Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2012

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 3: Defusing a Binary Bomb

Assigned on: **18th Oct 2012**
Due by: **25th Oct 2012**

Part 1: Hand-on project

Introduction

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*. 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

Each students will attempt to defuse his/her own personalized bomb. Each bomb is a Linux binary executable file that has been compiled from a C program. To obtain your bomb, update your SVN folder. A new subfolder `assignment3` will show up, which contains your individual bomb in a folder `bomb`*k*, where *k* is the unique number of your bomb. The folder contains 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.

Step 2: Defuse Your Bomb

Your job is to defuse the bomb.

You can use many tools to help you with this; please look at the **hints** section for some tips and ideas. The best way is to use your favorite debugger to step through the disassembled binary.

The phases get progressively harder to defuse, but 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.

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

```
    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.

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 is 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.

If you consider using brute force, please consider the following: 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 (wrong) assumptions that they all are less than 80 characters long and only contain lowercase 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. Here are some 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.

  – The CS:APP Student Site at `http://csapp.cs.cmu.edu/public/students.html` has a very handy `gdb` summary.

  – There are a number of tutorial for `gdb` on the web, like this: `http://www.cs.cmu.edu/~gilpin/tutorial/`.

  – For other documentation, type "`help`" at the `gdb` command prompt, or type "`man gdb`", or "`info gdb`" at a Unix prompt. Some people also like to run `gdb` under `gdb-mode` in `emacs`.

- `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!

- `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.

  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 in your bomb.

Looking for a particular tool? How about documentation? Don't forget, the commands `apropos` and `man` are your friends. In particular, `man ascii` might come in useful. Also, the web may also be a treasure trove of information. If you get stumped, feel free to ask your TA for help.

Part 1 hand-in instructions

Upload your input file named `psol.txt` to the `assignment3/bomb`*k* folder of your SVN folder. Commit new versions of the file as you progress through the phases of the bomb.

You can keep track of how you (and the other students) are doing by checking the link that will be provided in the course Web page's 'Assignments' section shortly after the release of the assignment. This web page is updated continuously to show the progress of each member of the bomb squad. You can use a nickname instead of your bomb number by adding a file `NICKNAME` to your `bomb`*k* folder. We will read the nickname form the first line of that file.

Part 2: Paper exercise

NOTE: Unless otherwise stated, the assembly code in this assignment is IA32 assembly code.

Question 1: Arrays in Assembly

Suppose the start address of a `short` integer array `S` and integer index `i` are stored in registers `%edx` and `%ecx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in register `%eax` if it is a pointer and in register element `%ax` if the result is a `short` integer. Note that each expression can be implemented with one single assembly instruction by choosing a suitable addressing mode.

```
S+3
S[5]
&S[i]
S[4*i+2]
S+2*i-7
```

Question 2: Structs

Consider the following structure declaration:

```
struct line {
    char *id;
    char  rgb[3];
    int   x;
    int   y;
    int   width;
    int   height;
    char  f;
} l;
```

a) Draw the memory layout of the structure including the address offset of each field.

b) How many total bytes does the structure require?

c) Optimize the structure in terms of memory consumption and state the new size.

d) The following procedure operates on the structure:

```
void obfuscatedoperation(struct line *lp) {
    ________ = ________;
    ________ = ________;
}
```

Using the original struct from above, the compiler generated the following assembly code for the body of the procedure:

```
movl    8(%ebp), %eax
sall    16(%eax)
sall    20(%eax)
```

From this assembly code, fill in the missing lines in `obfuscatedoperation(...)` and provide a more meaningful name for the procedure.

Part 2 hand-in instructions

This part of the assignment is a paper exercise. If you want your solution to be revised please hand it in during your exercise class on the due date.