233

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

234

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:0123456789012345678901234
0123456789012345678901234
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012345
Segmentation Fault
```

235

Buffer Overflow Disassembly

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
```

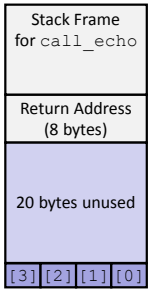
call_echo:

```
  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
```

236

Buffer Overflow Stack

Before call to gets



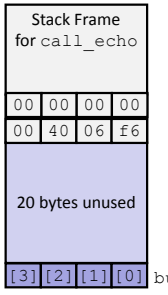

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

237

Buffer Overflow Stack Example

Before call to gets




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

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

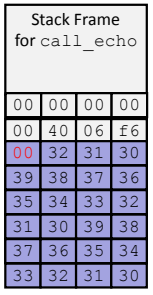
call_echo:

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

238

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
  . . .
```

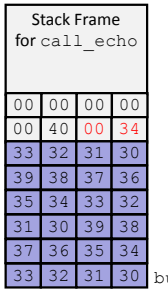
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

Overflowed buffer, but did not corrupt state

239

Buffer Overflow Stack Example #2

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
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

240

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
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:0123456789012345678901230123456789012345678901230
```

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

Buffer Overflow Stack Example #3 Explained

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
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

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`

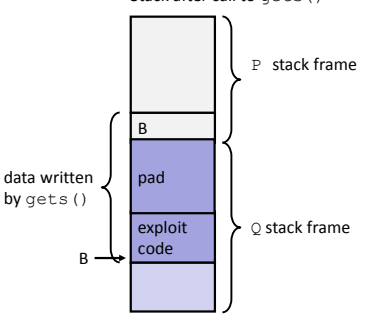
Code Injection Attacks

Stack after call to `gets()`

```
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*
- Distressingly common in real progams
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- You will learn some of the tricks in attacklab
  - Hopefully to convince you to never leave such holes in your programs!!

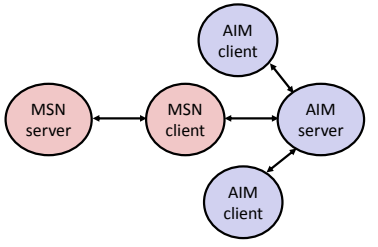
Example: the original Internet worm (1988)

- Exploited a few vulnerabilities to spread
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger "exploit-code padding new-return-address"`**
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet ☺)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…
  - and CERT was formed… still homed at CMU

245

Example 2: IM War

- July, 1999
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



246

IM War (cont.)

- August 1999
  - Mysteriously, Messenger clients can no longer access AIM servers
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes
    - At least 13 such skirmishes
  - What was really happening?
    - AOL had discovered a buffer overflow bug in their own AIM clients
    - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
    - When Microsoft changed code to match signature, AOL changed signature location

247

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

248

Aside: Worms and Viruses

- 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
- Both are (usually) designed to spread among computers and to wreak havoc

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each…

1. Avoid Overflow Vulnerabilities in Code (!)

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

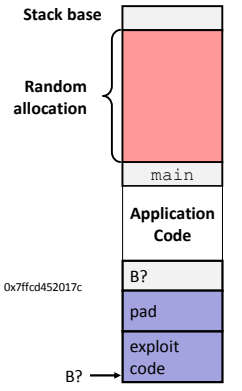
- For example, use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don’t use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

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
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

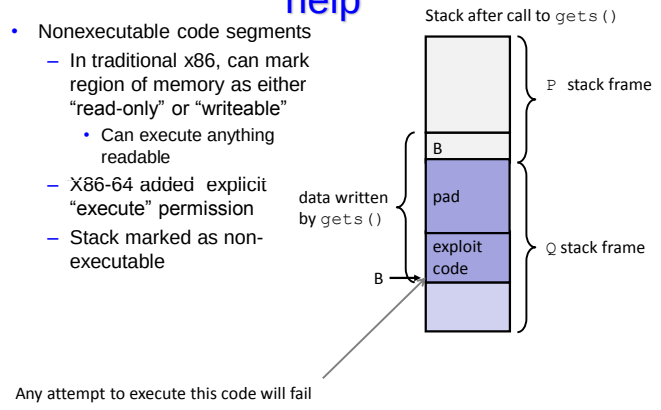
local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

  - Stack repositioned each time program executes



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



253

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 ***
```

254

Protected Buffer Disassembly

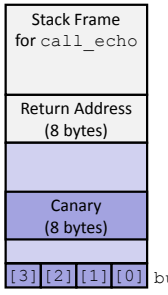
echo:

```
  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
```

255

Setting Up Canary

Before call to gets



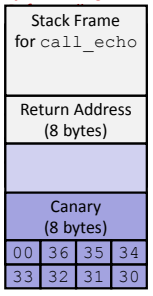

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

256

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:. . .
```

257

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*
- 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

258

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

259

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
```

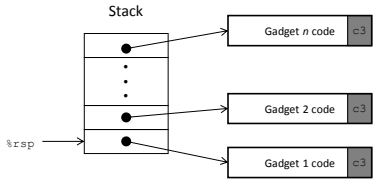
rdi ← rax

Gadget address = `0x4004dc`

- Repurpose byte codes

260

ROP Execution



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

261