CSE 374: Lecture 20

Memory Management

Buffer Overflow

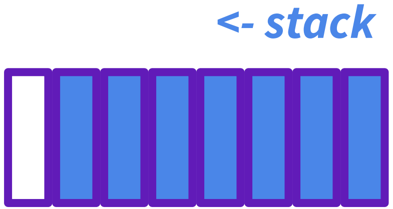
What is buffer overflow?

‘Gets’ doesn’t check for buffer size; if the string is more than 8 characters, it will write onto the memory at the end of buf.

Why is that so bad? (see the stack!)

```
void echo() {
        char buf[8];
        gets(buf);
        puts(buf);
}
```

The stack



Stack stores active functions & <u>local</u> variables

Each function gets a frame, moving down in memory

Last frame is completed, deleted

then the next most recent frame.

(Last in-first out)

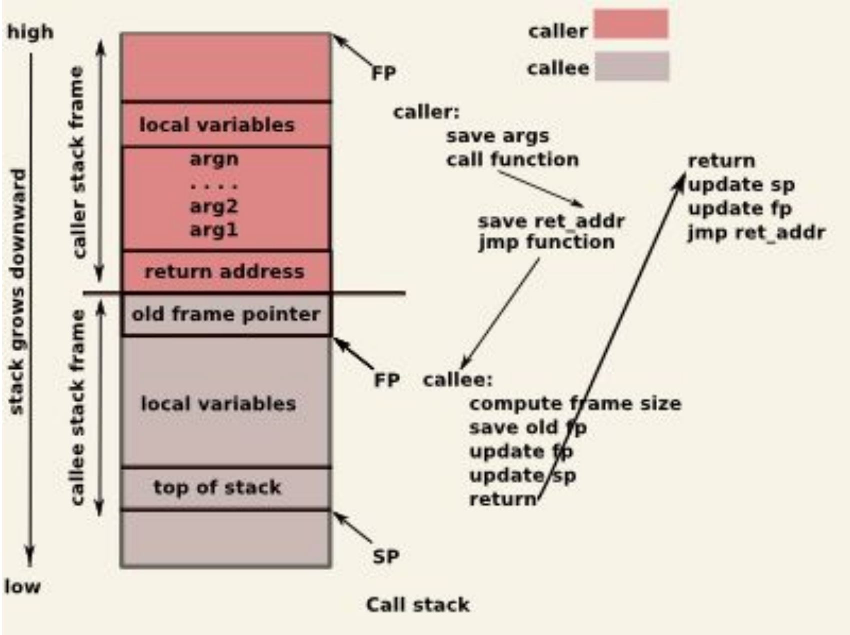
Each function call creates a frame

Containing:

Arguments, return address,

Pointer-to-last-frame,

local variables



Buffer Overflow

Writing past buf may overwrite other data, or the pointer to return to the calling code.

```
void echo()  {
      char buf[8];
      gets(buf);
      puts(buf);
}
```

Change return to last frame

```
void bufferplay (int a, int b, int c) {
  char buffer1[5];
  uintptr_t ret;  // holds an address

  // calculate the return address
  // change to be address of return
  ret = (uintptr_t) buffer1 + 0;

  // treat that number like a pointer,
  // and change the value in it
  *((uintptr_t*)ret) += 0;
}

int main(int argc, char** argv) {
  int x = 0;
  bufferplay (1,2,3);
  x = 1;  // want to skip this line
}
```

Use GDB:

```
break bufferplay
x buffer1       // prints the location of buffer1
info frame      // Look at "rip" to get the
    // location of the return address
print <rip-location> - <buffer1-location>
    // prints distance from buffer1 to return
    //  address.

disassemble main  // shows the machine
    // code  and how many bytes each
    // instruction takes up.
```

Replace command at return address

```
int bar(char *arg, char *out) {
  strcpy(out, arg);
  return 0;
}
void foo(char *argv[]) {
  char buf[256];
  bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
  foo(argv);
  return 0;
}
```

Idea:

Pass program a string in argv that contains nefarious code in a string

Take advantage of unprotected strcpy function so the return pointer on the stack is directed at the beginning of buf

When ‘foo’ exits, return ptr actually starts executing code passed in via string.

Defense against the dark-arts

- Avoid vulnerabilities in the first place.
  - Use library functions that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - %ns instead of %s in scanf
- System-level protections
  - Make stack non-executable
  - Have compiler insert “stack canaries”
  - Put a special value between buffer and return address
  - Check for corruption before leaving function

Allocating array memory

An array IS a pointer

A String is an array of char, terminating with \0

strsize returns length or string, minus the final \0 character

Allocate enough space for (strsize+1) chars

```c
// copy original string

int strsize = strlen(s)+1;
// result = (char *)malloc(strsize);
result =(char*)malloc(strsize*sizeof(char));
printf ("sizeof char: %d \n", sizeof(char));
strncpy(result, s, strsize);

// from final_reverse.c, lect. 11
```

Next up: C++ *(Want to read ahead?)*

Best place to start: C++ Primer, Lippman, Lajoie, Moo, 5th ed., Addison-Wesley, 2013

Every serious C++ programmer should also read: Effective C++, Meyers, 3rd ed., Addison-Wesley, 2005

Best practices for standard C++

Effective Modern C++, Meyers, O'Reilly, 2014

Additional "best practices" for C++11/C++14

Good online source: cplusplus.com