

[Skip to page content](#)

# Lab 2: Disassembling and Defusing a Binary Bomb

Assigned: Friday, April 12, 2024

Due Date: Friday, April 26, 2024 at 11:59 pm

Video(s):



Getting Started



(with captions)

and

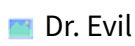


sscanf

## Overview

### Learning Objectives:

- Given x86-64 assembly instructions, students are able to understand how they work together to form higher-level language constructs (comparison, loops, switch statements, recursion, pointers, and arrays).
- Describe the operations of basic x86-64 assembly instructions (jmp, add, mul, mov, lea, etc.). Students should know what the x86-64 size modifiers are, and know how memory addressing syntax works.
- Given a program, students are able to run a program with the `gdb` debugger and know how to step through the program, print values, set breakpoints, and rerun the program with different arguments.

The nefarious  Dr. Evil has planted a slew of "binary bombs" on our machines. A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on `stdin` (standard input). If you type the correct string, then the phase is defused and the bomb proceeds to the next phase. Otherwise, the bomb explodes by printing "BOOM!!!" and then terminating. The bomb is defused when every phase has been defused.

There are too many bombs for us to deal with, so we are giving everyone a different bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date. Good luck, and welcome to the bomb squad!

## Code for this lab

Everyone gets a different bomb to defuse! Substitute `<username>` in the URL below with your UWNNetID in order to download yours. To avoid permission issues, make sure that you unzip `lab2-bomb.tar` while connected to `attu` or `cancun`.

Since each bomb for this lab is unique, each partnership should choose one bomb to work on (*i.e.*, send your partner your bomb). Ideally, the student whose bomb you defuse will be the one to submit, but things will work out either way.

**Terminal:** NOT SUPPORTED, ' `wget` ' command will NOT work!

**Browser:** Navigate to

`https://courses.cs.washington.edu/courses/cse351/24sp/labs/lab2/<username>/lab2-bomb.tar` (sign in with your UW credentials) to download it to your local machine and then copy the tarball to `attu/cancun`.

**Unzip:** Running `tar xvf lab2-bomb.tar` from the CSE Linux environment will extract the lab files to a directory called `bomb$NUM` (where `$NUM` is the ID of your bomb) with the following files:

- `bomb` - The executable binary bomb
- `bomb.c` - Source file with the bomb's main routine
- `defuser.txt` - File in which you write your defusing solution
- `lab2synthesis.txt` - File for your responses to the synthesis questions

## Lab 2 Instructions

### Lab Format

You should do this assignment on a 64-bit CSE Linux VM, a CSE lab Linux machine, or on `attu/cancun`. Be sure to test your solution on one of those platforms before submitting it, to make sure it works when we grade it! In fact, there is a rumor that Dr. Evil has ensured the bomb will always blow up if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so they say.

**Your job is to find the correct strings to defuse the bomb.** To avoid detonating the bomb, you will need to learn how to set breakpoints and single-step through the assembly code in `gdb` (our debugger) while inspecting the current state of the registers and memory. Look at the

Tools section for ideas and tools to use.

While you likely won't look at assembly much beyond this course, the debugger commands and debugging skills that you learn here will translate to code written in C and other programming languages and should pay big dividends later in this course as well as the rest of your career.

The bomb has 5 regular phases. The 6th phase is extra credit, and rumor has it that a secret 7th phase exists. If it does and you can find and defuse it, you will receive additional extra credit points. The phases get progressively harder to defuse, but the expertise you gain as you move from phase to phase should offset this difficulty. Nonetheless, the latter phases are not easy, so please don't wait until the last minute to start! If you're stumped, check the [Tips section](#) and the [Hints section](#).

During this lab, we **strongly recommend** that you keep notes of the steps you took in solving each stage. This will be immensely helpful in helping you to keep track of what's stored at important addresses in memory and in registers at different points in the program's execution. A good strategy for this might be to keep a notetaking app open on your computer so you can copy and paste values between it and `gdb`.

## Bomb Input

There are two different ways to use the bomb, as described below.

Dr. Evil added the second feature in a moment of weakness so you don't have to keep retyping the solutions to phases you have already defused, instead you can put them in `defuser.txt` as you go.

### From the keyboard

Running the bomb executable without arguments will take user input typed into the terminal (*i.e.*, the program will pause, you will type a string, then press [Enter]):

```
$ ./bomb
```

If you are running the bomb within `gdb`, then you use the standard `run` command and type input into the terminal when the program is paused and you don't see the usual `gdb` prompt:

```
(gdb) run
```

Blank input lines are ignored. The bomb will print some text in-between each phase which will indicate success or failure. On success, it will pause again for the next user input; on failure, it will exit back to the shell prompt.

### From a file

Running the bomb executable with the name of a file will read the phase input strings from the lines of the file. For this lab, we are dictating that you use `defuser.txt`:

```
$ ./bomb defuser.txt
```

If you are running the bomb within `gdb`, then you use append this argument to the `run` command:

```
(gdb) run defuser.txt
```

Blank input lines are ignored. The bomb will stop reading the file if it blows up. If the bomb reaches the EOF (end of file) before blowing up, then it will switch over to stdin (standard input from the terminal via the keyboard), meaning that you can use `defuser.txt` with partial bomb solutions.

When using `defuser.txt`, every phase string **MUST** end with the newline character. This just means that you should press [Enter] in-between each phase string that you put in the file but also press [Enter] after the last one (*i.e.*, you should have a blank line at the bottom of the file). We recommend checking this by either:

- Enabling line numbers in your text editor to double-check that the last line is blank.
- Using either `xxd defuser.txt` or `hexdump -C defuser.txt` to see if the last character is `0x0a` (the newline character).

## Defusing Tools

There are many ways of defusing your bomb. At the "highest" level, you can read the assembly code and reason through what it is doing without ever running the program. At the "lowest" level, you can step through the program execution in a debugger to determine what each assembly instruction does without looking up its behavior. As this lab is about learning to read assembly code as well as gaining familiarity with the debugger, we highly recommend a mixture of the two.

All we ask is that you do NOT use brute force! Otherwise, you are cheating yourself out of the knowledge and skills that this lab is supposed to impart on you and you will make the future labs that much more difficult for yourself.

This section is intended to give you the tools to follow both approaches, as desired, in analyzing your bomb, and hints on how to use them:

- `gdb` : The GNU debugger is a powerful command line debugger tool. Some of its many features include: setting breakpoints, line-by-line code execution, examining memory and registers, and viewing of the source code and assembly code (we are not giving you the source code for most of your bomb).
  - See our [GDB section](#) of the debugging tips page for guides, examples, and other resources to get started.
  - For this lab, you will find the following GDB commands most useful:
    - `disas <function>` will display the disassembly of the specified function.
    - `break <line>`, where `<line>` can be specified as a line number, a function name, or an instruction address, will create and set a breakpoint.
    - `run defuser.txt` will run the bomb using `defuser.txt` as the command-line argument until it encounters a breakpoint or terminates.



- `stepi <#>` and `nexti <#>` will move forward by `<#>` assembly instructions ( `stepi` will enter functions whereas `nexti` will go over functions). If omitted, `<#>` will default to 1.
- `print /<f> <expr>` will evaluate the expression `<expr>` and print out its value according to the format string `<f>`. The expression can use variable or register names. The format string can be omitted; see documentation for more details.
  - For other documentation, you can (1) type `help` at the `gdb` command prompt, (2) type `man gdb` or `info gdb` at the shell prompt, or (3) run `gdb` under `gdb-mode` in Emacs.
- `objdump -t bomb > bomb_sym` : This will print out the bomb's symbol table into a file called `bomb_sym` . The symbol table includes the names of all functions and global variables in the bomb, the names of all the functions the bomb calls, and their addresses. You may learn something by looking at the function names!
- `strings -t x bomb > bomb_strings` : This will print the printable strings in your bomb and their offsets within the bomb into into a file called `bomb_strings` .


Looking for a particular tool? How about documentation? Don't forget that the commands `apropos` and `man` are your friends. In particular, `man ascii` is more useful than you might think. If you get stumped, use office hours and the discussion board.

## Tips and Hints

### x86-64 assembly instructions

There are many online resources that will help you understand any assembly instructions you may encounter. In particular, the instruction references for x86-64 processors distributed by Intel and AMD are exceptionally valuable. They both describe the same ISA, but sometimes one may be easier to understand than the other.

-  Intel Instruction Reference
-  AMD Instruction Reference
- 

The instruction format used in these manuals is known as "Intel format". This format is **very different** than the format used in our text, in lecture slides, and in what is produced by `gcc` , `objdump` and other tools (which is known as "AT&T format". You can read more about these differences in the recommended textbook (CSPP p.177) or  on Wikipedia . ***The biggest difference is that the order of operands is SWITCHED.*** This also serves as a warning that you may see both formats come up in web searches.

### x86-64 Calling Conventions

The x86-64 ISA passes the first six arguments to a function in the following registers (in order): `%rdi` , `%rsi` , `%rdx` , `%rcx` , `%r8` , and `%r9` . The return value of a function is passed in `%rax` .

## Using `scanf`

First let's look at `scanf` ("scan format"), which reads in data from `stdin` (the keyboard) and stores it according to the format specifier into the *locations* pointed to by the additional arguments:

```
int i;
printf("Enter a number: ");
scanf("%d", &i);
```

- After `printf` prints the shown prompt, `scanf` will wait for the user to enter a number and hit [Enter] before storing the input into `i` with the format of an integer. Notice how `scanf` uses the *address* of `i` as the argument.

Lab 2 uses `sscanf` ("string scan format"), which is similar to `scanf` but reads in data from a string instead of `stdin`:

```
char* mystring = "123, 456";
int a, b;
sscanf(mystring, "%d, %d", &a, &b);
```

- The first argument, `mystring` , is the input string.
- The second argument, `"%d, %d"` is the format string that contains format specifiers to parse the input string with.
- After matching the input string to the format string, the extracted values are stored at the addresses given in the additional arguments. After this code is run, `a = 123` and `b = 456` .

Reference information can be found online for [!\[\]\(d5d7044e5caf6907399af2dced8d6ff8\_img.jpg\) `sscanf`](#) , [!\[\]\(0718ece108875f096be32ef1aea65831\_img.jpg\) `scanf`](#) , and [!\[\]\(5413432958edc783e108af585cf114e8\_img.jpg\) `printf`](#) .

## Phase Context Hints

The correct string has a particular format for each phase: It could be a phrase, sequence of numbers, characters, or both. We suggest figuring out what format is required before trying to understand what is happening at each phase! If you're still having trouble figuring out what your bomb is doing, here are some hints for what to think about at each stage:

1. **Comparison:** Dr. Evil is a fan of inspiring (or not so inspiring) quotes.
2. **Loops:** Phase 2 calls a function, then starts a loop. What inputs are required by the function? How is the loop using those inputs?
3. **Switch statements:** Figure out what inputs are required. For each input, what values would cause a call to `explode_bomb` ? Avoid those!
4. **Recursion:** What are the initial arguments of the recursive function? How are they manipulated before the recursive call occurs? What do we do with the final value once we exit out of the recursive function?
5. **Pointers and arrays:** Where are the arrays stored? What are their contents, and how are they being manipulated?

## 6. Linked lists (extra credit)

## Lab 2 Synthesis Questions

You will need to use the CSE Linux environment in order to get addresses that are consistent with our solutions.

Start with a *fresh* copy of `lab0.c` and examine `part_2()` using the following commands:

```
$ wget https://courses.cs.washington.edu/courses/cse351/24sp/labs/lab0.c
$ gcc -g -std=c18 -o lab0 lab0.c
$ gdb lab0
(gdb) layout split
(gdb) break fill_array
(gdb) break part_2
(gdb) run 2
```

Now answer the following questions. You will find the following GDB commands useful: `nexti` , `finish` , `print` , and `refresh` .

1. At what offsets (in bytes), relative to `%rsp` , are the variables `value` and `array` from `part_2()` stored when accessed before the last call to `fill_array` ? (Hint: Your answers should be a decimal number of bytes between the address of the variable and the address stored in `%rsp` ) [2 pt]
2. Give the *relative* addresses (*i.e.* of the form `<+#>` ) of the instructions that perform the initialization and update statements for the for-loop in `fill_array` . [2 pt]
3. What address is the string `"*** LAB 0 PART 2 ***"` stored at in memory? Which part of the memory layout is this? Give the relative instruction address (*i.e.* of the form `<function+#>` ) for the assembly instruction that references this memory address. [2 pt]

*⚠ The following questions are open-ended! They are less about "right" or "wrong" and more about you giving us your personal reflection.*

4. Looking back on your workflow for completing the lab's phases, what assembly tracing/reading strategy did you find most useful that you would recommend to a future student in 351? Provide an example of how this strategy was useful for you in debugging lab 2. [2 pt]
5. What aspect of x86-64 assembly code did you find the most difficult to learn/deal with? Why do you think that aspect was particularly difficult for you based on any of your relevant past experiences? [2 pt]


# Submission

You will submit: `defuser.txt` and `lab2synthesis.txt`.

It is your responsibility to make sure that your `defuser.txt` file works on your (or your partner's) assigned bomb and obeys the following formatting rules, otherwise our grading script is likely to conclude you defused zero phases:

- Put your answer for each phase in one line. Your answer for phase 1 should be in the first line, answer for phase 2 on the second line, and so on.
- Do **not** put your name or other information at the top of the file.
- Do **not** add numbering or other "comments" for your answers (e.g., `1. This is my answer for phase 1`).
- Make sure all your phase answers are followed by a newline character. When programming, the newline character is represented as `'\n'`, but it is a non-visible character in a text editor and is typically inserted by pressing [Enter] or [Return].

After submitting, please wait until the autograder is done running and double-check that you passed the "File Check" and "Compilation and Execution Issues" tests. If either test returns a score of -1, be sure to read the output and fix any problems before resubmitting. Failure to do so will result in a programming score of ZERO for the lab.

Submit your files to the "Lab 2" assignment on  Gradescope. Don't forget to add your partner, if you have one.

*If you completed the extra credit*, also submit the same `defuser.txt` file to the "Lab 2 Extra Credit" assignment.

It is fine to submit multiple times up until the deadline; we will only grade your last submission. If you do re-submit, you must re-submit BOTH files again AND add your partner again to the new submission.