Assignment #1

Release Date: Beginning of Week 1
Due Date: End of Week 3
TA emails:
Office Hours: Mon-Fri, 10AM-4PM

Introduction

The goal of this assignment is to gain hands on experience with secure coding and application development techniques. This assignment is broken into two parts. In part one you are tasked with developing a spell checker system in C. During part two you will write tests for and fuzz your developed program, likely discovering bugs. These bugs should then be fixed. Reports will be due for each part of this project. This project will apply the secure coding techniques you have learned in class, including the use of version control systems (VCS) and source code management, code testing, fuzzing, and other aspects of secure development.

Deliverables & Grading

This assignment requires you to submit both code and a write-up. The code must be submitted to gradescope. We require that the code you write be submitted in a file called **`spell.c`**. You should not submit a main function, though you should write a main function for yourself to test your functionality.

Reports, test cases, and fuzzing output should be submitted in a zip file through NYU classes. The reports should be submitted with the file name **`lname_fname_report1.pdf`** or **`lname_fname_report1.txt`**. This report should include a link to your repository so the graders can go through and check things like commit signing, commit messages, successful Travis integration, etc.

As the filenames above suggest, all report submissions should be in either PDF format or text format.

The grading split is as follows:

1. Program Correctness: 20 pts.
2. Tests and Fuzzing: 30 pts.
3. Reports: 50 pts.

Total 100 pts.

Week 1: Setup

Completion time – 1 to 2 hours

In order to complete this assignment, you are required to use the git VCS. Before beginning to write your code, you should first install git and set up a git repository to store your code in. The git binaries can be installed by your local package manager or at https://git-scm.com/downloads. For a cheat-sheet of git commands, please see https://github.com/nyutandononline/CLI-Cheat-Sheet/blob/master/git-commands.md. Although we will not be checking your commit messages or git log, it is recommended that you write descriptive commit messages so that the evolution of your repository is easy to understand. For a guide to writing good commit messages, please read https://chris.beams.io/posts/git-commit/ and https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/SubmittingPatches#n104 (lines 104 – 160).

After git is installed, you will want to configure your git user to sign your commits. This can be done by following the guide at https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

After installing and configuring git, you should then create a GitHub account at https://github.com, and add an ssh key using the guide found at https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account.

After creating a GitHub account, you are ready to create a git repository for the project you will be working on. The recommended way to do this is to create a repository using the GitHub Web interface. This is recommended because it can automatically generate important files for you, such as the license for your project and the gitignore file to avoid unintentionally tracking unnecessary files. After you create the repository on GitHub, you can clone it to your local machine using the following command.

```
$ git clone git@github.com:<your_username>/<your_repository_name>.git
```

The next step is to set up Travis CI. To do this, go to https://travis-ci.org and authorize Travis to link with your GitHub account. After this, you should create a `.travis.yml` file in the root of your repository. For information on how to set up a `.travis.yml` file, you can read the Travis tutorial (https://docs.travis-ci.com/user/tutorial/) and the guide to using Travis with C and C++ (https://docs.travis-ci.com/user/languages/c/).

Finally, you are required to set up Travis CI. This will take the form of creating a .travis.yml file which will contain the settings for Travis. More information about the .travis.yaml file for C projects can be found at https://docs.travis-ci.com/user/languages/c/. In Week 3 you will be expanding the functionality of Travis by including tests. By the end of Week 3, you should also include your Travis status button, as demonstrated at https://docs.travis-ci.com/user/status-

images/.

This assignment requires you to use a Hash map. For this project, we require that you use the Hash map hosted at the GitHub linked below. You should not modify this Hash map and you do not need to submit these files when you submit your program.

Hash map: https://github.com/kcg295/AppSecAssignment1

Week 2

Completion time – 3 to 10 hours depending on experience.

In part one of this project, you are tasked with writing a spell checker system in C. Your spell checker is expected to take as arguments a text file to check and dictionary file (which we provide, see the setup section above). It should be run as follows (this is just an example):

```
$ ./spell_check a_tale_of_two_cities.txt wordlist.txt
```

Your function to check the correctness of the words provided should be based on hash maps. When reading the dictionary file, you should populate a hash map with the words from the dictionary. Whenever you are checking if a word is spelled correctly, you can check to see if there exists an entry in the hash map.

Your program should contain at least three functions, a function to load the list of words into the hash map (called load_dictionary), a function to check if a word is correctly spelled (called check_word), and a function called check_words to tie them all together. The parameter to check_words will be a file pointer containing lines of words separated by spaces, punctuation, etc. The function prototypes are as follows[1]:

```
int check_words(FILE* fp, hashmap_t hashtable[], char* misspelled[]);

bool load_dictionary(const char* dictionary_file, hashmap_t
hashtable[]);

bool check_word(const char* word, hashmap_t hashtable[]);
```

These function prototypes are provided in the file `dictionary.h` provided to you. All of the code you write must be in a file called **`spell.c`**. This is required for the autograder to correctly grade your code. You should also create a main function to tie it all together, but this does not need to be submitted as part of the assignment.

Note: the C string dictionary_file is expected to be a filename. You must open this file and

1 We provide the function prototypes so we can use automatic graders. These function prototypes are required.

handle the input from it.

In addition to the development of this program, you are required to install git and create a GitHub account. Information on installing git can be found in the setup section. After creating a new repository, you should clone the new repository to your local machine.

In addition to these requirements and recommendations, you will need a dictionary of correctly spelled English words. We require that you use the list of words used by aspell. To get this list of words, first install aspell and the aspell-en word list using your distribution's package manager. Then generate the word list using the following command.

```
$ aspell dump master > wordlist.txt
```

Using a single word list allows us to uniformly grade all projects on equal footing, and ensures that nobody gets a poor grade because of a poor choice of dictionary file.

After finishing your program, you should analyze it using valgrind. You can download Valgrind from your distribution's package manager, or from valgrind.org. Be sure to fix any bugs that you discover from the output of Valgrind.

After you complete the program, you must add the following to your write-up.

- Your repository link
- exactly how the program works
- The output of Valgrind
- what bugs you expect may exist in your code
- why those bugs might occur
- what steps you took to mitigate or fix those bugs.

Week 3

Completion time – 3 to 5 hours depending on experience.

In part two of this assignment you are tasked with expanding the tests for the code you developed in part one. This will take two major forms, writing and running new tests and fuzzing your program.

When testing your program, be sure to attempt to get good code coverage, and tests for both common cases and potential edge cases your functions may encounter. For example, when a spell checking system is called on a string of characters that are all numeric, the spell checking system should determine that this "word" is spelled correctly. This, and other cases like this, are good things to ensure using tests. Don't forgtet to integrate these tests with Travis by editing your `.travis.yml` file and running the tests in the "script" section. Please also place a Travis status image on your project's README file. Instructions to add the button can be found at https://docs.travis-ci.com/user/status-images/.

After a build is completed, tests should be run to ensure that changes that were made did not break currently existing tests (though this should be covered by your use of Travis). In addition, whenever a bug is found in the code, a test should be written to ensure that similar cases do not show up by changes introduced in the future.

With C code, one of the ways to integrate your tests with Travis is by compiling and running your tests in your `Makefile` and including “make test” in the script portion of your `.travis.yml` file.

However, writing tests is not enough. It is impossible to determine if the tests written cover every execution path of a program, and it is overwhelmingly likely that they do not. For this reason, other testing methods must also be used. After finishing writing and running your tests, you are tasked with running the AFL fuzzer on your program. You can find the installation instructions at http://lcamtuf.coredump.cx/afl/. Information on how to use AFL can be found on their website. After the fuzzer finishes running, you should identify all causes of any bugs it reports and fix these discovered bugs. You should then write a new test to cover the case that caused that bug to trigger.

In addition to testing and fuzzing your code, you should add the following to your report.

- Your repository link.
- all of the bugs found by your tests
- all of the bugs found by fuzzing
- how the bugs were fixed
- how similar bugs can be avoided in the future.

This report should be submitted with your fuzzing output to NYU classes. The `spell.c` file should be submitted to Gradescope.

Extra Credit

Extra credit may be granted by the graders for assignments that go above and beyond the requirements specified in this document.

Hints

Make sure your functions match the prototypes provided exactly. Ideally you want your build to succeed and pass all of your tests.

Late Policy

Late assignments will not be accepted.