15-213 Recitation Bomblab

Your TAs

Friday, September 5th

Reminders

- `datalab` is due on ***Tuesday (Sep 9)***.
- `bomblab` is out! Due *September 16*.
- Bootcamp 2: *Debugging & GDB* is pre-recorded. Watch Ed for the link.

Agenda

- Assembly Refresher
- Preview: Calling Conventions
- Intro to `bomblab`
- `bomblab` defuse kit
- `gdb` activity

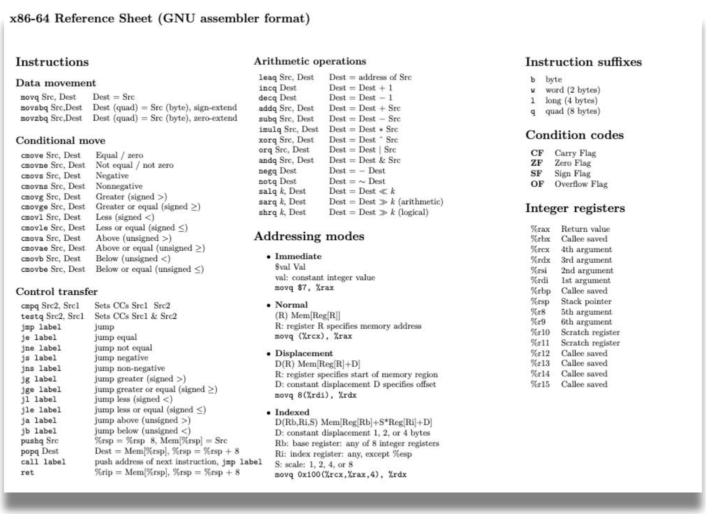
Assembly Refresher

Reading Assembly

- We will use ***AT&T*** syntax in this class:

AT&T	***Intel***
`movq Src, Dest` `addq Src, Dest`	`movq Dest, Src` `addq Dest, Src`

- If you get stuck, refer to our assembly cheat sheet!



x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed ≥)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed ≤)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned ≥)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned ≤)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest ∗ Src
`xorq` Src, Dest	Dest = Dest ˆ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = ∼ Dest
`salq` *k*, Dest	Dest = Dest ≪ *k*
`sarq` *k*, Dest	Dest = Dest ≫ *k* (arithmetic)
`shrq` *k*, Dest	Dest = Dest ≫ *k* (logical)

Addressing modes

- **Immediate**
  `$val` Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Reading Assembly: Operands

Constants ("Immediate" Values)

- Start with `$`

`$-15213` — *Decimal*

`$0x3b6d` — *Hex*

Registers

- Can store values or addresses
- Start with `%`

`%rax` — *"Return" Register*

`%eax` — *Low 32 bits of* `%rax`

Memory Locations

- Parentheses around a register, or an addressing mode

`(%rbx)` — *Normal*

`0x1c(%rax)` — *Displacement*

`0x4(%rcx, %rdi, 0x1)` — *Indexed*

Reading Assembly: Addressing Modes

Displacement

- D(R) Mem[Reg[R] + D]

```
movq 8(%rdi), %rdx
```

D: Constant **D**isplacement

R: **R**egister holding starting address

Indexed

- D(Rb, Ri, S) Mem[Reg[Rb] + S*Reg[Ri] + D]

```
movq 0x100(%rcx,%rax,4), %rdx
```

D: Constant **D**isplacement

Rb: **B**ase **R**egister holding starting address

Ri: **I**ndex **R**egister

S: **S**cale (1, 2, 4, 8)

Reading Assembly: Examples

Instruction	*Effect*
`mov %rbx, %rdx`	`rdx = rbx`
`add (%rdx), %r8`	`r8 += value at address in rdx`
`mul $3, %r8`	`r8 *= 3`
`sub $1, %r8`	`r8--`
`lea (%rdx, %rbx, 2), %rdx`	`rdx = rdx + rbx * 2`

No dereferencing!

Reading Assembly: Comparisons

Example

```
cmpl %r9, %r10
jg 8675309
```

- *"If the value of one register is greater than the value in the other, then jump to 8675309"*
- But which way around is it?
- Let's use the cheat sheet!

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed ≥)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed ≤)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned ≥)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned ≤)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest ∗ Src
`xorq` Src, Dest	Dest = Dest ˆ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = ∼ Dest
`salq` *k*, Dest	Dest = Dest ≪ *k*
`sarq` *k*, Dest	Dest = Dest ≫ *k* (arithmetic)
`shrq` *k*, Dest	Dest = Dest ≫ *k* (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)

- **`Src1`** is **`%r10`**, **`Src2`** is **`%r9`**
- Set CCs based on **`Src1 <op> Src2`**, where **<op>** := **>**

```
cmpl %r9, %r10
jg 8675309
```

- So we jump if: **`%r10 > %r9`**
- *"If the value of* **`%r10`** *is greater than the value in* **`%r9`***, then jump to 8675309"*

Reading Assembly: Jumps

Instruction	Condition	Description
`jmp`	`1`	Unconditional Jump
`je/jz`	`ZF`	Equal/Zero
`jne/jnz`	`~ZF`	Not Equal/Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Non-negative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF & ~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Reading Assembly: Jumps

```
cmp $0x15213, %r12
jge deadbeef
```

If **`%r12 >= 0x15213`**, then jump to **`0xdeadbeef`**.

```
cmp %rax, %rdi
jae 15213b
```

If the *unsigned* value in **`%rdi`** is greater than or equal to the *unsigned value* in **`%rax`**, jump to **`0x15213b`**.

```
test %r8, %r8
jnz *%rsi
```

If **`%r8`** is not zero, jump to the address stored in **`%rsi`**.

Preview: Calling Conventions

Calling Conventions: Passing Control

- How can we pass *control* from the assembly for the current function to the assembly for the function we want to call?
- How can we pass *control* back to the caller once we're done?

```
0000000000400540 <multstore>:
400540: push %rbx # Save %rbx
400541: mov %rdx,%rbx # Save dest
400544: call 400550 <mult2> # mult2(x,y)
400549: mov %rax,(%rbx) # Save at dest
40054c: pop %rbx # Restore %rbx
40054d: ret # Return
```

```
0000000000400550 <mult2>:
400550: mov %rdi,%rax # a
400553: imul %rsi,%rax # a * b
400557: ret # Return
```

Procedure Call: **`call label`**

- Push *return address* onto the stack (so that we can pass control back to the caller!)
- Jump to **`label`**

```
0000000000400540 <multstore>:
400540: push %rbx # Save %rbx
400541: mov %rdx,%rbx # Save dest
400544: call 400550 <mult2> # mult2(x,y)
400549: mov %rax,(%rbx) # Save at dest
40054c: pop %rbx # Restore %rbx
40054d: ret # Return
```

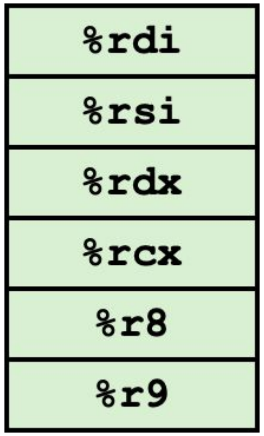
```
0000000000400550 <mult2>:
400550: mov %rdi,%rax # a
400553: imul %rsi,%rax # a * b
400557: ret # Return
```

Procedure Return: `ret`

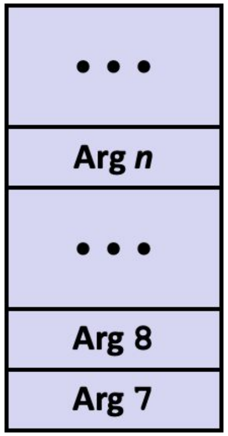
- Pop address from stack
  - This is the address of the next instruction of the *caller*
- Jump to that address

Calling Conventions: Passing Data

- How can we pass arguments to a procedure?



First 6 arguments passed in ***registers***.



Remaining arguments put at the end of the ***caller's stack frame***.

Calling Conventions: Passing Data

- How can we access the return value?

`%rax`

Return value placed in %rax
by convention.

Calling Conventions: Caller/Callee-Saved

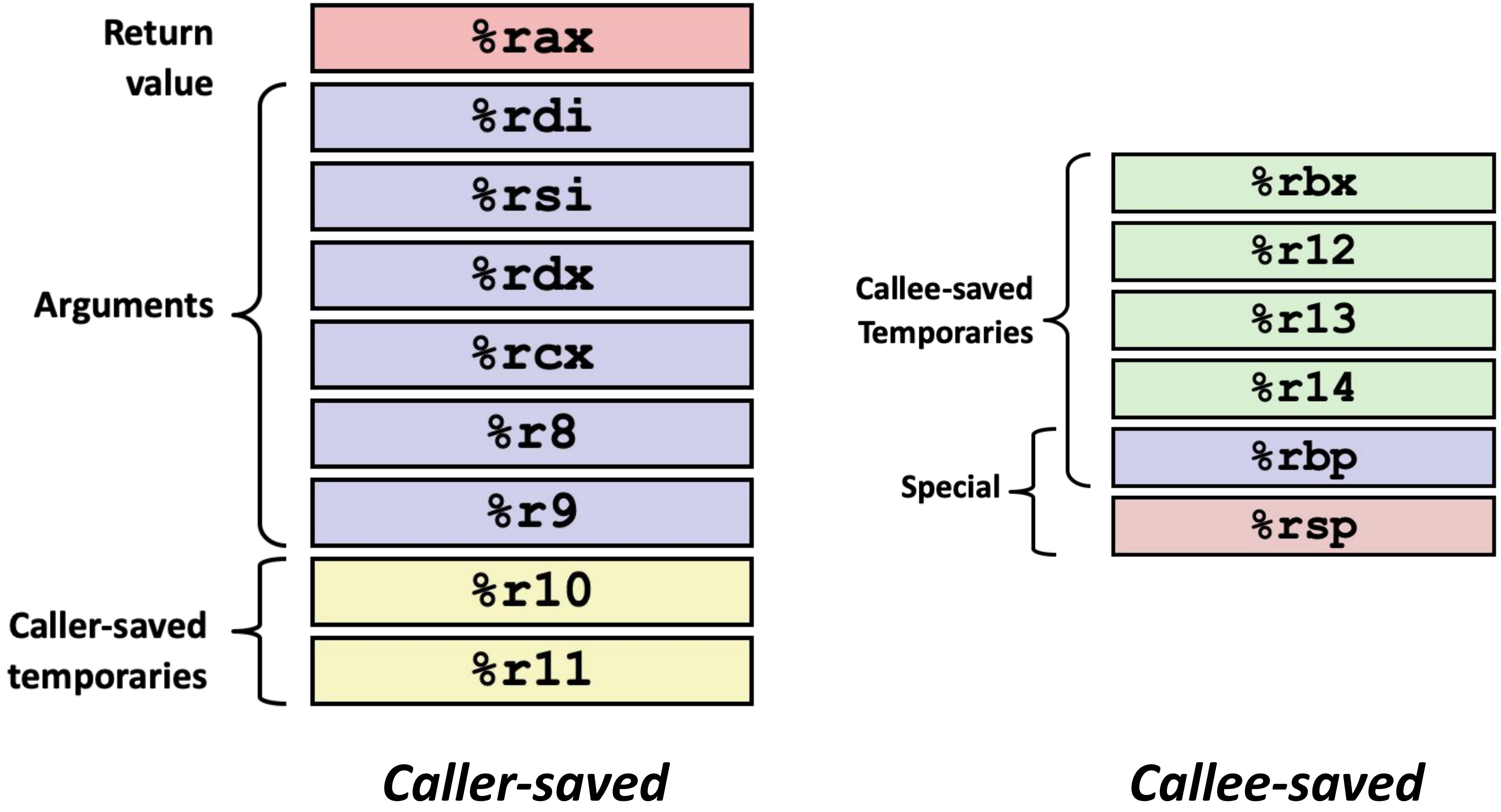
- If **`foo()`** calls **`bar()`**:
  - **`foo()`** is the ***caller***
  - **`bar()`** is the ***callee***
- Both **`foo()`** and **`bar()`** want to use registers.
  - How can **`bar()`** use a register without overwriting something **`foo()`** was using?
- Need a consistent ***convention*** for saving/overwriting registers so we don't lose data.

Calling Conventions: Caller/Callee-Saved

- ***Caller-Saved ("Call Clobbered")***
  - Procedures can overwrite these registers freely
  - So these registers can be "clobbered" (overwritten) by a call
  - So the caller has to save them!
- ***Callee-Saved ("Call Preserved")***
  - The callee will save these values on the stack before using them.
  - Before the callee returns, it restores these registers from the stack.

Calling Conventions: Caller/Callee-Saved

- Which registers are caller/callee-saved?



Bomblab

Bomblab: Premise

- ***Dr. Evil*** has planted *binary bombs* on our shark machines!
- Your task: defuse your bomb by passing the correct strings on **`stdin.`**
- You get:
  - A C source file for the *main program*
  - An executable (no C source code for the phases!)
- Have to reverse engineer the bomb using only **`gdb`** and the assembly code!

Bomblab: Getting Started

- Download your bomb from Autolab
- You must use the **Shark Machines** to extract (untar) and work on your Bomb.
- Run `autolab setup`
- 6 Progressively Harder Phases
  - Enter the correct string to move on to the next phase
- Read the write up! It has an entire page dedicated to hints!

Hints ***(Please read this!)***

There are many ways of defusing your bomb. You can examine it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to defuse it. This is probably the fastest way of defusing it.

Bomblab: Detonating Your Bomb

- Solving a phase automatically notifies Autolab and applies points to your score.
- If you let the bomb explode, Autolab will **deduct 0.5 points** *each time*.
- ***Do <u>not</u>*:**
  - Use gdb to jump between phases
  - Solve the phases out of order
  - Tamper with the bomb
  - Otherwise the bomb will explode!

Bomblab: Defuse Kit

Defuse Kit: `gdb`

- **`gdb`** = GNU Debugger
- Fully-featured debugger:
  - For bomblab, lets you trace the execution of assembly
  - Useful for future labs, and well beyond 213.
  - Expand your debugging toolkit beyond **`printf`**!

Defuse Kit: gdb

Examining Program State

`print (p)`

`print $rdi`

Print contents of %rdi

```
(gdb) print /d 0x3b6d
$2 = 15213
```

Print with format

`info`

`info registers`

Print all register contents

`x` (For eXamine)

- `x /[num][size][format]`
- `x /s 0x…` Examine contents of address as a string
- `x /64bx 0x…` View 64 bytes starting at the given address in Hex Format

GDB Demo

If you want to follow along… ***(you'll also need this for the activity)***

- Download today's activity handout from the *Schedule* page.

```
$ wget http://www.cs.cmu.edu/~213/activities/f25-rec2.tar
$ tar xvpf f25-rec2.tar
$ cd f25-rec2
$ make
```

Defuse Kit: Getting the Assembly

- Use **`objdump`** to get assembly code from your executable:
  - Then open and annotate in your favorite text editor!

```
objdump -d act1 > act1.asm
```

For syntax highlighting!

Defuse Kit: Getting the Assembly (pt2).

- In **`gdb`**, type **`disassemble <function_name>`**
- This will allow you to view the assembly for that function only (rather than for the entire executable, as in **`objdump`**)

Defuse Kit: Identifying inputs to `main()`

- We see **`int main(int argc, char** argv)`**
  - **`main`** is also a function - we follow calling conventions
  - **`argc => %rdi, argv => %rsi`**
- Note that **`argv`** is a pointer type (array of arguments), meaning we must dereference to access the arguments!
  - Look out for addressing mode around **`%rsi`**

Defuse Kit: Figuring out Input Format

- Phases use **`sscanf`** to parse input strings:

```
char *input_string = "123, 456";
int a, b;
sscanf(input_string, "%d, %d", &a, &b);
```

```
...
0x0000000000401ab4 <+15>:    mov    -0x8(%rsi,%rdi,8),%rdi
...
0x0000000000401ac3 <+30>:    lea    0xb453a(%rip),  %rsi        # 0x4b6004
0x0000000000401aca <+37>:    mov    $0x0,%eax
0x0000000000401acf <+42>:    call   0x40ba10 <__isoc99_sscanf>
...
```

We know that the format string is the second argument (**`%rsi`**)

`0x4b6004` is the address of that string!

Defuse Kit: Figuring out Input Format

```
...
0x0000000000401ac3 <+30>:    lea    0xb453a(%rip),  %rsi        # 0x4b6004
0x0000000000401aca <+37>:    mov    $0x0,%eax
0x0000000000401acf <+42>:    call   0x40ba10 <__isoc99_sscanf>
...
```

- If we can examine that memory address, we can recover the format string!
- Enter *gdb*:

```
(gdb) break main
Breakpoint 1 at 0x401aa5
(gdb) x /s 0x4b6004
0x4b6004: "%d, %d"
```

Examine memory address as a *string*.

We need two integers!

Warning: TUI Mode

TUI Mode

- Is very cool (can view assembly alongside **`gdb`** prompt).
- But can unexpectedly ***explode your bomb.***
- You will not get these points back.
- Can use **`vim`**/VSCode splitting instead.

GDB Activity

GDB Activity

- View the assembly and source code for **`act2`**
- Our objective is to match the source code to the assembly, identifying which sections correspond to each other!
- Get into groups of 3-4 and discuss together on how to interpret the assembly!
- If you understand the correlation fully along with the control flow in the assembly, feel free to try and solve the puzzle.