Assignment 6: Binary bomb

Due: Mon Nov 27 11:59 pm
No late submissions accepted.

Assignment by Michael Chang & Julie Zelenski
idea originated by Randal Bryant & David O'Hallaron (CMU)

Learning goals

Completing this assignment will give you:

- solid practice in reading and tracing assembly code
- an understanding of how data access, control structures, and function calls translate between C and asm
- experience reverse-engineering
- a compelling reason to invest in mastering the gdb debugger!

Overview

For the code-study exercises, you will examine a security vulnerability in a uncarefully-written program, exploit it using a stack buffer overflow attack, and work through how stack protector defends against such attacks. Then, it's on to the pièce de résistance: defusing a nefarious binary bomb! This is fun stuff and we hope you enjoy solving our puzzles as much as we enjoy creating them!

Getting started

Check out a copy of your cs107 repository with the command:

```
git clone /afs/ir/class/cs107/repos/assign6/$USER assign6
```

Do not start by running the bomb to "see what it will do..." You will quickly learn that "what it does" is explode :-) When started, it immediately goes into waiting for input and when you enter the wrong response, it will explode and deduct points. Thoroughly read the bomb information in this writeup and the advice page before attempting do anything with bomb!

1. Code study: buffer overflow

The program in `atm.c` simulates the operation of a simplified ATM. The code makes a good faith attempt at keeping your money safe, but its use of the deprecated `gets` function introduces a significant security vulnerability. This vulnerability is the focus on this exercise.

Skim the code in `atm.c` to understand its basic operation. This ATM offers only one option, a "fast cash" withdrawal of $200. A text file containing sunets and balances serves as the bank's database. Your login name is your sunet and the password for everyone is their sunet reversed.

Run `samples/atm_soln` to try to make a withdrawal. With a paltry $107 in your account, your request is denied— sad times! Never fear, just impersonate Michael Chang with `samples/atm_soln mchang91` to get the hookup. You know his password and that guy has so much coin, he'll never miss it. (If you run the program again, you'll note all balances return to their original levels. No money actually changes hands in this ATM; which is a blessing given its lack of security.) With a password scheme that lame, sure, anyone can get rich, but the more sophisticated exploit is getting the bank to hand you money, but without taking it from anyone else (and least of all from Michael!).

The program does take pains to try to maintain bank security by rejecting unknown users and invalid passwords and denying withdrawals in excess of the current balance. Yet a bit of hacking can score $200 free and clear with nary a complaint about insufficient funds. Your CS107 knowledge of assembly and the runtime stack is finally going to pay off!

The chink in the armor of this ATM is in its use of `gets` to read the user's password. We first looked at `gets` in assign3 (/class/cs107/assign3/) and recently revisited it in lab7 (/class/cs107/lab7). Answer the following questions in your `readme.txt` file.

a. It's not just this call to `gets` that is dangerous, any such call is. Explain why it is **impossible** to use `gets` safely.

The exploit you are to pull off is to login as yourself, view your account balance, and despite its lack of funds, convince the ATM to dispense $200 to you without decrementing your balance. Start your attack by studying the C code and disassembly for `fast_cash` to work out a precise

understanding of the stack layout.

b. On a piece of scratch paper, diagram the stack frame for `fast_cash` and work out where the local variables, saved registers, and stack housekeeping are placed. In your readme, summarize your diagram: indicate the total size of the stack frame for `fast_cash` and identify the order/position of the data within the frame. What data will be overwritten if you enter a password that is one character too long? What about a password that is 30 characters too long or 60 too long?

Your tactic will be to enter a password with excess characters, but not just any old letters, instead bytes carefully chosen to overwrite the stack housekeeping in such a way to take control of the execution. Look carefully at the disassembly for `main`. Where is the code supposed to return after a call to `fast_cash`? Where would you rather it return instead? There is a "password" you can give to `fast_cash` that will cause it to return to that more desirable location. Work out what this password is on paper using the stack diagram you sketched previously.

c. In your readme, identify the precise length and contents of the password to enter for your buffer overflow attack to reroute control so as to dispense money to you.

Now it's time to put your plan into action. You are going to create a file `password.txt` that contains the input you worked out above and feed that file to the ATM program. The characters in your "password" are a bit odd and your regular editor may frown upon them, so we provide a tiny program that lets you specify raw bytes as an array and writes to a file. Edit the raw bytes in `create_password.c`, then compile and run that program to write your `password.txt` file.

d. Verify that your password.txt works by piping it as input to the program like this: `samples/atm_soln <password.txt`. (You can also use sanity check to perform this test) It should log you in, report your too-small balance, and dispense $200 in cash to you, without changing your existing balance. (No question to answer in readme.txt for this sub-part, just confirm that you ran sanitycheck and successfully absconded with the cash!)

Pro-tip: If you enjoyed this little taste of security, you'll love CS155 (http://cs155.stanford.edu)!

2. Code study: stack protector

The `samples/atm_soln` was compiled from `atm.c` without stack protection (gcc flag `fno-stack-protector`); the `samples/protected_atm_soln` was compiled from the same `atm.c` file but with protection (`fstack-protector`). We first introduced stack protector in lab3 (/class/cs107/lab3), but that was before we had the assembly know-how that we now do. Let's dig into how this stack protection works. Answer the following questions in your `readme.txt` file.

a. What happens when you try to exploit the protected version with your password.txt input, e.g. `samples/protected_atm_soln <password.txt`?
b. Compare the disassembly for both `fast_cash` functions (one with and one without stack protection). The bulk of the instructions will be the same (although addresses will have shifted and operations may be slightly shuffled) but a few additional instructions have been inserted into the function prolog and epilog in the protected version. Identify those instructions.
c. Upon entry to a function, the protector code writes a value to the stack in a particular location. When leaving the function, it reads back the value from that location and verifies it was not perturbed. This value is called the *canary* (as in "canary in a coalmine"). How many bytes is the canary? What location on the stack is the canary written to? Why was that particular location chosen?
d. The canary is not a static value. Instead, its value varies from run to run. Run the program under gdb and set a breakpoint within `fast_cash` and use your mad gdb skills to dig out the value of the canary. Run the program again and observe the canary has a different value. Why is important that the canary value not be predictable? (Or phrased differently, how could you defeat stack protector if you knew the fixed value of the canary?)

3. Binary bomb

Those nefarious Cal students have broken into our myth machines and planted some mysterious executables we are calling "binary bombs." These programs are believed to be armed and dangerous. Without the original source, we don't have much to go on, but we have observed that the programs seem to operate in a sequence of levels. There are 4 levels in total. Each level challenges the user to enter a string. If the user enter the correct string, it defuses the level and the program proceeds on. But given the wrong input, the bomb explodes by printing an earth-shattering `KABOOM!` and terminating. To deactivate the entire bomb, one needs to successfully defuse each of its levels.

The Cal students have littered our systems with these landmines and we need your help. Each of you is given a bomb to disable. Your mission is to apply your best asm detective skills to work out the input required to pass each level and render the entire bomb harmless.

Your bomb is given to you as an executable, i.e. as compiled object code. From the assembly, you will work backwards to construct a picture of the original C source in a process known as *reverse-engineering*. Once you understand what makes your bomb "tick", you can supply each level with the input it requires and defuse it. The levels get progressively more complex, but the expertise you gain as you move up from each level should offset this difficulty. One confounding factor is that the bomb has a hair trigger, prone to exploding at the least provocation. Each time your bomb explodes, it notifies the staff, which deducts from your score. Thus, there are consequences to detonating the bomb-- you must tread carefully!

Reverse-engineering requires a mix of different approaches and techniques and will give you an opportunity to practice with a variety of tools. The most powerful weapon in your arsenal will be the debugger and an important learning goal of the assignment is to expand your gdb prowess. Building a well-developed gdb repertoire can pay big dividends the rest of your career!

Bomb logistics

Our counter-intelligence efforts been able to confirm a few things about how the bombs operate:

- If you start the bomb with no command-line argument, it reads input typed at the console.
- If you give an argument to the bomb:

  ```
  ./bomb input.txt
  ```

  the bomb will read lines from `input.txt` until it reaches EOF (end of file), and then switch over to reading from the console. This feature allows you to store inputs for solved levels in `input.txt` and avoid retyping them each time.

- Explosions can be triggered when executing at the shell or within gdb. However, gdb offers you tools you can use to intercept explosions, so your safest choice is to work under gdb and employ protective measures.
- The bomb in your repository was lovingly created just for you and is unique to your id. It is said that the bomb can detect if an impostor attempts to execute your bomb and won't play along.
- The bombs are designed for the myth computers (running on the console or logged in remotely). There is a rumor that the bomb will refuse to run anywhere else.
- The bombs were compiled from C code using gcc. Apparently Cal students don't know how to edit a Makefile to change the flags to achieve much obfuscation of the object code.
- The Cal students also weren't aware the function names would be visible in the object code, so they didn't take pains to disguise them. Thus, a function name of `initialize_bomb` or `read_five_numbers` can be a clue. Similarly, they played it straight with use of the standard C library functions, so if you encounter a call to `qsort` or `sscanf`, it is the real deal.
- Direct modification of the binary bomb executable can change its behavior, but be forewarned that we will test your submission against your original unmodified binary, so while hacking the executable is great fun, it won't be of much use as a strategy for solving the levels. (Although it can be an entertaining and educational exercise in suppressing explosions...)
- There is one important restriction: *Do not use brute force!* You could write a program to try every possible input to find a solution. But this is trouble for several reasons:
  - You lose points on every incorrect guess which explodes the bomb.
  - A notification is sent on each bomb explosion. Wild guessing will saturate the network, creating ill will among other users and attracting the ire of the system administrators who have the authority to revoke your privileges because you are abusing shared resources.
  - We haven't told you how long the strings are, nor have we told you what characters they can contain. Even if you made the (wrong) assumptions that they all are less than 80 characters long and only contain lowercase letters, you will have 26^{80} guesses for each level. Trying them all will take an eternity, and you will not have an answer before you graduate.
  - Part of your submission requires answering questions that show your understanding of the assembly code, which guessing will not provide. :-)

Bomb readme

The bulk of your effort on bomb goes into defusing the levels, but we have a few follow-up questions. Answer these questions in your `readme.txt` file.

a. What tactics did you use to suppress/avoid/disable explosions?
b. `level_1` contains an instruction of the form `mov $<hex>,%edi`. Explain how this instruction fits into the operation of `level_1`. What is this hex value and for what purpose is it being moved? Why can this instruction reference `%edi` instead of the full `%rdi` register?
c. `level_2` contains a `jle` that is not immediately preceded by a `cmp` or `test` instruction. Explain how a branch instruction operates in such a context. Under what conditions is this particular `jle` branch taken?
d. Explain how the loop in the `winky` function of `level_3` is exited.
e. The `read_array` function used in `level_4` declares a local variable that is stored on the stack at `0x8(%rsp)`. What is the type/size of this variable? Explain how can you discern its type from following along in the assembly, even though there is no explicit type information in the assembly instructions. Within read_array there is no instruction that writes to this variable. Explain how the variable is initialized (what value it is set to and when/where does that happen?).
f. Explain how the `cmp` function is used in `level_4`. What type of data is being compared and what ordering does it apply?

Advice/FAQ

Don't miss out on the good stuff in our companion document!

Go to advice/FAQ page (advice.html)

Grading

For this assignment, we expect a total number of points around 85, apportioned as follows:

- **readme.txt** (45 points) For the code-study and bomb questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions.
- **password.txt** (8 points) A successful exploit via password.txt earns these points.
- **Input.txt** (32 points) Each level you have solved earns 8 points. We will test by running `./bomb input.txt` on your submission. The `input.txt` file in your submission should contain one line for each level you have solved, starting from level 1.
- **Bomb explosions** (up to -6 points deducted) Each bomb explosion notification that reaches the staff results in a 1 point deduction, capped at -6 points total.

Finish and submit

The `input.txt` file should contain the strings that correctly defuse the levels that you solved. When grading, we will run your bomb on your input.txt to verify which levels you have successfully defused. Malformed entries in your input.txt or wrong line-endings (see FAQ at end of advice page (advice.html#faq)) will cause grading failures. To avoid surprises, be sure that you have verified your input.txt in the same way we will in grading (i.e. `./bomb input.txt`).

Review the How to Submit (/class/cs107/submit.html) page for instructions. There is **no grace period** for submissions on this assignment. The due date is firm for all students!

How did it go for you? Review the post-task self-check (/class/cs107/selfcheck.html#assign6).


KEEP
CALM
I'M A
ENGINEER
I CAN DEFUSE THIS!