CSE 351 Section 5 Solutions – Arrays and Buffer Overflow

Welcome back to section, we're happy that you're here ☺

Arrays

- Arrays are contiguously allocated chunks of memory large enough to hold the specified number of elements of the size of the datatype. Separate array allocations are not guaranteed to be contiguous.
- 2-dimensional arrays are allocated in row-major ordering in C (i.e. the first row is contiguous at the start of the array, followed by the second row, etc.).
- 2-level arrays are formed by creating an array of pointers to other arrays (i.e. the second level).

We have a two-dimensional matrix of integer data of size M rows and N columns. We are considering 2 different representation schemes:

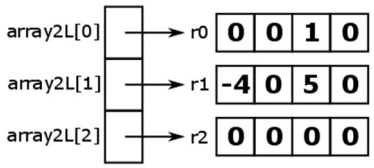
1) 2-dimensional array `int array2D[][]` // M*N array of ints
2) 2-level array `int* array2L[]` // M array of int arrays

Consider the case where $M = 3$ and $N = 4$. The declarations are given below:

2-dimensional array:	2-level array:
`int array2D[3][4];`	`int r0[4], r1[4], r2[4];` `int* array2L[] = {r0,r1,r2};`

For example, the diagrams below correspond to the matrix $\begin{bmatrix} 0 & 0 & 1 & 0 \\ -4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ for `array2D` and `array2L`:

0	0	1	0	-4	0	5	0	0	0	0	0
[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]	[2][1]	[2][2]	[2][3]



Fill in the following comparison chart:

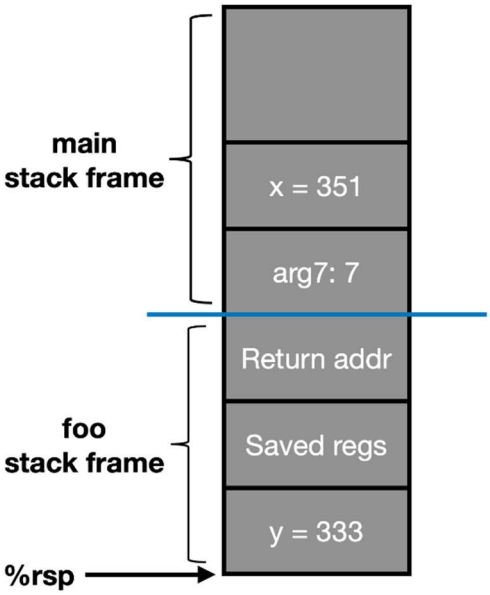
	2-dim array	2-level array
Overall Memory Used	M*N*sizeof(int) = 48 B	M*N*sizeof(int) + M*sizeof(int*) = 72 B
Largest *guaranteed* continuous chunk of memory	The whole array (48 B)	The array of pointers (24 B) > row array (16 B)
Smallest *guaranteed* continuous chunk of memory	The whole array (48 B)	Each row array (16 B)
Data type returned by:	`array2D[1]` int *	`array2L[1]` int *
Number of memory accesses to get `int` in the *BEST* case	1	2
Number of memory accesses to get `int` in the *WORST* case	1	2

Procedures and the Stack

The Stack is a region in memory which starts from the highest memory address and grows downwards when necessary as a programs executes. We consider the region with the highest address the Stack "bottom" and the region with the lowest address the Stack "top". **%rsp** is a dedicated special register which points to the current Stack top.

In x86-64, the Stack can be broken down into Stack Frames of functions. Consider the following lines of code:

```
int main() {
  int x = 351;
  foo(1,2,3,4,5,6,7);
}
void foo(int arg1, int arg2, …,int arg7){
  int y = 333;
}
```



Shown in the figure on the right are the stack frames of `main` and `foo` right before `foo` returns:

- Main
  - Return address (not pictured)
  - Potentially saved registers (not pictured)
  - Local variables
  - Arguments for called procedures
- Foo
  - Return address to main
  - Potentially saved registers
  - Local variables

Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07              movzbl (%rdi),%eax
  4005e9: 84 c0                 test %al,%al
  4005eb: 74 13                 je 400600 <rfun+0x1a>
  4005ed: 53                    push %rbx
  4005ee: 48 0f be d8           movsbq %al,%rbx
  4005f2: 48 83 c7 01           add $0x1,%rdi
  4005f6: e8 eb ff ff ff        callq 4005e6 <rfun>
  4005fb: 48 01 d8              add %rbx,%rax
  4005fe: eb 06                 jmp 400606 <rfun+0x20>
  400600: b8 00 00 00 00        mov $0x0,%eax
  400605: c3                    retq
  400606: 5b                    pop %rbx
  400607: c3                    retq
```

a) In terms of the C function, what value is being saved on the stack?

Value: *s.

The movsbq instruction at 0x4005ee puts *s into %rbx, which is pushed onto the stack by the pushq instruction at 0x4005ed.

b) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)?

0x4005fb

c) Assume `main` calls `rfun` with `char *s = "CSE 351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack?

Total frames: 9 Max depth: 8

main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4) -> rfun(s+5) -> rfun(s+6)-> printf()

The recursive call to rfun(s+6), which handles the null-terminator in the string (base case),still creates a stack frame since we consider the return address pushed to the stack during a procedure call to be part of the callee's stack frame.

d) Assume `main` calls `rfun` with `char *s = "CSE 351"`, as shown in the C code. After `main` calls `rfun`, we find that the return address to `main` is stored on the stack at address `0x7fffffffdb38`. On the first call to `rfun`, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "CSE 351" (i.e. `char *s = 0x4006d0`) during the **fourth** call to `rfun`.

*For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.*

Memory Address	Value	Description
0x7fffffffdb48	Unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	Unknown	Original %rbx
0x7fffffffdb28	0x4005fb	Return address
0x7fffffffdb20	*s, "C", 0x43	Saved %rbx
0x7fffffffdb18	0x4005fb	Return address
0x7fffffffdb10	*s, *(s+1), "S", 0x53	Saved %rbx
0x7fffffffdb08	0x4005fb	Return address
0x7fffffffdb00	*s, *(s+2), "E", 0x45	Saved %rbx

Buffer Overflow

Consider the following C program:

```
void main() {
  read_input();
}

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Here is a diagram of the stack at the beginning of the call to read_input():

Address	Value (hex)
%rsp+15	00
%rsp+14	00
%rsp+13	00
%rsp+12	00
%rsp+11	00
%rsp+10	~~40~~ 00 (null terminator)
%rsp+9	~~AF~~ 73
%rsp+8	~~3B~~ 72
%rsp+7	71
%rsp+6	70
%rsp+5	6F
%rsp+4	6E
%rsp+3	6D
%rsp+2	6C
%rsp+1	6B
%rsp+0	6A

a) What is the value of the return address stored on the stack?

0x40AF3B

Assume that the user inputs the string "`jklmnopqrs`"

b) Write the values in the stack before the "return 0;" statement is executed. Cross out the values that were overwritten and write in their new values.
(Hint: use the ASCII table at the bottom to convert from letters to bytes)

c) What is the new return address after the call to gets()?

0x7372

d) Where will execution jump to after the "`return 0;`"?

It will try to jump to 0x7372, but it will crash with a segfault

e) How many characters would we have to enter into the command line to overwrite the return address to `0x6A6B6C6D6E6F`?

14 = 8 for padding (the length of buf) + 6 for the length of the address in bytes. A null terminator is appended, but it's okay because the upper bytes were going to be 0x00 anyway

f) Create a string that will overwrite the return address, setting it to `0x6A6B6C6D6E6F`

"`abababababonmlkj`" (The first 8 characters don't matter since they're just padding)

In Lab 3, we are given a tool called `sendstring`, which converts hex digits into the actual bytes

```
>echo "61 62 63" | ./sendstring
abc
```

g) If we want to overwrite the return address to a stack address like 0x7FFFFFAB1234, we need to use a tool like sendstring to send the correct bytes.
Why can't we just manually type the characters like we did earlier with "`jklmnopqrs`"?

There is no character in ASCII we can type that will give us a byte value of 0x7F, 0xFF, or 0x12

Char	Hex
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A

Check out the Lab 3 video on Phase 0 before you start the lab! It's linked on the Lab 3 page