CS-2011, Machine Organization and Assembly Language, B-term 2020

Lab Assignment 2: Defusing a Binary Bomb

Professor Jonathan Weinstock

Introduction

The nefarious *Dr. Evil* has planted a slew of "binary bombs" on the course server. 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 *disarmed* 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 disarmed.

There are too many bombs for us to deal with, so we are giving each student a separate 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 One: Get Your Bomb

To obtain a bomb, you may visit the *Bomblab* server via

https://cs2011.cs.wpi.edu/

username: cs2011d20
password: d20/cs2011

and follow the links to Bomblab to get a bomb and see the scoreboard

This will display a binary bomb request form for you to fill in. Enter your WPI username and a nickname, and click the Submit button. The server will build a new bomb for you and return it to your browser as a **`tar`** file called **`bomb`*`k`*`.tar.gz`**, where ***`k`*** is the unique number of your bomb.

Save the **`bomb`*`k`*`.tar`** file to a directory where you plan to do your work. Then extract it using the command: **`tar xvfz bomb`*`k`*`.tar`** . This will create a directory called **`./bomb`*`k`*** containing the following files:

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

If you don't like the bomb you've been given, that's no big deal. Someone else will pick up the slack; you can always get a new one. Choose one bomb to work on and ignore the rest.

Note: You *must* use your WPI user name (i.e., login ID) when requesting a new bomb. Each bomb is associated with a unique student, and the user name is how the graders identify which bomb belongs to whom.

Note: The URL for downloading bombs might not be reachable from off campus due to firewalls.

We are unable to build bombs to run on CCC systems or for Macintosh or Windows platforms. Therefore, these bombs will only work on generic 64-bit Linux systems, such as the virtual machine you are using for this course.

Step Two: Defuse Your Bomb

Your job for this lab is to defuse your bomb. You must do the assignment on your virtual machine. In fact, there is a rumor that Dr. Evil really is evil, and the bomb will tend to blow up if run elsewhere. There are many other tamper-proofing devices built into the bomb as well, or so we hear. There is also an auto-solver out there for an older version of the Bomblab – rumor has it that if you try to use it, the bomb will automatically explode!

You can use many tools to help you defuse your bomb. Please look at the "**Hints — Please read this!**" section below for some tips and ideas. The best way is to use your favorite debugger to step through the disassembled binary.

Each time your bomb explodes it notifies the *bomblab* server, and you lose ½ point (up to a max of 20 points) in the final score for this assignment. So there are consequences to exploding the bomb. You must be careful!

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. Therefore, 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 two phases will challenge even the best students, so please don't wait until the last minute to start.

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

```
linux> ./bomb solution.txt
```

then it will read the input lines from **`solution.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 become *very good* at using a debugger. This is a crucial skill that will pay big dividends in the rest of your career.

Step Three: Hand in your Bomb

This is an individual project. All submissions are electronic. Please let the Professor and TAs know about problems and/or issues. Clarifications and corrections will be sent to the class via Canvas announcements and/or the Canvas discussion list.

The bomb will notify the graders automatically about your progress as you work on it. You can keep track of how you are doing by looking at the class scoreboard, linked off the main 2011 Webpage through the Bomblab link

This web page is updated continuously to show the progress for each bomb. (If you cannot reach it from off campus, check your firewall settings.) Check to make sure that your bomb is listed.

Even if your bomb is correctly listed in the scoreboard, you *must* submit your solution file and bomb number to *Canvas* under the assignment *Bomblab.*

The name of your solution file *must be* **`userName-bomb#.txt`** — e.g., **`weinstock-2130.txt`** for weinstock's bomb #2130. The penalty for not properly naming your solution file and identifying the bomb is 25% of the total project points. *Submitting a solution to a bomb not issued to your username will result in a grade of zero for the project.*

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 input string to find the right one. But this is no good for several reasons:

- You lose ½ 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 access to the server.
- 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 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, along with hints on how to use them. Many of these are already loaded on your VM.

- **`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 (source code is not provided for most of your bomb), set breakpoints, set memory watch points, and write scripts.

  The textbook authors' 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. There are many other on-line tutorials and manuals for the use of **`gdb`**. Here is one:

  http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

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 — particularly, at the point in the program that actually explodes bombs.
- For online documentation, type "**`help`**" at the **`gdb`** command prompt, or type "**`man gdb`**", or "**`info gdb`**" at a Linux prompt. Some people also like to run **`gdb`** under **`gdb-mode`** in **`emacs`**.

- **`Eclipse`** — the *Eclipse CDT* integrated development environment

  The *Eclipse* debug perspective is a nice, user-friendly front end to **`gdb`**. It lets you display source and/or assembly code and shows where your program counter is. It also lets you display data, registers, and other information, even without the source code.

  Eclipse CDT was installed on course virtual machines before creating the **`.ova`** file. To run it, simply type **`eclipse &`** in a Linux command shell.

  If, for some reason, you are running on a different virtual machine, you can download and install *Eclipse* yourself. There are several ways to do this. One is to select the icon on the left tool bar for the Ubuntu software installer. In the installer, search for *Eclipse CDT* and then let the installer install it.

  Alternatively, you can install *Eclipse* with the following shell command:

  ```
  sudo apt-get install eclipse-cdt
  ```

  Unfortunately, the version of Eclipse which Ubuntu installs is an older version, which does not support Git for source code control. So your best choice is to go to eclipse.org, download the latest version from there, and then install that. That is what we did for the course VM.

  See **Notes on Using Eclipse to Debug Binaries** at the end of this document. In case you are wondering, the "-cdt" tells the installer that you want the version of Eclipse set up for debugging C programs.

- **`peda`** — *Python* Exploit Development

  This is a set of plugins for **`gdb`** written in *Python* to improve the presentation of debugging information, including colorizing the display of disassembly code, registers, memory information, etc., during debugging. It is a relatively new tool and was introduced to the class three years ago by one of the undergraduate Teaching Assistants. Many students used it and liked it. You can obtain **`peda`** from

  https://github.com/longld/peda

  For more information, see

  https://github.com/longld/peda/blob/master/README.md

Note that these instructions tell you how to set up PEDA so it is automatically activated then you start gdb. This is convenient if you are debugging assembly language, but not so convenient if you want to want to go back to debugging C. When you want to go back to C, just rename the configuration file.

- **`DDD`** — the Data Display Debugger

  This graphical user interface is a user-friendly front end to **`gdb`**. It lets you display source and/or assembly code and shows where your program counter is. It also lets you display data, registers, and other information, even without the source code. **`DDD`** *used to be* one of the friendliest of debugging tools, but unfortunately it appears to be no long supported.[1]

  `DDD` should already be installed on the virtual machines of this course. If necessary, you can install it and its accompanying man page by executing the shell command

  ```
  sudo apt-get install ddd
  ```

  A comprehensive manual with tutorial can be found at

  http://www.gnu.org/software/ddd/#Doc

  Although most of the manual and tutorial is focused on debugging source code, you can use it just as easily for debugging assembly code, displaying registers, etc. You can also drop down into raw **`gdb`** commands as needed.

- **`Nemiver`** — a new C/C++ Debugger

  This is a more recent GUI debugger for *C* and *C*++. Like both *Eclipse* and *DDD*, it uses **`gdb`** as the "back end." A CS-2011 student in D-term 2014 found it and shared it with the class. He reported that **`nemiver`** is "very similar to **`DDD`**, yet updated to a far more recent interface and much less clunky to use." **`Nemiver`** can be installed on the virtual machine by opening the *Ubuntu Software Center* icon on the left toolbar and searching for "**`Nemiver`**" or by typing "**`sudo apt-get install nemiver`**" to a command shell.

  Note that the Professor has not extensively tested **`nemiver`**. However, see this footnote.[2]

- **`objdump -t`**

  This shell command 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!

---

1 Maintenance on **`DDD`** seems to have stopped in about 2007. It is, however, distributed as an option with most of the major Linux releases. **`DDD`** was the debugger of choice when *Bomblab* was first introduced to CS-2011 in 2013.

2 The student also wrote, "If you are fed up with **`DDD`** or using **`gdb`** alone in the terminal, this might really help you out! In testing **`nemiver`**, I found that it has a great workflow almost splendid for CS2011—it just lacked a few buttons in the toolbar for step/next over ASM instruction instead of lines of C code. In an effort to make **`nemiver`** better for CS2011 students, I modified it so that it has buttons in the toolbar that do exactly that—and I figured I should make this modified program available and easy to install for anyone interested. I put together a super easy Terminal command that automates the downloading of the program's source code, patching it with my mostly minimal modifications, installing dependencies, and compiling all from scratch. That command, tested with the Ubuntu 12.04 virtual machine for this course, is:

`wget -O script.sh` http://goo.gl/OpU438"

This script might not work correctly with the Ubuntu 18.04 virtual machine because it has specific version numbers embedded in it. You need to edit it for your version of Ubuntu. Also, the script has specific nemiver version numbers in it. You will need to update those, also.

- **`objdump -d`**

  Use this shell command 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.

  Although **`objdump -d`** gives you a lot of information, it doesn't tell you the whole story. Calls to system-level functions are displayed in a cryptic form. For example, a call to **`sscanf`** might appear as:

  ```
  8048c36:  e8 99 fc ff ff  call   80488d4 <_init+0x1a0>
  ```

  To determine that the call was to **`sscanf`**, you would need to disassemble within **`gdb`**.

- **`strings`**

  This utility will display the printable strings contained in a binary file such as your bomb.

- The **`disassemble`** (**`disas`**) facility in **`gdb`** and in most GUI debuggers.

  This allows you to print out or view the assembly language version of machine code at a particular location, such as at the start of a function or at the program counter.

Looking for a particular tool? How about documentation? Don't forget, the commands **`apropos`**, **`man`**, and **`info`** are your friends. In particular, **`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 or TAs for help.

Notes on Using Eclipse to Debug Binaries

To use *Eclipse* to debug your binary bomb, consult the *Eclipse* Help menu. Select

Contents>
  C/C++ Development User Guide >
    Tasks >
      Running and debugging projects >
        Debugging >
          Debugging an existing executable

This explains how to *import* an executable program such as your binary bomb and to turn it into an *Eclipse* project. This page is reproduced in the file DebuggingExistingBinaryInEclipse.pdf, which is available on Canvas.

When it asks you to select a binary parser, select the **`GNU Elf Parser`**. The binary parser is the tool that decodes the compiled binary file. Eventually, it will ask you to create a *launch configuration*. When it does, specify the text file containing your six lines of input in the *Argument* tab of this dialog.

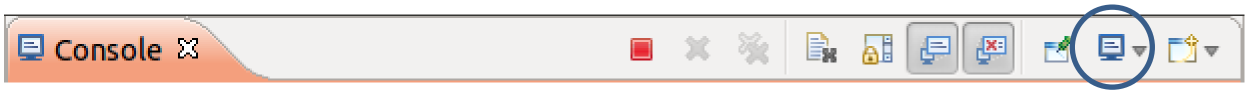
Now you are ready to debug your binary bomb. Simply invoke the *Debug* command from the *Run* menu. This opens the *Debug Perspective*. Note that in the group of tabs in the upper right of this perspective, a tab called *Registers* opens up, alongside the *Variables* tab.

Note: *Eclipse* will automatically find the **`bomb.c`** file that was part of your downloaded bomb. This is simply a wrapper for the real bomb, for which source is not available. However, **`bomb.c`** lets you get a handle on your program before it starts running.

You will need to open the *Disassembly* view, to show the current assembly code. In this view, you can set breakpoints, invoke the *Run To Line* command, etc. It may also be helpful to open the *Memory* view. Both of these views can be opened from the *Window > Show View* menu.

When debugging at the assembly language level, you will want to step instruction-by-instruction. To do this, click the *Instruction Stepping Mode* button at the top of the *Debug* tab in the *Debug* view. It looks like this:

Finally, you may want to drop down into **`gdb`** at times. To do this, go to the *Console* tab and pull down the *Display Selected Console* menu (circled in the picture below) to display the **`gdb`** console (*not* the **`gdb traces`** console!). Enter **`gdb`** commands into this console and see the results displayed by **`gdb`**.



You can also open a second console tab, in order to display both the output of the bomb and also the **`gdb`** console. To do this, pull down the menu to the right of the circled one and select the appropriate item.