xdKevin Zhang

Application Security CS GY 9163

**Homework 1: Beware if Geeks Bearing Gift Cards**

On NYU Classes, submit a link to your GitHub repository. The repository should be **private**, and you should add the instructor/TA's GitHub account as a contributor to give them access for grading.

For this section, your instructor is: Kevin Gallagher, GitHub ID `**kcg295**`

Your TA is: Evan Richter, GitHub ID `**evanrichter**`

The repository should contain:

**Part 1: Setting up Your Environment**

Relatively self-explanatory. The steps for Part 1 are detailed below:

GitHub Set-up

* Git-VCS has been installed.
* Git Bash has been installed (Windows)
* Git signed commits have been set-up
* GitHub Account created – kzhang112
* GitHub-Repo created (“AppSec\_1.1”) – set to “private”

Travis CI

* Travis-CI account created and tied to GitHub
* .travis.yml file created
* Iterations (changes of code) Travis builds until build is successful

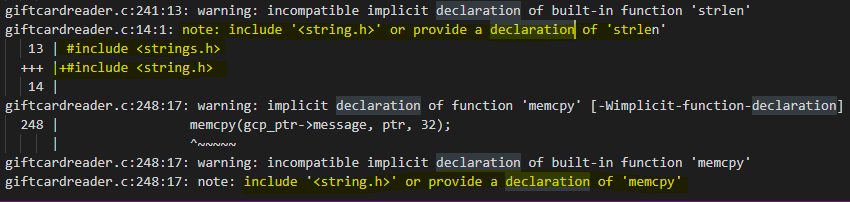
**Part 2: Auditing and Test Cases**

Set-up

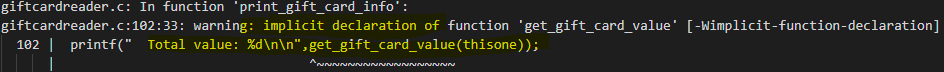
* Files added to GitHub Repo
  + “*examplefile.gft*”
  + “*giftcardexamplewriter.c*”
  + “giftcard.h”
  + “*giftcardreader.c*”
  + “*Makefile”*

After adding the above files to my GitHub Repo (“kzhang112/AppSec1.1”), I attempted to run both “*giftcardreader.c”* and “*giftcardwriter.c*” using my Visual Studio Code compiler (I ran the program through the Mingw-x32 for Windows). “*Giftcardwriter.c*” did not return any errors prior to running the program, and I was able to print out an example gift card (similar to the gift card provided in the original repository).

However, “*giftcardreader.c”* ran into an initial error indicating that I needed to include a string header.



After adding in “#include <string.h>”, I encountered another error.



By adding in an extra int for get\_gift\_card\_value prior to the code block where we see the error, it gives the compiler information that can be used later in the program (e.g. in this case, on line 102).

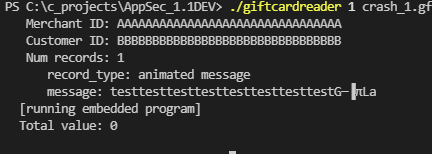


Finally, I was able to build and run the file on the example gift card that I had just created.

Find Flaws

* “*Crash\_1.gft*” created

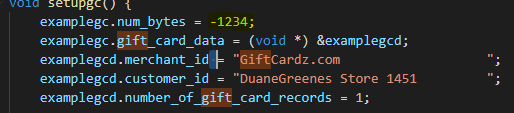
COME UP WITH SCREENSHOT ON HOW TO REPRODUCE CRASH GIFT CARD FILE



This initial has to do with the op-code 0x01 function on line 30 of "giftcardreader.c". After examining the code, I noticed that 0x01 pushes forward "regs[arg1] = \*mptr". Mptr is a pointer for the beginning of the message data, while arg1 provides the value for the register (e.g. regs[]).

Similar to the hang case below, this function interfaces with the "arg1 value" of the giftcard. If we set the arg1 value to anything higher than 16-bytes (storage capacity), we can trigger a segmentation fault and crash the program - this is due to a uffer overflow.

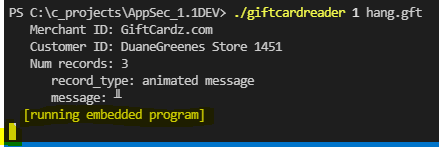
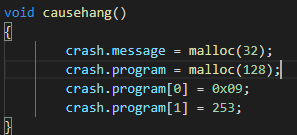
* “*Crash\_2.gft*” created



This crash is caused by setting "num\_bytes" section of 'giftcardwriter.c' to anything aside from a normal integer. Inputting a large negative number (e.g. -1234) creates a malformed gift card that can cause the malloc to break in 'giftcardreader.c', as this is outside the range of integers accepted.

“Giftcardexample.num\_bytes” stores the negative number, which is then used for memory allocation within the reader code. If we try to run the malformed gift card, we receive a segmentation fault (outputted by Travis when running it through a GitHub push), due to the fact that the fread cannot read anything negative. Adding in error handling for the negative number will cause the program to exit without crashing, while also informing the user of the error.

* “*Hang.gft*” created



The hanging program is leveraging a new function, titled causehang() in "giftcardwriter.gft," to force the program to loop instructions. This specifically refers to case opcode 0x09, referring to animate function. We alter the number of records of the card from 1 to 3 in this case, causing a loop (no chance of exit, error, or satisfying a condition). This is namely due to the presence of arg1 (in the line "pc += (char)arg1," line 64). Arg1 accepts 8bit integer values; however, does not know how to read extraneous information - an extra byte registers as a "-" sign.

Because of case opcode 0x09, memory location will keep increasing by "3" until a limit has been reached (256). If we input a specific Arg1 value (in this case 250), memory location will decrease, increase, and decrease again due to actions conducted in the beginning and end of the while loop (lines 22-61). Our fix, therefore, is to alter arg1 (specifically the code on line 64) to be and unsigned character, which would break in the case of a negative value.

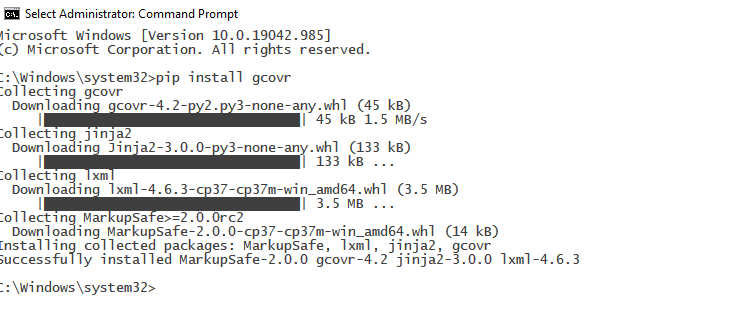
* *“Bugs.txt”* created, describing the above issues
* *Crash\_1.gft*, *Crash\_2.gft*, *hang.gft* added to Travis with commits
  + *Crash\_1.gft* and *Crash\_2.gft* both cause Travis to exit with a code of “2,” while *hang.gft* causes Travis to continuously run. The hang Travis build failed on its own after 10 minutes of running, showing that the program did hang on “running the embedded program.”
* Bugs highlighted in “*Bugs.txt*”fixed in “*giftcardreader.c*”
* Fixes added to Travis and run to show that the bugs were indeed fixed.

**Part 3: Fuzzing and Coverage**

In order for me to utilize GCC with LCOV (as I am running on a Windows computer), I followed the following tutorial (<https://txt.arboreus.com/2015/05/29/howto-get-lcov-test-coverage-on-windows.html>). This involved obtaining MSYS2, a fork of MinGW/MSYS. This will allow me to utilize LCOV on my builds.

Afterwards, I ran LCOV on giftcardreader.c with --coverage, and included my test suite within the folder under “part\_3”.

<https://github.com/gcovr/gcovr> - through cmd prompt



<https://marketplace.visualstudio.com/items?itemName=alexdima.vscode-lcov> – through visual studio code

<https://marketplace.visualstudio.com/items?itemName=JacquesLucke.gcov-viewer> – GCOV visual studio marketplace

You should notice that there are portions of the program that are *uncovered* (i.e., the code was not executed while processing your test suite). Pick two lines of code from the program that are currently not covered and create test cases that cover them.

An easy and effective way of finding crashes and getting higher coverage in a program is to fuzz it with a fuzzer like AFL. Fuzz the program using AFL, following the [quick-start instructions](https://lcamtuf.coredump.cx/afl/QuickStartGuide.txt). To make the fuzzing more effective, you should provide AFL with all of the test files you have created in its input directory. Let the fuzzer run for at least two hours, and then examine the test cases (in the queue directory) and crashes/hangs (in the crashes and hangs directories).

Add the generated test cases (but not the crashes or hangs) to your test suite, and produce a new coverage report. You should see that the tests generated by the fuzzer reach more parts of the gift card program.

Finally, pick two crashes/hangs and fix the bugs in the program that cause them. Then, re-run the program on all the crashes and hangs found by AFL. You should find that not all of the crashes found by AFL originally crash the program now---although AFL tries its best to figure out which crashes are caused by unique bugs, if often overcounts.

Add the generated tests to your repository and have Travis run them. Note that depending on how long you ran the fuzzer and how fast your machine is, there may be a lot of redundant test cases! To keep only the ones that exercise new behavior in your program, you can use the afl-tmin tool.

To complete the assignment, commit your updated code, your handwritten tests, the fuzzer-generated tests, and a brief writeup explaining the bugs you found and fixed in this part.

Total points: 100

Part 1 is worth 20 points:

* 10 points for signed commits
* 10 points for Travis configuration

Part 2 is worth 40 points:

* 10 points for your test cases and fixes
* 20 points for the bug writeup
* 10 points for Travis regression testing

Part 3 is worth 40 points:

* 10 points for handwritten tests
* 10 points for fuzzer-generated tests
* 10 points for your code fixes
* 10 points for writeup
* Part 1
  + Your .travis.yml
  + At least one signed commit
* Part 2
  + A directory named part2 that contains crash1.gft, crash2.gft, hang.gft, and bugs.txt
  + An updated .travis.yml that runs your tests
  + A commit with the fixed version of the code (if you like, this commit can also contain the files mentioned above)
* Part 3
  + A directory named part3 that contains cov1.gft, cov2.gft, fuzzer1.gft, fuzzer2.gft, and writeup.txt
  + An updated .travis.yml that runs the new tests
  + A commit with the fixed version of the code (if you like, this commit can also contain the files mentioned above)