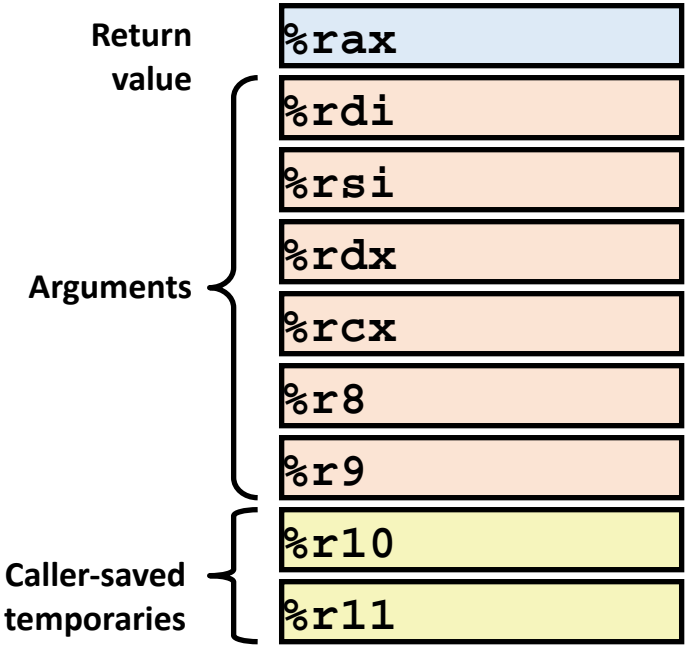
Reverse Engineering with gdb

Background for Lab 2: Bomblab, CS 208

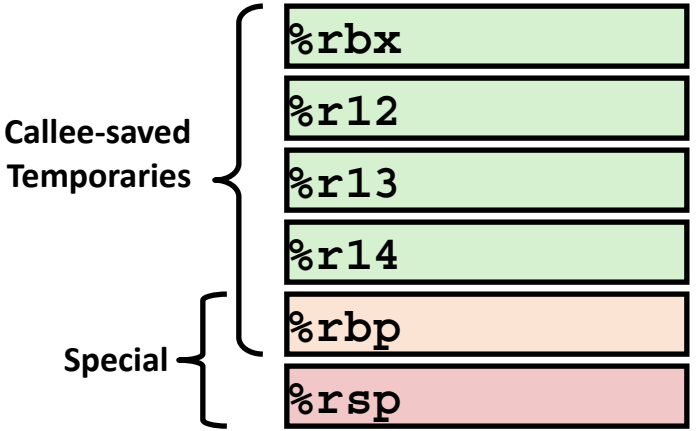
x86-64 Linux Register Usage #1

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, …, `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`
  - Callee-saved
  - Callee must save & restore
  - (i.e., these registers must have the same value when the procedure returns as they did when it started)
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as a frame pointer
  - Can mix & match
- `%rsp`
  - Stack pointer, special form of callee save
  - Restored to original value upon exit from procedure



x86-64 Linux Register Usage #3

Most Important Registers:

- `%rax:` return value
- `%rsp:` stack pointer
- `%rdi:` first argument
- `%rsi:` second argument

Helpful GDB Commands

disassemble: displays assembly

```
int squareInt(int x) {
    return x * x;
}
```

```
(gdb) disassemble squareInt

Dump of assembler code for function
squareInt:
0x000000000040091d <+0>: mov %edi,%ea
0x000000000040091f <+2>: imul %edi,%e
0x0000000000400922 <+5>: retq
End of assembler dump.
```

** disas != disa in gdb! Be careful with these shortcuts on bomblab

Helpful GDB Commands

Breakpoints: stops execution of program when it reaches certain point

- `break function_name`: breaks once you call a specific function
- `break *0x…`: breaks when you execute instruction at a certain address
- `info b`: displays information about all breakpoints currently set
- `disable #`: disables breakpoint with id equal to #

Helpful GDB Commands

Navigating through assembly:

- `stepi`: moves one instruction forward, will step into functions encountered
- `nexti`: moves one instruction forward, skips over functions called
- `c`: continues execution until next breakpoint is hit

What to do

- Don’t understand what a big block of assembly does? GDB
- Need to figure out what’s in a specific memory address? GDB
- Can’t trace how 4 – 6 registers are changing over time? GDB
- Have no idea how to start the assignment? Writeup
- Need to know how to use certain GDB commands? Writeup
  - Also useful: http://csapp.cs.cmu.edu/3e/docs/gdbnotes-x86-64.pdf
  - GDB intro video: https://courses.cs.washington.edu/courses/cse351/videos/tutorials/gdb.mp4
  - Many resources: http://cs.carleton.edu/faculty/awb/cs208/f21/#gdb-resources
- Don’t know what an assembly instruction does? Topic notes/textbook
- Confused about control flow or stack discipline? Topic notes/textbook

Basic GDB tips

- Many commands have shortcuts. Dissasemble → disas. Disable → dis
  - Do not mix these up! Disable will disable all your breakpoints, which may cause you to blow up your bomb.

- (gdb) print [any valid C expression]
  - This can be used to study any kind of local variable or memory location
  - Use casting to get the right type (e.g. print *(long *)ptr)

- (gdb) x [some format specifier] [some memory address]
  - Examines memory. See the handout for more information. Same as print *(addr), but more convenient.

- (gdb) set disassemble-next-line on
  (gdb) show disassemble-next-line
  - Shows the next assembly instruction after each step instruction

- (gdb) info registers Shows the values of the registers
- (gdb) info breakpoints Shows all current breakpoints
- (gdb) quit Exits gdb

Quick Assembly Info

- $rdi holds the first argument to a function call, $rsi holds the second argument, and $rax will hold the return value of the function call.
- Many functions start with “push %rbx” and end with “pop %rbx”. Long story short, this is because %rbx is “callee-saved”.
- The stack is often used to hold local variables
  - Addresses in the stack are usually in the 0x7fffffff… range
- Know how $rax is related to $eax and $al.
- Most cryptic function calls you’ll see (e.g. callq … <_exit@plt>) are calls to C library functions. If necessary, use the Unix man pages to figure out what the functions do.

Quick Assembly Info

- $ objdump -d [name of executable] > [any file name]
  - Saves the assembly code of the executable into the file.
  - Feel free to annotate the assembly in your favorite text editor.

```
[dalud@angelshark:~/.../15213/s17/bomb16] $ objdump -d example > example.asm
```

```
0000000000400560 <function>:
  400560: 48 83 ec 18           sub    $0x18,%rsp              // Setting things up
  400564: 48 89 7c 24 08        mov    %rdi,0x8(%rsp)
  400569: 48 83 7c 24 08 00     cmpq   $0x0,0x8(%rsp)          // Checks $rdi against 0
  40056f: 74 0a                 je     40057b <function+0x1b>  // Jumps to the "if branch" if equal
  400571: b8 00 00 00 00        mov    $0x0,%eax
  400576: e8 0a 00 00 00        callq  400585 <quit>           // Calls "quit" (else branch)
  40057b: b8 01 00 00 00        mov    $0x1,%eax               // Makes $rax = 1 (if branch)
  400580: 48 83 c4 18           add    $0x18,%rsp              // Cleaning stuff up
  400584: c3                    retq                           // Looks like the return value is 1
```

Activity Walkthrough

- $ make
- $ cat gdb-activity.c // display the source code of gdb-activity
- $ gdb gdb-activity
- (gdb) disassemble compare
- Q. Where is the return value set in compare?

- (gdb) break compare
- Now run gdb-activity with two numbers
- Q. Using nexti or stepi, how does the value in register %rbx change, leading to the cmp instruction?

- (gdb) run 200 3
- About to run push %rbx
- $rdi = 200
- $rsi = 3
- $rbx = [$rbx from somewhere else]
- $rax = [garbage value]

- Stack:

[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run mov %rdi, %rbx
- $rdi = 200
- $rsi = 3
- $rbx = [$rbx from somewhere else]
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run add $0x5, %rbx
- $rdi = 200
- $rsi = 3
- $rbx = 200
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run add %rsi, %rbx
- $rdi = 200
- $rsi = 3
- $rbx = 205
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run cmp 0xd0, %rbx
  & other instructions
- $rdi = 200
- $rsi = 3
- $rbx = 208 (= 0xd0)
- $rax = [garbage value]

- Stack:

[$rbx from somewhere else]

[some old stack items]

- (gdb) nexti
- (gdb) nexti
- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run pop %rbx
- $rdi = 200
- $rsi = 3
- $rbx = 208 = 0xd0
- $rax = 1

- Stack:

[$rbx from somewhere else]
[some old stack items]

- (gdb) nexti

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

- About to run retq
- $rdi = 200
- $rsi = 3
- $rbx = [$rbx from somewhere else]
- $rax = 1

- Stack:

[some old stack items]

```
push   %rbx
mov    %rdi,%rbx
add    $0x5,%rbx
add    %rsi,%rbx
cmp    $0xd0,%rbx
sete   %al
movzbq %al,%rax
pop    %rbx
retq
```

What is Bomb Lab?

- An exercise in reading x86-64 assembly code.
- A chance to practice using GDB (a debugger).

- Why?
  - x86 assembly is low level machine code. Useful for understanding security exploits or tuning performance.

  - GDB can save you days of work in future labs *cough Malloc cough* and can be helpful long after you finish this class.

Downloading Your Bomb

- Here are some highlights of the write-up:
  - Each bomb is unique
  - Bombs have six phases which get progressively harder.
  - Make sure to read the writeup for more tips and common mistakes you might make.

Detonating Your Bomb

- Blowing up your bomb doesn’t cost you, but it does print “BOOM!!!”
  - It’s very easy to prevent explosions using break points in GDB.
- Inputting the correct string moves you to the next phase.
- Don’t tamper with the bomb. Skipping or jumping between phases detonates the bomb.
- You have to solve the phases in order they are given.

Bomb Hints

- ***Mr. Dr. The Professor*** may be evil, but he isn't cruel. You may assume that functions do what their name implies
  - i.e. phase_1() is most likely the first phase. printf() is just printf(). If there is an explode_bomb() function, it would probably help to set a breakpoint there!
- Use the man pages for library functions.
  - Although you can examine the assembly for snprintf(), we assure you that it's easier to use the man pages ($ man snprintf) than to decipher assembly code for system calls.
- Most cryptic function calls you'll see (e.g. callq … <_exit@plt>) are also calls to C library functions.
  - You can safely ignore the @plt as that refers to dynamic linking.