CS61 Section Notes 4

(Week of 10/8 - 10/12)

Topics to be covered
1. Code
2. Questions
3. More Assembly + Homework tips
4. Exceptions

Topics to be covered

- Arrays
- Structs
- Buffer overflow

1. Code

A terrible programmer wrote the following code. It is designed to generate information about students and fill that information into a data structure. Two of the students, however, are celebrities, and therefore their information is sensitive. To generate their data, a password must be provided.

```c
#include <stdio.h>
#include <string.h>

struct student_t {
    char name[3][7];
    int age;
    char grade;
};

struct student_t global_pair[2];

void generate_students(struct student_t *students[3]) {
    struct student_t sally = {
        "Sally","Henthorn","Ro"},
        15,
        'B'
    };
```
struct student_t psyche = {
    {"Psyche", "Lazy", "Murphy"},
    2,
    'A'
};
struct student_t harvey = {
    {"Harvey", "Dexter", "Glenn"},
    2,
    'A'
};
students[0] = &psyche;
students[1] = &harvey;
students[2] = &sally;
students += 1;

printf("%s %s %s\n", sally.name[0],sally.name[1],sally.name[2]);
}

void generate_secret_students(struct student_t *students[2],
                               char *password) {
    struct student_t secret_1 = {
        {"Secret", "Oscar", "Meyer"},
        15,
        'C'
    };
    struct student_t secret_2 = {
        {"Secret", "Betty", "Crock"},
        16,
        'A'
    };
    char buffer[9];
    strcpy(buffer, password);
    if (!strcmp(buffer, "secure")) {
        students[0] = &secret_1;
        students[1] = &secret_2;
    }
}

int main() {
    struct student_t *students[3];
    struct student_t *secret_students[2];
    int sneaky_length = 9 + sizeof(struct student_t) * 2 + 4 + 4 + 1;
    char sneaky[sneaky_length];
    sneaky[sneaky_length-5] = 0xef;
    sneaky[sneaky_length-4] = 0xbe;
    sneaky[sneaky_length-3] = 0xad;
    sneaky[sneaky_length-2] = 0xde;
    sneaky[sneaky_length-1] = 0x0;
68    generate_students(students);
69    generate_secret_students(secret_students, sneaky);
70
71    printf("First student name: %s %s %s\n",
72        students[0]->name[0],
73        students[0]->name[1],
74        students[0]->name[2]);
75
76    return 0;
77 }

2. Questions
Assume this program is compiled for Linux using gcc running on an x86 processor.

Q1: What is the layout of a struct student_t in memory?

Q2: If global_pair points to memory location 0x80049110, what memory location do each of the following refer to?
   a. global_pair[0].name
   b. global_pair[0].name[0]
   c. global_pair[0].name[1]
   d. &(global_pair[0].grade)
   e. &(global_pair[1].grade)

Q3: What will line 34 print and why?

Q4: What does the stack look like at the end of the generate_students() function?

Q5: What is the overall structure of the data pointed to by the argument "students" at the end of the generate_students() function?

Q6: What is the overall structure of the data pointed to by the "students" array declared in main()?
Q7: Suppose (just for this question) that the function main was defined as follows.

```c
100   int main() {
101       struct student_t *students[3];
102       struct student_t *secret_students[2];
103
104       generate_students(students);
105       generate_secret_students(secret_students, “wrongpwd”);
106
107       printf("First student name: %s %s %s\n",
108           students[0]->name[0],
109           students[0]->name[1],
110           students[0]->name[2]);
111
112       return 0;
```

What will line 107 print and why?

Q8: What are lines 60 through 67 doing?

Q9: Why is sneaky_length the size that it is?

3. More Assembly + Homework tips

The following code has some very familiar behavior. Try to work through print and figure out what it is doing. Hint: Try to figure out what data structure is being used.

```
0804849e <print>:
  0804849e:  55   push %ebp
  0804849f:  89 e5  mov %esp,%ebp
  080484a1:  53   push %ebx
  080484a2:  83 ec 14  sub $0x14,%esp
  080484a5:  8b 1d 24 a0 04 08  mov 0x804a024,%ebx  # this is a global pointer
  080484ab:  85 db  test %ebx,%ebx
  080484ad:  74 19  je 80484c8 <print+0x2a>
  080484af:  8b 03  mov (%ebx),%eax
  080484b1:  89 44 24 04  mov %eax,0x4(%esp)
  080484b5:  c7 04 24 00 86 04 08  movl $0x8048600,(%esp)  # This value is “%d\n”
  080484bc:  e8 7f fe ff ff  call 8048340 <printf@plt>
  080484c1:  8b 5b 04  mov 0x4(%ebx),%ebx
  080484c4:  85 db  test %ebx,%ebx
  080484c6:  75 e7  jne 80484af <print+0x11>
  080484c8:  83 c4 14  add $0x14,%esp
  080484cb:  5b   pop %ebx
  080484cc:  5d   pop %ebp
```
Q10: What does this function do?

Magic is a slightly more complicated function, but is working on the same data structure.

08048444 <magic>:
08048444:  55   push  %ebp   # Save callee-save registers
08048445:  89 e5  mov  %esp,%ebp
08048447:  56   push  %esi
08048448:  53   push  %ebx
08048449:  83 ec 10  sub  $0x10,%esp  # Set up the stack frame
0804844c:  8b 75 08  mov  0x8(%ebp),%esi
0804844f:  8b 1d 24 a0 04 08  mov  0x804a024,%ebx  # A global pointer
08048455:  c7 44 24 04 08 00 00  movl  $0x8,0x4(%esp)
0804845c:  00
0804845d:  c7 04 24 01 00 00 00  movl  $0x1,(%esp)
00
# void *calloc(size_t nmemb, size_t size)
08048464:  e8 17 ff ff ff  call  8048380 <calloc@plt>
08048469:  89 30  mov  %esi,%eax
080486b:  85 db  test  %ebx,%ebx
080486d:  75 07  jne  8048476 <magic+0x32>
080486f:  a3 24 a0 04 08  mov  %eax,0x804a024
0804874:  eb 21  jmp  8048497 <magic+0x53>
0804876:  3b 33  cmp  (%ebx),%esi
0804878:  7d 0c  jge  8048486 <magic+0x42>
080487a:  89 58 04  mov  %ebx,0x4(%eax)
080487d:  a3 24 a0 04 08  mov  %eax,0x804a024
0804882:  eb 13  jmp  8048497 <magic+0x53>
0804884:  89 d3  mov  %edx,%ebx
0804886:  8b 53 04  mov  0x4(%ebx),%edx
0804889:  85 d2  test  %edx,%edx
080488b:  74 04  je  8048491 <magic+0x4d>
080488d:  3b 32  cmp  (%edx),%esi
080488f:  7f f3  jg  8048484 <magic+0x40>
0804891:  89 50 04  mov  %edx,0x4(%eax)
0804894:  89 43 04  mov  %eax,0x4(%ebx)
0804897:  83 c4 10  add  $0x10,%esp
080489a:  5b  pop  %ebx
080489b:  5e  pop  %esi
080489c:  5d  pop  %ebp
080489d:  c3  ret

Q11: What is this code doing?

4. Exceptions

Exceptions are changes in control flow that are due to a change in the processors state. Exception handlers are run in kernel mode as opposed to regular code which is run in user mode. This gives them access to the kernel's stack and data. There are several types of
Exceptions are incredibly important as they are the way that your program can interact with the rest of the computer. It is necessary to understand how many of these exceptions work, to understand bugs that may occur and how to avoid them.

Run the example program and show how to debug a seg fault.

<table>
<thead>
<tr>
<th>Type of Exception</th>
<th>Cause of Exception</th>
<th>Return behavior</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Signal from I/O device</td>
<td>Returns to the next instruction</td>
<td>When the hard drive has finished fetching data, and is ready to return it</td>
</tr>
<tr>
<td>Trap</td>
<td>Intentional exceptions that occur during normal program flow</td>
<td>Returns to the next instruction</td>
<td>System Calls: e.g. fork, file i/o</td>
</tr>
<tr>
<td>Fault</td>
<td>Error conditions it is possible to recover from</td>
<td>Might return to the next instruction</td>
<td>Page faults, Divide by Zero errors, Seg faults</td>
</tr>
<tr>
<td>Abort</td>
<td>Error conditions it is not possible to recover from</td>
<td>Never returns to the program</td>
<td>Fatal hardware errors such as bit corruption</td>
</tr>
</tbody>
</table>