PROJECT 3:
ASSEMBLY BOMBS AND EXPLOITS

Shinwoo Kim
Teaching Assistant
shinwookim@pitt.edu
https://www.pitt.edu/~shk148/

Spring 2023, Term 2234
Friday 12 PM Recitation
5502 Sennott Square

Course News!

- Project 3 is out!
  - Due: Friday, March 3rd, 2023
  - Multiple parts, submission via gradescope
- Mid-Term exam has been delayed
  - See announcement on Discord

Project 3: The Bomb

Please read the write-up before starting this lab!

1. Introduction
   - You will each be given a *binary bomb* which consists of a sequences of *phases* (*traps*)
   - Each phase expects you to type a particular string
     - via the standard input
   - You can defuse each trap by entering the correct string (*'code'*)
   - Entering an incorrect code will cause your bomb to explode
     - Don't worry…the bomb doesn't actually cause your computer to explode
   - The bomb is *defused* when all phases have been defused
   - Your task is to defuse your bomb before the deadline

Welcome to the bomb squad!

The Bomb: Getting your bomb

2. Getting your bomb & Setting-up
    - Each student will be supplied with a unique bomb
        - Constructed with a random choice of phases; assigned in random order
    - **See the project write-up for instructions on downloading your bomb**
    - Once you have received your bomb, save it in a secure directory
        - IT IS YOUR RESPONSIBILITY TO MAKE SURE YOUR CODE IS SECURE!
        - The `~/private` directory on Thoth should be useful
    - While the bomb is designed to run on top of any x86-64 machine running a sufficiently recent Linux installation, we highly recommend you do this project on Thoth
        - No lost points will be returned because you did the project in a different environment than Thoth and have not tested it on Gradescope before the deadline

The Bomb: Defusing & Grading

3. Defusing your bomb

- Take a look at the files you downloaded
- The bomb is an executable not a `.c` source file.
  - An executable is a binary file containing the code and data
  - A binary file which consists of 1s and 0s
- To make this binary executable more readable (for humans like you), we can convert this file into assembly

4. Project submission

- You will submit your *codes* and the bomb executable via gradescope
- To do so, place the codes in a text file: bomb.txt
- Test to make sure the file is formatted correctly
  - Run the `bomb` using this file as your input
  - `./bomb < bomb.txt`
    - The `<` symbol redirects the contents of the file into the programs standard input
- This is how gradescope will grade your code!

bomb.txt

```
This is my answer
123 123 32 32
Blarg foo bar
askdj
<make sure to include an empty line>
```

The Bomb: Defusing hints

- There are many ways to defuse your bomb
  1. You can examine the disassembly to figure out the correct *codes* without ever running the program
     - This is a useful technique, but it may not give you enough information about what the bomb is doing.
  2. You can run the bomb with a debugger, inspect its memory locations, watch what it does step by step, and use this information to defuse it.
  3. A combination of both techniques is probably the strongest combination you can use.
- 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 approach will be probably no good because:
    - 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) assumption that the codes are less than 80 characters long (size of your terminal) and contain only lowercase letters, then you will still have 2680 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.
  - This will also **unnecessarily hog-up resources** on Thoth for everyone
    - The course staff reserve the right to terminate your process if we suspect if this is the case

The Bomb: Defusing hints

1. How can we examine the disassembly?
   - The `objdump` program can be used to display the *disassembly* of binary executables
     - I.e., we can view the assembly code of the program
   - `objdump -d bomb` disassembles all the bomb code
   - `objdump -d bomb > bomb.s` stores the disassembly to a file
2. How can we view the status of the machine while the bomb is running?
   - `gdb bomb` will open up the GNU debugger (with the `bomb` *loaded*)
   - With GDB, you can view the contents of:
     - Registers, stack (memory), instruction stream, etc.
   - We can also set breakpoints
     - Hint: If we set a breakpoint on the function that explodes the bomb…our bomb will never blow up (unless we want it too)
   - Once again…see lab0 on a walkthrough of GDB!

Project 3: The Attack.

1. Introduction
   - Download the handout from the course website using `wget`
   - Copy the handout to your `~/private/` directory and un-zip using `unzip``
   - You are to exploit a buffer overflow to change the value of a constant, stack-allocated variables.
   - You will also craft a buffer overflow attack to hijack the control flow
   - Unlike The Bomb, the targets don't explode!
   - Read the lab instructions carefully!

- It is very important to understand how the stack *grows* in this project
  - *The stack grows in what direction?*
  - Drawing the stack helps!

Attack: Buffer Overflow

```
void read_my_number() {

   // stack allocate some variables
   const int my_number = 72; // my_number is a constant => cannot change
   char buf[BUFSIZE]; // BUFSIZE = 32

`  // print favorite number
   printf("My number is %d and nothing can change that\n", my_number);

   gets(buf); // get a string from stdin

   // print favorite number again bc its a great number
   printf("My number is %d and nothing can change that\n", my_number);
}
```

How can we overflow this buffer to modify `my_number`?

Output

```
My number is 72 and it will always be 72 and nothing can change that
1234567 # user input
My number is 72 and it will always be 72 and nothing can change that
Returned to main safe and sound
```

Attack 1: Modify `my_number` to hold `449`

- Goal: Provide an input string that modifies `my_number`
  - Using buffer overflow
  - Make sure the program does not segfault
  - Output: `My number is 449 and nothing can change that`
- Hints
  1. Size of the buffer is 32 bytes (`#define BUFSIZE 32`)
     - Hence buffer overflow occurs if we supply a input that is larger than 32 bytes
  2. Thoth stores data in **little-endian** (LSB first!)
  3. Since we are providing the input as *strings*, you need a way to map hexadecimal data into ASCII characters
     - A helper program `HEX2RAW` is provided to aid you in this process
     - See appendix A in handout
     - To store `0xdeadbeef`, pass `ef be ad de` into `HEX2RAW` (note the endianness of the input)
     - **C strings must always be appended with the null terminator** `\0`

Attack 2: Calling a function maliciously

- Goal: Hack the call stack and redirect the program to execute another existing procedure
  - Disassembling `TARGET` should reveal a function called `hack()`
  - Create a exploit string causing a buffer overflow that results in the execution of the hack function rather than returning to `main()`
  - Output: `You've been HACKED!`
- Hints
  1. `retq` loads an address into `%rip` from the stack
     - We can modify the return address with a buffer overflow attack (and `retq` will copy it into `$rip`)
  2. Looking at the assembly code may be in determining the new return address
     - `objdump -d` helps get us the disassembly
     - `gdb`'s `disas` too!
  3. Again, little-endian ordering!

Attack: Inserting malicious code

```
void hack(unsigned val) {
   printf("You’ve been HACKED!\n");
   if (val == COOKIE) {
       printf("Yes! Called passing the right argument (%d)\n", val);
   } else {
       printf("Misfire! Called it passing the wrong argument (%d)\n", val);
   }
   exit(0);
}
```

How can we figure out the value of COOKIE?

Attack 3: Inserting malicious code

- Goal: Hack the call stack and redirect the program to execute another existing procedure
  - Same as Attack 2
  - But this time, make it seem like we passed in the correct argument
- Hints
  1. The C ABI tells us where the function arguments go
     - `%rdi` holds the first argument
  2. Hence, we can inject some code to update `%rdi` and call `hack()`
     - Update `%rip` (like before)
     - But now, we should transfer control to our injected code (not `hack()`)
  3. How do we inject code?
     - Write some assembly, store it on the stack
     - Update `%rip` to address on the stack
     - But how do we get the assembly code to ASCII? See appendix B

Submission

To receive credits for the project, you need to upload the following files to Gradescope:

- Bomb defusing (3+4+5+6+7 points)
  - bomb (your executable!)
  - bomb.txt (the answers to the traps)
- Exploiting memory (25 points)
  - exploit1.txt (5 points)
  - exploit2.txt (10 points)
  - exploit3.txt (10 points)

Helps and Hints

- You may find the GDB help videos attached in lab 4 to be quite helpful for this process
- Slides from last recitation also goes in-depth about assembly
- As always, review lecture slides if you are struggling to understand conceptual materials