Assembly: Bomb Lab, procedures, and function calls.

Yipeng Huang

Rutgers University

March 23, 2021

Table of contents

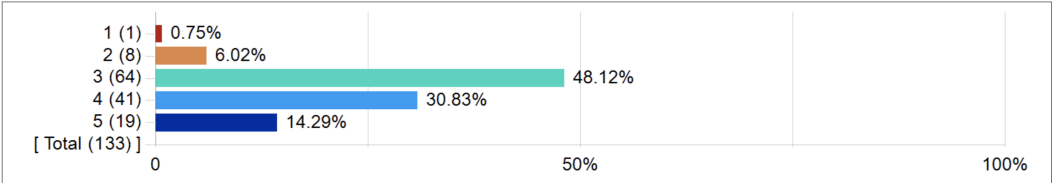
Looking ahead

Class plan

1. Today, Tuesday, 3/23: Assembly procedures and function calls.
2. Programming Assignment 3 on bits, bytes, integers, floats due tomorrow, Wednesday 3/24.
3. Thursday, 3/25: Finish assembly with arrays and structs.
4. Programming Assignment 4 on Defusing a Binary Bomb out. Due Tuesday, 4/6.
5. Starting next week: The memory hierarchy. Reading assignment, CS:APP Chapter 6.

Midcourse feedback: workload question

Compared to other classes in the computer science department, the workload of this class is: 1: much lighter, 2: lighter, 3: the same, 4: heavier, 5: much heavier.



Statistics	Value
Response Count	133
Mean	3.52
Median	3.00

Figure: Midcourse survey comparative workload question results

Table of contents

Programming Assignment 4: Defusing a Binary Bomb

Goals

- Learning to learn to use important tools like GDB.
- Understand how high level programming constructs compile down to assembly instructions.
- Practice reverse engineering and debugging.

Setup

- Programming assignment description PDF on Canvas.
- Web interface for obtaining bomb and seeing progress.
- Unpacking.

Unpacking and gathering information about your bomb

What comes in the package

- `bomb.c`: Skeleton source code
- `bomb`: The executable binary

```
objdump -t bomb > symbolTable.txt
```

- 000000000040143a g F .text 0000000000000022 explode_bomb

```
objdump -d bomb > bomb.s
```

Different phases correspond to different topics about assembly programming in the CS211 lecture slides, in the CS:APP slides, and in the CS:APP book.

- phase_1
- phase_2
- explode_bomb

```
strings -t x bomb > strings.txt
```

Example phase_1 in example bomb from CS:APP website

```
0000000000400ee0 <phase_1>:
  400ee0: 48 83 ec 08           sub    $0x8,%rsp
  400ee4: be 00 24 40 00        mov    $0x402400,%esi
  400ee9: e8 4a 04 00 00        callq  401338 <strings_not_equal>
  400eee: 85 c0                 test   %eax,%eax
  400ef0: 74 05                 je     400ef7 <phase_1+0x17>
  400ef2: e8 43 05 00 00        callq  40143a <explode_bomb>
  400ef7: 48 83 c4 08           add    $0x8,%rsp
  400efb: c3                    retq
```

Understanding what we're seeing here

- Don't let `callq` to `explode_bomb` at instruction address `400ef2` happen...
- so, must ensure `je` instruction does jump, so we want `test` instruction to set ZF condition code to 0.
- so, must ensure `callq` to `strings_not_equal()` function returns 0.

Using GDB to carefully step through execution of the bomb program

```
gdb bomb
```

Finding help in GDB

- `help`: Menu of documentation.
- `help layout`: Useful tip to use either `layout asm` or `layout regs` for this assignment.
- `help aliases`
- `help running`
- `help data`
- `help stack`

Using GDB to carefully step through execution of the bomb program

`gdb bomb`

Setting breakpoints and running / stepping through code

- `break explode_bomb` or `b explode_bomb`: Pause execution upon entering explode_bomb function.
- `break phase_1` or `b phase_1`: Pause execution upon entering phase_1 function.
- `run mysolution.txt` or `r mysolution.txt`: Run the code passing the solution file.
- `continue` or `c`: Continue until the next breakpoint.
- `nexti` or `ni`: Step one instruction, but proceed through subroutine calls.
- `stepi` or `si`: Step one instruction exactly. Steps into functions / subroutine calls.

Using GDB to carefully step through execution of the bomb program

`gdb bomb`

Printing and examining registers and memory addresses

- `print /x $eax` or `p /x $eax`: Print value of %eax register as hex.
- `print /d $eax` or `p /d $eax`: Print value of %eax register as decimal.
- `x /s 0x402400`: Examine memory address 0x402400 as a string.

Table of contents

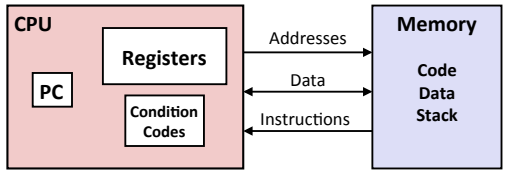
Procedures and function calls

To create the abstraction of functions, need to:

- Transfer control to function and back
- Transfer data to function (parameters)
- transfer data from function (return type)

CPU and memory state in support of procedures and functions

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Relevant state in CPU:

- %rip register / instruction pointer / program counter
- %rsp register / stack pointer

Relevant state in Memory:

- Stack

Stack instructions: `push` and `pop`

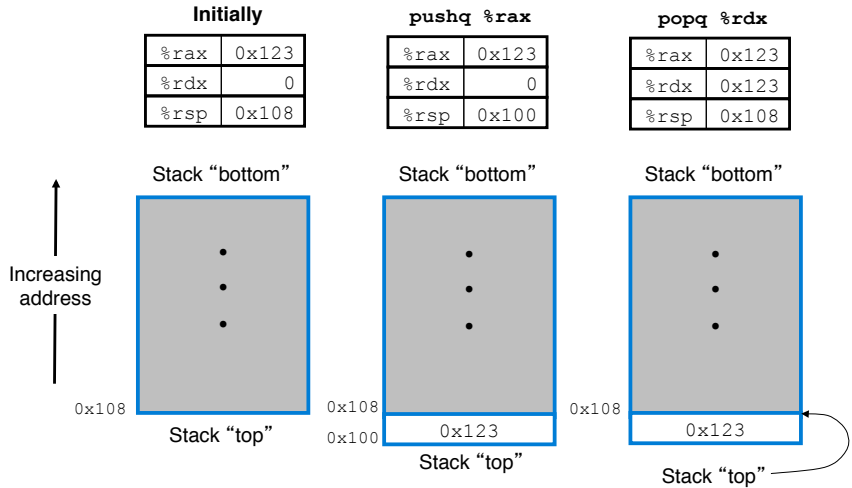


Figure: x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of %rsp is implied. Image credit: CS:APP.

Procedure call and return: `call` and `ret`

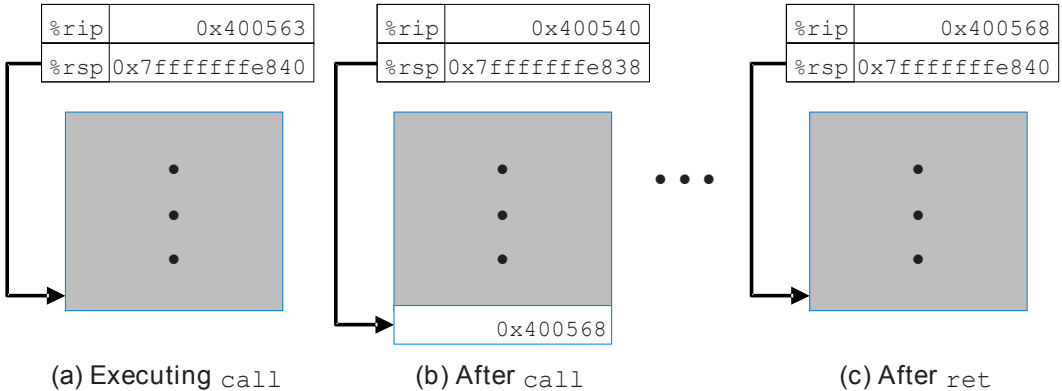


Figure: Effect of `call 0x400540` instruction and subsequent return. `call` and `ret` instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.

Table of contents

Procedures and function calls: Transferring data

For purposes of this class, the Bomb Lab, and the CS:APP textbook, we study the x86-64 Linux Application Binary Interface (ABI). Would be different on ARM or in Windows. So, don't memorize this, but it is helpful for PA4 Bomb Lab.

Passing parameters

Parameter	Register / stack	Subset registers	Mnemonic[1]
1st	%rdi	%edi, %di	Diane's
2nd	%rsi	%esi, %si	silk
3rd	%rdx	%edx, %dx, %dl	dress
4th	%rcx	%ecx, %cx, %cl	cost
5th	%r8	%r8d	$8
6th	%r9	%r9d	9
7th and beyond	Stack		

[1] http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html

Procedures and function calls: Transferring data

Passing function return data

Function return data is passed via:

- the 64-bit %raX register
- the 32-bit subset %eax register
- the 16-bit subset %ax register
- the 8-bit subset %al register