**CSE278: Introduction to Systems Programming (Systems I)**

**Lab #9**

**Due: Mon/Tue May 4/5 during Lab time**

Maximum Points: 50

|  |
| --- |
| **Submission Instructions**  This part of the homework assignment must be turned-in electronically via Canvas. Ensure you name this document *Lab9\_MUID*.docx, where *MUid* is your Miami University unique ID. (Example: Lab9\_ahmede.docx) |
| Copy pasting from online resources is **Plagiarism**. Instead you should read, understand, and use your own words to respond to questions. |
| **Submission Instructions:**  Once you have completed answering the questions save this document as a PDF file (**don’t just rename the document; that is not the correct way to save as PDF**) and upload it to Canvas.  **General Note**: Upload each file associated with homework (or lab exercises) individually to Canvas. Do not upload archive file formats such as zip/tar/gz/7zip/rar etc. |

|  |
| --- |
| **Objective** |
| The objective of this Lab is to review basic concepts of:   * Linux ABI * Debugging with gdb |

|  |  |
| --- | --- |
| **Name:** | **Ben Hilger** |

**Required reading**

* Lecture Slides & ClassNotes: Security
* Lecture Slides & ClassNotes: ComputerArchitecture

# PART A: Linux Application Binary Interface (ABI)

**Goals**

* Details of how a program executes on a Unix/Linux system

.

**Linux ABI defines most of the low-level details of a program including:**

* The register layout (rip, rsp, rbp, rax,, rbx, rcx, rdx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14, and r15)
* The stack frame
  + Pushing to the stack *subtracts* from the stack pointer
  + Popping from the stack *adds* to the stack pointer
  + call
  + ret
* Function prologs and epilogs
* The calling convention (that is, parameter passing)
* Exception handling
* Virtual memory layout
* The binary object format (ELF) (begins with 0x7F)

1. Consider the following code fragment:

|  |
| --- |
| int test()  {  int i = 1;  int j = 2;  return i + j;  }  int main (void)  {  test();  } |
|  |

1. Compile the Lab9.cpp as follows:

**g++ Lab9.cpp**

1. Disassemble the resulting binary using the command

**objdump –S a.out**

1. Attach a screen shot

A close up of text on a white background

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

1. Now modify the code as follows:

|  |
| --- |
| int test( int val1, int val2)  {  return val1 + val2;  }  int main (void)  {  auto ret = test (42, 42);  } |

1. Again from the resulting binary, report the generated assembly language code for relevant for the **main()** function and **test()** function
2. Issue the following command:

**readelf –SW a.out**

How many sections reported there?

There are 28 sections reported

1. To know more details of readelf, issue the command

**man readelf**

1. Issue the following command:

**readelf --debug-dump=frames a.out**

Report the contents of the .eh\_frame table

A screenshot of a cell phone

Description automatically generated

The .eh\_frame contains sections of CIE (Common Information Entry) and FDE (Frame Description Entry) that handle certain parts of exceptions. Every CIE is followed by one or more FDE in the output and each memory address comes one after the other.

1. Look at the *hexdump* of the resulting a.out ELF by issuing the following command:

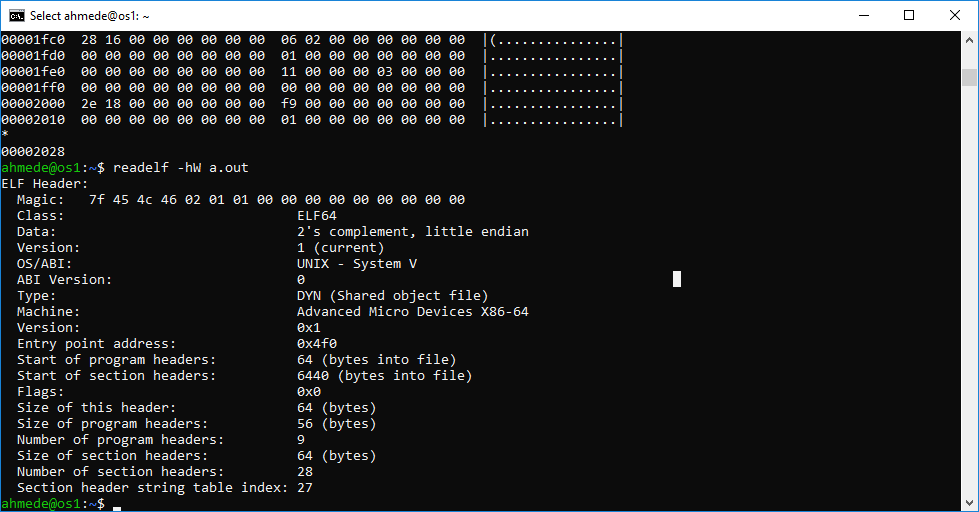
**hexdump –C a.out**

Every ELF file begins with the hex number 0x7F, and continues with the ELF string.

Issue the following command to view the ELF file’s header:

**readelf –hW a.out**

You should see a result like this:



Attach the screen shot what you get from previous command.

**A screenshot of a cell phone

Description automatically generated**

1. **ELF sections**

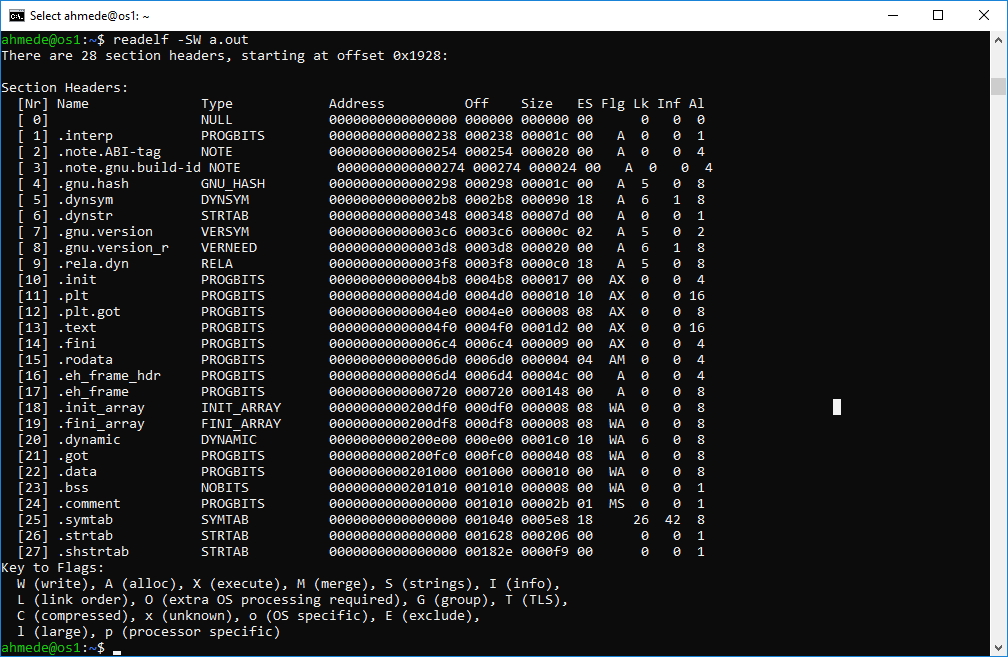
To see a list of all the sections, use the following command:

**readelf –SW a.out**

This will result in something like the following output:

A close up of text on a white background

Description automatically generated



1. What you get from the above command, explain the following sections:

|  |
| --- |
| * eh\_frame/.eh\_frame\_hdr   The eh\_frame and .eh\_frame\_hdr are in charge of handling exceptions and provide tables in how to unwind the stack. According to the report above, it is the PROGBITS type, which means it’s a section that contains either initialized data and instructions or instructions only.  .eh\_frame\_hdr has a memory location of 0000000000000928  offset: 000928  size: 000054 (84 bits)  .eh\_frame has a memory location of 0000000000000980  Offset: 000980  Size: 000168 (360 bits)  0000000000000980 000980 000168   * .init\_array/.fini\_array/.init/.fini   The .init\_array/.fini\_array/.init/.fini sections are in charge of the initialization anf termination functionality. According to the report, the .init\_array is of type INIT\_ARRAY, which means that this section contains an array of pointers to initializations functions. Then the .fini\_array is of type FINI\_ARRAY, which means that this section contains an array of pointers to termination functions. The .init and. .fini are both of type PROGBITS, which means they only contained already initialized data and instructions or instructions only.  .init\_array has a memory location of 0000000000200d88  Offset: 000d88  Size: 000010  .finit\_array has a memory location of 0000000000200d98 Offset: 000d98  Size: 000008  .init has a memory location of 0000000000000688  Offset: 000688  Size: 000017  .fini has a memory location of 0000000000000904  Offset: 000904  Size: 00000   * .dynsym   The .dynsym section contains the symbol table necessary to support dynamic linking. According to the report, the .dynsym has a memory location of 00000000000002c0  Offset: 0002c0  Size: 000108 |
|  |
|  |
|  |

1. Recompile the Lab9.cpp, this time using –v flag

**g++ Lab9.cpp –v**

From the report generated, identify the **PIE** (Position Independent Executable)

A PIE is an executable that can be run from anywhere in the primary memory without worrying about the absolute memory address. This means that now that the file has been configured with PIE as shown by the flags in the report, it can be run anywhere in the primary memory and still function properly.

There are some notable sections that should be pointed out:

* .text (contains most of the code associated with the program)
* .data (contains global variables initialized other than 0)
* .bss (global variables that should be initialized to 0 )

1. a. Now, modify the Lab9.cpp code, just to print one line of text as: “The answer is: 42\n”. This time don’t call any of the test() function from main().

Recompile the code

1. Issue the following command:

**hexdump –C a.out | grep “The” -B1 –A1**

Attach the screen shot what you see from above command.A screenshot of a cell phone

Description automatically generated

1. Answer the following questions

|  |
| --- |
| * What is a pipe in Linux? How we represent pipe?   We represent the pipe with the “|” character and a pipe is a form of redirecting data from one command to another. In the instance above, we are redirecting the output from the hexdump command to the grep command, which then the grep filters the data and then displays that to the console.   * Elaborate the meaning of the command grep.   Grep stands for the Global regular expression print and searches files for a specified pattern of characters and then returns the memory addresses that contain that specified pattern.   * Make an *educated guess* what B and A might refer to in the above command.   The B1 flag displays the contents of one (hence B1) memory address found after the main address found using grep  The A1 flag displays the contents of one (hence A1) memory address after the main address found using grep   * Use different B and A values for the above, then attach another screen shot what you get * A picture containing keyboard    Description automatically generated |

1. Issue the following command to see the loadable components:

**readelf –lW a.out**

Attach the screen shot what you see from above command

# A screenshot of a cell phone Description automatically generated

# PART B: Debugger (gdb)

**Goals**

* How to write more efficient code
* Where to look when hard-to-find bugs arise.

Note: You may use the supplied **crash.cpp** in os1.csi.miamioh.edu (compile using -g switch) to experiment with all the commands that are available in gdb, *alternatively*, you may use your own sample code to experiment and should use NetBeans8.2 built-in gdb debugger. Here are the operations you should try:

1. Try setting and removing breakpoints
2. Step line-by-line in a method (may be the main()) and observe changes in a variable
3. Try navigating the stack frames in the debugger (observe how the call stack looks)
4. Finally, make a screenshot of your whole NetBeans window showing:
   1. The current line of code
   2. The stack trace tab
   3. The variables tab

A screenshot of a social media post

Description automatically generated

* Starting gdb
  + gdb nameOfExecutable
* Running a Program
  + (gdb) run
* Stack Trace
  + (gdb) backtrace
* Examining Variables
  + (gdb) print x
* Listing the Program
  + (gdb) list
* Setting Breakpoints
  + (gdb) help breakpoint

From command shell, issue

**$** gdb –help

It will show all the commands usage

The same help can be found when running gdb already:

$gdb

(gdb) help

To quit from debugger, use the command q

(gdb) q

|  |  |
| --- | --- |
| **Command** | **Purpose** |
| q /quit | Quit from gdb |
| L | List |
| l myfunction |  |
| b main | Puts a breakpoint at the main |
| break 5 | Set a breakpoint at line 5 and run the program |
| Run |  |
| b N | Puts breakpoint at line N |
| r /run | Start script |
| Where |  |
| Up |  |
| p x | Print variable |
| Cont | Continue |
| commnads | End with a line saying just “end” |

# Submission

* No late assignments will be accepted!
* This work is to be done individually
* The submission file will be saved with the name ***Lab9\_yourMUID.pdf***
* Assignment is due by Mon/Tue May 4/5 during Lab time.
* On or before the due time, drop the *electronic copy* of your work in the *canvas*
* Don’t forget to Turn in the files! Lab9\_yourMUID.pdf & Lab9\_yourMUID\*.cpp