CS 3410 Lab 7

Spring 2026

Cornell Bowers CIS
Computer Science

Agenda

1 Intro to Buffer Overflow

2 Memory & Stack Layout

3 Useful GDB commands

4 Endianness

Intro to Buffer Overflow

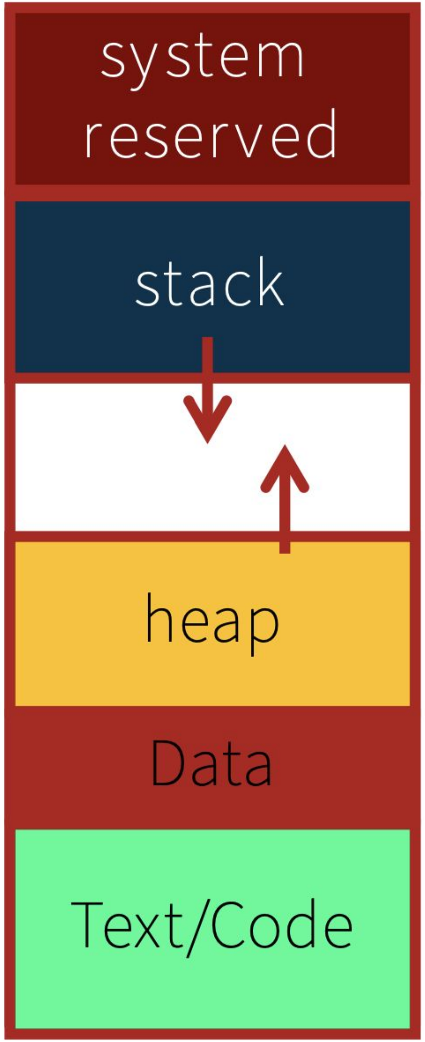
Buffer Overflow

- Well-known security vulnerability that can be used to effectively exploit unprotected programs.
- Occurs when some user input is written to a fixed length buffer and the size of the user input exceeds the size of the buffer being written into.
- If this happens, the extra input overflows into adjacent memory locations on the stack, overwriting their data.
- Can cause major problems if important data like the sp or ra are overwritten.
- E.g - overwriting the ra can lead to a segfault (caused by attempting to access restricted / nonexistent memory addresses).
- Hackers can use these vulnerabilities to force programs to do what they want (this type of deliberate attack is called stack smashing).

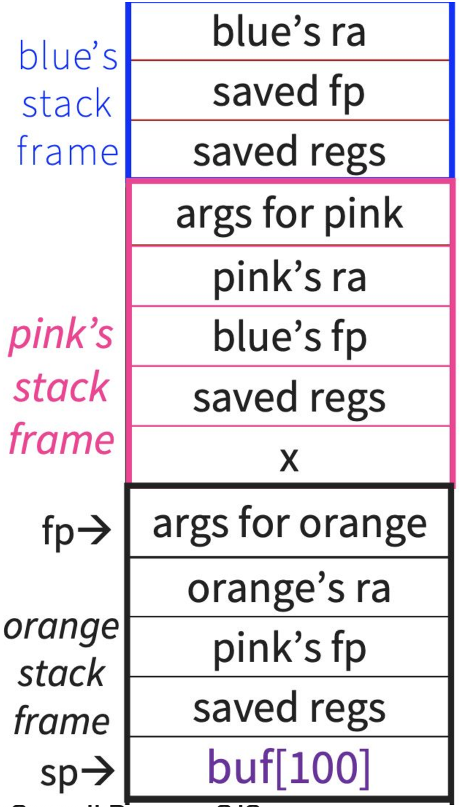
Memory & Stack Layout

Review of memory Segments (you have learned)

- Stack- Memory segment used for function calls.
  - stack frames for each function call
  - frames store local variables, function args, frame pointer, register backups, and return addresses.
- Heap - Dynamically allocated memory memory segment
  - used for objects and data that persist beyond function calls.
- Data - Stores global and static variables.
- Text/Code- Contains the compiled executable code (machine instructions to be run) of a program.



Stack Layout with a Buffer



blue's stack frame

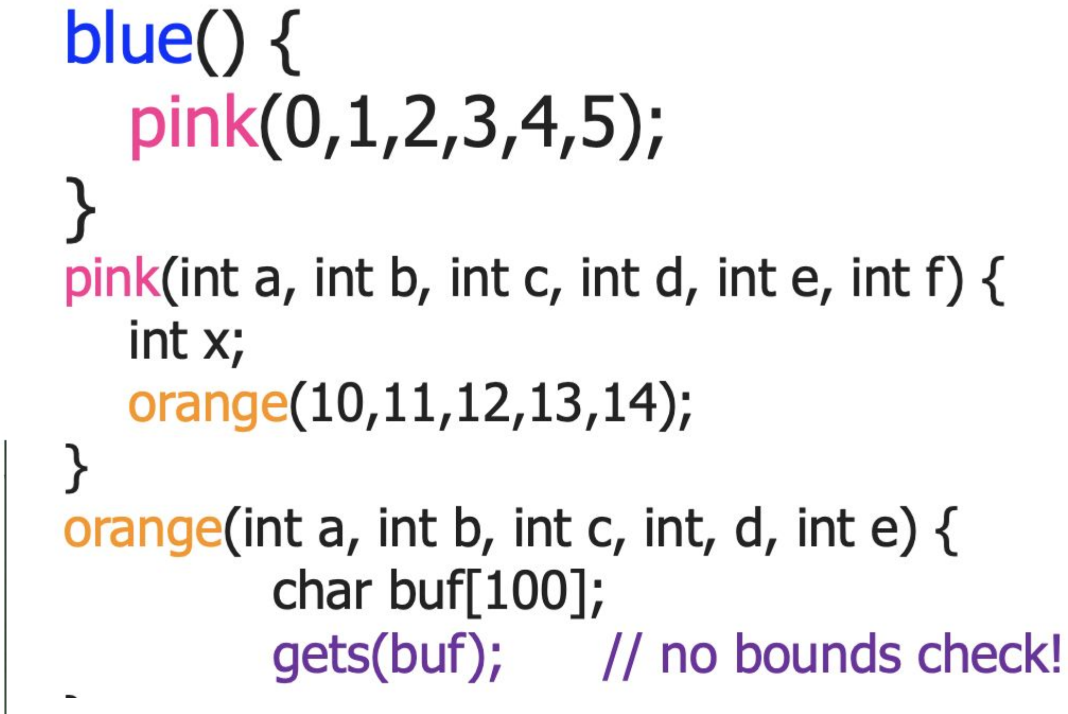
- blue's ra
- saved fp
- saved regs

pink's stack frame

- args for pink
- pink's ra
- blue's fp
- saved regs
- x

orange stack frame

- fp→ args for orange
- orange's ra
- pink's fp
- saved regs
- sp→ buf[100]



```
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    int x;
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
            char buf[100];
            gets(buf);      // no bounds check!
-
```

Useful GDB Commands

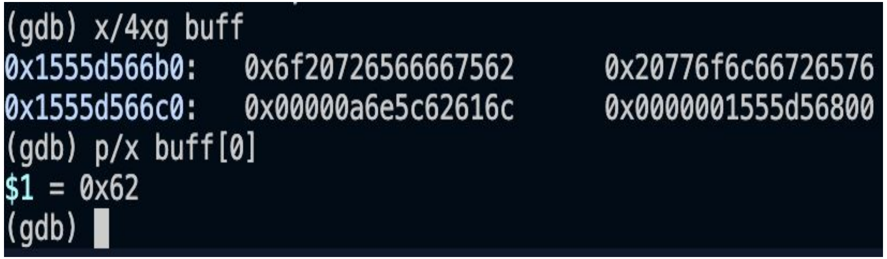
Useful GDB Commands

- info frame/ i f - lists information about the stack frame
- info registers - prints out the value of all RISC-V registers
- p buff - prints out the value of the variable buff
- p &buff - prints out the address where buff is located
- x/4xg buff - prints out 4 double words in hex starting at the address buff is pointing to
- p/x buff[0] - prints the first character of buff in hex
- stepi - step one instruction
- display/i $pc - print the current instruction
- Can also directly print out data at a particular address, whether or not that address is associated with a C variable (e.g., with the command x/4xg <address>)
- Other useful commands can be found in the gdb lab

Endianness

Endianness

- How bytes are read out from memory.
- In little-endian representation, lsbs are stored in the lowest memory address.
- In big endian representation, msbs are stored at the smallest address.
- You are working with a little endian machine which makes a lot of sense (since lsbs are in lowest addresses)
- Sometimes confuses people when doing "print debugging", since we read from left-to-right (i.e., msb to lsb)
- When crafting a buffer and storing particular values (e.g., an address) in it, be sure that your script prints the lsbs in the double word first.



```
(gdb) x/4xg buff
0x1555d566b0:   0x6f20726566667562      0x20776f6c66726576
0x1555d566c0:   0x00000a6e5c62616c      0x0000001555d56800
(gdb) p/x buff[0]
$1 = 0x62
(gdb) 
```

- Here, the byte at address 0x1555d566b0 is 0x62.
- The byte at address 0x1555d566b1 is 0x75.
- The list of bytes starting at address 0x1555d566b0 and ending at address 0x1555d566b7 is [0x62, 0x75, 0x66, 0x66….]
- But gdb printed the double word in opposite order.

Good Luck!