Buffer Overflows

- Buffer overflow vulnerabilities exist when a user of your program can store data outside of the buffer that you allocated for it
- These are a problem in languages without bounds checking
- Stack smashing—when you overflow a buffer allocated on the stack
- Today we will be exploring stack smashing in particular

Buffer Overflows

Some examples from your Lab 5's:

```
char access[2];
re = fscanf(f,"%s %x", access ,&address);

char filename[100];
strcpy( filename, argv[optind]);
```

Buffer Overflows

- The goal of a stack-based buffer overflow is to overwrite values on the stack that will allow you to hijack control flow of the problem
- This is commonly the return address
- Recall MIPS:
  - sw ra, 20(sp)
  - . . .
  - lw ra, 20(sp)

Buffer Overflows

- Consider this layout:

	Stack	For example:	With input *0x123456789abcdef0deadbeef*
Addresses ↑ Stack grows ↓	Return address	0x12345678	0xdeadbeef
	Local variables	int a;	0x9abcdef0
	Buffer	char buf[4]	0x12345678

Buffer Overflows

- If we were to overwrite the return address, what would we overwrite it to? Some possibilities...
  - The buffer. Then we could put our own machine code in the buffer.
  - Code that will jump into the buffer. For instance, if $a0 points to the buffer, jump to code that contains “jr $a0”. Then, again, put our own machine code in the buffer. This is a form of return-to-libc attack.
  - Code that takes a large number of argument that will require looking on the stack for the arguments. Take control by putting your arguments in the buffer. This is another form of return-to-libc attack.

Buffer Overflows

- Ways to mitigate buffer overflows...
  - Randomized stack locations. This means that you cannot hard code the location of your buffer when you overwrite the return address, since it will be different each time you run the program
    - But you can still use a return-to-libc attack to jump to, e.g., “jr $a0”, where $a0 points to the buffer
    - Or you can find out where the stack is

Buffer Overflows

- Ways to mitigate buffer overflows...
  - Place a canary with a random value before the return address. Check if the canary is different (has died) before returning. Abort the program if different.
    - But you can often overwrite the canary with, e.g., format string vulnerabilities

Buffer Overflows

- Ways to mitigate buffer overflows...
  - Have a non-executable stack
    - But you can still use return-to-libc attacks

Buffer Overflows

- Why MIPS is more difficult to exploit than 32-bit x86...
  - Many instructions have null bytes in them that cause string functions to not copy the entire buffer
    - But, with some effort, you can use instructions with no null bytes
  - Word-aligned instructions—you can only jump to addresses divisible by 4
  - Word-aligned... words—you can only write words to addresses divisible by 4
  - First four 32-bit functions have dedicated arguments, so we do not have to go to the stack for them
    - But functions that take more than four 32-bit arguments (or variadic functions) may still be applicable
  - I-Cache vs. D-Cache

Buffer Overflows

- Best solution: just write non-exploitable code
- Let's look at some code...
  - Some code to exploit a buffer overflow for privilege escalation
  - First attempt assumes no stack randomization
  - Second attempt assumes and defeats stack randomization
  - Press escape...