# CISC-221 Computer Architecture Lab Machine Representation of Programs Lab 4: The Bomblab February, 2018

Instructor: Dave Dove ( [dove@cs.queensu.ca](mailto:dove@cs.queensu.ca) ).

Name: Alejandra Kudo Student Number: 10136014

**1 Introduction**

The nefarious Dr. Evil has planted a binary bomb in a seemingly innocent program. The binary bomb asks for an input string from the keyboard. If you type the correct string, then the bomb is defused. Otherwise the bomb explodes by printing "BOOM!" and then terminating.

A usually reliable informant seems to think that the defuse string was something like "Now is the time for GDB!" but that the bomb was booby-trapped as well.

Your mission, which you have no choice but to accept, is to defuse the bomb before the end of the lab. Good luck, and welcome to the bomb squad!

**2 Logistics**

You may work individually or in pairs for this assignment. All submissions are electronic to the course OnQ site. Clarifications and corrections will be posted to the course web site.

If you are working as a pair, only one submission is required for two students but make sure that both your names are included in the submission. Either student may make the submission.

The TAs will begin each TA session this week with a short tutorial introduction to gdb. Gdb will be used to inspect and analyze the bomb, and figure out how to defuse it.

**Before the due date and time, submit a text file (word or pdf) to the Lab4 Submission item on the course web site. The file should contains the following:**

1. **The correct defuse string. (This is not worth anything without support information as described next)**
2. **Support Information (required). In order of diminishing value:  
   An equivalent C language program OR algorithm of what the program does in detail, referencing labels embedded in the program, OR a description of what you have figured out so far (a commented copy of the source program would be ideal) and how you have used gdb to analyze the program and determine the correct decode string.**

**If you get stuck (don't know how to proceed), ask a TA – they're there to help you!**

**3 Instructions**

The file you need to start this assignment is ***lab4.s*** – the assembly language source of the program. This is available in the CASLAB domain, on your Ydrive in the cisc221/labs2018. It’s also included in this document. If you are outside of the CASLAB domain, log into your team’s raspberry pi and retrieve the file from the above ydrive folder at /cas/course/ydrive. To make compilation very simple, just use the following command to compile the lab4.s source file: gcc –g –o lab4 lab4.s. This will create an executable program called lab4. Then use gdb to debug the program by running the command: ***gdb lab4***.

**4 Helpful Details**

There is nothing in this lab that isn't done in the *HLL Translations - C to ARM* item found in the Extras folder. That folder also contains two gdb command reference sheets.

The vast majority of the code was produced by the gcc compiler by compiling a C program to assembly language. I did this so that the lab will be especially helpful to those who are compiling their project1 C code to assembly language instead of manually translating their C code. (Either approach is acceptable and each has its advantages and disadvantages).

I have put strategic comments in the code to help you isolate sections of code. You will be able to concentrate on understanding the heart of the code rather than those sections involved in inputting and printing strings.

There is one noticeable difference between compiled C code and the code examples in the Extras folder and it relates to how we specify the address of an operand by using the =varname pseudo code. Example ldr r0, =var1 which will load the address of var1 into register r0.

The compiler does this a little differently by creating a block of variable addresses within the text section of code (look at lab4.s starting at label .L8 (shown below).

.L8:

.word .LC0

.word msg0

.word stdin

.word ch

.word x

.word arr

.word fusedata

.word msg1

.word .LC1

.word msg2

Each line above will contain the address of the listed variable. We can then load an address into a register with the following examples:

ldr r0, .L8+4 @load r0 with the address msg0. Equivalent to ldr r0, =msg0

ldr r0, .L8+12 @load r0 with address of ch. Equivalent to ldr r0, =ch

ldr r0, .L8+16 @load r0 with address of x. Equivalent to ldr r0, =x

….and so on

.file "lab4.c"

//Create ch: 40 byte character buffer

.comm ch,40,4 //alternative to .space 40.

.global msg0

.data

.align 2

.type msg0, %object

.size msg0, 23

msg0:

.ascii "Enter diffuse string:\000\000"

.global msg1

.align 2

.type msg1, %object

.size msg1, 35

msg1:

.ascii "BOOM!! Bomb Exploded. Bye! Bye!\012\000\000"

.global msg2

.align 2

.type msg2, %object

.size msg2, 27

msg2:

.ascii "Bomb Defused! Good work!\012\000\000"

.global fusedata

.align 2

.type fusedata, %object

.size fusedata, 40

fusedata:

.ascii "abcdefghijklmnopqrstuvwxyz0123456789 \012\000\000"

.global arr

.align 2

.type arr, %object

.size arr, 72

arr:

.word 15

.word 24

.word 38

.word 10

.word 20

.word 38

.word 21

.word 9

.word 38

.word 21

.word 14

.word 6

.word 38

.word 32

.word 38

.word 8

.word 5

.word 3

.global x

.bss

.align 2

.type x, %object

.size x, 4

x:

.space 4

.section .rodata

.align 2

.LC0:

.ascii "%s\000"

.align 2

.LC1:

.ascii "arr is %d\012\000"

.text

.align 2

.global main

.type main, %function

main:

//following 2 lines start main as just another function. Exit at .L4 will also exit like a function.

@ args = 0, pretend = 0, frame = 0

@ frame\_needed = 1, uses\_anonymous\_args = 0

stmfd sp!, {fp, lr}

add fp, sp, #4

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*printf msg\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ldr r3, .L8

mov r0, r3

ldr r1, .L8+4

bl printf

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*fgets(ch,sizeof(ch),stdin)\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ldr r3, .L8+8

ldr r3, [r3, #0]

ldr r0, .L8+12

mov r1, #40

mov r2, r3

bl fgets

b .L2 @branch to head of main loop at .L2

/\* start of innermost part of loop consisting of .L6, .L3, .L2 \*\*\*\*\*\*\*\*/

.L6:

R3=x

R3=x

R2=ch

R3=x

R1=x

R3=arr

R3=arr[x]

R3=arr[x]-2

Fusedata

If ch[x]==fusedataarr[x]-2], then go to 13

ldr r3, .L8+16

ldr r3, [r3, #0]

ldr r2, .L8+12

ldrb r2, [r2, r3]

ldr r3, .L8+16

ldr r1, [r3, #0]

ldr r3, .L8+20

ldr r3, [r3, r1, asl #2]

sub r3, r3, #2

ldr r1, .L8+24

ldrb r3, [r1, r3] r3=fusedata[arr[x]-2]

cmp r2, r3

beq .L3 @good news! current character matches diffuse string

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*printf msg1 and exit\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ldr r3, .L8

mov r0, r3

ldr r1, .L8+28

bl printf

mov r3, #1

b .L4 @exit loop and exit

.L3:

ldr r3, .L8+16

ldr r3, [r3, #0]

add r2, r3, #1

ldr r3, .L8+16

str r2, [r3, #0]

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*head of main loop consisting of .L6, .L3, .L2 \*\*\*\*\*\*\*\*\*\*\*\*/

.L2:

ldr r3, .L8+16

ldr r3, [r3, #0]

ldr r2, .L8+12

ldrb r3, [r2, r3] @ zero\_extendqisi2 (byte data needs to be etended to register size)

cmp r3, #0

beq .L5

ldr r3, .L8+16

ldr r3, [r3, #0]

ldr r2, .L8+12

ldrb r3, [r2, r3] @ zero\_extendqisi2 (byte data needs to be etended to register size)

cmp r3, #10

bne .L6

.L5:

ldr r3, .L8+32

mov r0, r3

mov r1, #72

bl printf

ldr r3, .L8+16

ldr r3, [r3, #0]

cmp r3, #18

beq .L7

ldr r3, .L8

mov r0, r3

ldr r1, .L8+28

bl printf

mov r3, #1

b .L4

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*printf msg2 and exit \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.L7:

ldr r3, .L8

mov r0, r3

ldr r1, .L8+36

bl printf

mov r3, #0

.L4: @exit

mov r0, r3

ldmfd sp!, {fp, pc}

.L9:

.align 2

.L8:

.word .LC0

.word msg0

.word stdin

.word ch

.word x

.word arr

.word fusedata

.word msg1

.word .LC1

.word msg2

.size main, .-main

.ident "GCC: (Debian 4.6.3-14+rpi1) 4.6.3"

.section .note.GNU-stack,"",%progbits

**Answer (Translation/Algorithm):**

I used gdb the debugger to correctly test and diffuse string. I used word and correct index to decipher the string (n-2) and tested in gdb to check if we got message 1 (incorrect diffused string) or message 2 (correctly diffused string).

C-Code

**Diffused String**: nw is th tme 4 gdb

int main() {

char ch[40]

char msg0[23] = “Enter diffuse string: \n”;

char msg1[35] = “BOOM! Bomb Exploded. Bye! Bye!\n\0”;

char msg2[27] = “Bomb Defused! Good Work!\n”;

char fusedata[40] = “abcdefghijklmnopqrstuvwxyz0123456789 \n\0”;

int arr[18] = [15,24,38,10,20,38,21,9,38,21,14,6,38,32,38,8,5,3];

int x;

printf(“%s”, msg0);

fgets(ch, sizeof(ch), stdin);

x=0;

while(ch[x] != ‘\0’ && ch[x] == ‘\n’) { //.L2

if(ch[x] != fusedata[arr[x]-2]) { //.L6

printf(“%s”, msg1);

return 0;

} // end if

x=x+1

} // end while loop

printf(“arr is %d\n”, sizeof(arr)); //.L5

if (x !=18) {

printf(“%s”, msg1);

return 0;

}

else {

printf(“%s”, msg2);

}

return 0;

} // end main