CS 4400

Computer Systems

LECTURE 9

Structs and alignment

Buffer overflow

New to C?: Structures

- In C, a user-defined type is accomplished with a `struct`.
- *Example:*

```
struct element {
  char name[10];
  char symbol[5];
  float weight;
  float mass;
};
```

- The new type is `struct element`.
- Declaration of a structure variable

```
struct element e1;
```

allocates contiguous storage for all structure members.

at least 10 + 5 + 2 * `sizeof(float)` bytes

More on Structures

- To access a member of the structure variable, use the dot `.` operator.

```
e1.mass = 3.0;
strcpy(e1.name, "hydrogen");
```

- Use `typedef` to avoid the awkward two-word type.

```
typedef struct element {
  char name[10];
  char symbol[5];
  float weight;
  float mass;
} ELT;

ELT e1;
```

- What is the difference in a structure and an array?

Pointers to Structures

- As with objects in C++, the pointer operator `->` can be used with pointers to structures.

```
ELT e1;
ELT* elt_ptr = &e1;
printf("%s", (*elt_ptr).symbol);
printf("%s", elt_ptr->symbol);
```

- A self-referential structure has a member that is a pointer of the same type as the structure itself.

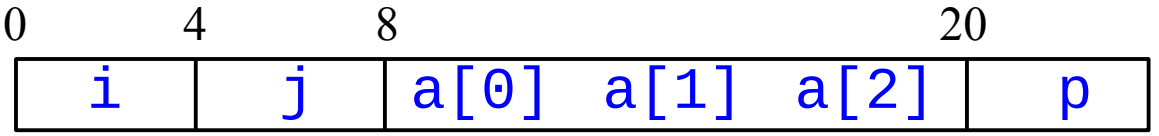
```
typedef struct node {
  int data;
  struct node* next;
} NODE;
... x->next->next->data ...
```

Structs

- The compiler maintains information about each structure.
  - indicating byte offset of each field

- *Example*:
```
struct rec {
  int i;
  int j;
  int a[3];
  int* p;
};
```

24 bytes:



- Generated code adds the appropriate offset.
  - suppose `r` (type `struct rec *`) is in `%edx`, to copy element `r->i` to element `r->j`:

```
movl (%edx),%eax
movl %eax,4(%edx)
```

Exercise: Structs

```
struct prob {
  int* p;
  struct {
    int x;
    int y;
  } s;
  struct prob* next;
};

void sp_init(struct prob* sp) {

  sp->s.x = _________________ ;

  sp->p = _________________ ;

  sp->next = _________________ ;
}
```

```
movl 8(%ebp),%eax
movl 8(%eax),%edx
movl %edx,4(%eax)
leal 4(%eax),%edx
movl %edx,(%eax)
movl %eax,12(%eax)
```

- Offset of each field? Total number of bytes?
- Fill in function, given assembly code for its body.

Clicker Question

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

What is the offset of field `f` in `struct d` ?

A. 0

B. 4

C. 8

D. 12

E. 16

F. none of the above

```
struct a {
  int b;
  int c;
};

struct d {
  struct a* e;
  float f;
};
```

New to C?: Unions

- Unions provide a way for a single object to be referenced according to multiple types.
- *Example:*

```
union u {
  char c;
  int i[2];
  double v;
} x;
x.v = 4.5;
printf("%d %d\n", x.i[0], x.i[1]);
```

- `sizeof(union u)` is the max size of any of its fields.
- Technically, you should only read the variant you wrote.

Unions

- The byte offset of each field is 0.
- *Example*:

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

8 bytes

- Assembly code lacks any information about type.

```
unsigned f2u(float f) {
  union {
    float f;
    unsigned u;
  } temp;
  temp.f = f;
  return temp.u;
}
```

```
movl 8(%ebp),%eax
```

Alignment

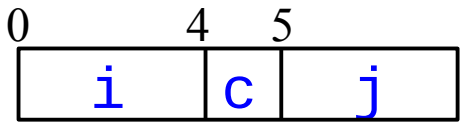
- Many systems restrict the addresses allowed for primitive types—they must be a multiple of *k*.
- Alignment restrictions simplify the interface between processor and memory.
  - avoids an 4-byte `int` straddling two 4-byte memory blocks
- Linux/IA32 alignment policy:
  - addresses of 1-byte data types are not restricted
  - addresses of 2-byte data types must be multiples of 2
  - addresses of larger data types must be multiples of 4

Struct Alignment

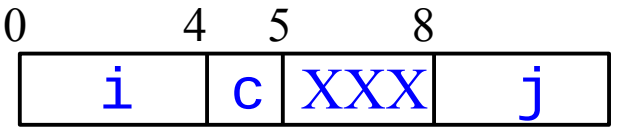
- The compiler may need to insert gaps in field allocation to ensure each structure element is aligned.
- *Example*:

```
struct S1 {
  int i;
  char c;
  int j;
};
```

9 bytes (unaligned):



12 bytes (aligned):



- Is a gap required if we make `char c` the third field?

Exercise: Struct Alignment

Given the Linux/IA32 alignment policy, how is each structure aligned?

- `struct P1 { int i; char c; int j; char d; };`
- `struct P2 { int i; char c; char d; int j };`
- `struct P3 { short w[3]; char c[3]; }`
- `struct P4 { short w[3]; char* c[3]; }`
- `struct P5 { struct P1 a[2]; struct P2 *p };`

Clicker Question

Given the Linux/IA32 alignment policy, what is the total number of bytes required for `s`?

A. 12

B. 16

C. 20

D. 24

E. 28

F. none of the above

```
struct {
  char a[3];
  short b;
  double c;
  char* d;
} s;
```

Clicker Question

If reordering of fields is allowed, is it possible to avoid padding at all in `s`?

A. yes

B. no

C. I don't know

```
struct {
  char a[3];
  short b;
  double c;
  char* d;
} s;
```

Out-of-Bounds Memory References

- C does no bounds checking for array references.
  - Do any programming languages perform bounds checking?
- Recall that the run-time stack is used to store local variables, as well as, register values and return address.
- What happens when an out-of-bounds element of a local array is written?
  - program “state” is potentially corrupted
  - examples?

Buffer Overflow

- A common source of state corruption.
- *Typically*: A `char` array is allocated to the stack, but a string is written which exceeds the allocated space.

```
char* gets(char* s) {
  int c; char* dest = s;
  while((c=getchar()) != '\n' && c != EOF)
    *dest++ = c;
  *dest = '\0';
  if(c == EOF)
    return NULL;
  return s;
}

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

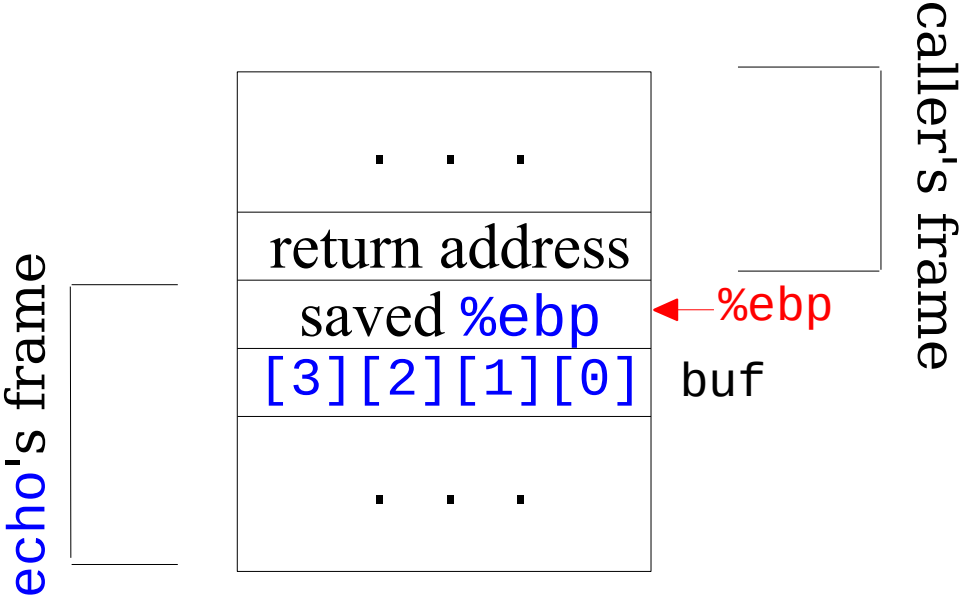
Any potential problems with `gets`?

Example: Buffer Overflow

```
echo:
  pushl %ebp           ;save to stack
  movl %esp,%ebp       ;set new fr_ptr
  subl $20,%esp        ;alloc space
  pushl %ebx           ;save to stack
  addl $-12,%esp       ;alloc more space
  leal -4(%ebp),%ebx   ;buf is %ebp-4
  pushl %ebx           ;push buf
  call gets
```

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

- What values of `buf` will corrupt the saved value of `%ebp`?
- What values will corrupt the return address?
- How can buffer overflow be avoided in this example?



Exploit Code

- When the byte encoding of executable code is fed into a program as an input string, buffer overflow can be used to get a program do something it otherwise would not.
  - Also include extra bytes to overwrite the return address with the address of this exploit code.
  - The effect of `ret` is to jump to (and execute) the exploit code.
- In Lab 3, you will get first-hand experience mounting a buffer-overflow attack.
  - Requires deep understanding of run-time stack organization, byte ordering, and instruction encoding.

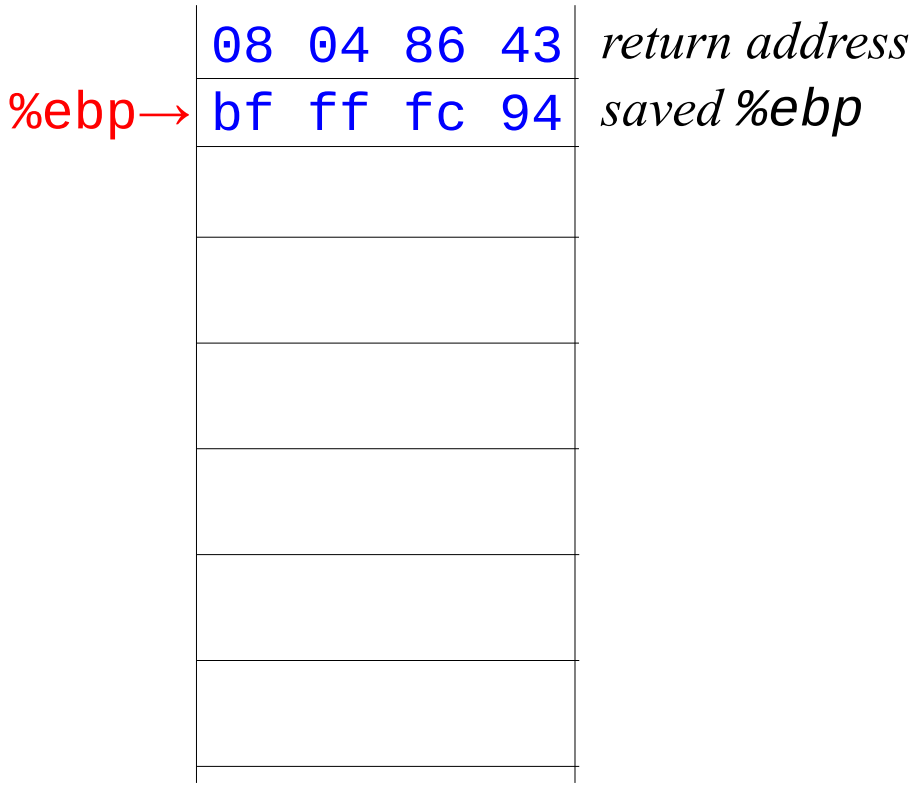
Exercise: Buffer Overflow

```
char* getline() {
   char buf[8];
   char* result;
   gets(buf);
   result = malloc(strlen(buf)+1);
   strcpy(result, buf);
   return result;
}
```

Disassembly of `getline`*:*

```
push %ebp
mov %esp, %ebp
sub $0x10,%esp
push %esi
push %ebx
add $0xfffffff4,%esp
lea 0xfffffff8(%ebp),%ebx
push %ebx
... call gets ...
```

%esi 0x1 %ebx 0x2



- If input is `012345678901`, program terminates with seg-fault. Error occurs during return of `getline`.
- Fill in stack just before `add`, and then after call to `gets`.
- To where does the program try to return?
- What registers have corrupted values?