FUTCEFS

THE STATE UNIVERSITY OF NEW JERSEY

Computer Architecture (CS-211) Recitation

Abu Awal Md Shoeb

Topics

- HW4 (Bomb Lab)
- Assembly Language

* Some materials are collected and compiled from previous year's CS 211 lectures and TAs

GDB: GNU Debugger

- Find bugs in a program
- Print out variables' values
- Check the logic of the program
- Debug using gdb

- GDB: can trace the program execution
- Step through the program one line at a time
- Monitor / modify internal variables

- How to use GDB to debug
- Need to compile your source code with -g
- gcc -g simple.c -o simple
- Use gdb after compiling: gdb [your exe]

A GDB tutorial: https://www.youtube.com/watch?v=sCtY--xRUyI

Some GDB commands

- Start debugging the program in gdb
  - `(gdb) run` or `r`
- End debugging
  - `(gdb) quit` or `q`
- Set a breakpoint
  - `(gdb) break [func name]` or `b [func name]`
  - `(gdb) break [line number]` or `b [line number]`
- Execute next line of code
  - `(gdb) next` or `n`
- Print the value of a variable
  - `(gdb) print [var name]` or `p [var name]`
- List code around current line
  - `(gdb) list` or `l`

Some GDB commands

- Display a variable's value each time gdb stops
  - `(gdb) display [var name]`
- Continue execution until next breakpoint
  - `(gdb) continue` or `c`
- Set a temp value to a variable
  - `(gdb) print [var name]=[value]` or `p [var name]=[value]`
- Show all breakpoints
  - `(gdb) info breakpoint` or `info b`
- Clear a breakpoint
  - `(gdb) clear [func name]`
  - `(gdb) clear [line number]`
  - `(gdb) delete breakpoint [# of breakpoint]`
- Clear all breakpoints
  - `(gdb) delete`

Using GDB without source code

- Not compiled with -g
  - You cannot see the source code in gdb
  - E.g: PA3-bomblab
- Disassemble the binary (binary to assembly code)
  - `objdump –d simple`
- Use GDB to debug assembly code
  - Start GDB: gdb [exec]
    - gdb simple

GDB commands to debug assembly code

- `run` and `quit` are the same
- Set a breakpoint
  - `(gdb) break [func name]` or `b [func name]`
  - `(gdb) break *[address]` or `b *[address]`
- Execute next instruction of assembly code
  - `(gdb) nexti` or `ni`
- Clear a breakpoint
  - `(gdb) clear [func name]`
  - `(gdb) clear *[address]`
  - `(gdb) delete breakpoint [# of breakpoint]`
- Clear all breakpoints
  - `(gdb) delete`

GDB commands to debug assembly code

- Print value of a register in a given format
  - `(gdb) print/[format] [expression]`
    - Useful formats (default is decimal)
      - d: decimal
      - x: hex
      - t: binary
      - i: instruction
      - c: character
    - Expression can be program variables or registers,
    - A register is represented as $eax instead of %eax
    - E.g. p/x $eax (print the value in register eax in hex)
  - `(gdb) info r [register name]`
    - `info r eax` or `i r eax`
    - will print all registers' contents without register name

GDB commands to debug assembly code

- Print value of a specified memory address
  - `(gdb) x/[count][format] [address]`
  - `format` is the same as for `print`
  - Address can be symbolic (e.g. main) or numeric (e.g. 0x804848a) or register name (e.g. $eax)

- Step into and out of a function
  - Step in: `(gdb) step i` or `si`
  - Step out: `(gdb) step`

HW4 - Bomblab

- Find the solutions by debugging using GDB
  - `gdb bomb` (run in gdb)
  - Set break point for each phase (e.g. `(gdb) b phase_1`)
    - When running, it will step into the function of each phase
    - Guess the input according to the instructions in each phase function
- Generate the assembly code from binary
  - Disassemble the code ($ objdump -d bomb)
- Other useful informaiton
  - Print bomb's symbol table ($ objdump -t bomb)
  - Display printable strings ($ strings -t x bomb)

How to Defuse It!

```
8048b6f:    e8 c0 09 00 00          call   8049534 <read_line>
8048b74:    89 04 24                mov    %eax,(%esp)
8048b77:    e8 04 01 00 00          call   8048c80 <phase_1>
8048b7c:    e8 ad 0a 00 00          call   804962e <phase_defused>
8048b81:    c7 04 24 40 a4 04 08    movl   $0x804a440,(%esp)
8048b88:    e8 f3 fc ff ff          call   8048880 <puts@plt>
8048b8d:    e8 a2 09 00 00          call   8049534 <read_line>
8048b92:    89 04 24                mov    %eax,(%esp)
8048b95:    e8 2a 01 00 00          call   8048cc4 <phase_2>
8048b9a:    e8 8f 0a 00 00          call   804962e <phase_defused>
8048b9f:    c7 04 24 81 a3 04 08    movl   $0x804a381,(%esp)
8048ba6:    e8 d5 fc ff ff          call   8048880 <puts@plt>
8048bab:    e8 84 09 00 00          call   8049534 <read_line>
8048bb0:    89 04 24                mov    %eax,(%esp)
8048bb3:    e8 30 01 00 00          call   8048ce8 <phase_3>
8048bb8:    e8 71 0a 00 00          call   804962e <phase_defused>
8048bbd:    c7 04 24 9f a3 04 08    movl   $0x804a39f,(%esp)
8048bc4:    e8 b7 fc ff ff          call   8048880 <puts@plt>
8048bc9:    e8 66 09 00 00          call   8049534 <read_line>
8048bce:    89 04 24                mov    %eax,(%esp)
8048bd1:    e8 9c 01 00 00          call   8048d72 <phase_4>
8048bd6:    e8 53 0a 00 00          call   804962e <phase_defused>
8048bdb:    c7 04 24 6c a4 04 08    movl   $0x804a46c,(%esp)
8048be2:    e8 99 fc ff ff          call   8048880 <puts@plt>
8048be7:    e8 48 09 00 00          call   8049534 <read_line>
8048bec:    89 04 24                mov    %eax,(%esp)
8048bef:    e8 d6 01 00 00          call   8048dca <phase_5>
8048bf4:    e8 35 0a 00 00          call   804962e <phase_defused>
8048bf9:    c7 04 24 b0 a3 04 08    movl   $0x804a3b0,(%esp)
8048c00:    e8 7b fc ff ff          call   8048880 <puts@plt>
8048c05:    e8 2a 09 00 00          call   8049534 <read_line>
8048c0a:    89 04 24                mov    %eax,(%esp)
```

HW4 – Generate the assembly code

```
-bash-4.1$ ls
bomb  bomb.c  bomb.s  defuser.txt  README
-bash-4.1$ objdump -d bomb > bomb.s
-bash-4.1$
```

Address and name of function

```
8048aa2:   e8 af 06 00 00          call   8049156 <initialize_bomb>
8048aa7:   c7 04 24 64 a2 04 08    movl   $0x804a264,(%esp)
8048aae:   e8 5d fd ff ff          call   8048810 <puts@plt>
8048ab3:   c7 04 24 a0 a2 04 08    movl   $0x804a2a0,(%esp)
8048aba:   e8 51 fd ff ff          call   8048810 <puts@plt>
8048abf:   e8 50 09 00 00          call   8049414 <read_line>
8048ac4:   89 04 24                mov    %eax,(%esp)
8048ac7:   e8 04 01 00 00          call   8048bd0 <phase_1>
8048acc:   e8 77 0a 00 00          call   8049548 <phase_defused>
8048ad1:   c7 04 24 cc a2 04 08    movl   $0x804a2cc,(%esp)
8048ad8:   e8 33 fd ff ff          call   8048810 <puts@plt>
8048add:   e8 32 09 00 00          call   8049414 <read_line>
8048ae2:   89 04 24                mov    %eax,(%esp)
8048ae5:   e8 2a 01 00 00          call   8048c14 <phase_2>
8048aea:   e8 59 0a 00 00          call   8049548 <phase_defused>
8048aef:   c7 04 24 0d a2 04 08    movl   $0x804a20d,(%esp)
```

Phase_1

HW4 – A glance at phase_1

```
08048bd0 <phase_1>:
 8048bd0:   83 ec 2c                sub    $0x2c,%esp
 8048bd3:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
 8048bda:   00
 8048bdb:   8d 44 24 1c             lea    0x1c(%esp),%eax
 8048bdf:   89 44 24 08             mov    %eax,0x8(%esp)
 8048be3:   c7 44 24 04 b4 a5 04    movl   $0x804a5b4,0x4(%esp)
 8048bea:   08
 8048beb:   8b 44 24 30             mov    0x30(%esp),%eax
 8048bef:   89 04 24                mov    %eax,(%esp)
 8048bf2:   e8 89 fc ff ff          call   8048880 <__isoc99_sscanf@plt>
 8048bf7:   83 f8 01                cmp    $0x1,%eax
 8048bfa:   74 05                   je     8048c01 <phase_1+0x31>
 8048bfc:   e8 84 07 00 00          call   8049385 <explode_bomb>
 8048c01:   81 7c 24 1c 5c 02 00    cmpl   $0x25c,0x1c(%esp)
 8048c08:   00
 8048c09:   74 05                   je     8048c10 <phase_1+0x40>
 8048c0b:   e8 75 07 00 00          call   8049385 <explode_bomb>
 8048c10:   83 c4 2c                add    $0x2c,%esp
 8048c13:   c3                      ret
```

Be careful !!

- Execute the instruction line by line using `ni`
- Find out what it does before call the function *explode_bomb*
- Quit from gdb if it's going to execute *call xxx <explode_bomb>*

SSH Tunnel with Firefox

Linux

https://ubuntuforums.org/showthread.php?t=723025

$ ssh -D 9999 -C netId@iLab

Windows

https://www.sotechdesign.com.au/browsing-the-web-through-a-ssh-tunnel-with-firefox-and-putty-windows/

More GDB Commands

```
$ gcc –m32 hello.c –g –o hello
$ gdb hello
(gdb) run
(gdb) c - continue
(gdb) layout asm – gui for assembly code
(gdb) ni  - next instruction
(gdb) si  - step in (e.g. step into function)
(gdb) step  - step out
(gdb) disas  - disassemble instructions
(gdb) until  *addr – jump to the given addr
(gdb) i r – print all reg values
(gdb) x/s addr – print value of the addr (similarly x/d)
```

More Commands DRAFT

see contents of the registers and memory
(gdb) info registers or i r

print out the contents of the ECX register in decimal, hexadecimal, and binary, respectively
print/d $ecx
print/x $ecx
print/t $ecx

The gdb command "info display" will list all the active displays. Use "undisplay" to remove an item on this list.
display $eax
display/i $eip

then the contents of the EAX register will be printed to the screen every time the program is halted.

http://csapp.cs.cmu.edu/2e/docs/gdbnotes-ia32.pdf
https://www.csee.umbc.edu/~cpatel2/links/310/nasm/gdb_help.shtml

PA 3 – Scoreboard

- Remember : You will lose **0.5** points for each explodes!

Bomb Lab Scoreboard

This page contains the latest information that we have received from your bomb. If your solution is marked **invalid**, this means your bomb reported a solution that didn't actually defuse your bomb.

Last updated: Tue Mar 7 12:18:30 2017 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status
1	bomb3	Tue Feb 28 19:02	9	0	100	valid
2	bomb19	Sat Mar 4 18:50	9	0	100	valid
3	bomb15	Mon Mar 6 19:28	9	8	96	valid
4	bomb32	Sun Mar 5 06:25	8	8	86	**invalid phase 9**
5	bomb17	Mon Mar 6 23:07	6	0	60	**invalid phase 7**
6	bomb20	Tue Mar 7 11:27	7	1	75	**invalid phase 8**
7	bomb24	Sat Mar 4 20:56	5	0	45	**invalid phase 6**
8	bomb26	Sun Mar 5 09:23	5	1	45	**invalid phase 6**
9	bomb6	Thu Mar 2 19:21	5	33	29	**invalid phase 6**
10	bomb16	Mon Mar 6 20:39	4	4	33	**invalid phase 5**
11	bomb51	Mon Mar 6 23:51	3	0	25	**invalid phase 4**
12	bomb31	Sun Mar 5 14:17	3	1	25	**invalid phase 4**
13	bomb48	Tue Mar 7 00:06	3	1	25	**invalid phase 4**
14	bomb5	Tue Mar 7 08:00	2	14	8	**invalid phase 3**
15	bomb28	Sat Mar 4 21:10	1	5	3	**invalid phase 2**
16	bomb37	Sun Mar 5 14:43	0	1	0	**invalid phase 1**
17	bomb47	Mon Mar 6 08:47	0	1	0	**invalid phase 1**
18	bomb41	Sun Mar 5 19:47	0	2	-1	**invalid phase 1**
19	bomb44	Sun Mar 5 22:57	0	2	-1	**invalid phase 1**
20	bomb18	Sat Mar 4 15:11	0	3	-1	**invalid phase 1**
21	bomb30	Mon Mar 6 16:50	0	10	-5	**invalid phase 1**
22	bomb34	Mon Mar 6 22:11	0	17	-8	**invalid phase 1**
23	bomb61	Tue Mar 7 11:27	0	10266140	-40	**invalid phase 1**

Summary [phase:cnt] [1:1] [2:1] [3:3] [4:1] [5:3] [6:1] [7:1] [8:1] [9:3] total defused = 2/23

GDB (GNU Debugger)

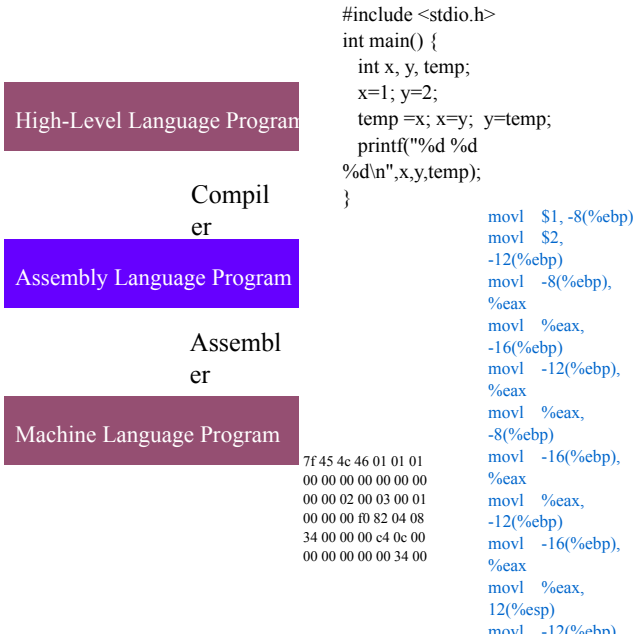
```
8048b6f:       e8 c0 09 00 00          call   8049534 <read_line>
8048b74:       89 04 24                mov    %eax,(%esp)
8048b77:       e8 04 01 00 00          call   8048c80 <phase_1>
8048b7c:       e8 ad 0a 00 00          call   804962e <phase_defused>
8048b81:       c7 04 24 40 a4 04 08    movl   $0x804a440,(%esp)
8048b88:       e8 f3 fc ff ff          call   8048880 <puts@plt>
8048b8d:       e8 a2 09 00 00          call   8049534 <read_line>
8048b92:       89 04 24                mov    %eax,(%esp)
8048b95:       e8 2a 01 00 00          call   8048cc4 <phase_2>
8048b9a:       e8 8f 0a 00 00          call   804962e <phase_defused>
8048b9f:       c7 04 24 81 a3 04 08    movl   $0x804a381,(%esp)
8048ba6:       e8 d5 fc ff ff          call   8048880 <puts@plt>
8048bab:       e8 84 09 00 00          call   8049534 <read_line>
8048bb0:       89 04 24                mov    %eax,(%esp)
8048bb3:       e8 30 01 00 00          call   8048ce8 <phase_3>
8048bb8:       e8 71 0a 00 00          call   804962e <phase_defused>
8048bbd:       c7 04 24 9f a3 04 08    movl   $0x804a39f,(%esp)
8048bc4:       e8 b7 fc ff ff          call   8048880 <puts@plt>
8048bc9:       e8 66 09 00 00          call   8049534 <read_line>
8048bce:       89 04 24                mov    %eax,(%esp)
8048bd1:       e8 9c 01 00 00          call   8048d72 <phase_4>
8048bd6:       e8 53 0a 00 00          call   804962e <phase_defused>
8048bdb:       c7 04 24 6c a4 04 08    movl   $0x804a46c,(%esp)
8048be2:       e8 99 fc ff ff          call   8048880 <puts@plt>
8048be7:       e8 48 09 00 00          call   8049534 <read_line>
8048bec:       89 04 24                mov    %eax,(%esp)
8048bef:       e8 d6 01 00 00          call   8048dca <phase_5>
8048bf4:       e8 35 0a 00 00          call   804962e <phase_defused>
8048bf9:       c7 04 24 b0 a3 04 08    movl   $0x804a3b0,(%esp)
8048c00:       e8 7b fc ff ff          call   8048880 <puts@plt>
8048c05:       e8 2a 09 00 00          call   8049534 <read_line>
8048c0a:       89 04 24                mov    %eax,(%esp)
```

Help

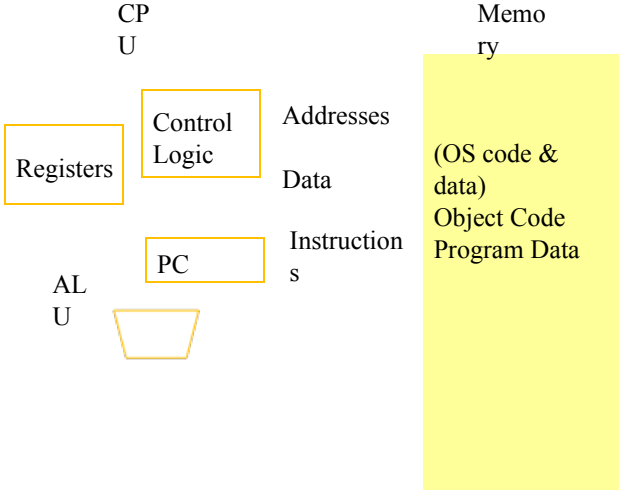
https://www.cse.umbc.edu/~cpatel2/links/310/nasm/gdb_help.shtml

https://www.youtube.com/watch?v=Z6zMxp6r4mc

Programming Meets Hardware



Assembly Programmer's View



Registers

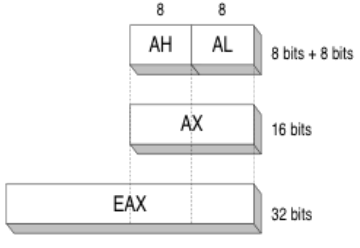
- Registers are CPU components that hold data and address
- Much faster to access than memory
- It is used to speed up CPU operations
- Categories
- General registers
- Data registers (Holds operands)
- Pointer & index registers (Holds references to addresses as well as indices)
- Control Register (e.g. CF,ZF)
- Segment registers (Holds starting address of program segments)
- CS, DS, SS, ES

Data Registers 1

- AX is the primary accumulator
- Used in most arithmetic instruction
- BX is the base register
- Could be used in indexed addressing
- CX is the count register
- Store the loop count in iterative operations
- DX is the data register
- Used in input / output operations

Data Registers 2

Can use 8-bit, 16-bit, or 32-bit name



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Pointer Registers

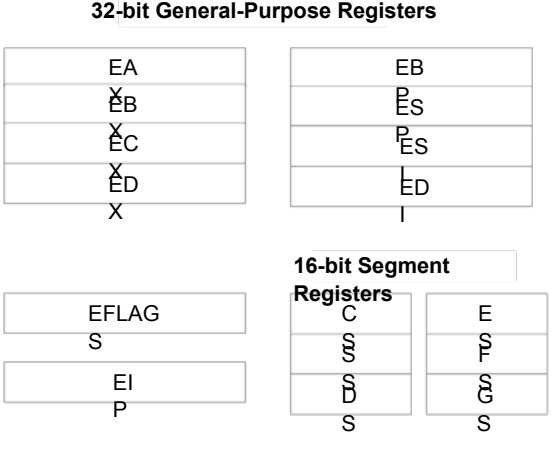
- ESP is stack pointer
- It refers to be current position of data or address within the program stack
- Changed by push, pop instructions
- EBP is frame pointer
- Referencing the parameter variables passed to a subroutine
- EIP is instruction pointer
- It stores the offset address of the next instruction to be executed

Control Registers

- Overflow flag (OF)
- Indicates the overflow of a high-order bit
- Carry flag (CF)
- Contains the carry of 0 or 1 from high-order bit after arithmetic operation
- Stores the last bit of a shift or rotate operation
- Sign flag (SF)
- Shows the sign of the result of an arithmetic operation
- Positive -> 0, Negative -> 1
- Zero Flag (ZF)

Registers Overview

Named storage locations inside the CPU, optimized for speed



Condition Codes

- Single Bit Registers (set after each instruction)
  - CF: Carry Flag, operation generated a carry out
  - SF: Sign Flag, operation yielded a negative value
  - ZF: Zero Flag, operation yielded zero
  - OF: Overflow Flag, operation caused 2's complement overflow

Assembly Language

- Instruction format
  - opcode operands
- 3 types of operands:
  - Immediate
    - Constant values
  - Register
    - Contents of registers
  - Memory
    - Contents of memory
- Number of operands depends on the command

Addressing Mode

- Immediate addressing
  - Used for constant values
  - $ in front of immediate operand
  - The value is found immediately
    - E.g., movl $0x4040, %eax

- Register mode addressing
  - Use % to denote register
    - E.g., %eax
  - Examples:
    - movl %eax, %ebx
      - Copy content of %eax to %ebx

Addressing mode

- Memory addressing

Type	Example	Comments
Direct	movl %eax, 0x0000f	copy value in eax to memory location 0x0000f
Indirect	movl (%ebp), %eax	copy value from memory location whose address is in ebp into eax
Indexed	movl (%ebp, %esi), %eax	copy value from memory at (address = ebp + esi) into eax
Scaled Indexed	movl 0x80(%ebx, %esi, 4), %eax	copy value from memory at (address = ebx + esi*4 + 0x80) into eax 0x80 is an address offset

Some arithmetic operation

Instruction		Computation
addl	Src, Dest	Dest = Dest + Src
subl	Src, Dest	Dest = Dest - Src
imull	Src, Dest	Dest = Dest * Src
sall	Src, Dest	Dest = Dest << Src (left shift)
sarl	Src, Dest	Dest = Dest >> Src (right shift)
xorl	Src, Dest	Dest = Dest ^ Src
andl	Src, Dest	Dest = Dest & Src
orl	Src, Dest	Dest = Dest | Src
incl	Dest	Dest = Dest + 1
decl	Dest	Dest = Dest - 1
negl	Dest	Dest = - Dest
notl	Dest	Dest = ~ Dest

Jump Operation

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
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)

Movement operation

- Five possible combinations of source and destination types
  - No memory to memory transfers with single instruction

```
1    movl $0x4050,%eax        Immediate--Register, 4 bytes
2    movw  %bp,%sp            Register--Register,  2 bytes
3    movb (%edi,%ecx),%ah     Memory--Register,    1 byte
4    movb $-17,(%esp)         Immediate--Memory,   1 byte
5    movl %eax,-12(%ebp)      Register--Memory,    4 bytes
```

Instruction		Effect
MOV	*S*, *D*	$D \leftarrow S$
movb		Move byte
movw		Move word
movl		Move double word

Are the following operations legal? If no, why?

(a) movl (%eax), %ecx

(b) movl 0x8(%eax), (%ebp)

(c) movl 0x80(%eax,%ebx,%ecx), %edx

Usef Commands for Bomblab

- One way to do it by debugging using GDB
  - $ gdb bomb (run in gdb)
  - Set break point for each phase (e.g. (gdb) break phase_1) (this will help you not to explode the bomb)
  - Run the program ((gdb) run)

- Useful Commands for binary bomb
  - Print bomb's symbol table ($ objdump -t bomb)
  - Disassemble the code ($ objdump -d bomb)
  - Display printable strings ($ strings -t x bomb)

- You can save output of commands into file
  - Example : $ objdump -d bomb > bomb-assembly.txt

http://csapp.cs.cmu.edu/2e/docs/gdbnotes-ia32.pdf

PA3 – Generate the assembly code

```
-bash-4.1$ ls
bomb  bomb.c  bomb.s  defuser.txt  README
-bash-4.1$ objdump -d bomb > bomb.s
-bash-4.1$
```

Address and name of function

```
8048aa2:   e8 af 06 00 00          call   8049156 <initialize_bomb>
8048aa7:   c7 04 24 64 a2 04 08    movl   $0x804a264,(%esp)
8048aae:   e8 5d fd ff ff          call   8048810 <puts@plt>
8048ab3:   c7 04 24 a0 a2 04 08    movl   $0x804a2a0,(%esp)
8048aba:   e8 51 fd ff ff          call   8048810 <puts@plt>
8048abf:   e8 50 09 00 00          call   8049414 <read_line>
8048ac4:   89 04 24                mov    %eax,(%esp)
8048ac7:   e8 04 01 00 00          call   8048bd0 <phase_1>
8048acc:   e8 77 0a 00 00          call   8049548 <phase_defused>
8048ad1:   c7 04 24 cc a2 04 08    movl   $0x804a2cc,(%esp)
8048ad8:   e8 33 fd ff ff          call   8048810 <puts@plt>
8048add:   e8 32 09 00 00          call   8049414 <read_line>
8048ae2:   89 04 24                mov    %eax,(%esp)
8048ae5:   e8 2a 01 00 00          call   8048c14 <phase_2>
8048aea:   e8 59 0a 00 00          call   8049548 <phase_defused>
8048aef:   c7 04 24 0d a2 04 08    movl   $0x804a20d,(%esp)
```

Phase_1

PA3 – A glance at phase_1

```
08048bd0 <phase_1>:
 8048bd0:   83 ec 2c                sub    $0x2c,%esp
 8048bd3:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
 8048bda:   00
 8048bdb:   8d 44 24 1c             lea    0x1c(%esp),%eax
 8048bdf:   89 44 24 08             mov    %eax,0x8(%esp)
 8048be3:   c7 44 24 04 b4 a5 04    movl   $0x804a5b4,0x4(%esp)
 8048bea:   08
 8048beb:   8b 44 24 30             mov    0x30(%esp),%eax
 8048bef:   89 04 24                mov    %eax,(%esp)
 8048bf2:   e8 89 fc ff ff          call   8048880 <__isoc99_sscanf@plt>
 8048bf7:   83 f8 01                cmp    $0x1,%eax
 8048bfa:   74 05                   je     8048c01 <phase_1+0x31>
 8048bfc:   e8 84 07 00 00          call   8049385 <explode_bomb>
 8048c01:   81 7c 24 1c 5c 02 00    cmpl   $0x25c,0x1c(%esp)
 8048c08:   00
 8048c09:   74 05                   je     8048c10 <phase_1+0x40>
 8048c0b:   e8 75 07 00 00          call   8049385 <explode_bomb>
 8048c10:   83 c4 2c                add    $0x2c,%esp
 8048c13:   c3                      ret
```

Be careful !!

- Execute the instruction line by line using `ni`
- Find out what it does before call the function *explode_bomb*
- Quit from gdb if it's going to execute *call xxx <explode_bomb>*

```
/* Do all sorts of secret stuff that makes the bomb harder to defuse. */
initialize_bomb();

printf("Welcome to my fiendish little bomb. You have 9 phases with\n");
printf("which to blow yourself up. Have a nice day!\n");

/* 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. */
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder.  No one will ever figure out
 * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2.  Keep going!\n");

/* I guess this is too easy so far.  Some more complex code will
 * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("One step closer.\n");

/* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one.  Try this one.\n");
```

bmb.c

Q&A

Thanks!