CS213 Fall 2024
Bomb Lab: Understanding Machine-level Programs
Due: Oct 29, 11:59PM

1 Introduction

The nefarious *Dr. Evil* has planted a slew of “binary bombs” on our class 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`. 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 each student a 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!

Step 1: Get Your Bomb

You can obtain your bomb by pointing your web browser at:

```
http://moore.wot.eecs.northwestern.edu:15213/
```

Be patient. This website might take 20 seconds to load. After it does, it will display a binary bomb request form for you to fill in. Enter the NetID and email address of you and your partner (or just the first NetID if you are working independently) and hit the Submit button. The server will build your bomb and return it to your browser in a `tar` file called `bombk.tar`, where k is the unique number of your bomb.

If you’re working with a partner, you can just email them a copy of the bomb. It’s okay to have multiple copies of the bomb files. Just be sure you put both of your NetIDs in the fields when requesting it.

After you download that file, you’ll need to upload it to a class server so you can work on it. You can do so with the ‘scp‘ command. It copies files between computers. The following command will copy the file from your own computer into your home directory on Moore (replace $bombk$ with the filename for your bomb and $netid$ with your NetID).

```
scp bombk.tar netid@moore.wot.eecs.northwestern.edu:~/
```

You MUST run the command on your own computer inside the directory which has your `bombk.tar`. It should work fine on MacOS. On Windows, use the same command via WSL, Git Bash, or Windows's built-in ssh/scp commands.

This will put your `bombk.tar` file into your home directory on Moore.

You could also use an SFTP tool, such as WinSCP or FileZilla, to upload the file to a class server. You'll have to use one of these if you use PuTTY to connect from Windows.

Move the `bombk.tar` file to a (protected) directory in which you plan to do your work. Then give the command: `tar -xvf bombk.tar`. This will create a directory called `./bombk` with the following files:

- `README`: Identifies the bomb and its owners.
- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine and a friendly greeting from Dr. Evil.

If for some reason you request multiple bombs, this is not a problem. Choose one bomb to work on and delete the rest.

Step 2: Defuse Your Bomb

Your job for this lab is to defuse your bomb.

You must do the assignment on one of the class servers. In fact, there is a rumor that Dr. Evil really is evil, and the bomb will **always blow up** if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so we hear. Moore, Hanlon, or any of the Batman-themed Wilkinson machines are all safe. Moore is probably the easiest.

To see how the bomb works at a high level, take a look at the code in `bomb.c`. It demonstrates that there are six phases which read lines from the input to determine passphrases for each phase.

You can use many tools to help you defuse your bomb. Please look at the **hints** section for some tips and ideas. The best way is to use gdb (or another debugger) to step through the disassembled binary.

Each time your bomb explodes it notifies the bomblab server, and you lose 1/2 point (up to a max of 20 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful!

If you lose too many points from explosions, you may request a new bomb. However, the solutions will be different...

The first four phases are worth 10 points each. Phases 5 and 6 are a little more difficult, so they are worth 15 points each. So the maximum score you can get is 70 points.

Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. However, the last phase will challenge even the best students, so please don't wait until the last minute to start.

You may pass a text file to the bomb. If you run:

```
linux>  ./bomb psol.txt
```

then it will read the input lines from `psol.txt` until it reaches EOF (end of file), and then switch over to `stdin`. In a moment of weakness, Dr. Evil added this feature so you don't have to keep retyping the solutions to phases you have already defused.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the nice side-effects of doing the lab is that you will get very good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

Clarifications and corrections will be posted on Piazza.

Handin

There is no explicit handin. The bomb will notify your instructor automatically about your progress as you work on it. You can keep track of how you are doing by looking at the class scoreboard at:

```
http://moore.wot.eecs.northwestern.edu:15213/scoreboard
```

Again, this page might take a bit to load. This web page is updated continuously to show the progress for each bomb.

If you have exploded your bomb too many times and are receiving a low score, you can always request a new bomb. Be aware that the solutions to defuse the bomb are unique to each bomb though. . .

Hints *(Please read this!)*

There are many ways of defusing your bomb. You can examine it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to defuse it. This is probably the fastest way of defusing it.

We do make one request, *please do not use brute force!* You could write a program that will try every possible key to find the right one. But this is no good for several reasons:

- You lose 1/2 point (up to a max of 20 points) every time you guess incorrectly and the bomb explodes.
- Every time you guess wrong, a message is sent to the bomblab server. You could very quickly saturate the network with these messages, and cause the system administrators to revoke your computer access.
- We haven't told you how long the strings are, nor have we told you what characters are in them. Even if you made the (incorrect) assumptions that they all are less than 80 characters long and only contain letters, then you will have 26^{80} guesses for each phase. This will take a very long time to run, and you will not get the answer before the assignment is due.

There are many tools which are designed to help you figure out both how programs work, and what is wrong when they don't work. Here is a list of some of the tools you may find useful in analyzing your bomb, and hints on how to use them.

- `gdb`

  The GNU debugger, this is a command line debugger tool available on virtually every platform. You can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts.

  The CS:APP web site `http://csapp.cs.cmu.edu/public/students.html` has a very handy single-page `gdb` summary that you can print out and use as a reference. Here are some other tips for using `gdb`.

  – To keep the bomb from blowing up every time you type in a wrong input, you'll want to learn how to set breakpoints.

  – For online documentation, type "`help`" at the `gdb` command prompt, or type "`man gdb`", or "`info gdb`" at a Unix prompt.

  – `gdb` is a command-line tool, but there are numerous higher-level user interfaces that have been built on top of it. Some of these include the TUI (built into gdb), the ddd visual debugger, development environments (Eclipse, CLion, KDevelop). Some people also like to run `gdb` under `gdb-mode` in `emacs`.

  – To have a better visual layout for GDB, we recommend you install the "GDB Dashboard" extension for it. CS211 had you do so by default, but you can also find it here: `https://github.com/cyrus-and/gdb-dashboard` (follow the "Quickstart" instructions)

- `objdump -t`

  This will print out the bomb's symbol table. 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! Use the '>' operator to write to a file: "objdump -t > name.txt"

- `objdump -d`

  Use this to disassemble all of the code in the bomb. You can also just look at individual functions. Reading the assembler code can tell you how the bomb works. Use the '>' operator to write to a file: "objdump -d > name.s"

  The objdump argument "–prefix-addresses" can be a way to improve the clarity of jump instructions.

  While `objdump -d` gives you a lot of information, it cannot tell you everything. For example, some call instructions may be indirect, i.e. the function being called is not known until runtime. Only GDB can divine such dynamic information.

- `strings`

  This utility will display the printable strings in your bomb.

Looking for a particular tool? How about documentation? The commands `apropos`, `man` (short for "manual"), and `info` are your friends. For example, `man ascii` might come in useful. `info gas` will give you more than you ever wanted to know about the GNU Assembler. Also, the web may also be a treasure trove of information. If you get stumped, feel free to ask your instructor for help.

Getting Started

Once you've gotten the lab files onto a lab computer, you can start working on them. The file `bomb` is an executable program you can run. However, you don't know the passwords it's looking for yet, so running it is pretty dangerous. Instead, we need to figure out how to analyze and run it safely.

If you DO run the bomb without knowing the passwords, press Ctrl+C several times to abort.

Get Assembly Code

The `objdump` tool, as described above is capable of "disassembling" the bomb. This will transform the executable back into readable assembly code, although it won't generate readable C code for you. So you'll end up having to read and understand the assembly code in order to figure out how the bomb works.

While you can just run `objdump -d` every time you want to look at the assembly, a better way would be to write the disassembled code into a file so you can read it in a text editor whenever you want. To do so, you can redirect the output of `objdump` into a file like this:

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

That will make you a new file, "bomb.s", which is really just a text file with the assembly instructions. You can open it in an editor to look at the contents. It'll even syntax highlight in most editors, since we gave it the right file extension (`.s` for assembly source files). Note that this is just a copy of the original assembly, so you're welcome to edit this without having an affect on the original bomb, and you can make a new fresh copy whenever you want by re-running that `objdump` command.

Run the Bomb with GDB

To understand what's going on inside the bomb, inspecting the assembly isn't enough. You also need to look at the values in memory to see what's going on. First, we recommend that you get the "GDB Dashboard" environment explained in the previous section. It'll make `GDB` much easier to use.

To run `GDB` on the bomb, you can do the following:

```
linux>  gdb bomb
```

Or, later in the lab if you want to provide a text file with your already-discovered passphrases, you can provide arguments to the executable with the following (but replace `input.txt` with whatever your input filename is called:

```
linux>  gdb --args bomb input.txt
```

Place a Breakpoint in GDB

Before doing anything else in GDB, you should make a breakpoint so that the bomb doesn't explode by accident. To do so, type the following into GDB:

```
>>>  break explode_bomb
```

A breakpoint causes GDB to stop executing code if it ever reaches that point. So this will mean that instead of running the `explode_bomb()` function, GDB will pause your code there. Be careful though, if you ever reach that breakpoint and keep stepping through or running code, your bomb will explode. You'll have to set this breakpoint each time you start GDB. Don't forget!

If you ever do reach the breakpoint, you'll want to either exit GDB which you can do by typing "quit" or you can restart your bomb by typing "run". Generally, you can always use the "run" command to restart your code in GDB, and since the session is still running, all of your breakpoints will still be there.

Step Through a Function with GDB

If you want to step through the assembly instructions of a certain function, you'll first want to place a breakpoint at the start of it. For example, `break phase_1`

Then once you're in there, you can use the "`nexti`" command to move to the next assembly instruction. That will step over function calls (not follow them). If you want to follow function calls, use the "`stepi`" command instead.

If you reach a breakpoint and find that what you really want to do is just continue running code until the next breakpoint, you can do that with the "`continue`" command.

Inspect Memory with GDB

Finally, once you have GDB running on your bomb, you'll want to use it to view values in memory while it's running. Some phases might compare against certain values in memory, which you'll need to take a look at.

To print memory while paused in GDB, you can use the "x" command to examine the contents of memory. A really good guide for this command is here: https://visualgdb.com/gdbreference/commands/x

Here are some examples of how to examine memory with the "x" command.

View 16 bytes of data in hexadecimal starting at address 0x400000:

```
>>>  x/16xb 0x400000
```

View ten 32-bit signed integer values starting at address 0x00007fffffffe098:

```
>>>  x/10dw 0x00007fffffffe098
```

View 64 bytes of data in hexadecimal starting 32 bytes before `%rsp`:

```
>>>  x/64xb $rsp-32
```