CS61: Systems Programing and Machine Organization

Fall 2011

Section 3: Monday 26 September – Friday 30 September

Topics to be covered:

- **Arrays**
- **Structs**
- **Buffer overflow**

A terrible programmer (in this case one of the TFs) 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 }
```

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

1. What is the layout of a `struct student_t` in memory?

2. 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)

3. What will line 34 print and why?

4. What does the stack look like at the end of the generate_students() function?

5. What is the overall structure of the data pointed to by the argument “students” at the end of the generate_students() function?

6. What is the overall structure of the data pointed to by the “students” array declared in main()?

7. What will line 72 print and why?

8. What are lines 60 through 67 doing?

9. Why is sneaky_length the size that it is?

Pointers and Arrays

A supplement to Section Notes for Week 3.

In class, we learned that in C there is a strong relationship between pointers and arrays. Any operation that can be achieved by array subscripting can also be achieved using pointers.

There are, however, differences between arrays and pointers.

An array declaration defines an array of the declared size. For example, `int foo[5]` defines an array called `foo` that contains 5 integers. Declaration `float bar[6][4]`, defines a multidimensional array called `bar` that consists of 6 rows, and 4 columns, for a total of 24 `floats`. Declaration `char *strings[10]` defines an array called `strings` that contains 10 pointers.

You cannot assign to an array (e.g., `foo = 4` is illegal, as is `foo++`). You can, however, use an array in an expression, in which case it will evaluate to the address of the first element of the array (e.g., `int *x = foo;` is equivalent to `int *x = &foo[0];`)

Note that if you declare an array as a local variable (e.g., `void f(void) { int foo[5]; ... }`), the compiler will allocate space on the stack for the array (e.g., `5*sizeof(int)` bytes will be allocated in `f`'s stack frame). However, there will not be space allocated for a pointer to the array (e.g., it is not the case that there is space on the stack for a variable called `foo` that points to the array; `foo` *is* the array.)

Note also that a procedure argument that has an array type is actually a pointer. For example, `void f(char s[])` is equivalent to `void f(char* s)` (and the latter form is preferred). In the body of procedure `f`, you can assign to the argument `s`, for example, `s = s + 1;`

To test your knowledge, figure out what the following code outputs, assuming that the address of `foo` is 0x100.

```
int foo[3][4] = { {1,  2,  3,  4},
                  {5,  6,  7,  8},
                  {9, 10, 11, 12} };

void g(int a[], int *b) {
  printf("Output 7:  %x\n", *a);
  printf("Output 8:  %x\n", *b);

  a++;
  printf("Output 9:  %x\n", *a);
  printf("Output 10:  %x\n", *b);
}

int main(void) {
   printf("Output 1:  %x\n",  foo);
   printf("Output 2:  %x\n",  foo[1]);
   printf("Output 3:  %x\n",  &foo[1]);

   printf("Output 4:  %x\n",  foo[2]);
   printf("Output 5:  %x\n",  foo[1][2]);
   printf("Output 6:  %x\n",  &foo[1][2]);

   g(foo[2], foo[2]);
}
```