Software Security
COSC 466/566
Spring 2023

Dr. Doowon Kim

THE UNIVERSITY OF
TENNESSEE

Today’s class

Background

The C Language

Machine Code, Assembly, the C Language

- Machine code
  - The actual bits executed by the CPU
  - Non-human readable
- Assembly code
  - Human-readable representation of machine code
- C language
  - Machine-independent abstraction
    - Compiles to machine code

The C Language

- Created in 1972 by Dennis Ritchie
- Procedural programming language
  - Not object-oriented, though you can shoe-string it in
- Designed to be very low-level
  - Closely maps to underlying assembly code
  - Can include inline assembly code
- Includes direct access to memory
  - Pointers allow direct and arbitrary access to memory
  - Correct usage of memory is left up to the developer

The Great Sins of the C Language

1. Pointers can be null
2. Manual memory management
   - C-strings
   - Non-bounded array access
   - Orphaned memory
3. Written to map to assembly, not human understanding
   - Implicit type conversions
   - Varargs
   - Exception handling

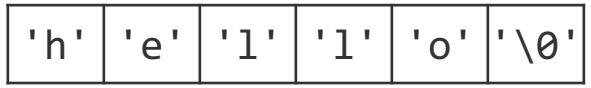
Background

Strings and Buffers

Two Philosophies

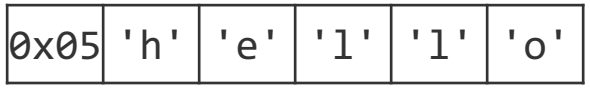
C-String

- Simply a pointer to memory where the string is stored
- End of the data is indicated by a special value ('\0')



Pascal String

- A pointer to a structure
  - First value is the length
  - Second value is the string itself
- No terminating value



Two Philosophies

C-String

- Advantages
  - Easier to implement for the compiler (non-issue)
- Disadvantages
  - No way to know length of string other than scanning it
    - Can store and reuse length
  - Can't store '\0' in string
    - Problematic for byte strings

Pascal String

- Advantages
  - Length always known
  - Can store arbitrary data
- Disadvantages
  - More work to implement for the compiler (non-issue)

Static Allocation of Memory for Strings

- String literals
  - `char* value = "Hello world";`
  - Automatically allocates the necessary amount of memory (length + 1)
  - May be read only
- Character array initialized with string literal
  - `char[] value = "Hello world";`
    - `typeof(char[]) == typeof(char*)`
  - Automatically allocates the necessary amount of memory (length + 1)
  - Will be modifiable
    - Developer must ensure that modification is in-bounds (requirement for c-strings)

Dynamic Allocation of Memory for Strings

1. Create a buffer to store the string
   - char value[12];
   - char* value = malloc(12 * sizeof(char));
2. Move the string into the buffer
   - strcpy(value, "Hello world");

- Used everywhere
  - Formatting strings using other data
  - Reading in data from external sources

Developer is responsible for ensuring the string and the terminator byte ('\0') will fit within the buffer

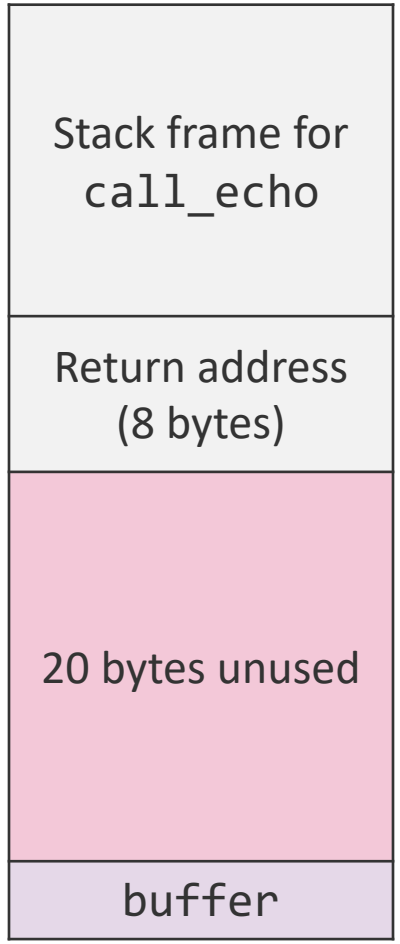
The Sin

4 Rules to Live By

1. Treat all input as malicious
2. Apply the principle of least privilege
3. Leverage defense in depth
4. You are not as clever as you think

Overflowing a Buffer

```
void echo() {
  char buffer[4]; // Way too small!
  gets(buffer);
  puts(buffer);
}

void call_echo() {
  echo();
}
```

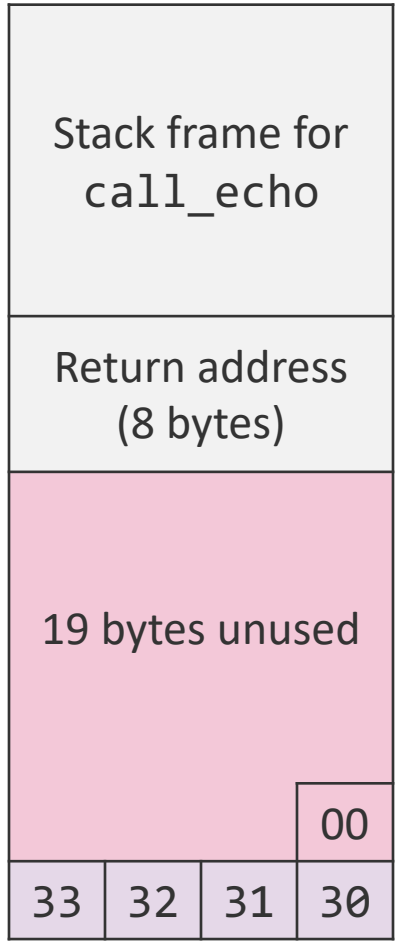


Overflowing a Buffer

```
void echo() {
  char buffer[4]; // Way too small!
  gets(buffer);
  puts(buffer);
}

void call_echo() {
  echo();
}
```

```
input:  0123
```

```
output: 0123
```

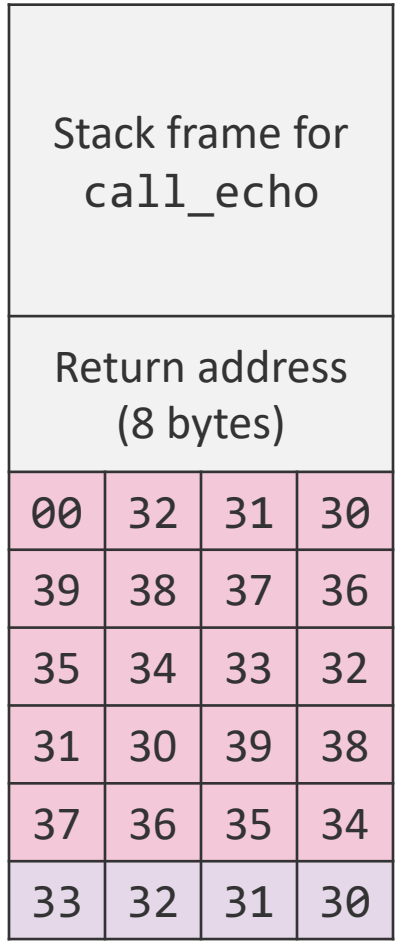


Overflowing a Buffer

```
void echo() {
  char buffer[4]; // Way too small!
  gets(buffer);
  puts(buffer);
}

void call_echo() {
  echo();
}
```

```
input:  0123456789012345678901 2
```

```
output: 0123456789012345678901 2
```

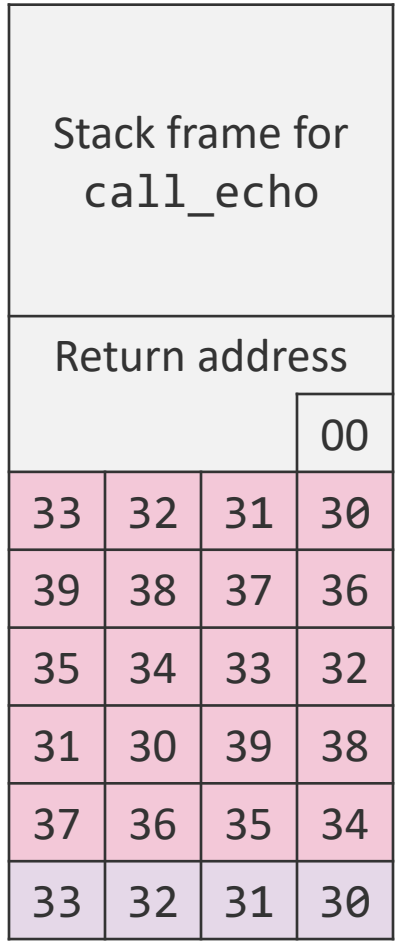


Overflowing a Buffer

```
void echo() {
  char buffer[4]; // Way too small!
  gets(buffer);
  puts(buffer);
}

void call_echo() {
  echo();
}
```

```
input:  0123456789012345678901 23
```

```
output: 0123456789012345678901 23
```

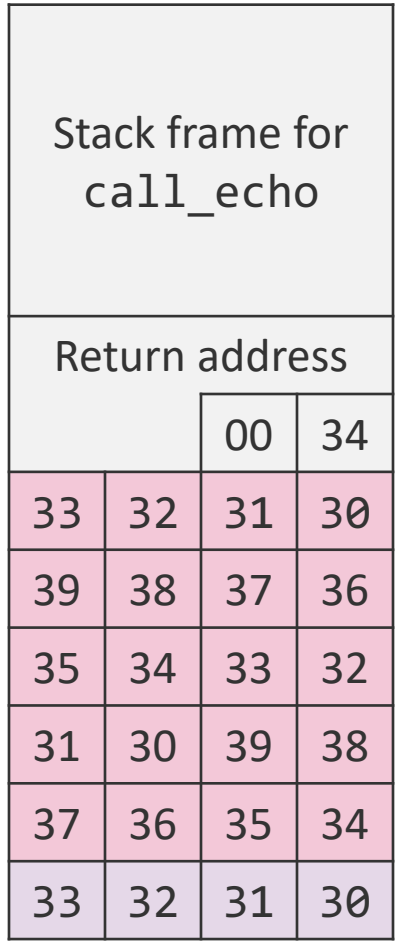


Overflowing a Buffer

```
void echo() {
  char buffer[4]; // Way too small!
  gets(buffer);
  puts(buffer);
}

void call_echo() {
  echo();
}
```

```
input:  0123456789012345678901234
```

```
output: SegFault
```



When Do Buffer Overflows Happen

- Writing to a buffer without checking the size of the input or the amount of room remaining in the buffer
- Using functions that write to a buffer without limiting how much data can be used
  - E.g., `strcpy`, `strcat`, `sprintf`, `gets`
- Incorrectly calculating the amount of data in the buffer

Common functions that cause overflow

- Recall: In C, string are character arrays terminated with a null character
  - ‘\0’ which is represented by a byte of all zeroes

```
1 #include <string.h>
2 #include <stdio.h>
3 void main () {
4     char src[40]="Hello world \0 Extra string";
5     char dest[40];
6
7     // copy to dest (destination) from src (source)
8     strcpy (dest, src);
9 }
```

Common functions that cause overflow

strcpy(char *to, char *from)

Copies ‘from’ into ‘to’ until it reaches the null character in from

Does not take into account the size of either

Overflows **`to`** whenever **`strlen(from)`**
is greater than the size of **`to`**

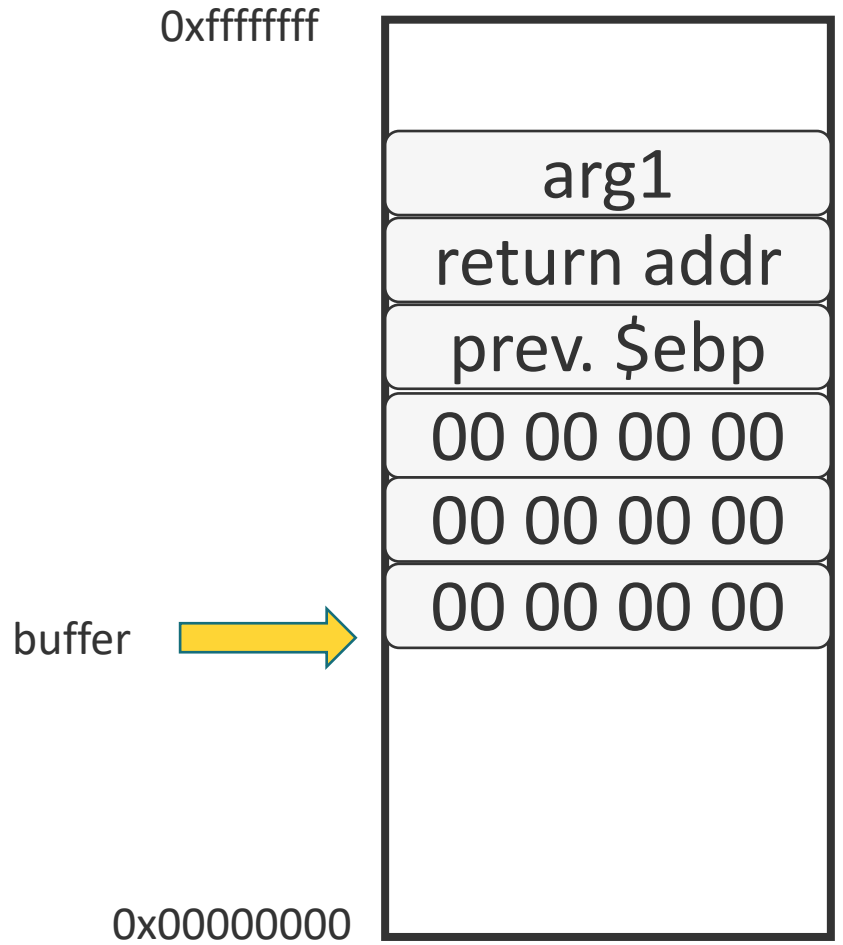
Common functions that cause overflow

- What if the string to copy is larger than the size of buffer?

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```

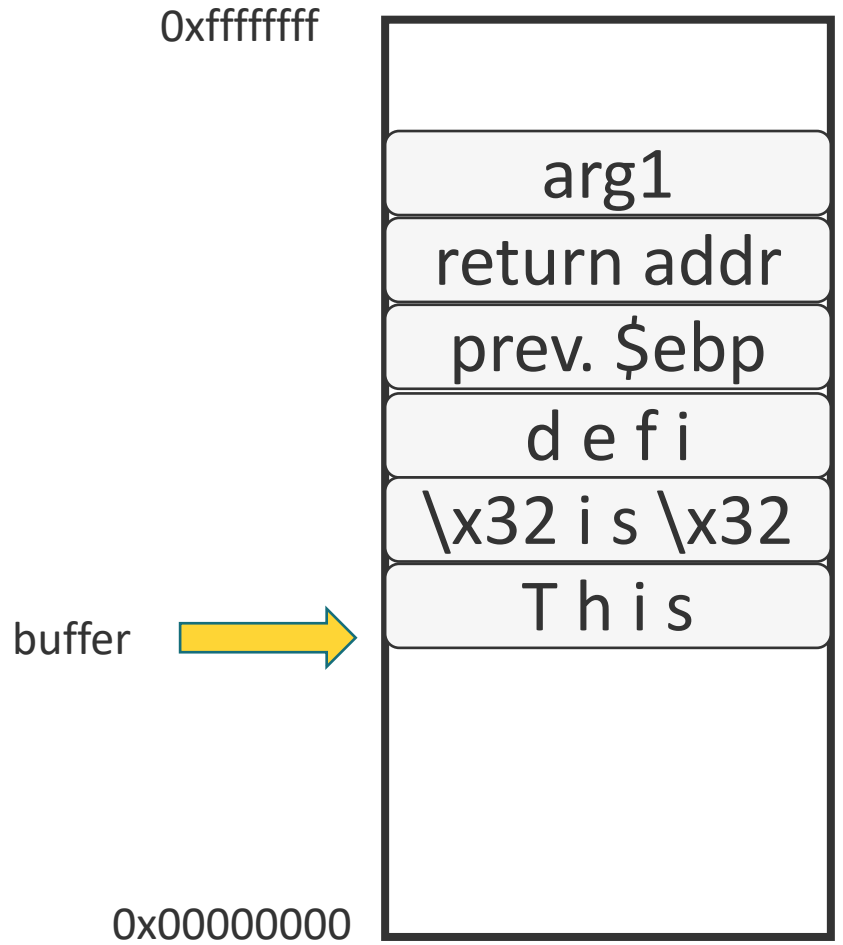
Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



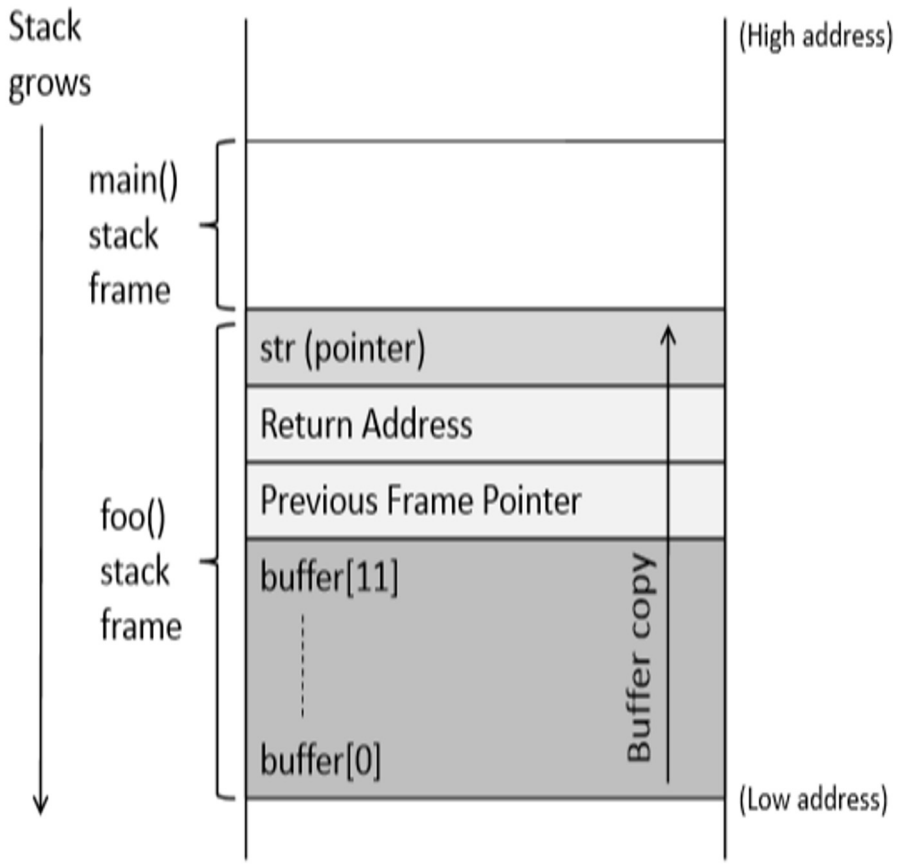
Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



Common functions that cause overflow

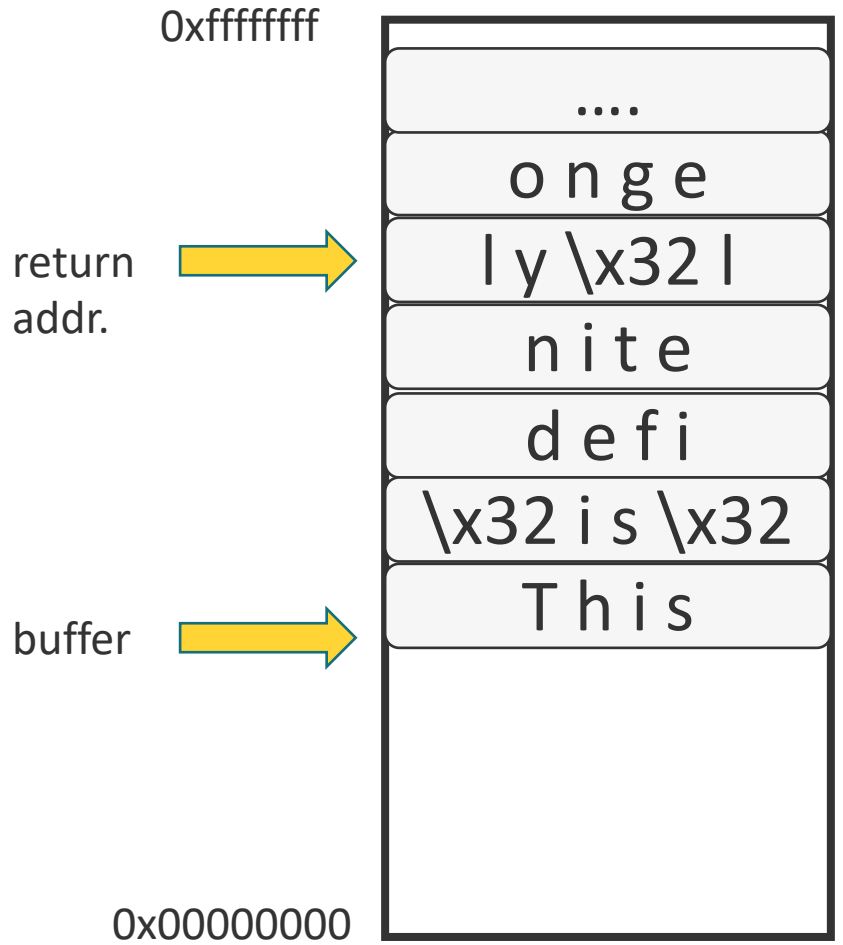
```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```

SEGFAULT

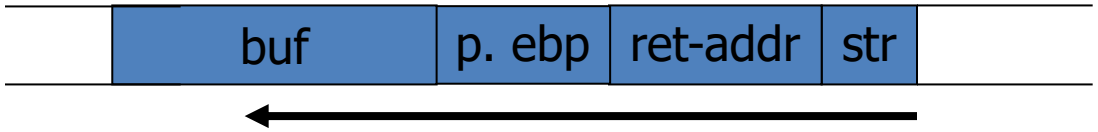


Example of a Stack-based Buffer Overflow

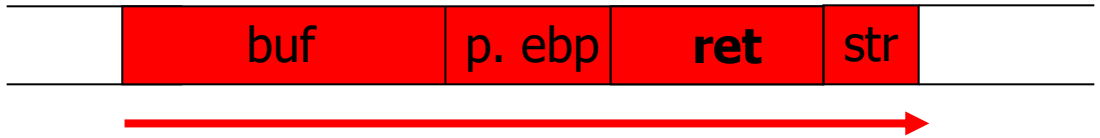
- Suppose a web server contains a function:

```
void 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`:



What’s the common things?

- Functions does not check the length.

Some Unsafe C Lib Functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)

sprintf (conts char *format, …)

What’s the common things?

- Functions does not check the length.
- User-supplied strings can result in serious problems

User-supplied strings

- In these examples, we were providing our own strings

- But they come from users in myriad ways
  - Text input
  - Network packets
  - Environment variables
  - File input
  - …

What Can An Adversary Do With This?

- Two general forms of attack
- Option 1) Change the value of local variables outside of normal control flow
  - For example an account number stored on the stack
  - Or an integer storing say the current EUID stored on the stack…
  - Can change values of variables in higher (calling) stack frames as well
    - A little more complicated, but certainly not impossible
- Option 2) Alter what the return address points to
  - Pointing it to code we want to run
  - Where could we place such code???

Effects of Overflowing the Buffer

- Nothing
  - If overflow is into unused space
  - Overwrites data that is no longer needed
- Minor issues
  - Overwrites return address to another benign part of the program
  - Overwrites data that is still used, but does not significantly modify behavior

Effects of Overflowing the Buffer

- Serious
  - Changing the return address to an invalid address (SEGFAULT)
    - Can be used as part of a denial-of-service attack
  - Attacker changes program data to change program execution flow
- Catastrophic
  - Attacker changes the return address to hijack control of the program and allow the adversary to execute arbitrary machine code
    - Return address could point to machine code stored in the buffer or anywhere else in memory
  - Commonly used to provide an adversary with a shell that can be used to further compromise the machine

Where Do Buffer Overflows Attacks Happen

- Anywhere you have a buffer
  - Stack
  - Heap
  - Data (global variables)
- Attacking the stack is the most straightforward (and common) due to the return address being on the stack