CSE 351 Section 6

Buffer Overflow, Lab 3, Structs, Arrays

Administrivia

- **Midterm!!!**
  - Due Friday, Feb 11 at 11:59 PM (and zero seconds!!!)
  - No lecture this Friday tomorrow :D
  - Use the midterm reference sheet!
- **Homework 15.2 (Memory & Caches I)**
  - Due Monday, Feb 14
- **Lab 3**
  - Due Wednesday, Feb 16

Structs

What is a Struct?

Structs are contiguously allocated chunks of memory that hold a programmer-defined collection of potentially disparate variables.

A general form and example of a struct declaration:

```
struct name {
  type field_1;
  type field_2;
  …
  type field_n;
};
```

```
typedef struct st {
  int id;
  struct st* next;
  short vals[3];
} Node;
```

typedef so we can type “Node” instead of “struct st” everywhere!

Using Structs

A struct allows us to "make our own data types" consisting of collections of smaller pieces of data. We can declare, get the address of, dereference, etc. a struct just like any other variable.

```
typedef struct st {
  int id;
  short vals[3];
  struct st* next;
} Node;
```

```
struct su {
  int key;
  int value;
} pair;
```

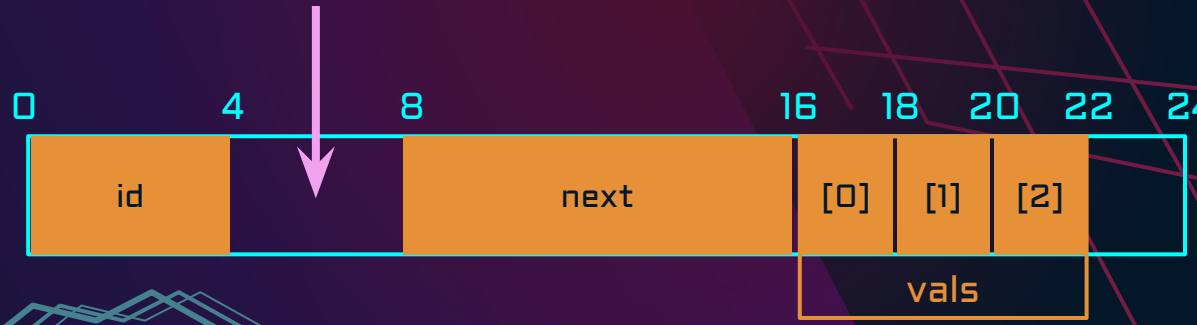
```
struct st a;
Node b;
Node* bp = &b;
a.id = pair.key;
b.vals[2] = 16;
bp->next = a
a = *bp;
```

Structs in Memory

In memory, the individual fields appear in the struct in the order that they are declared!

Each field follows its own alignment requirement (typically the size of one element of the data type). If we must put a space in between fields, that's ***internal fragmentation***.

```
typedef struct st {
  int id;
  struct st* next;
  short vals[3];
} Node;
```



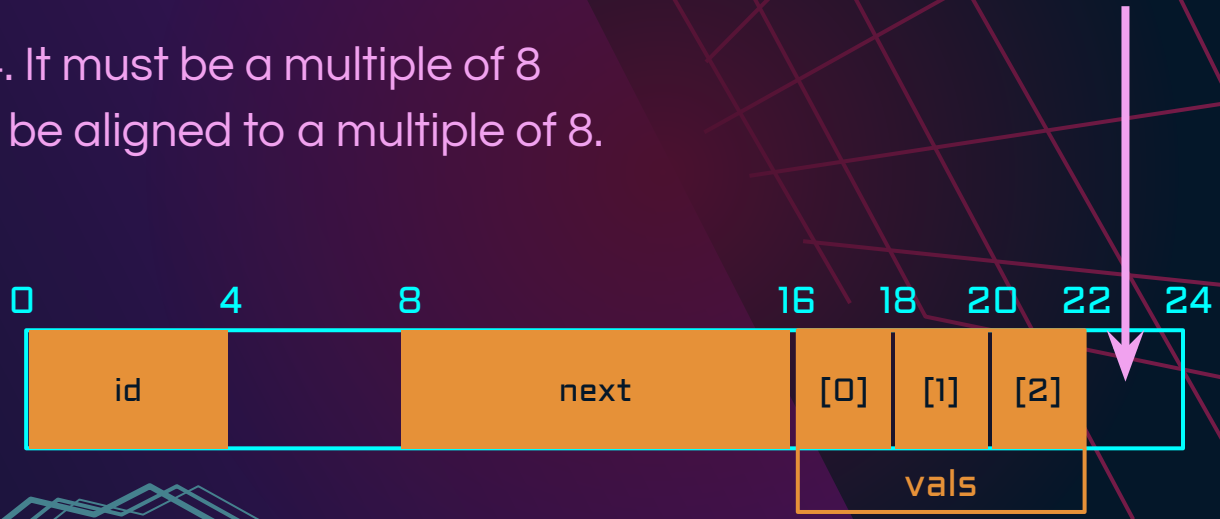
Structs in Memory

The entire struct is aligned to the size of its largest field requirement.

To fill up any empty space at the end, we add ***external fragmentation!***

The size of this struct is 24. It must be a multiple of 8 because the pointer must be aligned to a multiple of 8.

```
typedef struct st {
  int id;
  struct st* next;
  short vals[3];
} Node;
```



Exercise 1

```
struct student {
  int id;
  char* name;
  char age;
};
```

Which bytes correspond to which field? How much fragmentation? What is the total size? Is it possible to reorder fields to reduce this?

Exercise 2

a) Give the following quantities as number of bytes:
   i) `sizeof(cse_building)`
   ii) Internal fragmentation
   iii) External fragmentation
b) More efficient space arrangement?

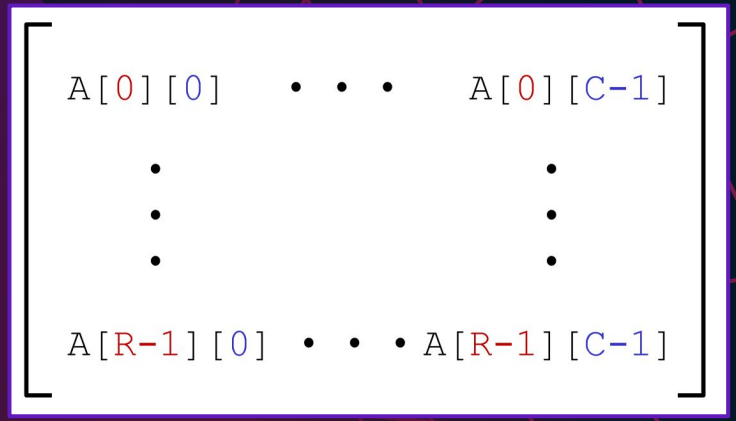
```
typedef struct {
  int num_donors;
  char** donor_names;
  char has_soft_seats;
  char location_name[10];
  short height;
} cse_building;
```

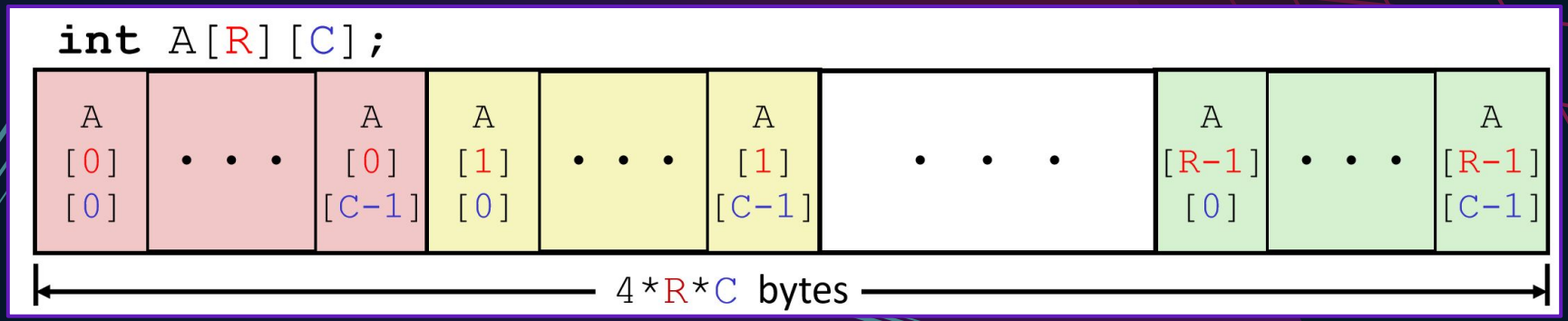
Arrays

Arrays

Multidimensional Arrays

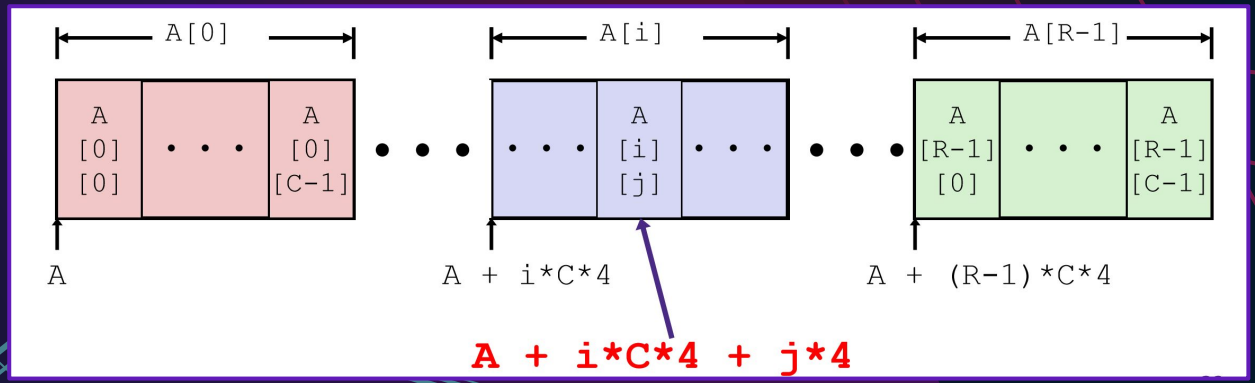
- `T A[R][C];`
  - 2D array of type T with R rows and C columns
  - Row-major
  - Contiguously
- Can extend to more dimensions
  - Add more brackets: `T A[X][Y][Z];`





Multidimensional Arrays

- `T A[R][C];`
  - A still returns a pointer to the array
- `A[i]` gets a ***pointer*** to a row of the array
  - The same as `A + i * (C * sizeof(T))`
- `A[i][j]` gets an ***element*** of the array
  - The same as `*(A[i] + j * sizeof(T))`

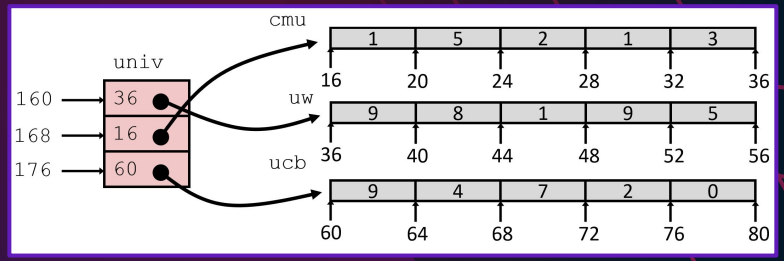


Multilevel Arrays

- `T* A[R];`
  - Array of pointers to arrays
- Same indexing ***notation*** as multidimensional arrays
- `univ[0]` gets a pointer to `uw`
  - `*(univ + 0 * sizeof(int*))`
- `univ[0][1]` gets the second element of `uw`
  - `*(univ[0] + 1 * sizeof(int))`
  - Notice we have 2 dereferences!

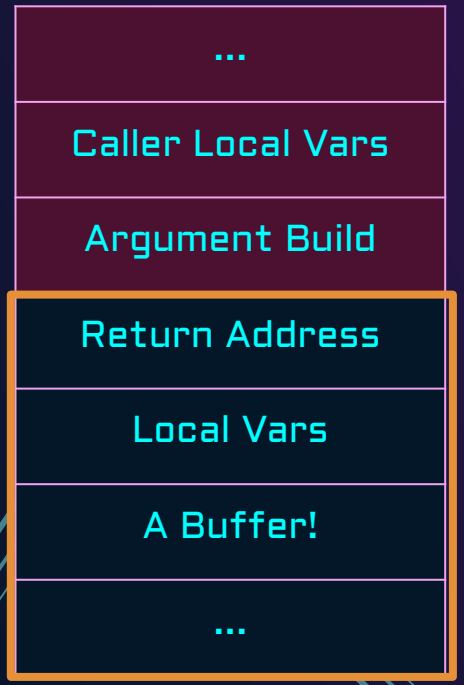
```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = { uw, cmu, ucb };
```



Buffer Overflow

Buffer Overflow Review

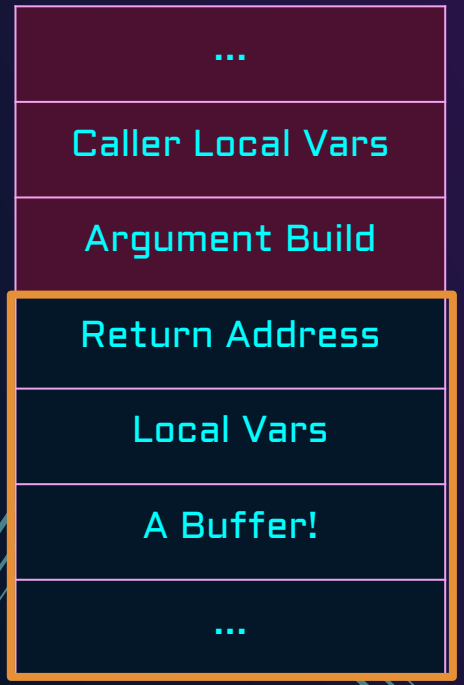


Simplified Memory Layout

Current Stack Frame

What can we overwrite and how might that affect the execution of our program?

Buffer Overflow Review



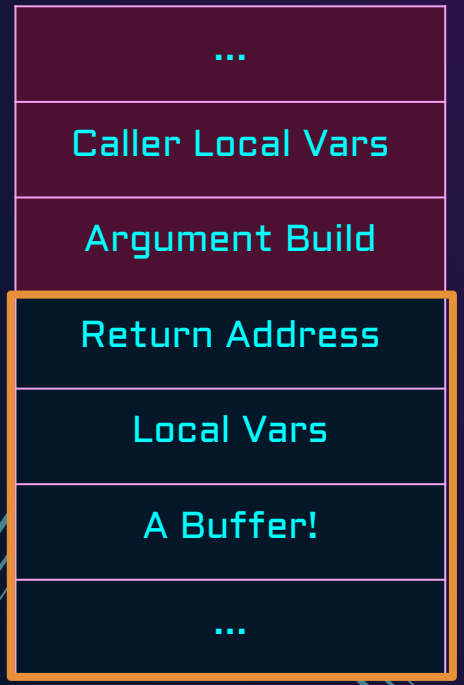
Simplified Memory Layout

Current Stack Frame

What can we overwrite and how might that affect the execution of our program?

How can we defend against it?

Buffer Overflow Review



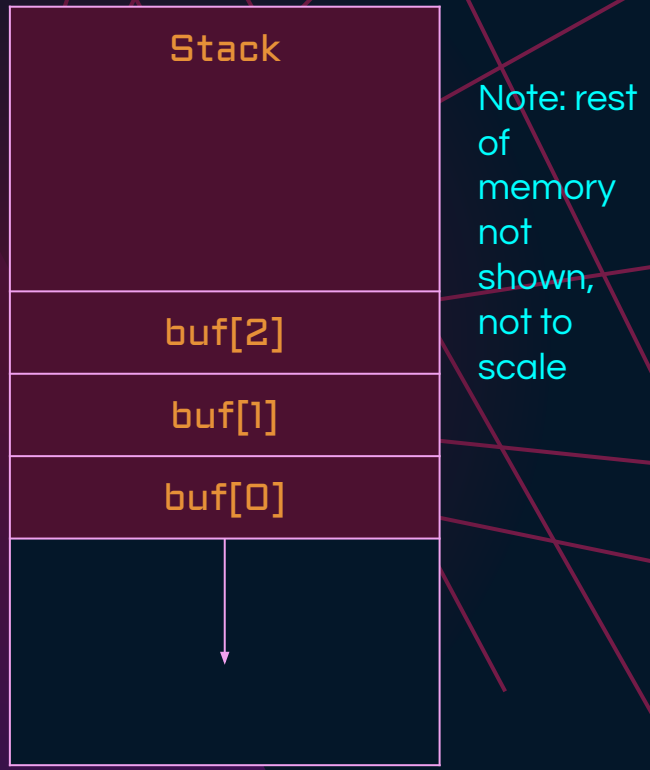
What can we overwrite and how might that affect the execution of our program?

How can we defend against it?

- Stack canaries
- Non-executable segments
- “Safe” functions or languages

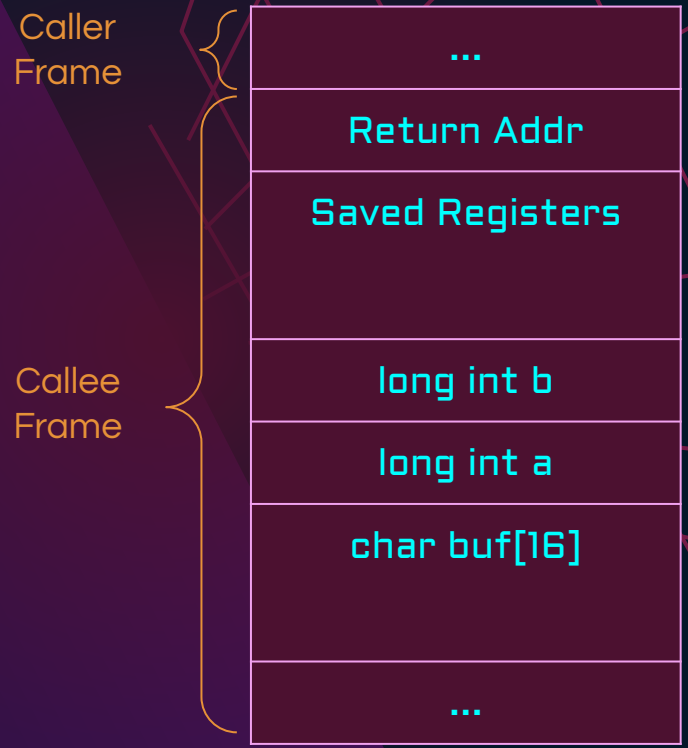
Memory Addresses and Arrays

- Stack is “upside down”
  - Top of stack is at lower addresses than rest of stack
- Each additional element in an array is at a higher address than the previous
  - Tip: Think about pointer arithmetic!
  - `buf[i] = *(buf + i)`
- Since each element is higher than the previous, writing to the buffer writes “up” toward the return address



Stack Layout

- To which byte does buf[17] refer to in this example?
- In buffer overflow *attacks*, malicious users pass values to attempt to overwrite important parts of the stack or heap
- E.g. An attacker could overwrite the return instruction pointer with the address of a malicious block of code



Protecting Against Overflows

- **f**gets(char* s, int size, FILE* stream)
  - Takes a size parameter and will only read that many bytes from the given input stream
- str**n**cpy(char* dest, const char* src, size_t n)
  - Will copy at most n bytes from `src` to `dest`
- **Stack canaries**
  - Use a random integer before return instruction pointer
  - Check if tampered
- **Data execution prevention**
  - Mark some parts of the memory (notably the stack) as non-executable

Exercise 3

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Address	Value (Hex)
%rsp + 15	00
%rsp + 14	00
%rsp + 13	00
%rsp + 12	00
%rsp + 11	00
%rsp + 10	40
%rsp + 9	AF
%rsp + 8	3B
%rsp + 7	
%rsp + 6	
%rsp + 5	
%rsp + 4	
%rsp + 3	
%rsp + 2	
%rsp + 1	
%rsp + 0	

Lab 3 Intro

Handout!

There is a handout on ed that walks you through the first phase of lab 3!

Make sure you're familiar with the `sendstring` utility: how we use it, and why we need it.

Lab 3: Understand the Tools

- `sendstring` - Use to generate your malicious strings
  - Takes a list of space-separated hex values and formats them in raw bytes suited for exploits
- `gdb` - You will use this a lot to inspect your code
  - `set args -u <username>`
    - Set the argument to the program
  - `x/40wx ($rsp - 40)`
    - Show the 40 bytes above rsp
    - Change `w` to `g` to check the value in 8-byte chunks
  - `b * (&getbuf + 12)`
    - Create a breakpoint at 12 bytes away after the start of getbuf
- `bufbomb -u [UW_NetID]` --- Everyone's lab is different
  - Your username alters the lab slightly

Level 0: Call smoke()

Goal: call the smoke() function from getbuf()

```
int getbuf() {
    char buf[36];
    gets(buf);
    return 1;
}
```

How?

- Overwrite the return address so we "return" to smoke()

The stack in getbuf()

36 bytes

That's All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to office hours).

See you all next week!