Homework #2: Bomb Defuse Game

Due: Apr. 9, 11:59 PM
Responsible TA: Mingi Jung (wjdaslrl4475@unist.ac.kr)

The nefarious Dr. Wi has planted a slew of *binary bombs* on the CSE251 System Programming server machines! Each of these bombs is a program that consists of a sequence of phases. We managed to recover the `bomb.c` file that he used, but we couldn't recover the code contained in the phases. We do know however that each phase expects you to type a particular string on `stdin`. If you type the correct string, that phase will be defused and the bomb will proceed 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 happy hacking!

- **Warning #1**: This homework can be very difficult if you don't understand what's going on. Make sure you read all the documentation before trying to do the homework!
- **Warning #2**: This homework may be challenging for students who do not have much experience with low-level assembly and GDB. I strongly recommend that you start early.

- **Environment.** To complete this homework, you must access the server at `10.20.18.55` via `ssh` using the account information provided to you. Note that this server can only be accessed from the UNIST internal network, so make sure to use a VPN if you are connecting from outside.

  In the `hw2` directory under your home directory, you will find the `bomb` and `bomb.c`.

  - `bomb`: The executable binary bomb.
  - `bomb.c`: Source file with the bomb's main routine.

  Note that the `bomb` is specifically compiled for the architecture of our course server and will not run in other environments. Therefore, you must complete this hoemwork on the course server.

- **Main task: defuse your bomb.** Your goal is to defuse your bomb. This homework consists of seven phases. You will earn points for each phase you defuse, with a maximum total of 145 points. The point distribution for each phase is shown in the table below:

	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	Phase 6	Phase 7	Total
Point Value	10	10	15	25	25	30	30	**145**

  Be careful! Each time the bomb explodes (i.e., `explode_bomb()` is invoked), you will lose 3 points. Penalties apply only to the first 20 explosions; any explosions after that will not result in additional deductions. In other words, the maximum penalty is -60 points.

  Each time you defuse a phase or trigger the bomb, the corresponding event is sent to the instructor's scoring server to track each student's score.

- **Bomb usage.** To run the `bomb`, you can use the command:

```
$ ./bomb
```

Then, a message like the one below is displayed, and the program receives an input value from standard input (keyboard) for each phase.

```
...
Welcome to my fiendish little bomb. You have 7 phases with
which to blow yourself up. Happy hacking!

[*] Phase 1
Input:
```

Once you type an input string and press `Enter`, the `input` variable will point to the string entered so far (`\n` will be replaced with `\0`). If your input is `[END]`, the program will terminate immediately. You can also terminate the program immediately by pressing `Ctrl+c`. Refer to the `read_line` function in `bomb.c`.

If you want to provide input values using a prepared file, use the `<` shell redirection operator. For example, you can provide the contents of `input.txt` as standard input to `bomb`:

```
$ cat input.txt
Input string for phase 1
Input string for phase 2
[END]
$ ./bomb < input.txt
```

To avoid accidentally detonating the bomb, you will need to learn how to step through the assembly code at the instruction level and how to set breakpoints in GDB. 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 homework is that you will get very good at using a debugger, a crucial skill that will pay big dividends the rest of your career.

You can use many tools to help you defuse your bomb. Refer to the Hint section for some tips and ideas. The best way is to use GDB to step through the disassembled binary.

- **Submission guidelines.** You should submit your report. You should upload a single PDF file on BlackBored.
  - Your report must describe the answer to each question in this homework. Your report can be written in either English or Korean.
  - The name of the PDF file should have the following format: `Your_ID-hw2.pdf`. For example, if your ID is 20231234, then you should submit a file named `20231234-hw2.pdf`.
  - If your solution includes some code (assembly, C, etc.), you should embed them in your PDF.
- **Late submission policy.** Late submission will be assessed a penalty of 10% per day (We will only accept late submissions of up to 3 days).
- **Scoreboard.** You can check your current score by looking at our scoreboard: `http://10.20.18.55:2060/`
  - Your score on the scoreboard is not your final score for this homework. Your work will be evaluated based on both (1) your scoreboard score and (2) the quality of your submitted report. Be sure to write the report as clearly and thoroughly as possible.
  - Do not attack the scoring (e.g., scoreboard, web service) environments! Our scoring system logs all of your accesses. Any suspicious action, even a seemingly minor one, will be treated as cheating and will result in a score of zero for this homework.
  - If you think the services are not working properly or have any questions, please publicly upload your question on the BlackBored.
  - ⋆ Bonus points will be awarded to (1) the first three students who quickly defuse all phases (regardless of the number of bomb explosions) and the (2) three students with the highest final scores on the scoreboard.

Phase 1 (10 report points)

(a) (3 points) Reverse engineer the function `phase_1`, and show the corresponding C psudocode. What does this program do?

(b) (2 points) Reverse engineer the `eq` function, and show the corresponding C psudocode. What does `eq` function do?

(c) (5 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

Phase 2 (10 report points)

(a) (3 points) Reverse engineer the function `phase_2`, and show the corresponding C psudocode. What does this program do?

(b) (2 points) Explain what `sscanf` (at `0x401bee` in `read_numbers`) does.

(c) (1 points) Analyze the `read_numbers` function. How many numbers does this function take as input?

(d) (4 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

Phase 3 (15 report points)

(a) (5 points) Reverse engineer the function `phase_3`, and show the corresponding C (psudo) code. What does this program do?

(b) (5 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

(c) (5 points) How many possible inputs are there that can defuse this phase? List all valid inputs.

Phase 4 (15 report points)

(a) (5 points) Reverse engineer the function `phase_4`, and show the corresponding C psudocode. What does this program do?

(b) (5 points) Reverse engineer the function `func4`, and show the corresponding C psudocode. What does this program do?

(c) (5 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

Phase 5 (15 report points)

(a) (7 points) Reverse engineer the function `phase_5`, and show the corresponding C psudocode. What does this program do?

(b) (8 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

Phase 6 (20 report points)

(a) (8 points) Reverse engineer the function `phase_6`, and show the corresponding C psudocode. What does this program do?

(b) (2 points) Describe the layout of the `struct` used in this function.

(c) (10 points) Explain in detail how you defused this phase. If you show your work, you can get partial points even if you cannot defuse this phase.

Phase 7 (20 report points)

(a) (10 points) Reverse engineer the function `phase_7`, and show the diagram of the stack layout after the execution of instruction of `0x401916` (`call 401040 <puts@plt>`).

(b) (10 points) What is the requirement for your input in order to directly invoke the function `phase_defused`? Explain in detail how you defuse this phase. If you used a script to defuse this phase, include the script in the writeup. If you show your work, you can get partial points even if you cannot defuse this phase.

Note

- **Analysis scope.** You do not need to analyze the internal mechanisms of `explode_bomb()` and `phase_defused()`. Your goal is simply to avoid triggering `explode_bomb()` and to reach `phase_defused()`.
- **Grading policy.** Your work will be evaluated based on both (1) your scoreboard score and (2) the report score. Be sure to write the report as clearly and thoroughly as possible.
- **Be careful about plagiarism!** We will conduct strict cross-plagiarism detection for evaluation, including a series of answer generated by ChatGPT, code found online, and your submissions. Therefore, we do not recommend consulting ChatGPT or online solutions. Last semester, we identified several cases of plagiarism using an automated tool. Also, do not share your input values. It is easy to verify from your report PDF whether you solved the bomb on your own.
- **Do not use brute force!** While it is theoretically possible to write a program that brute-forces every possible value, doing so is a very bad idea for several reasons. You lose 3 points every time you enter an incorrect answer and the bomb explodes. In addition, each incorrect guess sends a message to the scoring server. A large number of such attempts could quickly saturate the network and may lead the system administrators to revoke your access to the server. Furthermore, we have not disclosed the length of the strings or the set of characters they contain. Even under the incorrect assumption that every string is shorter than 80 characters and consists only of letters, each phase would still involve 26^{80} possible guesses. Even at 100,000,000,000 guesses per second, brute-forcing would take far too long and would not produce the answer before the assignment deadline.
- **Questions.** If you have any requests or questions (technical difficulties, late submission due to inevitable circumstances, etc.), please ask the TAs on Blackboard. We generally encourage the use of Blackboard for discussions. However, for urgent issues or secret issues, you can send an email to the responsible TA.

Hint (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.

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.` GDB, or GNU Debugger, is a command-line tool available on virtually every platform, designed for debugging programs in languages like C and C++. It allows you to trace through a program line by line, examine memory and register contents, and inspect both source code and assembly code. For our purposes GDB is essential because it will allow you to reverse-engineer the binary `bomb` file without access to the original source code. By setting breakpoints, monitoring memory changes, and examining assembly instructions, you will systematically analyze the program's behavior, identify the required inputs, and progress through each bomb "phase" safely.

To open GDB, simply type:

```
$ gdb bomb
```

The metarial [1] 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.
- It is recommended to use `layout asm` and `layout regs` in GDB, as they make it easier to inspect the assembly code and register values during debugging.

`$ objdump -t bomb.` 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 -M intel -d bomb.` 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.

`$ strings -t x bomb.` This utility will display the printable strings in your bomb.

Writing byte sequences to the file. In this example, we describe how to write (append) byte sequences to `input.txt` using Python 3.

```
import sys

payload = b``
payload += b`A' * 4   \\ Add \x41 for 4 times
payload += b`\xef\xf9\x12\xea'  \\ Add 0xea12f9ef to the payload in
                                  \\ little-endian manner

with open('input.txt', 'ab') as file:
    file.write(payload)
```

References

[1] CSE251: System Programming. 2026. Summary of GDB commands for x86-64 Systems. https://websec-lab.github.io/courses/2026s-cse251/metarials/gdb_reference.pdf.