Introduction to GDB & Bomblab

Tang Tang. Mar 12

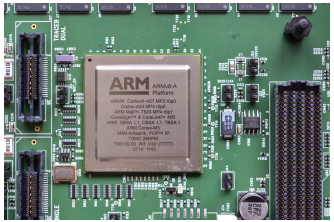
Outline

- Bomblab Intro
  - Why Bomblab
  - Bomb Structure
- X86-64 Recap
  - Syntax
  - Registers
  - Operands
  - Control
  - Calling Conventions
- Bomb Defuse Kit
  - Objdump
  - GDB

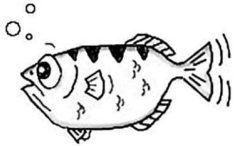
Bomblab Intro

Why bomblab ?

- A chance to read X86-64 Assembly code
  - Hardware/Software Interface
  - System programming (compiler, os) and embedding system
  - Performance optimization
  - More easier to learn modern **RISC** assembly code
- A chance to practice using GDB (a debugger)
  - Command Line Interface
  - Effectively debugging (backtrace, break point, print …)
  - Much stronger than printf()
  - Reverse engineering



RISC-V®



GDB
The GNU Project
Debugger

You can not learn well by books only ！

Bomb structure

- Bomb structure
  - All you need is to **input a string** at each phase
  - 6 phases (easy to hard)
  - Secret phase
    - Where to enter ?
    - How functions is called ?
  - **Defusing bombs phase by phase !!!**
- Be careful to **sscanf(str, format, args …)**
  - First argument is a string
  - Second argument is format string
  - **Reading from a string instead of stdin**
  - **Return # of arguments successfully reading into defined by format string**

```
/* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                    */
    phase_1(input);                  /* Run the phase                */
    phase_defused();                 /* Drat!  They figured it out!
                    * Let me know how they did it. */

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();

    /* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
    input = read_line();
    phase_4(input);
    phase_defused();

    /* Round and 'round in memory we go, where we stop, the bomb blows! */
    input = read_line();
    phase_5(input);
    phase_defused();

    /* This phase will never be used, since no one will get past the
     * earlier ones.  But just in case, make this one extra hard. */
    input = read_line();
    phase_6(input);
    phase_defused();
```

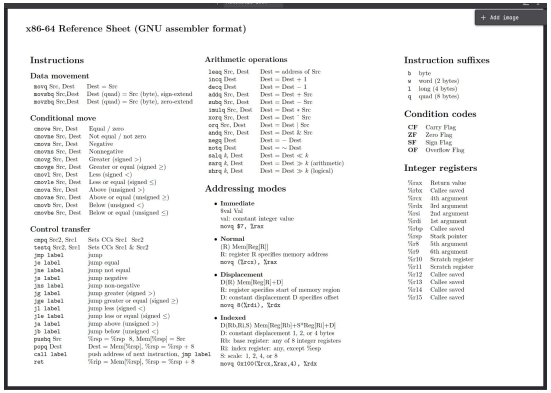
X86-64 Assembly code Recap

X86-64 Recap: Syntax

- AT&T syntax
  - GNU/GCC
  - Default format for objdump (GNU Binutils)
- Intel syntax
  - Microsoft (Visual Studio)
- Differences
  - Prefix
    - $ vs None
  - Directions of Operands
  - Memory Operands
    - () vs []

Only AT&T is used in this class !!!

Refer to 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 ^ 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
  final 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 %rsp
  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

Intex Syntax		AT&T Syntax	
mov	eax,1	movl	$1,%eax
mov	ebx,0ffh	movl	$0xff,%ebx
int	80h	int	$0x80

Intex Syntax		AT&T Syntax	
instr	dest,source	instr	source,dest
mov	eax,[ecx]	movl	(%ecx),%eax

Intex Syntax		AT&T Syntax	
mov	eax,[ebx]	movl	(%ebx),%eax
mov	eax,[ebx+3]	movl	3(%ebx),%eax

X86-64 Recap: Registers

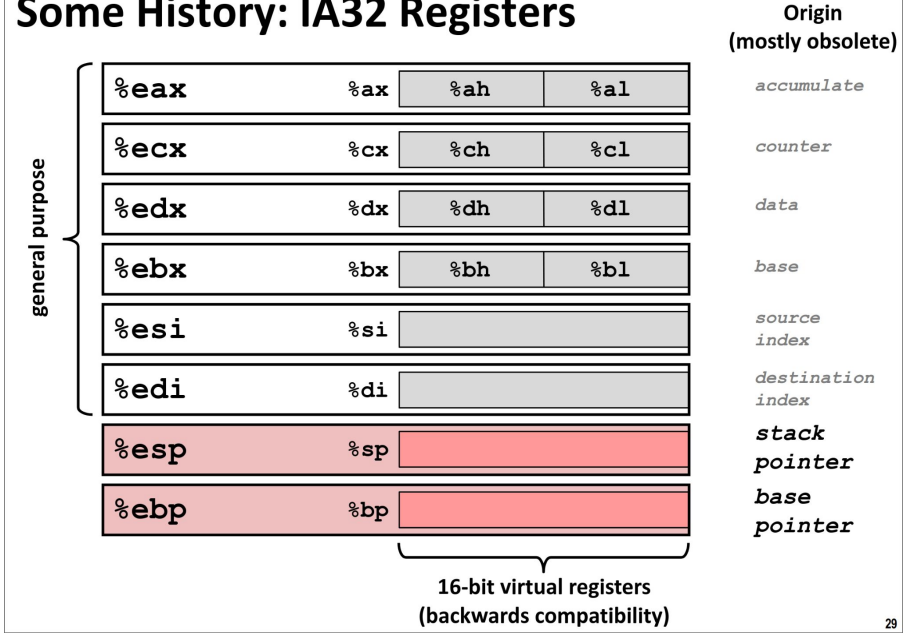
- 64-bit CPU ?
- 64-bit Operating System ?

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers



- Program Counter (PC)
  - **%rip**: points to the next instruction address

X86-64 Recap: Operands

- Constants (“Immediate” Values)
  - Start with $
    - $5060
    - $0x148e (5060)
- Registers
  - Store values or memory address
  - Start with %
- Memory
  - (%rax) (Indirect)
    - Mem[Reg(%rax)]
  - **D**(%rax) (**D**isplacement)
    - Mem[D + Reg(%rax)]
    - D default to **0x0** (if D is omitted)
  - **D**(%rax, %rbx, **S**) (**S**caled indexed)
    - Mem[D + Reg(%rax) + Reg(%rbx) * S]
    - S default to 0x1 (if S is omitted)

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax,%rdx)	0x11	Address 0x10C
260(%rcx,%rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

X86-64 Recap: Operands

Type	Form	Operand value	Name
Immediate	$$\$Imm$$	$$Imm$$	Immediate
Register	$$\mathtt{r}_a$$	$$\mathtt{R}[\mathtt{r}_a]$$	Register
Memory	$$Imm$$	$$\mathtt{M}[Imm]$$	Absolute
Memory	$$(\mathtt{r}_a)$$	$$\mathtt{M}[\mathtt{R}[\mathtt{r}_a]]$$	Indirect
Memory	$$Imm(\mathtt{r}_b)$$	$$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b]]$$	Base + displacement
Memory	$$(\mathtt{r}_b,\mathtt{r}_i)$$	$$\mathtt{M}[\mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i]]$$	Indexed
Memory	$$Imm(\mathtt{r}_b,\mathtt{r}_i)$$	$$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i]]$$	Indexed
Memory	$$(,\mathtt{r}_i,s)$$	$$\mathtt{M}[\mathtt{R}[\mathtt{r}_i] \cdot s]$$	Scaled indexed
Memory	$$Imm(,\mathtt{r}_i,s)$$	$$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_i] \cdot s]$$	Scaled indexed
Memory	$$(\mathtt{r}_b,\mathtt{r}_i,s)$$	$$\mathtt{M}[\mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i] \cdot s]$$	Scaled indexed
Memory	$$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$$	$$\mathtt{M}[Imm + \mathtt{R}[\mathtt{r}_b] + \mathtt{R}[\mathtt{r}_i] \cdot s]$$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

X86-64 Recap: Operations

movq %rdx, %rax	=>	Reg(%rax) = Reg(%rdx)
movq (%rdx), %rax	=>	Reg(%rax) = Mem[Reg(%rdx)]
movq 0xc(%rdx), %rax	=>	Reg(%rax) = Mem[Reg(%rdx) + 0xc]
movq 0xc(%rdx, %rbx, 0x4)	=>	Reg(%rax) = Mem[Reg(%rdx) + Reg(%rbx)*0x4 + 0xc]
add $0x5060, %rax	=>	Reg(%rax) = Reg(%rax) + 0x5060
sub (%rax) , %rbx	=>	Reg(%rbx) = Reg(%rbx) - Mem[Reg(%rax)]
pushq %rax	=>	Reg(%rsp) **-=** 0x8, Reg(%rax) = Mem[Reg(%rsp)]
popq %rax	=>	Reg(%rax) = Mem[Reg(%rsp)], Reg(%rsp) **+=** 0x8
lea (%rdx, %rbx, 0x2), %rax	**=>**	**Reg(%rax) = Reg(%rdx) + Reg(%rbx) * 0x2**

X86-64 Recap: Control

- cmpq %rdx, %rax
  - **%rax - %rdx**, then set CCs
- test %rdx, %rax
  - **%rax & %rdx**, than set CCs

```
cmpq %rdx, %rax
jge 5060
```

If %rax >= $rdx, jmp to **0x5060**

```
cmpq %rdx, %rax
jae 5060
```

If **unsigned value** in %rax >= **unsigned value** in $rdx, jmp to **0x5060**

```
testq %rax, %rax
jne 5060
```

If %rax != 0, jmp to **0x5060**

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

X86-64 Recap: Control

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)

X86-64 Recap: Calling Conventions

Registers

First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

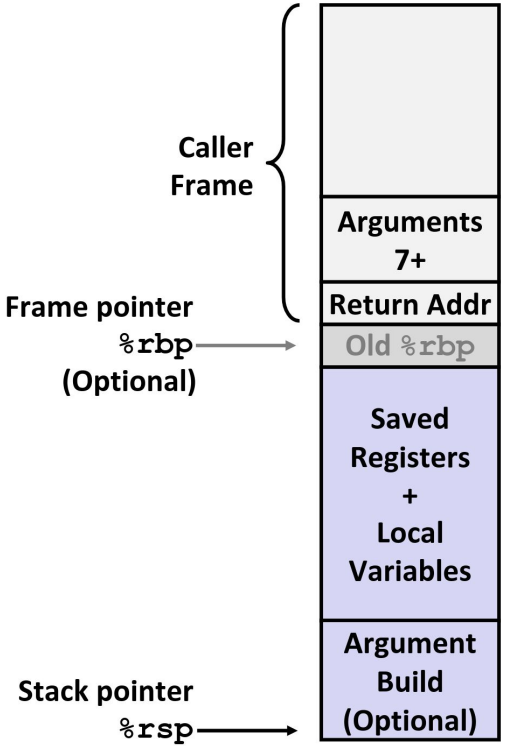
Return value

%rax

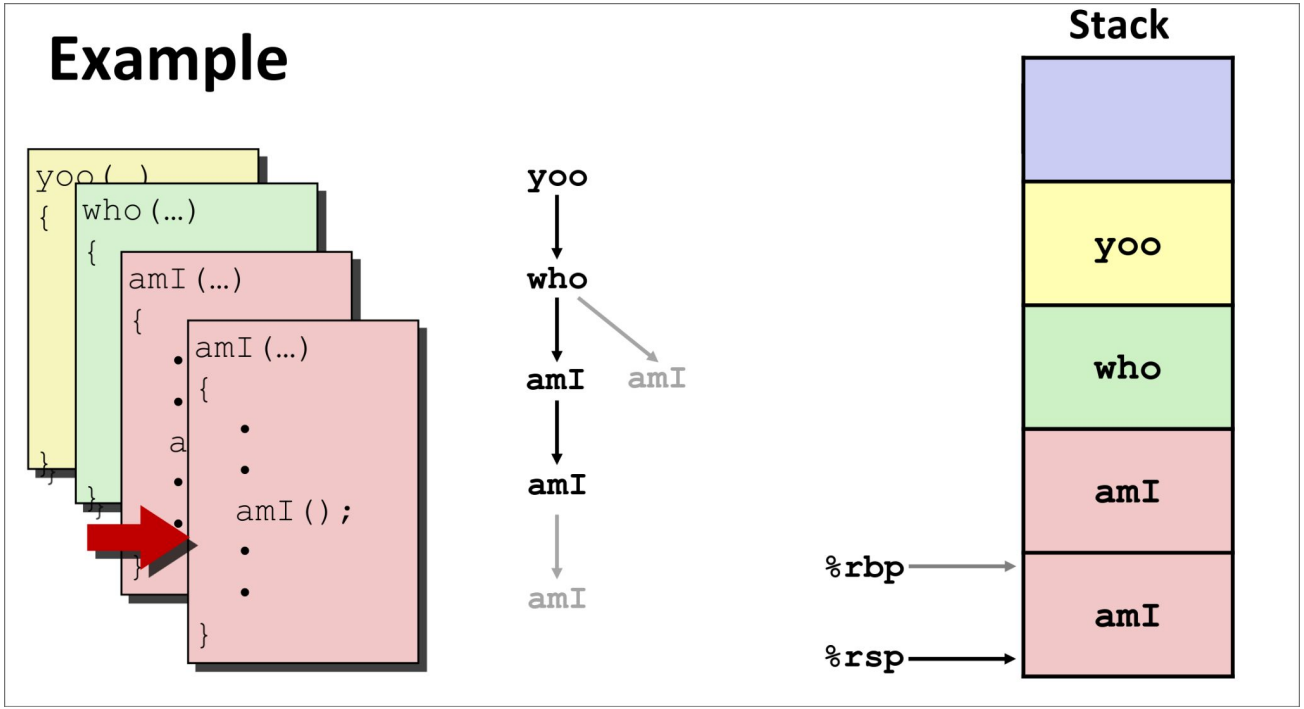
Stack

• • •
Arg *n*
• • •
Arg 8
Arg 7

Only allocate stack space when needed



X86-64 Recap: Recursion



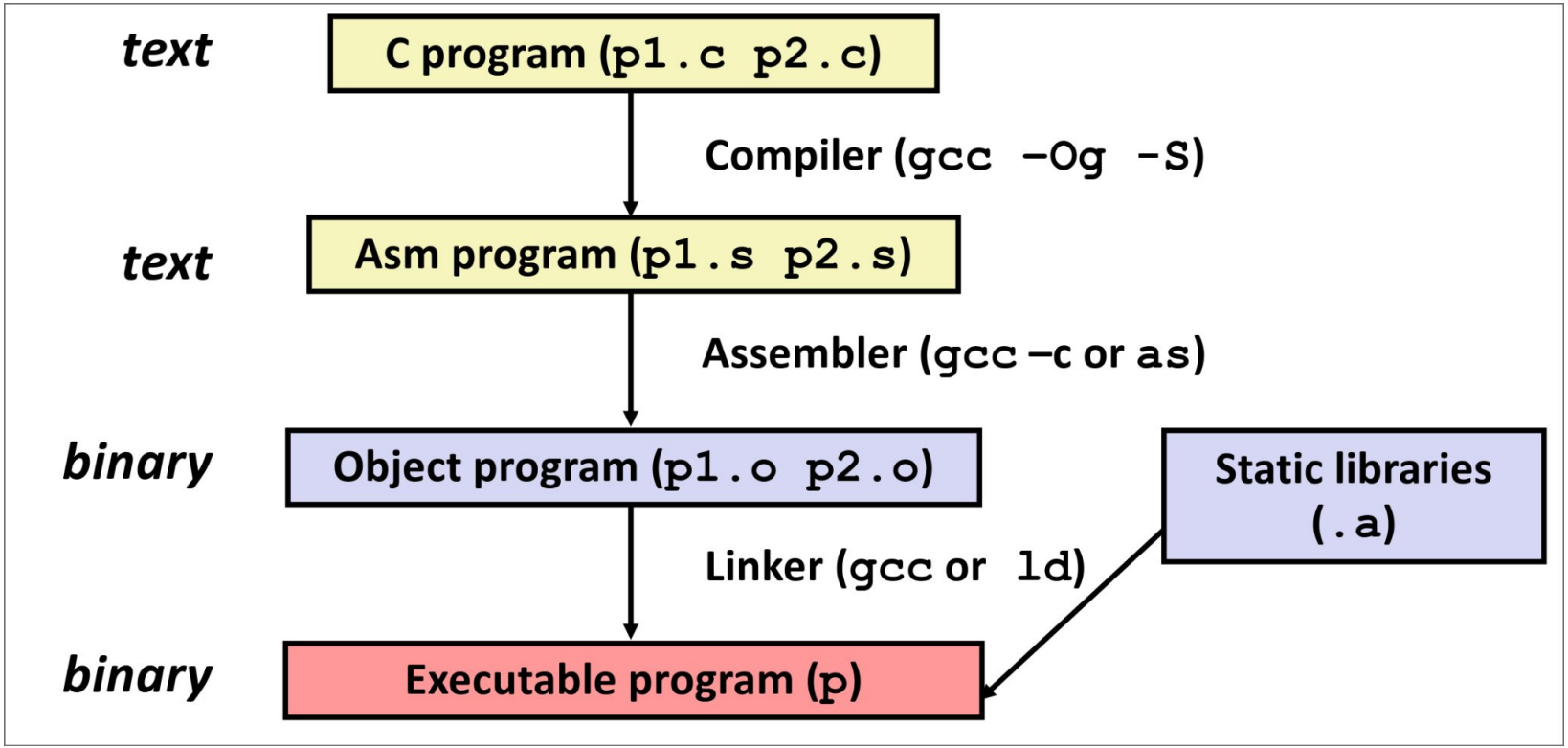
Stack overflow !

```
ulimit -s
8192
```

Usually 8MB (8192KB) limit

Bomb Defusing

Recap: C code to machine code



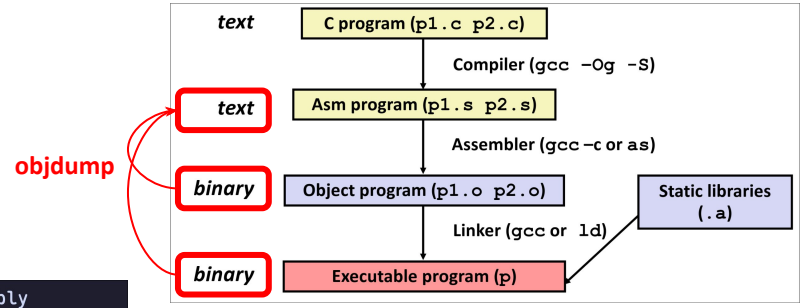
General defusing step

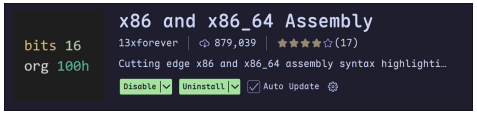
- Get the assembly code
  - Use **objdump**
- Reading the assembly
  - input format
  - core function
  - you may defuse the bomb immediately
- If we need to examine values of registers / memory location
  - Using **gdb**
  - break at read_line
  - reading the assembly

```
/* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                  */
    phase_1(input);                  /* Run the phase              */
    phase_defused();                 /* Drat!  They figured it out!
                      * Let me know how they did it. */

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();

    /* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
    input = read_line();
    phase_4(input);
    phase_defused();

    /* Round and 'round in memory we go, where we stop, the bomb blows! */
    input = read_line();
    phase_5(input);
    phase_defused();

    /* This phase will never be used, since no one will get past the
     * earlier ones.  But just in case, make this one extra hard. */
    input = read_line();
    phase_6(input);
    phase_defused();
```

Step1: Getting the assembly

- Use **objdump** to get the assembly code
  - objdump -d ./bomb > bomb.**asm** (syntax highlighting)
- Open it in the text editor






```
00000000000015ab <phase_1>:
    15ab:   f3 0f 1e fa             endbr64
    15af:   48 83 ec 08             sub    $0x8,%rsp
    15b3:   48 8d 35 f6 1a 00 00    lea    0x1af6(%rip),%rsi        # 30b0 <_IO_stdin_used+0xb0>
    15ba:   e8 20 05 00 00          call   1adf <strings_not_equal>
    15bf:   85 c0                   test   %eax,%eax
    15c1:   75 05                   jne    15c8 <phase_1+0x1d>
    15c3:   48 83 c4 08             add    $0x8,%rsp
    15c7:   c3                      ret
    15c8:   e8 26 06 00 00          call   1bf3 <explode_bomb>
    15cd:   eb f4                   jmp    15c3 <phase_1+0x18>
```

Demo: defusing phase A !

Phase A

- Getting the assembly

```
0000000000040154a <phase_a>:
  40154a: 53                    push   %rbx
  40154b: 48 89 fb              mov    %rdi,%rbx
  40154e: e8 99 fe ff ff        call   4013ec <string_length>
  401553: 48 83 f8 03           cmp    $0x3,%rax
  401557: 76 0e                 jbe    401567 <phase_a+0x1d>
  401559: 48 89 df              mov    %rbx,%rdi
  40155c: e8 b1 ff ff ff        call   401512 <func_1>
  401561: 84 c0                 test   %al,%al
  401563: 74 07                 je     40156c <phase_a+0x22>
  401565: 5b                    pop    %rbx
  401566: c3                    ret
  401567: e8 49 ff ff ff        call   4014b5 <explode_bomb>
  40156c: e8 44 ff ff ff        call   4014b5 <explode_bomb>
```

Keep in mind:

- **%rdi: address of our input string**

- Prologue
  - Make room for this stack frame
  - Save callee saved registers

- Get the length of input string
  - Length is saved in %rax

- **Boom** if length **<= 3**
  - Input string length must > 3

- Call func_1
  - Return value is saved in %rax
  - Boom if **%rax == 0**

- Postlude
  - Restore callee saved registers
  - Free spaces of current frame
  - Return to caller function

Phase A

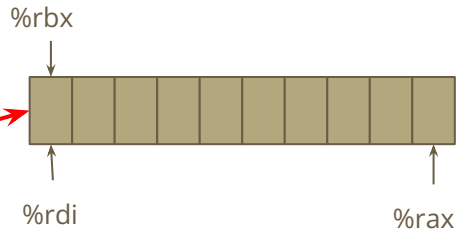
- What does func_1 do ?

```
0000000000401512 <func_1>:
  401512: 53                    push   %rbx
  401513: 48 89 fb              mov    %rdi,%rbx
  401516: e8 d1 fe ff ff        call   4013ec <string_length>
  40151b: 48 83 e8 01           sub    $0x1,%rax
  40151f: ba 00 00 00 00        mov    $0x0,%edx
  401524: eb 08                 jmp    40152e <func_1+0x1c>
  401526: 48 83 c2 01           add    $0x1,%rdx
  40152a: 48 83 e8 01           sub    $0x1,%rax
  40152e: 48 39 c2              cmp    %rax,%rdx
  401531: 73 10                 jae    401543 <func_1+0x31>
  401533: 0f b6 0c 03           movzbl (%rbx,%rax,1),%ecx
  401537: 38 0c 13              cmp    %cl,(%rbx,%rdx,1)
  40153a: 74 ea                 je     401526 <func_1+0x14>
  40153c: b8 00 00 00 00        mov    $0x0,%eax
  401541: eb 05                 jmp    401548 <func_1+0x36>
  401543: b8 01 00 00 00        mov    $0x1,%eax
  401548: 5b                    pop    %rbx
  401549: c3                    ret
```

Func_1 arguments ?

- **%rdi: address of our input string**
- **%rbx also points to that start of input string now**

- Get the length of input string
  - Length is saved in %rax

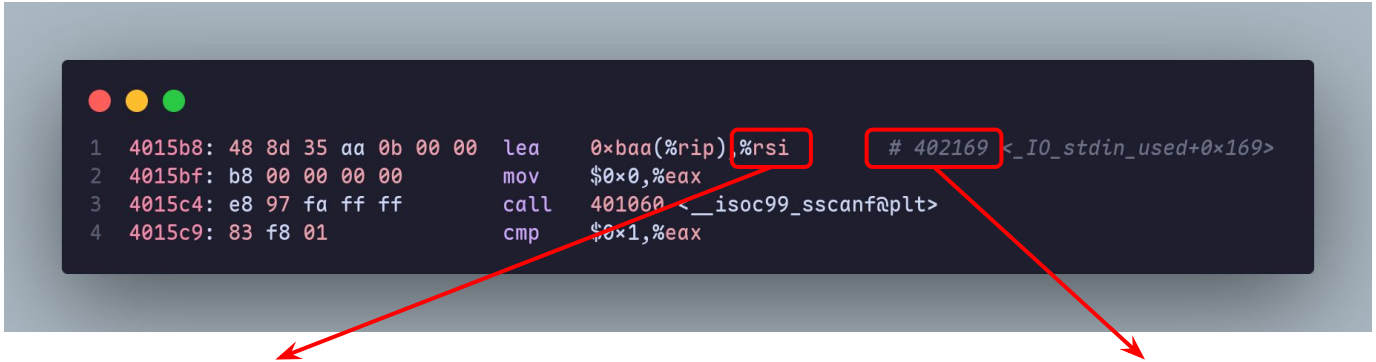


- What does loop do ?
  - If the char that %rax points to **is equal to** the char that %rdx points to, continue
  - Otherwise, quit with **%rax = 0**
  - Exit when **%rdx >= %rax** with **%rax = 1**
- **Palindromic string with length >= 3 !**

```
└─ ./bomb
This is the bomb.
Please enter input for Phase A:
abcba
Phase A defused. How about the next one?
```

Step1: Getting the assembly

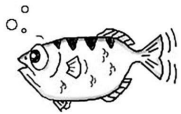
- Phase uses **sscanf()** to parse input string
  - sscanf(string, “%d, %d”, &a, &b)



The format string is the second argument (%rsi)

0x402169 is the address of that string !

- How to examine that memory address ?
  - **GDB is all you need !**



GDB
The GNU Project Debugger

Step2: Starting GDB

- Just open it in your shell
  - $ gdb
  - (gdb) file <executable>
  - (gdb) r (run)
- Specify what program to run
  - $ gdb <executable> ($ gdb ./bomb)
- Run with command line arguments
  - $ gdb --args <executable> <arguments> // gdb --args ./bomb solution.txt
  - $ gdb ./bomb
    - (gdb) r < solution.txt
- Quitting gdb
  - (gdb) q (quit)
  - Or just type <Ctrl-d>
- More options
  - $ gdb –h (--help)

Step2: Breakpoints

- A breakpoint makes your execution stop when the certain point is reached
- Set breakpoint
  - (gdb) b (break) function_name // breaks when the specified function is called
  - (gdb) b (break) *0x… // breaks when this address is reached
- List breakpoint
  - (gdb) i (info) b // display information about all currently set breakpoints
- Disable breakpoint
  - (gdb) disable # // disables breakpoint with ID equal to #
- Delete breakpoint
  - (gdb) clear [location] // delete breakpoints according to where they are

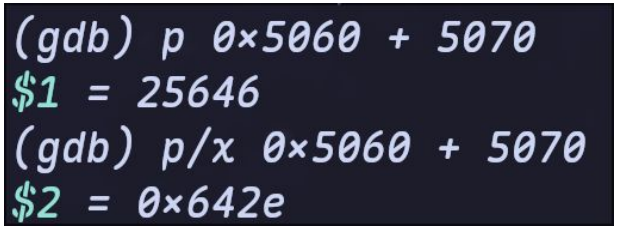
```
(gdb) b phase_a
Breakpoint 1 at 0x40154a
```

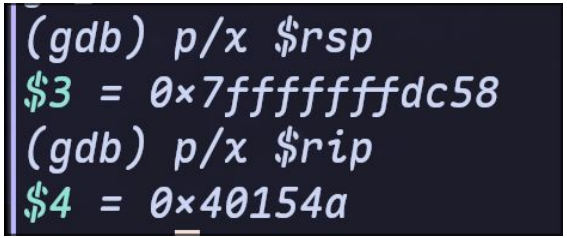
```
Breakpoint 1, 0x000000000040154a in phase_a ()
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040154a <phase_a>
        breakpoint already hit 1 time
(gdb)
```

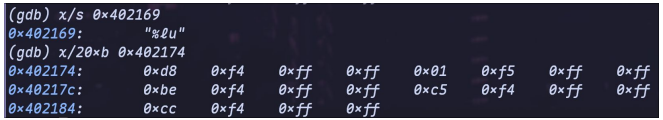
```
(gdb) clear phase_a
Deleted breakpoint 1
```

Step2: Print values and examining memory

- Print values
  - (gdb) p (print) <expr> // print the expression (value will be stored in variables start with $)
  - (gdb) p (print) /x <expr> // print in the hex format (also /d (decimal) /t (binary) …)
  - (gdb) i (info) r (registers) // print values of all registers
- Examine memory
  - (gdb) x/[nfu] <addr> // n: memory unit #, f: format (/x /d /t /s (string) …), u: memory unit size(b: 1 byte, h: 2 bytes, w: 4 bytes, g: 8 bytes)
    - (gdb) x /s 0x… // view the content as a string
    - (gdb) x /64xb 0x… // view 64 bytes from the given address in hex format







Step2: Displays assembly

- Disassemble
  - (gdb) disas function_name // disassemble source code into assembly code

This line is **ready** to be executed but not now

```
(gdb) p/x $rip
$1 = 0x40154a
```

```
(gdb) disas phase_a
Dump of assembler code for function phase_a:
=> 0x000000000040154a <+0>:     push   %rbx
   0x000000000040154b <+1>:     mov    %rdi,%rbx
   0x000000000040154e <+4>:     call   0x4013ec <string_length>
   0x0000000000401553 <+9>:     cmp    $0x3,%rax
   0x0000000000401557 <+13>:    jbe    0x401567 <phase_a+29>
   0x0000000000401559 <+15>:    mov    %rbx,%rdi
   0x000000000040155c <+18>:    call   0x401512 <func_1>
   0x0000000000401561 <+23>:    test   %al,%al
   0x0000000000401563 <+25>:    je     0x40156c <phase_a+34>
   0x0000000000401565 <+27>:    pop    %rbx
   0x0000000000401566 <+28>:    ret
   0x0000000000401567 <+29>:    call   0x4014b5 <explode_bomb>
   0x000000000040156c <+34>:    call   0x4014b5 <explode_bomb>
End of assembler dump.
```

Step2: Control your program

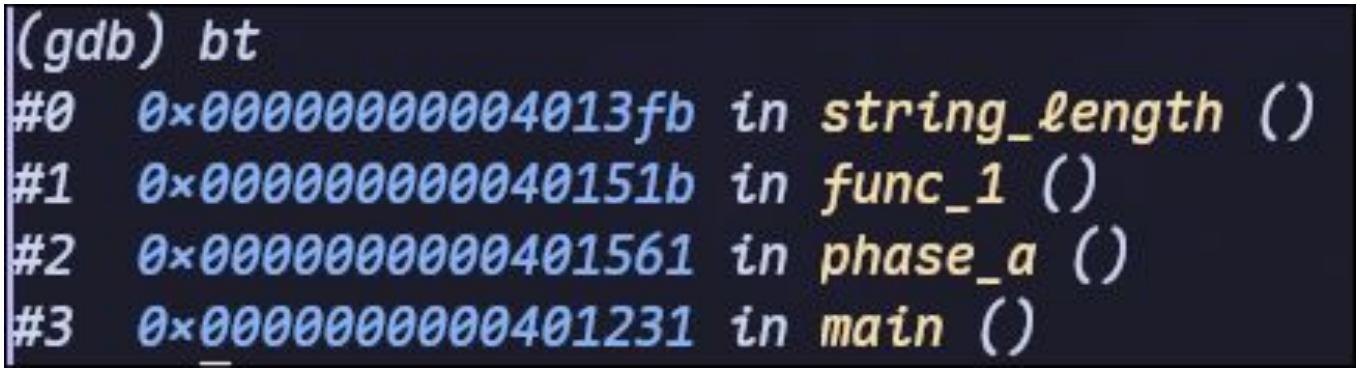
- Next
  - (gdb) n (next) // run next line of your program and **do not** step into function
  - (gdb) nexti // run next line of assembly code of your program and **do not** step into function
- Step
  - (gdb) s (step) // run next line of your program and step into function
  - (gdb) stepi // run next line of assembly code of your program and step into function
- Continue
  - (gdb) c (continue) // continues until next breakpoint

```
(gdb) si
0x00000000004013ec in string_length ()
```

```
(gdb) ni
0x0000000000401553 in phase_a ()
```

Step2: Backtrace

- Show the function calling chain
  - (gdb) bt (backtrace)



- Useful when reading complex project
  - Linux kernel
  - glibc

Demo: defusing phase B !

Phase B

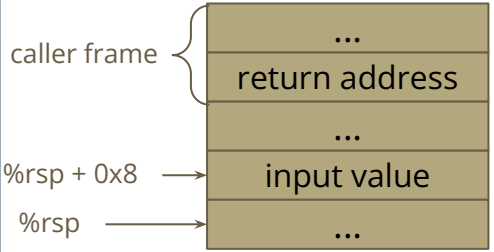
- Get the assembly

```
00000000004015af <phase_b>:
  4015af: 48 83 ec 18           sub    $0x18,%rsp
  4015b3: 48 8d 54 24 08        lea    0x8(%rsp),%rdx
  4015b8: 48 8d 35 aa 0b 00 00  lea    0xbaa(%rip),%rsi        # 402169 <_IO_stdin_used+0x169>
  4015bf: b8 00 00 00 00        mov    $0x0,%eax
  4015c4: e8 97 fa ff ff        call   401060 <__isoc99_sscanf@plt>
  4015c9: 83 f8 01              cmp    $0x1,%eax
  4015cc: 75 15                 jne    4015e3 <phase_b+0x34>
  4015ce: 48 8b 7c 24 08        mov    0x8(%rsp),%rdi
  4015d3: e8 99 ff ff ff        call   401571 <func_2>
  4015d8: 48 83 f8 07           cmp    $0x7,%rax
  4015dc: 75 0a                 jne    4015e8 <phase_b+0x39>
  4015de: 48 83 c4 18           add    $0x18,%rsp
  4015e2: c3                    ret
  4015e3: e8 cd fe ff ff        call   4014b5 <explode_bomb>
  4015e8: e8 c8 fe ff ff        call   4014b5 <explode_bomb>
```

Keep in mind:

- **%rdi: address of our input string**
- **%rsi: format string**
- **%rdx: formatted value address**
- **We should input an integer (unsigned)**

```
(gdb) x /s 0x402169
0x402169:       "%lu"
```

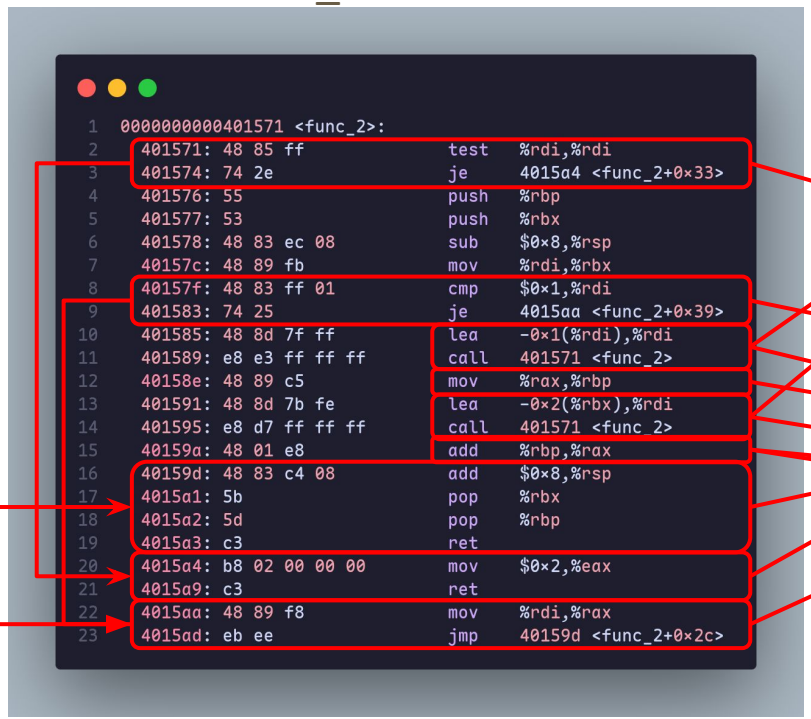


- Examine func_2
  - Calls with one integer
  - Return with 7

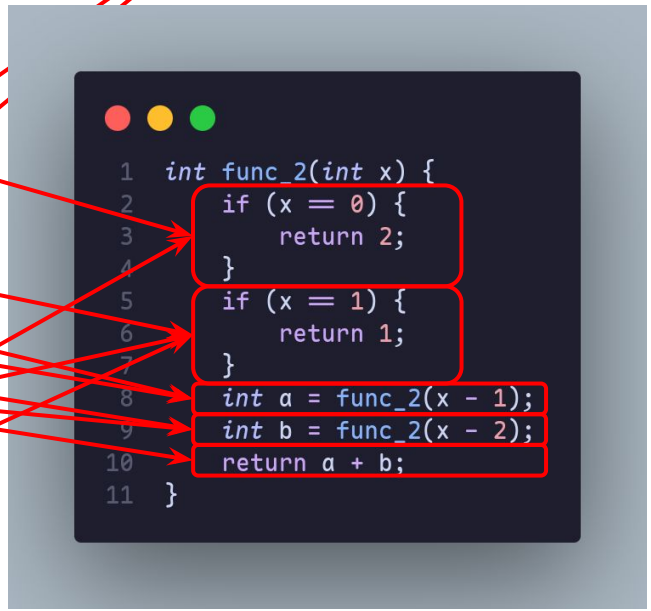
Phase B

Which x can cause func2 to return 7 ?

- What does func_2 do ?



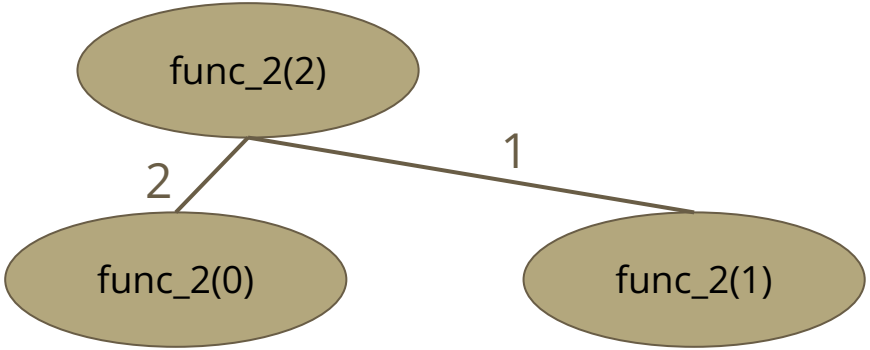
Recursion !



Phase B

- How to return 7 ?

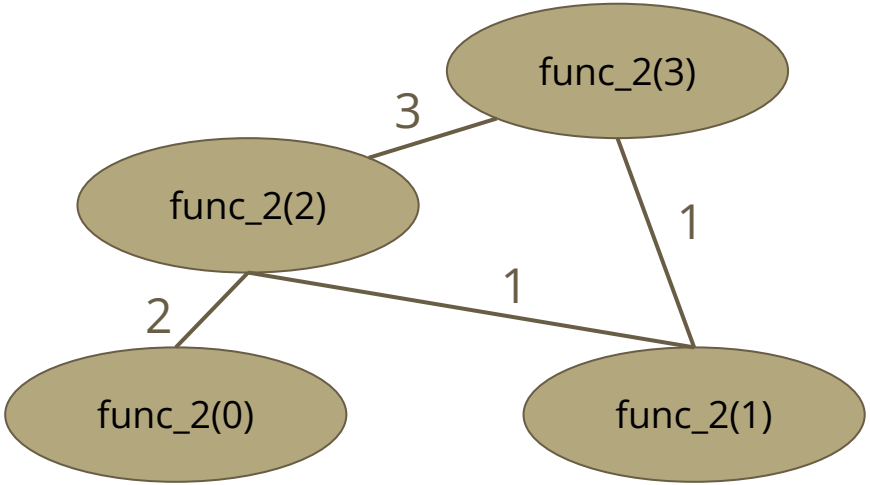
```c
int func_2(int x) {
    if (x == 0) {
        return 2;
    }
    if (x == 1) {
        return 1;
    }
    int a = func_2(x - 1);
    int b = func_2(x - 2);
    return a + b;
}
```



Phase B

- How to return 7 ?

```c
int func_2(int x) {
    if (x == 0) {
        return 2;
    }
    if (x == 1) {
        return 1;
    }
    int a = func_2(x - 1);
    int b = func_2(x - 2);
    return a + b;
}
```

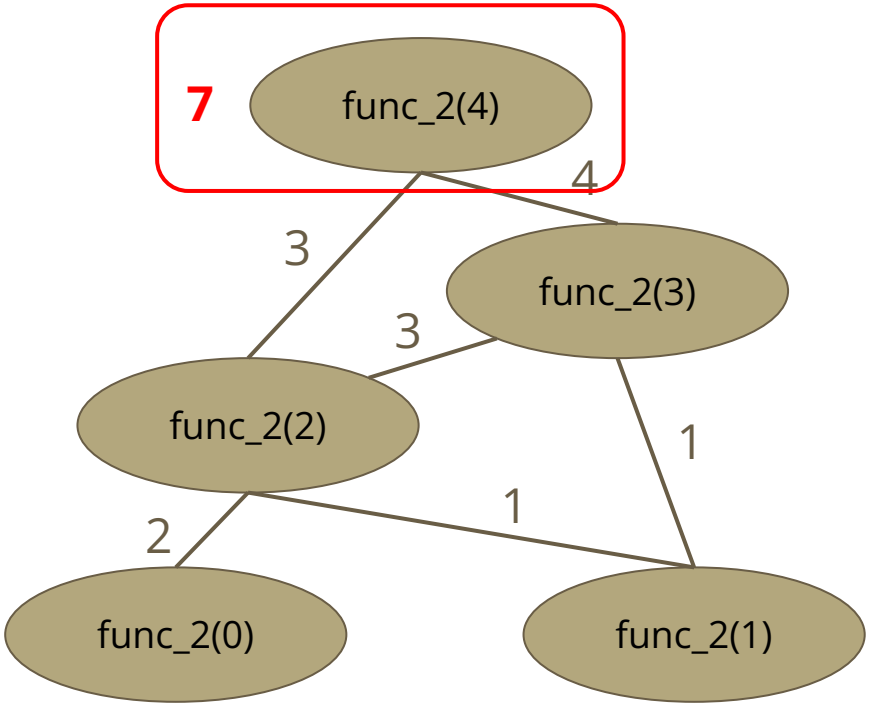


Phase B

- How to return 7 ?

```
4
Phase B defused.  Keep going!
```

```c
int func_2(int x) {
    if (x == 0) {
        return 2;
    }
    if (x == 1) {
        return 1;
    }
    int a = func_2(x - 1);
    int b = func_2(x - 2);
    return a + b;
}
```

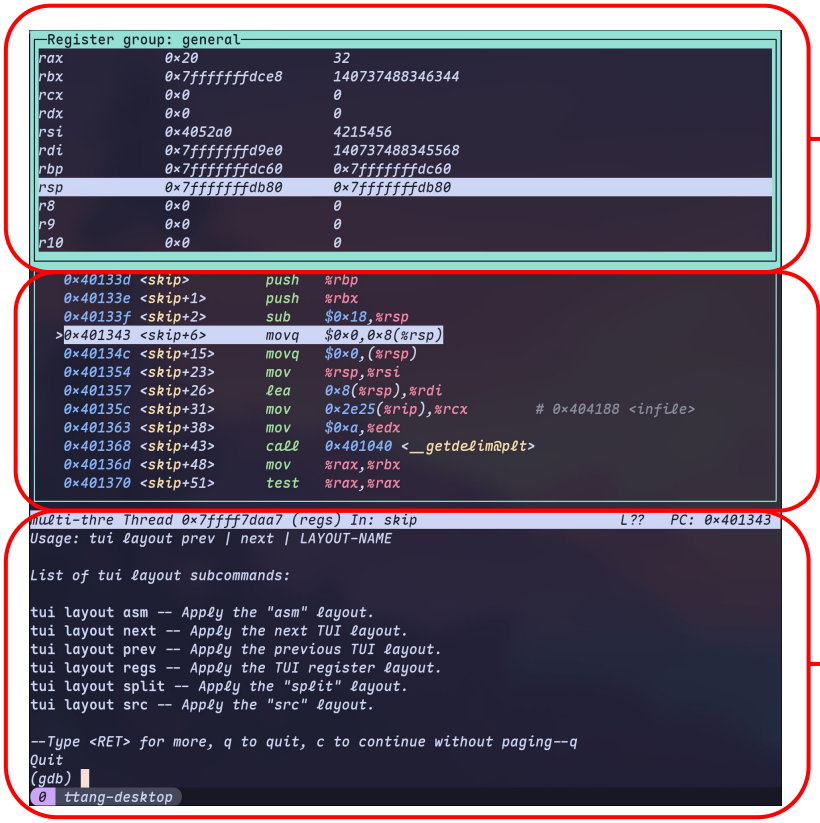


Demo: defusing phase C !

Step3: Using TUI

- Start TUI
  - (gdb) tui layout asm // display assembly code
  - (gdb) tui layout regs // display registers information
  - (gdb) tui layout src // display source code
- Switch between windows
  - (gdb) focus regs // switch focus to register window, also asm, src …
  - (gdb) <Ctrl-x> + o
- More options
  - (gdb) help tui layout

Step3: Using TUI



Registers (focused on)

Assembly

Command

Phase C: Much longer

- Get the Assembly

```
1  00000000004015ed <phase_c>:
2    4015ed: 48 83 ec 18           sub    $0x18,%rsp
3    4015f1: 48 8d 54 24 08        lea    0x8(%rsp),%rdx
4    4015f6: 48 8d 35 6c 0b 00 00  lea    0xb6c(%rip),%rsi        # 402169 <_IO_stdin_used+0x169>
5    4015fd: b8 00 00 00 00        mov    $0x0,%eax
6    401602: e8 59 fa ff ff        call   401060 <__isoc99_sscanf@plt>
7    401607: 83 f8 01              cmp    $0x1,%eax
8    40160a: 75 21                 jne    40162d <phase_c+0x40>
9    40160c: 48 8b 44 24 08        mov    0x8(%rsp),%rax
10   401611: 48 2d 6d 3b 00 00     sub    $0x3b6d,%rax
11   401617: 48 83 f8 04           cmp    $0x4,%rax
12   40161b: 77 2a                 ja     401647 <phase_c+0x5a>
13   40161d: 48 8d 15 50 0b 00 00  lea    0xb50(%rip),%rdx        # 402174 <_IO_stdin_used+0x174>
14   401624: 48 63 04 82           movslq (%rdx,%rax,4),%rax
15   ...
16   401647: e8 69 fe ff ff        call   4014b5 <explode_bomb>
17   40164c: b8 02 00 00 00        mov    $0x2,%eax
```

```
(gdb) x /s 0x402169
0x402169:       "%lu"
```

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

What value should we input ?

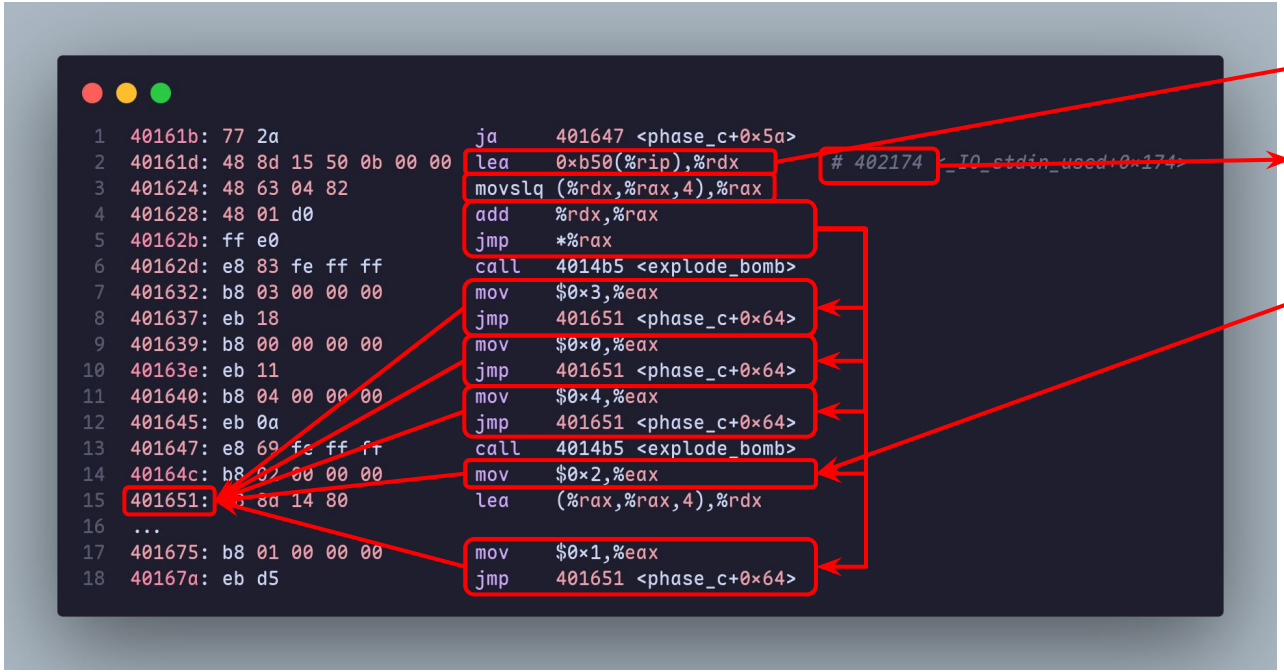
- x >= 15213
- x - 15213 <= 4
- 15213 => %rax = 0
- 15214 => %rax = 1
- 15215 => %rax = 2
- 15216 => %rax = 3
- 15217 => %rax = 4

Phase C: Much longer

- Continue …

Where to go ?

- 15213 => %rax = 0 => %rax = 2
- 15214 => %rax = 1 => %rax = 1
- 15215 => %rax = 2 => %rax = 3
- 15216 => %rax = 3 => %rax = 0
- 15217 => %rax = 4 => %rax = 4



Phase C: Much longer

- Keep going …

```
Phase B defused.  Keep going!
15214
Phase C defused!
You have defused all of the phases :-(
```

```
1  40164c: b8 02 00 00 00        mov    $0x2,%eax
2  401651: 48 8d 14 80           lea    (%rax,%rax,4),%rdx
3  401655: 48 8d 05 44 2a 00 00  lea    0x2a44(%rip),%rax        # 4040a0 <sharkNames>
4  40165c: 48 8d 3c 90           lea    (%rax,%rdx,4),%rdi
5  401660: 48 8d 35 06 0b 00 00  lea    0xb06(%rip),%rsi         # 40216d <_IO_stdin_used+0x16d>
6  401667: e8 95 fd ff ff        call   401401 <strings_not_equal>
7  40166c: 84 c0                 test   %al,%al
8  40166e: 75 0c                 jne    40167c <phase_c+0x8f>
9  401670: 48 83 c4 18           add    $0x18,%rsp
10 401674: c3                    ret
11 401675: b8 01 00 00 00        mov    $0x1,%eax
12 40167a: eb d5                 jmp    401651 <phase_c+0x64>
13 40167c: e8 34 fe ff ff        call   4014b5 <explode_bomb>
```

Where to go ?

- 15213 => %rax = 0 => %rax = 2 => %rdx = 40 => “makoshark”
- **15214 => %rax = 1 => %rax = 1 => %rdx = 20 => “requin”**
- 15215 => %rax = 2 => %rax = 3 => %rdx = 60 => “milkshark”
- 15216 => %rax = 3 => %rax = 0 => %rdx = 0 => “roughshark”
- 15217 => %rax = 4 => %rax = 4 => %rdx = 80 => “blueshark”

%rdx = %rax + %rax * 4

- %rdx = 5 * %rax

%rdx = %rdx * 4

- %rdx = 20 * %rax

```
(gdb) x/s 0x4040a0
0x4040a0 <sharkNames>:  "makoshark"
(gdb) x/s 0x4040a0 + 20
0x4040b4 <sharkNames+20>:       "requin"
(gdb) x/s 0x4040a0 + 40
0x4040c8 <sharkNames+40>:       "milkshark"
(gdb) x/s 0x4040a0 + 60
0x4040dc <sharkNames+60>:       "roughshark"
(gdb) x/s 0x4040a0 + 80
0x4040f0 <sharkNames+80>:       "blueshark"
```

```
(gdb) x/s 0x40216d
0x40216d:       "requin"
```

%rdi = address of selected string
%rsi = address of target string

Acknowledgement

- https://www.cs.cmu.edu/~213/bootcamps/lab2_slides.pdf
- https://www.cs.cmu.edu/~213/recitations/rec02_slides.pdf
- https://www.cs.cmu.edu/~213/activities/s25-rec3.tar
- https://ics.dfshan.net/GDB-Usage-Tutorial
- https://ics.dfshan.net/uploads/slide/bomblab&&attacklab.pdf
- https://sourceware.org/gdb/

Thanks !