CS 107
Lecture 18: Review and Wrap-up

Friday, March 17, 2023

Computer Systems
Winter 2023
Stanford University
Computer Science Department

Lecturer: Chris Gregg

Today's Topics

1. Comments on Heap Allocator: You Did It!
2. Review / Examples
   - Major final topics
   - Topics from midterm to review
3. Wrap-up
   - Future courses in CS?
   - Why is *X* coded in C?

Comments on Heap Allocator and Debugging

- The heap allocator assignment is challenging! **Great job finishing up!**
  - Many students hand in incomplete explicit heap allocators
  - Repeat: *many students hand in incomplete explicit heap allocators*
- There are two primary reasons we don't look at your code in office hours:
  1. You really do need to struggle with debugging your own code. At this point in the class, we've given you all the tools to do just that. The debugging aspect of this assignment is more important than simply writing a heap allocator! (perhaps the last assignment isn't the best time to force you to debug your own code, and I'm happy to brainstorm other ideas for future classes).
  2. Everyone's solution is different, and we would never have time in office hours to figure out exactly what you are trying to do (and again, see point 1)

Major Final Topics

Generics
x86-64 Assembly
Runtime Stack
Managing the heap / heap allocation

CS107 Recap:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1234/recap/

x86-64 Example Problem

Convert the assembly on the right to the original C code on the left:

```
long mystery(long *arr, size_t nelems)
{
    for (                               ) {

        size_t sum = ___________________;

        if (________________________)

            return __________;
    }

    return _______________;
}
```

```
Dump of assembler code for function mystery:
   0x400566 <+0>:  mov    $0x0,%edx
   0x40056b <+5>:  jmp    0x40057d <mystery+23>
   0x40056d <+7>:  mov    (%rdi,%rdx,8),%rax
   0x400571 <+11>: add    $0x1,%rdx
   0x400575 <+15>: add    (%rdi,%rdx,8),%rax
   0x400579 <+19>: test   $0x1,%al
   0x40057b <+21>: jne    0x40058d <mystery+39>
   0x40057d <+23>: lea    -0x1(%rsi),%rax
   0x400581 <+27>: cmp    %rax,%rdx
   0x400584 <+30>: jb     0x40056d <mystery+7>
   0x400586 <+32>: mov    $0xffffffffffffffff,%rax
   0x40058d <+39>: repz retq
End of assembler dump.
```

x86-64 Example Answer

Convert the assembly on the right to the original C code on the left:

```
long mystery(long *arr, size_t nelems)
{
    for (size_t i=0; i < nelems-1; i++) {

        size_t sum = arr[i] + arr[i+1];

        if (sum % 2 == 1)

            return sum;
    }

    return -1;
}
```

```
Dump of assembler code for function mystery:
   0x400566 <+0>:  mov    $0x0,%edx
   0x40056b <+5>:  jmp    0x40057d <mystery+23>
   0x40056d <+7>:  mov    (%rdi,%rdx,8),%rax
   0x400571 <+11>: add    $0x1,%rdx
   0x400575 <+15>: add    (%rdi,%rdx,8),%rax
   0x400579 <+19>: test   $0x1,%al
   0x40057b <+21>: jne    0x40058d <mystery+39>
   0x40057d <+23>: lea    -0x1(%rsi),%rax
   0x400581 <+27>: cmp    %rax,%rdx
   0x400584 <+30>: jb     0x40056d <mystery+7>
   0x400586 <+32>: mov    $0xffffffffffffffff,%rax
   0x40058d <+39>: repz retq
End of assembler dump.
```

Runtime Stack Example Problem

```
int authenticate()
{
    char goodpw[8];
    get_one_time_pw(goodpw);

    char pw[8];
    printf("What is your password?\n");
    gets(pw);

    if (strcmp(pw,goodpw) != 0) {
        printf("Sorry, wrong password!\n");
        return 0; // user not okay
    } else {
        printf("You have been authenticated!\n");
        return 1; // user okay
    }
}

int main(int argc, char **argv)
{
    int authenticated;
    authenticated = authenticate();
    if (authenticated) {
        printf("Welcome to the US Treasury!\n");
    }
    return 0;
}
```

Now that you've finished CS 107, you have been hired by a security firm. The first job you have is to find out how a hacker was able to become authenticated on a client's system. Here is what you know:

1. The code to the left is the C code to grant access.
2. The hacker had access to the binary for the C code, but could only run it on their own system to test. The hacker did not have access to the **`get_one_time_pw`** function, which grants a one-time password that changes each time the program is run. (continued...)

Runtime Stack Example Problem

```c
int authenticate()
{
    char goodpw[8];
    get_one_time_pw(goodpw);

    char pw[8];
    printf("What is your password?\n");
    gets(pw);

    if (strcmp(pw,goodpw) != 0) {
        printf("Sorry, wrong password!\n");
        return 0; // user not okay
    } else {
        printf("You have been authenticated!\n");
        return 1; // user okay
    }
}

int main(int argc, char **argv)
{
    int authenticated;
    authenticated = authenticate();
    if (authenticated) {
        printf("Welcome to the US Treasury!\n");
    }
    return 0;
}
```

You open the program in gdb, and you break it right before the call to **`gets`** as shown in the disassembly below:

```
   0x0000000000400609 <+0>:  sub    $0x28,%rsp
   0x000000000040060d <+4>:  lea    0x10(%rsp),%rdi
   0x0000000000400612 <+9>:  callq  0x4005f6 <get_one_time_pw>
   0x0000000000400617 <+14>:mov    $0x40072c,%edi
   0x000000000040061c <+19>:mov    $0x0,%eax
   0x0000000000400621 <+24>:callq  0x4004b0 <printf@plt>
   0x0000000000400626 <+29>:mov    %rsp,%rdi
=> 0x0000000000400629 <+32>:callq  0x4004e0 <gets@plt>
   0x000000000040062e <+37>:lea    0x10(%rsp),%rsi
   0x0000000000400633 <+42>:mov    %rsp,%rdi
   0x0000000000400636 <+45>:callq  0x4004d0 <strcmp@plt>
   0x000000000040063b <+50>:test   %eax,%eax
   0x000000000040063d <+52>:je     0x400655 <authenticate+76>
   0x000000000040063f <+54>:mov    $0x400744,%edi
   0x0000000000400644 <+59>:mov    $0x0,%eax
   0x0000000000400649 <+64>:callq  0x4004b0 <printf@plt>
   0x000000000040064e <+69>:mov    $0x0,%eax
   0x0000000000400653 <+74>:jmp    0x400669 <authenticate+96>
   0x0000000000400655 <+76>:mov    $0x40075c,%edi
   0x000000000040065a <+81>:mov    $0x0,%eax
   0x000000000040065f <+86>:callq  0x4004b0 <printf@plt>
   0x0000000000400664 <+91>:mov    $0x1,%eax
   0x0000000000400669 <+96>:add    $0x28,%rsp
   0x000000000040066d <+100>:   retq
```

Runtime Stack Example Problem

You print out some details of the variables, and also the initial bytes on the stack and find the following:

```
(gdb) p goodpw
$1 = "hunter2"
(gdb) p &goodpw
$2 = (char (*)[8]) 0x7fffffffe960
(gdb) p &pw
$3 = (char (*)[8]) 0x7fffffffe950
(gdb) x/32bx $rsp
0x7fffffffe950:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x7fffffffe958:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x7fffffffe960:  0x68  0x75  0x6e  0x74  0x65  0x72  0x32  0x00
0x7fffffffe968:  0x00  0x05  0x40  0x00  0x00  0x00  0x00  0x00
(gdb)
```

This gives you enough information to determine how the hacker was successful!

1. Using the assembly code, the stack trace, and your knowledge of the C library, explain how the hacker could have gained access to the system by running the program.
2. Fill in the `create_password.c` program on the following slide with bytes that will create a password suitable for gaining access to the system.

Runtime Stack Example Problem

Change the bytes in the create_password.c program to build a program that will create a password that will allow access to the user.

```
// file: create_password.c

int main(int argc, char *argv[])
{
    const char *filename = argc > 1 ? argv[1] : "password.txt";
    FILE *fp = fopen(filename, "w");
    if (!fp) error(1, errno, "%s", argv[1]);

    char bytes[] = {'c','s','1','0','7',0,
    }; // edit bytes as desired

    fwrite(bytes, 1, sizeof(bytes), fp);
    fclose(fp);
    printf("Wrote password to file '%s'.\n", filename);
    return 0;
}
```

Runtime Stack Example Problem

Change the bytes in the create_password.c program to build a program that will create a password that will allow access to the user.

```
// file: create_password.c

int main(int argc, char *argv[])
{
    const char *filename = argc > 1 ? argv[1] : "password.txt";
    FILE *fp = fopen(filename, "w");
    if (!fp) error(1, errno, "%s", argv[1]);

    char bytes[] = char bytes[] = {'a',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                'a',0,
    }; // edit bytes as desired

    fwrite(bytes, 1, sizeof(bytes), fp);
    fclose(fp);
    printf("Wrote password to file '%s'.\n", filename);
    return 0;
}
```

Heap

- You should review your implicit and explicit heap allocator solutions
- You should expect to write some code for a similar but somewhat unique heap allocator problem
- See the practice final exams for examples of the types of questions we might ask

Possible topics from before the midterm

void * arrays and generic functions
function pointers
bits/bytes

Future CS Classes?

CS107 prepares you for:

- CS111
    - File systems
    - Multiprocessing and threading, deadlock, race conditions
- CS112: Operating Systems
- CS144: Networking
- CS149: Parallel Computing
- CS143: Compilers (kind of)

Why is *X* coded in C?

https://sqlite.org/whyc.html

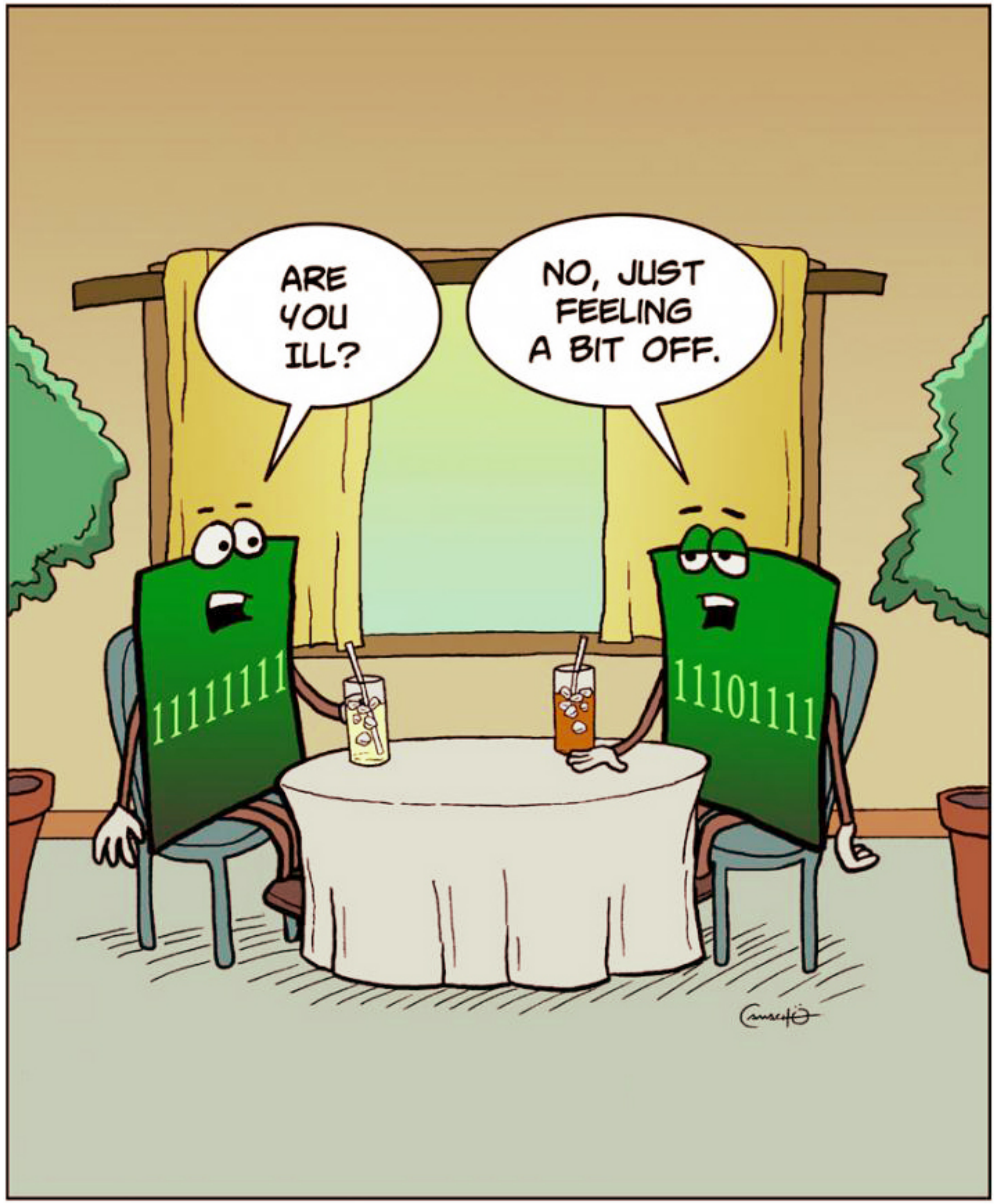
https://www.quora.com/Why-is-Linux-kernel-written-in-C-and-not-C++-given-that-C++-is-more-flexible-and-one-can-write-C-code-in-C++-as-well

https://news.ycombinator.com/item?id=2405387

https://stackoverflow.com/questions/580292/what-languages-are-windows-mac-os-x-and-linux-written-in

More programs than you think are written in C -- hopefully you now understand why!

Finally



You have learned a *ton* of information this quarter! (including the ability to understand low-level humor)

You are better programmers, and you now know what is going on "under the hood" of your programs.

Be proud of your accomplishments, and know that you are now part of the "took CS107" club!

Congratulations!