CS 3410 Lab 3.5

Spring 2026

Cornell Bowers C·IS
Computer Science

Agenda

1	C Review
2	Address Sanitization
3	GDB Intro
4	GDB Exercise

Cornell Bowers C·IS
Computer Science

C Review

Address Sanitization

Out-of-bounds memory read

This code reads from uninitialized memory!

```c
#include <stdio.h>

int main() {
    int x[] = {42, 3410};
    printf("%d", x[2]);
}
```

Running and compiling normally prints an arbitrary value like "1440050536"

Why did it print garbage?

- `x[2]` is `*(x + 2)`

- x only has two elements…

- `x[2]` will read out of bounds!

- But this is still totally valid
    - Memory is bytes, and bytes form an `int`

- Address sanitization (ASan) adds checks

Address	Variable	Value
0x12	x	0x42
0x3E	--	0
0x42	x[0]	42
0x46	x[1]	3410
0x4A	--	1440050536

garbage

Cornell Bowers C·IS
Computer Science
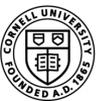
Address sanitization to the rescue!

Enable ASan with the `-fsanitize=address` gcc flag and rerun…

```
> rv gcc -fsanitize=address bad.c && rv qemu a.out
================================================================
==1==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x001557b09028 at pc
 0x000000010af8 bp 0x001555d56b50 sp 0x001555d56b38
READ of size 4 at 0x001557b09028 thread T0
    #0 0x10af6 in main (/root/a.out+0x10af6)
    #1 0x15564bb922 in __libc_start_call_main (/lib/libc.so.6+0x2b922)
    #2 0x15564bba0e in __libc_start_main@GLIBC_2.27 (/lib/libc.so.6+0x2ba0e)
    #3 0x108aa in _start (/root/a.out+0x108aa)

Address 0x001557b09028 is located in stack of thread T0 at offset 40 in frame
    #0 0x1095a in main (/root/a.out+0x1095a)

  This frame has 1 object(s):
    [32, 40) 'x' (line 4) <== Memory access at offset 40 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind me
chanism, swapcontext or vfork
    (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/root/a.out+0x10af6) in main
```

ASan found the out of bounds read!

Cornell Bowers C·IS
Computer Science

GDB Intro

Introduction to GDB

- GDB can start, stop, and inspect the execution of a program (and more!)

- After initial setup, GDB will prompt you to enter commands

```
Reading symbols from a.out...
(gdb) <you would enter a gdb command here>
```

- A "breakpoint" is a line of source code where you would like to stop execution

- Typical usage: set breakpoint → run until it's hit → inspect program state

Note: using GDB is a bit tricky with rv, read our setup instructions carefully!

Cornell Bowers C·IS
Computer Science

Common GDB commands

- **break** *func* and **break** *line* set breakpoints (**b** for short)

- **next** (**n**), **step** (**s**), **continue** (**c**), and **finish** move the debugger forward

- **info locals** shows variables, **info args** shows arguments

- **print** *expr* will evaluate *expr* and print the result (**p** for short)
  - **print/a** *expr* interprets *expr* as an address, **/t** does binary
  - **print *arr@***num* will print the first *num* elements of **arr**

- **list** shows the next 10 lines around the debugger location

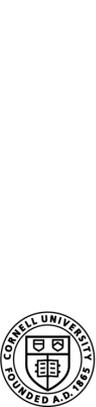Cornell Bowers C·IS
Computer Science

Stepping through a bugged program

- This simple program has a bug!

- Should print "a = 6, b = 5"

- Actually prints "a = 6, b = 6"

- We will debug this with GDB

```c
#include <stdio.h>

void swap(int *x, int *y) {
    *x = *y;
    *y = *x;
}

int main() {
    int a = 5;
    int b = 6;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```

Cornell Bowers C·IS
Computer Science

Stepping through a bugged program

Our debugging workflow:

1. Set breakpoint on main

2. Continue execution until main

3. Step until inside swap

```
(gdb) break main
Breakpoint 1 at 0x105ac: file bug.c, line 9.
(gdb) continue
Continuing.

Breakpoint 1, main () at bug.c:9
9          int a = 5;
(gdb) step
10         int b = 6;
(gdb) step
11         swap(&a, &b);
(gdb) step
swap (x=0x1555d56bbc, y=0x1555d56bb8) at bug.c:4
4          *x = *y;
```

Cornell Bowers C·IS
Computer Science

Stepping through a bugged program

Our debugging workflow:

1. Set breakpoint on main

2. Continue until main

3. Step until inside swap

4. Print values when stepping

5. Show all local variables

We found the bug!

```
(gdb) print *x
$1 = 5
(gdb) print *y
$2 = 6
(gdb) step
5             *y = *x;
(gdb) print *x
$3 = 6
(gdb) print *y
$4 = 6
(gdb) step
6         }
(gdb) step
main () at bug.c:12
12            printf("a = %d, b = %d\n", a, b);
(gdb) info locals
a = 6
b = 6
```

Cornell Bowers C·IS
Computer Science

13

That's 1% of GDB's true power…

- This usage was similar to print statement debugging

- GDB can do much more, such as:

    - Stop program execution when a condition is true

    - Inspect the state of the program, such as viewing and searching memory

    - Print a backtrace and inspect variables after a segfault

    - View the assembly corresponding to the current instruction

Cornell Bowers C·IS
Computer Science

GDB Exercise

Common GDB commands, for reference:

- **break *func*** and **break *line*** set breakpoints (**b** for short)

- **next** (**n**), **step** (**s**), **continue** (**c**), and **finish** move the debugger forward

- **info locals** shows variables, **info args** shows arguments

- **print *expr*** will evaluate ***expr*** and print the result (**p** for short)
    - **print/a *expr*** interprets ***expr*** as an address, **/t** does binary
    - **print *arr@*num*** will print the first ***num*** elements of **arr**

- **list** shows the next 10 lines around the debugger location

Cornell Bowers C·IS
Computer Science