CS240: Programming in C

Lecture 10: Assertions and Error Handling.

PURDUE UNIVERSITY

Application of #ifndef: include files

- To include a include file only once

```
#ifndef _MY_INCLUDE_FILE_
#define _MY_INCLUDE_FILE_
 header file

#endif /* _MY_INCLUDE_FILE_ */
```

Application of #ifdef: Print debug information

```
#ifdef DEBUG
#define DPRINTF(args) printf args
#else
#define DPRINTF(args)
#endif
```

- Specify how you want to macro to expand by specifying the DEBUG variable at compilation time in the Makefile
- gcc -D option

How can this code fail?

```
#include <stdio.h>

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

  a = 10;
  b = getchar() - 48; /* ascii for 0 is 48*/
  c = a/b;

  return 0;
}
```

ALWAYS VALIDATE USER INPUT

- Never assume that user input is correct
- For the example before
  - Check that b !=0
- SO what if b is 0? What now?

Common Software Vulnerabilities

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

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 Does Buffer Overflow Happen?

- 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 Matters

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

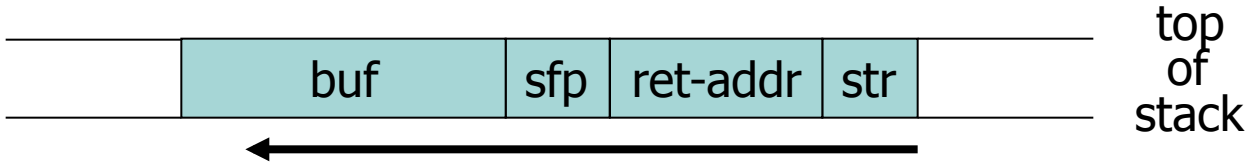
Example of a Stack-based Buffer Overflow

- Suppose a web server contains a function:

```
void my_func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(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, …)

How can this code fail: Part 2

```
#include <stdio.h>

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

  a = 10;
  b = some_function_computes_something();
  c = a/b;

  return 0;
}
```

Assertions

```
#include <stdio.h>
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

- In a function, at the end of a function, if you know at that point you should return success

- assert(ret == SUCCES);

How can this code fail?

```
#include <stdio.h>
#define MAX 10

char * my_function(char s1[]);

int main() {
  char *ptr;
  char str1[MAX];

  ptr = my_function(str1);
  printf("%c\n", *ptr);

  return 0;
}

char *my_function(char s1[]) {
  char *p = NULL;

  /* does stuff*/

  return p;
}
```

How can this code fail?

```
#include <stdio.h>
#include <assert.h>
#define MAX 10

char * my_function(char s1[]);

int main() {
  char *ptr;
  char str1[MAX];

  ptr = my_function(str1);
  assert (ptr != NULL);
  printf("%c\n", *ptr);

  return 0;
}

char *my_function(char s1[]) {
  char *p = NULL;

  /* does stuff*/

  return p;
}
```

What to think about/check for …

- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code

- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Under allocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators