

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.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 struct student_t {
5     char name[3][7];
6     int age;
7     char grade;
8 };
9
10 struct student_t global_pair[2];
11
12 void generate_students(struct student_t *students[3]) {
13     struct student_t sally = {
14         {"Sally", "Henthorn", "Ro"},
15         15,
16         'B'
17     };
```

```

18 struct student_t psyche = {
19     {"Psyche", "Lazy", "Murphy"},
20     2,
21     'A'
22 };
23 struct student_t harvey = {
24     {"Harvey", "Dexter", "Glenn"},
25     2,
26     'A'
27 };
28
29 students[0] = &psyche;
30 students[1] = &harvey;
31 students[2] = &sally;
32 students += 1;
33
34 printf("%s %s %s\n", sally.name[0],
sally.name[1], sally.name[2]);
35 }
36 void generate_secret_students(struct student_t *students[2],
37     char *password) {
38     struct student_t secret_1 = {
39         {"Secret", "Oscar", "Meyer"},
40         15,
41         'C'
42     };
43     struct student_t secret_2 = {
44         {"Secret", "Betty", "Crock"},
45         16,
46         'A'
47     };
48     char buffer[9];
49
50     strcpy(buffer, password);
51     if (!strcmp(buffer, "secure")) {
52         students[0] = &secret_1;
53         students[1] = &secret_2;
54     }
55 }
56
57 int main() {
58     struct student_t *students[3];
59     struct student_t *secret_students[2];
60     int sneaky_length = 9+sizeof(struct student_t)*2+4+4+1;
61     char sneaky[sneaky_length];
62
63     sneaky[sneaky_length-5] = 0xef;
64     sneaky[sneaky_length-4] = 0xbe;
65     sneaky[sneaky_length-3] = 0xad;
66     sneaky[sneaky_length-2] = 0xde;
67     sneaky[sneaky_length-1] = 0x0;

```

```
68
69 generate_students(students);
70 generate_secret_students(secret_students, sneaky);
71
72 printf("First student name: %s %s %s\n",
73        students[0]->name[0],
74        students[0]->name[1],
75        students[0]->name[2]);
76
77 return 0;
78 }
```

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.

```
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>:
804849e: 55                push  %ebp
804849f: 89 e5            mov   %esp,%ebp
80484a1: 53              push  %ebx
80484a2: 83 ec 14        sub   $0x14,%esp
80484a5: 8b 1d 24 a0 04 08 mov   0x804a024,%ebx    # this is a global pointer
80484ab: 85 db          test  %ebx,%ebx
80484ad: 74 19          je   80484c8 <print+0x2a>
80484af: 8b 03          mov   (%ebx),%eax
80484b1: 89 44 24 04    mov   %eax,0x4(%esp)
80484b5: c7 04 24 00 86 04 08 movl  $0x8048600,(%esp)  # This value is "%d\n"
80484bc: e8 7f fe ff ff call  8048340 <printf@plt>
80484c1: 8b 5b 04      mov   0x4(%ebx),%ebx
80484c4: 85 db          test  %ebx,%ebx
80484c6: 75 e7          jne  80484af <print+0x11>
80484c8: 83 c4 14      add   $0x14,%esp
80484cb: 5b            pop   %ebx
80484cc: 5d            pop   %ebp
```

```
80484cd:    c3                ret
```

Q10: What does this function do?

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

```
08048444 <magic>:
8048444:    55                push %ebp           # Save callee-save registers
8048445:    89 e5             mov  %esp,%ebp
8048447:    56                push %esi
8048448:    53                push %ebx
8048449:    83 ec 10         sub  $0x10,%esp    # Set up the stack frame
804844c:    8b 75 08         mov  0x8(%ebp),%esi
804844f:    8b 1d 24 a0 04 08 mov  0x804a024,%ebx # A global pointer
8048455:    c7 44 24 04 08 00 00 movl $0x8,0x4(%esp)
804845c:    00
804845d:    c7 04 24 01 00 00 00 movl $0x1,(%esp)
                                # void *calloc(size_t nmemb, size_t size)
8048464:    e8 17 ff ff ff   call 8048380 <calloc@plt>
8048469:    89 30             mov  %esi,(%eax)
804846b:    85 db             test %ebx,%ebx
804846d:    75 07             jne 8048476 <magic+0x32>
804846f:    a3 24 a0 04 08   mov  %eax,0x804a024
8048474:    eb 21             jmp 8048497 <magic+0x53>
8048476:    3b 33             cmp  (%ebx),%esi
8048478:    7d 0c             jge 8048486 <magic+0x42>
804847a:    89 58 04         mov  %ebx,0x4(%eax)
804847d:    a3 24 a0 04 08   mov  %eax,0x804a024
8048482:    eb 13             jmp 8048497 <magic+0x53>
8048484:    89 d3             mov  %edx,%ebx
8048486:    8b 53 04         mov  0x4(%ebx),%edx
8048489:    85 d2             test %edx,%edx
804848b:    74 04             je   8048491 <magic+0x4d>
804848d:    3b 32             cmp  (%edx),%esi
804848f:    7f f3             jg   8048484 <magic+0x40>
8048491:    89 50 04         mov  %edx,0x4(%eax)
8048494:    89 43 04         mov  %eax,0x4(%ebx)
8048497:    83 c4 10         add  $0x10,%esp
804849a:    5b                pop  %ebx
804849b:    5e                pop  %esi
804849c:    5d                pop  %ebp
804849d:    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:

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

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.