CS 107
Lecture 19: Review and Wrap-up

Friday, March 16, 2018

Computer Systems
Winter 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg
Today's Topics

1. Review / Examples
   • Major final topics
   • Minor final topics
   • Topics from midterm to review

2. Wrap-up
   • Future courses in CS?
   • Why is X coded in C?
Major Final Topics

Floating Point
x86-64 Assembly
Runtime Stack
Managing the heap / heap allocation
a. Convert decimal 24.0 to 32-bit IEEE floating point format.
b. Convert the 32-bit IEEE float \(0 \ 10000000 \ 10000000000000000000000\) to decimal format.
c. What will the following print?

```c
float a = FLT_MAX;
float b = 1;
printf("%s\n", a == a + b ? "true" : "false");
```

Notes: 10000000b == 128
a. Convert decimal -24.0 to 32-bit IEEE floating point format.
   \[1 10000011 100000000000000000000000000000000\]

b. Convert the 32-bit IEEE float \[0 10000000 10000000000000000000000\]
to decimal format. \[3\]

c. What will the following print? \textbf{true}

```c
float a = FLT_MAX;
float b = 1;
printf("%s\n", a == a + b ? "true" : "false");
```
Convert the assembly on the right to the original C code on the left:

```c
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
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.
Convert the assembly on the right to the original C code on the left:

```c
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
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.
```
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...)
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
0x000000000040062e <+37>:leaq 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>
0x0000000000400644 <+59>:mov $0x0,%eax
0x0000000000400649 <+64>:callq 0x4004e0 <gets@plt>
0x000000000040064e <+69>:mov $0x0,%eax
0x0000000000400655 <+74>:jmp 0x4000669 <authenticate+96>
0x000000000040065c <+76>:mov $0x400040649e <authenticate+96>
0x000000000040065a <+81>:mov $0x0,%eax
0x000000000040065f <+86>:callq 0x4004b0 <printf@plt>
0x0000000000400664 <+91>:mov $0x1,%eax
0x0000000000400669 <+96>:add $0x28,%rsp
0x000000000040066d <+100>: retq
```

```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 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.
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.

```c
// 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;
}
```
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.

```c
// 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[] = {'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;
}
```
Managing the Heap Example Problem

See review material on the website :)
Program optimization (manual and compiler)
  git
gcc / ELF format
  Makefiles
What kind of optimization does gcc produce in the following cases?

```c
#include<stdlib.h>
int optimize1(int *x, size_t nelems) {
    int sz = sizeof(x[0]);
    int multiplier = 5;
    return nelems * sz * multiplier;
}
```

<table>
<thead>
<tr>
<th>Optimizer Level</th>
<th>Instruction Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O0</code></td>
<td><code>optimize1: pushq %rbp</code></td>
</tr>
<tr>
<td><code>leal (%rsi,%rsi,4), %eax</code></td>
<td></td>
</tr>
<tr>
<td><code>sall $2, %eax</code></td>
<td></td>
</tr>
<tr>
<td><code>ret</code></td>
<td></td>
</tr>
<tr>
<td><code>-O2</code></td>
<td><code>optimize1: pushq %rbp</code></td>
</tr>
<tr>
<td><code>leal (%rsi,%rsi,4), %eax</code></td>
<td></td>
</tr>
<tr>
<td><code>sall $2, %eax</code></td>
<td></td>
</tr>
<tr>
<td><code>ret</code></td>
<td></td>
</tr>
</tbody>
</table>
What kind of optimization does gcc produce in the following cases?

```c
#include<stdlib.h>
int optimize1(int *x, size_t nelems) {
    int sz = sizeof(x[0]);
    int multiplier = 5;
    return nelems * sz * multiplier;
}
```

**optimization level:**

**-O0**

- pushq %rbp
- movq %rsp, %rbp
- movq %rdi, -24(%rbp)
- movq %rsi, -32(%rbp)
- movl $4, -4(%rbp)
- movl $5, -8(%rbp)
- movl -4(%rbp), %eax
- cltq
- movl %eax, %edx
- movl -32(%rbp), %rax
- imull %edx, %eax
- movl -8(%rbp), %edx
- imullq %edx, %rdx
- imull %edx, %eax
- popq %rbp
- ret

**-O2**

- leal (%rsi,%rsi,4), %eax
- sall $2, %eax
- ret

**constant folding**
What kind of optimization does gcc produce in the following cases?
git Example Problem

1. You have a git repository in your directory, and you have made changes to the `implicit.c` file that added a coalescing function. How would you commit the change to git?

2. You run the `git log` command, and you want to take a look at the code where you wrote your initial `mymalloc` function.

   ```
   $ git log
   commit 76947b8b728fc84fc70f18ad5b5f35ec63f7a948
   Author: Chris Gregg <tofergregg@gmail.com>
   Date:   Fri Mar 16 12:16:21 2018 -0700
           changed myfree function to use header bit correctly
   commit 198bbed69046ab5a2f346af0beca4a6be287c047
   Author: Chris Gregg <tofergregg@gmail.com>
   Date:   Fri Mar 16 12:14:55 2018 -0700
           created header struct
   commit 3c480dcf149d1fa0636b00c0066bd36686731814
   Author: CS107 tools <cs107@cs.stanford.edu>
   Date:   Wed Feb 28 15:05:20 2018 -0800
           wrote initial mymalloc function
   ```

   What command do you type to revert to that change so you can look at it?

3. When you are finished looking at the code, what command do you run to get back to your latest commit?
1. You have a git repository in your directory, and you have made changes to the `implicit.c` file that added a coalescing function. How would you commit the change to git?

   ```sh
git commit -m "added coalescing function to implicit.c"
   ```

2. You run the `git log` command, and you want to take a look at the code where you wrote your initial `mymalloc` function.

   ```sh
   $ git log
   commit 76947b8b728fc84fc70f18ad5b5f35ec63f7a948
   Author: Chris Gregg <tofergregg@gmail.com>
   Date:   Fri Mar 16 12:16:21 2018 -0700
   changed myfree function to use header bit correctly
   commit 198bbed69046ab5a2f346af0beca4a6be287c047
   Author: Chris Gregg <tofergregg@gmail.com>
   Date:   Fri Mar 16 12:14:55 2018 -0700
   created header struct
   commit 3c480dcf149d1fa0636b00c0066bd36686731814
   Author: CS107 tools <cs107@cs.stanford.edu>
   Date:   Wed Feb 28 15:05:20 2018 -0800
   wrote initial mymalloc function
   ```

   What command do you type to revert to that change so you can look at it?

   ```sh
git checkout 3c480dcf149d1fa0636b00c0066bd36686731814
   ```

3. When you are finished looking at the code, what command do you run to get back to your latest commit?

   ```sh
git checkout master
   ```
1. What does the `.rodata` section of an assembly language program designate?

2. What is the difference between static linking and dynamic linking?
1. What does the `.rodata` section of an assembly language program designate?

The `.rodata` section holds the read only constants and strings in a program that have an initial value.

2. What is the difference between static linking and dynamic linking?

Static linking puts all external functions into your executable, such as the library functions called in your program. Dynamic linking only creates an offset table and functions are loaded from the library when the function is called in your program.
Makefiles Example Problem

You have the following Makefile:

```
PROGRAMS = hello goodbye
all: $(PROGRAMS)
hello: hello.c hello.h
    gcc -o $@ hello.c -Wall -Wextra
goodbye: goodbye.c hello.h
    gcc -o $@ goodbye.c -Wall -Wextra
clean:
    rm -f $(PROGRAMS)
```

1. You type the following:
   $ make clean
   $ make
   $ make
   What is the output of the second command?

2. You type the following:
   $ touch hello.c
   $ make
   What is the output of the second command?
You have the following Makefile:

```makefile
PROGRAMS = hello goodbye
all: $(PROGRAMS)
hello: hello.c hello.h
    gcc -o $@ hello.c -Wall -Wextra
goodbye: goodbye.c hello.h
    gcc -o $@ goodbye.c -Wall -Wextra
clean:
    rm -f $(PROGRAMS)
```

1. You type the following:
   
   $ make clean
   $ make

   What is the output of the second command?
   
   ```
   gcc -o hello hello.c -Wall -Wextra
   gcc -o goodbye goodbye.c -Wall -Wextra
   ```

2. You type the following:
   
   $ touch hello.c
   $ make

   What is the output of the second command?
   
   ```
   gcc -o hello hello.c -Wall -Wextra
   ```
Possible topics from before the midterm

void * arrays
function pointers
bits/bytes
Future CS Classes?

CS 107 prepares you for:

• CS 110
  • File systems
  • Multiprocessing and threading, deadlock, race conditions
  • Networking
  • MapReduce
• CS 140
  • Operating Systems
• CS 144
  • Networking
Why is X coded in C?

https://sqlite.org/whyc.html


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 CS 107" club!

Congratulations!