Lecture 9
Assertions and Error Handling

CS240

The C preprocessor

- The C compiler performs *Macro expansion and directive handling*
  - Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros
- Example:
  - Specify how you want to macro expand by specifying the DEBUG variable at compilation time in the Makefile
  - gcc -D option

```
#ifdef DEBUG
#define DPRINT(s) fprintf(stderr,”%s\n”,s)
#else
#define DPRINT(s)
#endif
```

The C preprocessor

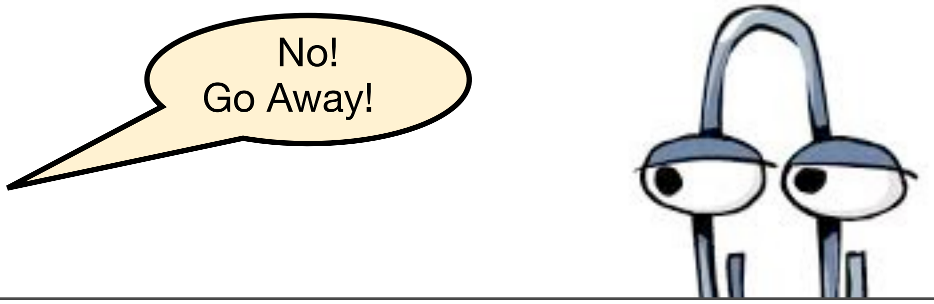
- The C compiler performs *Macro expansion and directive handling*
    - Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros

```
DPRINT( makeString(“error”, CAUSE) );
```

- Question:
    - *Is function makeString() called?*

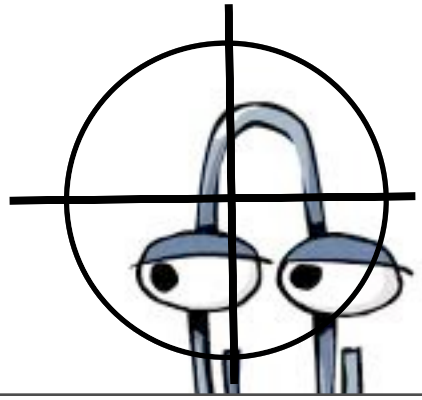
How can this code fail?

```
  #include <stdio.h>

int main() {
   int a, b, c;

   a = 10;
   b = getchar() - 48;
   c = a/b;

   return 0;
}
```



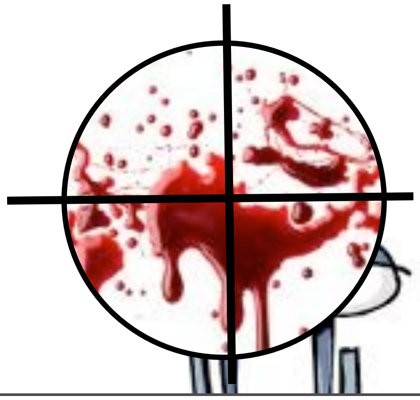
Common Software Vulnerabilities

- Buffer overflows
- Input validation
- Format string problems
- Integer overflows
- Failing to handle errors
- Other exploitable logic errors



Weak Types and Errors

- Would strong typing prevent these kinds of vulnerabilities?
  - What kind of errors do type systems typically catch?
    - Structural violations: think of types as sets
- Not all elements in a set are sensible in all contexts
  - Think of buffers as arrays
    - Buffer overflow arises when arrays of different sizes than expected are constructed.
  - Similar reasoning for overflow and underflow
- Failure to enforce temporal and logical relations



Is C more vulnerable than...

- Weak typing means data can be interpreted in multiple ways
  - This can lead to errors
  - A single datum associated with multiple types
- Memory can be indexed arbitrarily, beyond the range(s) of declared arrays and structures
  - Overwrite stack contents
- Dangling pointers
  - Pass a pointer to an object allocated locally within a function to the function's caller

What is a Buffer Overflow?

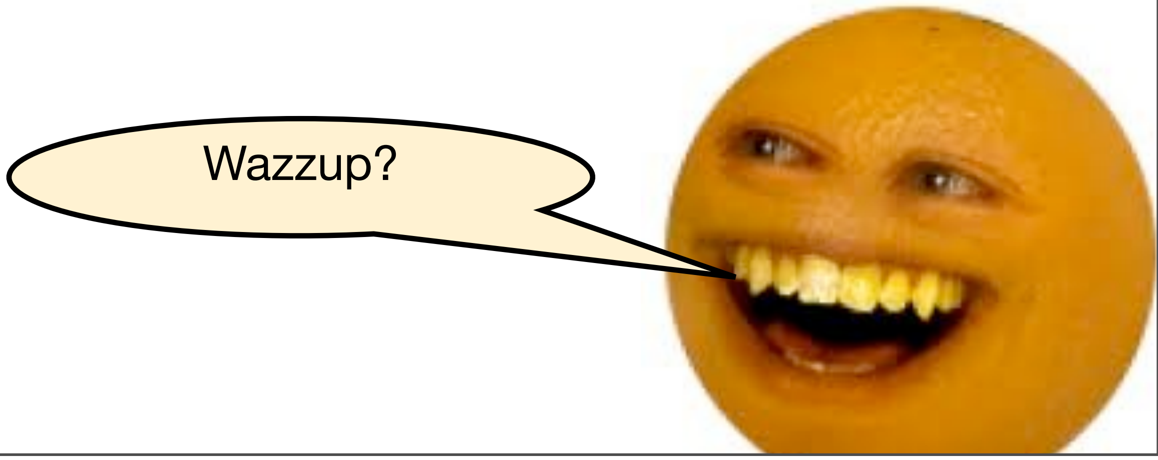
- Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold
- Very dangerous because the extra information may:
  - Affect user's data
  - Affect user's code
  - Affect system's data
  - Affect system's code

Why Buffer Overflows?

- No check on boundaries
  - Programming languages give user too much control
  - Programming languages have unsafe functions
  - Users do not write safe code
- C and C++, are more vulnerable because they provide no built-in protection against accessing or overwriting data in any part of memory
  - Can’t know the lengths of buffers from a pointer
  - No guarantees strings are null terminated

Why Buffer Overflow Matter

- Overwrites:
  - other buffers
  - variables
  - program flow data
- Results in:
  - erratic program behavior
  - a memory access exception
  - program termination
  - incorrect results
  - breach of system security

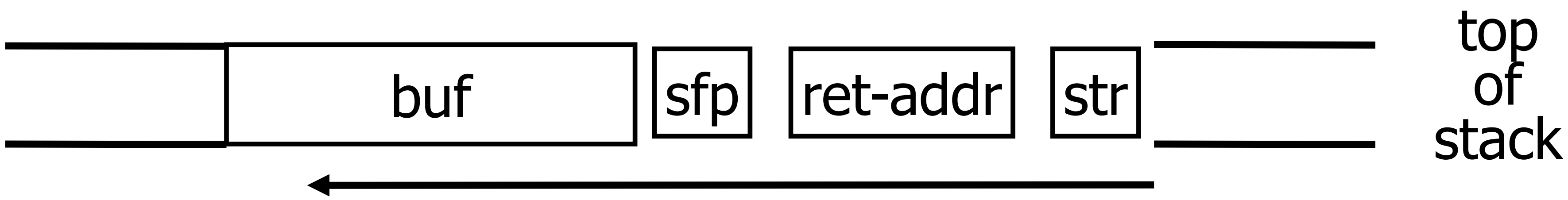


Example

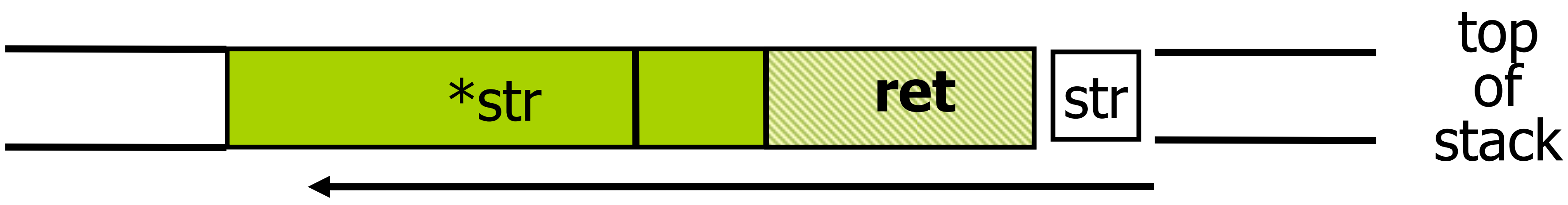
- Suppose a web server contains a function:

```
void f(char *str) {
  char buf[128];
   strcpy(buf, str);
   do(buf);
}
```

- When the function is invoked the stack looks like:



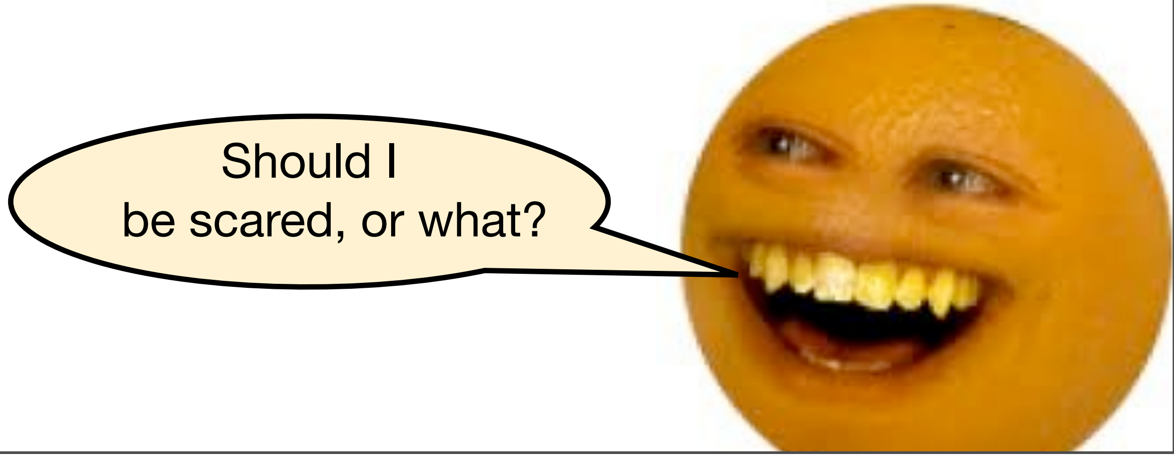
- What if `*str` is 136 bytes long? After `strcpy`:



Some Unsafe C lib Functions

```
  strcpy (char *dest,  const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf ( const char *format, … )
printf (conts char *format, … )
```

⋮

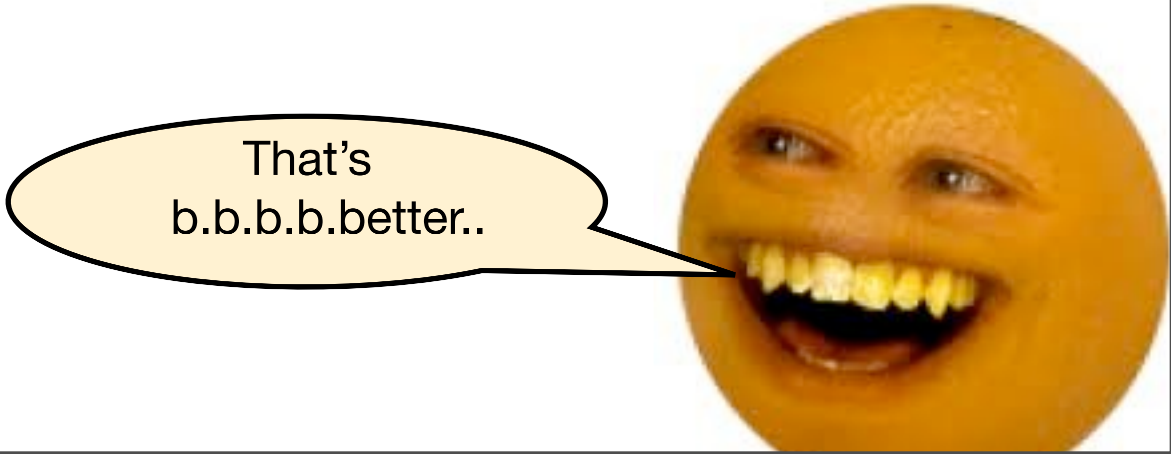


Assertions

```
 int main() {
int a, b, c;

a = 10;
b = some_function_computes_something();

c = a/b;
return 0;
}
```

Assertions

```
int main() {
 int a, b, c;

 a = 10;
 b = some_function_computes_something();
 assert(b!=0);
 c = a/b;
 return 0;
}
```



Assertions

- Used to help specify programs and to reason about program correctness.
- precondition
  - an assertion placed at the beginning of a section of code determines the set of states under which the code is expected to be executed.
- postcondition
  - placed at the end – describes the expected state at the end of execution.
- #include <assert.h>

```
assert (predicate);
```

Examples

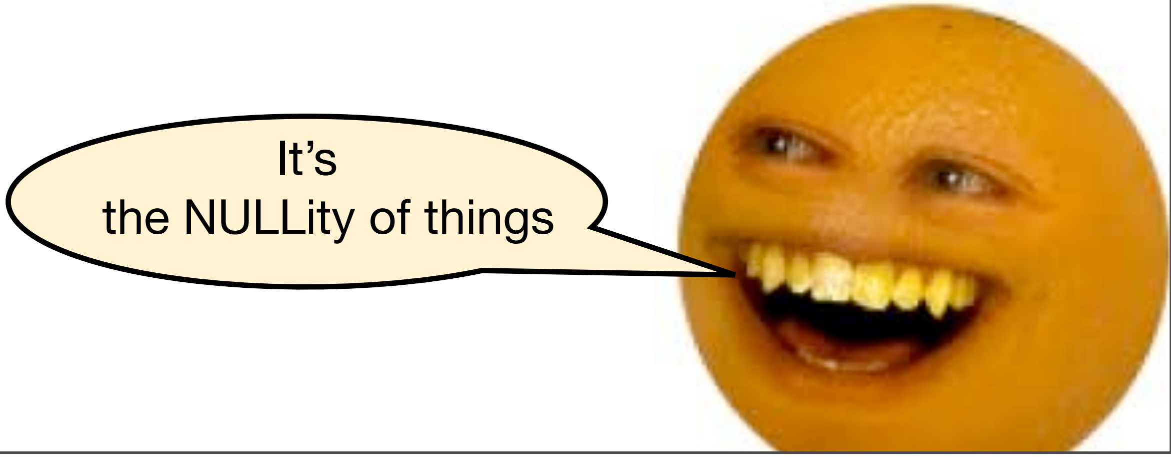
```
 (assert b!=0);
c = a/b
```

- At the end of a function, if you know you should return success

```
assert(ret == SUCCES);
```

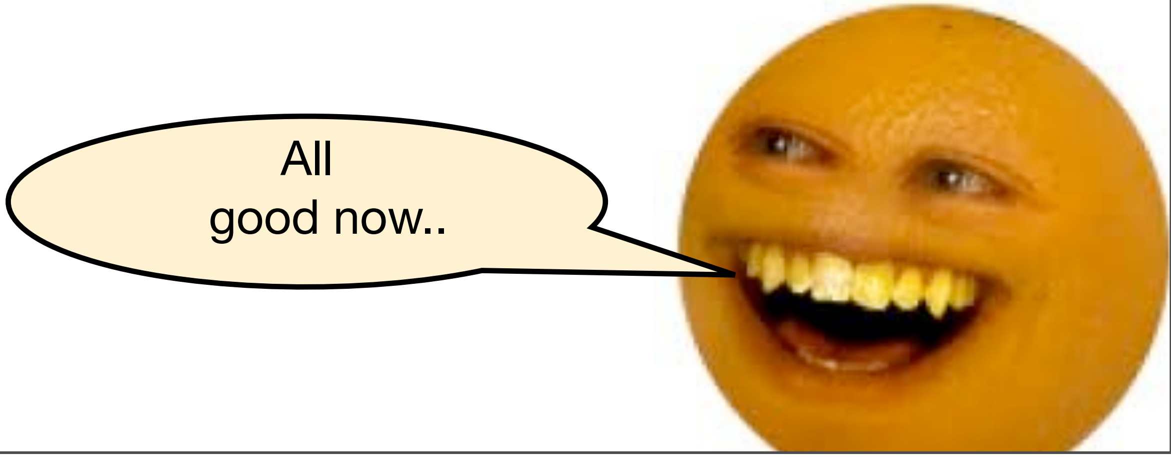
How can this code fail?

```
#include <stdio.h>

#define MAX 10
char* f(char s[]);

int main() {
  char str[MAX];
  char *ptr=f(str);

  printf("%c\n", *ptr);
  return 0;
}

char *my_function(char s[]) {
  char *p = NULL;
  /* does stuff*/
  return p;
}
```



How can this code fail?

```
#include <stdio.h>
#include <assert.h>
#define MAX 10
char* f(char s[]);

int main() {
  char str[MAX];
  char *ptr=f(str);
  assert(ptr!=NULL);
  printf("%c\n", *ptr);
  return 0;
}

char *my_function(char s[]) {
  char *p = NULL;
  /* does stuff*/
  return p;
}
```



What to think about/check for...

- Null pointer dereference
- Use after free
- Double free
- <u>Array indexing errors</u>
- Mismatched array new/delete
- Potential stack/heap overrun
- <u>Return pointers to local variables</u>
- <u>Logically inconsistent code</u>
- <u>Uninitialized variables</u>
- <u>Invalid use of negative values</u>
- <u>Passing large parameters by value</u>
- Under allocations of dynamic data
- Memory leaks
- File handle leaks
- Unhandled return codes
- <u>Use of invalid iterators</u>