CS 161: Computer Security

Lecture 16

November 5, 2015

Why I love computer security

- Doing computer security requires knowing all levels of security

- Need ***full*** understanding of hardware, machine architecture, compiler, programming languages, operating systems, algorithms, mathematics of encryption

- Hacking illustrates some of the scope of material we must know

Please review

- Please review
  - Notes from CS 61C on assembly language
  - Notes from CS 61C on stack and heap storage and function calls
  - Notes from CS 61C on using gdb

main.c

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

```
gcc -S -o main.s main.c
```

main.s

```
 .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

main.s

labels

```
 .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

main.s

labels

.cfi – call frame instructions

```
 .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

main.s

```
 .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

labels

.cfi – call frame instructions

.x – assembly directives

main.s

```
main:
        pushq   %rbp

        movq    %rsp, %rbp
        movl    $0, %eax
        popq    %rbp
        ret
```

main.s

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $0, %eax
        popq    %rbp
        ret
```

main.s

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $0, %eax
        popq    %rbp
        ret
```

`%r__` are 64 bit registers

`%e__` are 32 bit registers

q suffix on instruction indicates a “quad-word” (64 bit) instruction

l suffix on instruction indicates a “long-word” (32 bit) instruction

main.s

```
main:
        pushq   %rbp            #save %rbp on stack
        movq    %rsp, %rbp      #store %rsp value in %rbp
        movl    $0, %eax        #store 0 in %eax
        popq    %rbp            #restore %rbp from stack
        ret                     #return from function
```

What is going on with the **`%rbp`** and **`%rsp`** juggling?

What does **`ret`** (return) actually do?

function.c

```
void foo(int a, int b)
{
}

int main()
{
    foo(4, 6);
    return 0;
}
```

function.s

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        popq    %rbp
        ret
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $6, %esi
        movl    $4, %edi
        call    foo
        movl    $0, %eax
        popq    %rbp
        ret
```

%rbp is special register storing frame (base) ptr
%rsp is special register storing stack pointer
space between them is function's working space
stack grows down!

params stored -4 & -8 bytes

below base pointer

This code does nothing with parameters.
Let's try changing that.

function2.c

```
int foo(int a, int b)
{
    return a + b;
}

int main()
{
    foo(4, 6);
    return 0;
}
```

New code in `function2.s`

```
movl    %edi, -4(%rbp)
movl    %esi, -8(%rbp)
movl    -8(%rbp), %eax
movl    -4(%rbp), %edx
addl    %edx, %eax
```

- Parameters passed in **`%edi`** and **`%esi`**
- Stored on stack, and then copied into **`%eax`** & **`%edx`**
- Answer returned in **`%eax`**
- C convention: always return value in **`%eax`** or **`%rax`**

function3.c

```
int foo(int a, int b)
{
    return a + b;
}

int main()
{
    int x = foo(4, 6);
    return 0;
}
```

`main` in `function3.s`

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp  #this is new
        movl    $6, %esi
        movl    $4, %edi
        call    foo
        movl    %eax, -4(%rbp) #getting answer from foo()
        movl    $0, %eax
        leave              #this is new too
        ret
```

- The assembler knows to look for **`foo()`** return value in **`%eax`**
- Copied onto stack for later use (although it is never used)
- **`subq`** moves stack pointer, reserving local storage space on stack
- **`leave`** is syntactic sugar for **`movq %rbp,%rsp`** followed by **`popq %rpb`**
- Releases frame (space between **`%rsp`** and **`rbp`**) & restores prev stack frame
- Opposite of what you see at the at beginning of function (**`pushq`** & **`movq`**)

More memory space?

- We’ve seen two storage locations for memory
- Registers
- Stack
  - Make space by decrementing stack pointer
  - Reference space relative to base pointer
    - e.g. **`-4(%rbp)`**
- There are only about a million bytes of stack
- Only a few registers
- Where is the rest of memory?
- Heap!

heap.c

```c
#include <stdlib.h>

int main()
{
    int *x = (int *) malloc(sizeof(int));
    int y = *x;
    free(x);
    return 0;
}
```

heap.s

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $4, %edi
        call    malloc
        movq    %rax, -16(%rbp)
        movq    -16(%rbp), %rax
        movl    (%rax), %eax
        movl    %eax, -4(%rbp)
        movq    -16(%rbp), %rax
        movq    %rax, %rdi
        call    free
        movl    $0, %eax
        leave
        ret
```

heap.s

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $4, %edi #move 4 to %edi to for malloc param
        call    malloc #call malloc (get heap space)
        movq    %rax, -16(%rbp) #store malloc ret val on stack
        movq    -16(%rbp), %rax #get it back from stack
        movl    (%rax), %eax #parens = mov val pt’d by %rax
        movl    %eax, -4(%rbp) #store value on stack
        movq    -16(%rbp), %rax #get malloc val from stack
        movq    %rax, %rdi #pass to %rdi for free param
        call    free #call free (release heap space)
        movl    $0, %eax
        leave
        ret
```

Walking through x86-64 assembly

- For midterm 3 and HW 7, you need to at least know the following parts of x86-64 assembly
- Slightly simplified in version that follows
- Full documentation at http://www.x86-64.org/documentation/abi.pdf
- We use AT&T syntax
  - Source on left, destination on right
  - Note ABI standard uses opposite syntax!

General purpose registers

Register	Callee-save	Description
`%rax`		Result register, also used in **`imul`** & **`idiv`**
`%rbx`	yes	Miscellaneous register
`%rcx`		Fourth argument register
`%rdx`		Third argument register (used in **`imul`** & **`idiv`**)
`%rsp`		Stack pointer
`%rbp`	yes	Frame pointer (base pointer)
`%rsi`		Second argument register
`%rdi`		First argument register
`%r8`		Fifth argument register
`%r9`		Sixth argument register
`%r10`		Miscellaneous register
`%r11`		Miscellaneous register
`%r12-%r15`	yes	Miscellaneous register
`%rip`		Instruction pointer

Calling

- **`call`** pushes the address of next instruction (i.e., the return address) onto stack & transfers control to operand address
- **`leave`** sets stack pointer (%rsp) to frame pointer (%rbp) & sets frame pointer to saved frame pointer (popped from the stack)
- **`ret`** pops return address off stack & jumps to it

Addressing memory

Syntax	Address	Description
(reg)	reg	Base addressing
d(reg)	reg + d	Base addressing + displacement
d(reg, s)	(s x reg) + d	Scaled index + displacement (s = 2, 4, or 8)
d(reg1, reg2, s)	reg1 + (s x reg2) + d	Base + scaled index + displacement (s = 2, 4, or 8)

Opcodes

- X86-64 has many, many opcodes
- Here are some families you should know (but you may encounter more in assignments):

`add`, `and`, `call`, `cmp`, `idiv`, `imul`, `jmp`, `lea` (load effective address), `mov`, `nop`, `or`, `pop`, `push`, `ret`, `sal`, `sar`, `shr` (shift codes), `sub`, `xor`

Result flags

- CF – carry flag
- PF – parity flag
- ZF – zero flag
- SF – sign flag
- OF – overflow flag

Jump opcodes

Instructions	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JE, JZ	Jump if equal (zero)	ZF = 0
JNE, JNZ	Jump if not equal (not zero)	ZF = 1
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JP, JPE	Jump if parity (parity even)	PF = 1
JNP, JPO	Jump if not parity (parity odd)	PF = 0

Jump opcodes (unsigned)

Instructions	Description	Flags
JB, JNAE, JC	Jump if below (carry, not above or equal)	CF = 1
JNB, JAE, JNC	Jump if not below (not carry, above or equal)	CF = 0
JBE, JNA	Jump if below or equal (not above)	CF = 1 or ZF = 1
JA, JNBE	Jump if above (not below or equal)	CF = 0 and ZF = 0

Jump opcodes (signed)

Instructions	Description	Flags
JL, JNGE	Jump if less (not greater or equal)	SF ≠ OF
JGE, JNL	Jump if greater or equal (not less)	SF = OF
JLE, JNG	Jump if less or equal (not greater)	ZF = 1 or SF ≠ OF
JG, JNLE	Jump if greater (not less or equal)	ZF = 0 and SF = OF