Nicolas Gomez
Professor: Kevin Gallagher
Application Security
September 22, 2019

Unit 1

In this assignment we will gain hands on experience on using tools like Github, Travis-CI, Valgrind, and AFL Fuzzer to learn how secure coding and application development techniques that we learn during class are implemented.

Week 1

Version Control Systems or VCS are extremely important when developing an application since it will help track any modifications made to the source code of your application. By managing the source code changes, we can limit who is allowed to submit changes to the repositories, it will give us a summary of the changes made, allow for code reviews, and try to prevent changes that might break the application. For our application development, we will be using Github and Git as our Version Control System to submit changes to our code.

To start, I made sure Git was installed on my virtual machine by running the "git" command and I did not receive the message: "This command was not found." After that I went to the GitHub page, created an account, and then created the following repository: https://github.com/ng1968/Application-Security. I proceeded to create a couple of GPG keys by following Gits documentation on Signing Your Work and added them to my Github profile. The use of GPG keys with Git take advantage of Public Key algorithms to ensure that the changes submitted to the repository came in from a trusted source. Once that was completed, I proceeded to add an SSH key to my GitHub profile to authenticate my machine so that I would not have to enter my username and password every time I needed to clone my repository.

The next step of setting up our environments for the application development is to setup Travic-CL and authorize it to link my Github account and the repository I will be working on. In my repository, I added the ".travis.yml" file necessary to tell Travis what it needs to do to build the application and run any other steps needed like installing dependencies and running tests.

Week 2

Now that I have the environment setup to save our changes in GitHub and run tests, I can move forward with the development of our application. The application that I will be working on is a spell checker that will take in two arguments, the first argument will be the file to check and the second argument is the dictionary file to create a hash table. This program will be written in C and will reference many files from the professors repository where the definitions of the functions are and where test files to run against the program are located. The main part that I will be working on is the "spell.c" file which has the code for how the functions will execute. In this file there are three functions designed to load a list of words from a dictionary into a hash table for fast reading of date, a function to check if an individual word is in the hash table, and a function that will go through a text file and see if each work in that text is properly spelled by running the function previously mentioned. Below is an in depth description on how each function operates:

1. **load_dictionary function:**
    a. Function definition: **bool load_dictionary(const char* dictionary_file, hashmap_t hashtable[]);**

This function takes two arguments, the first one is a pointer to a string containing the path to the dictionary file, the second argument is the hashtable to be populated. This function also returns a boolean variable that determines whether it is true or false that the dictionary was loaded properly. Once the function is called, the hashtable is initialized via "for" loop assigning NULL to every item in the hashtable from index zero to HASH_SIZE which is 2000. This means that the hash table should only hold 2000 words.

Once the hashtable is initialized with all NULLs, the dictionary file pointer is open to only read the file. It checks if the file is empty, if so, it returns false and the function ends. If not, the function will read the file line by line and copy each word up to 45 characters in a buffer variable called "word." For each word in the dictionary the program does the following:

- Each of the words being read has a new line character at the end since the new line character is not part of the word and replaced with a null character.
- Since the function will be adding nodes into the hash table, the function allocates some space in memory for the nodes to be stored in. Each node has a word stored and which one is the next node.
- The hash value of the word is calculated to tell us where in the hash table it should be stored. The function then checks if that current position is already taken or not. If it is not, the function will store the new node containing the word and a NULL next node. If it is occupied, we assign the node in that position next's node as the new node.

After every word in the dictionary file is put into the hash table, the function returns "true" which confirms that the dictionary was loaded properly.

2. **check_word function:**
    a. Function definition: **bool check_word(const char* word, hashmap_t hashtable[]);**

This function takes two arguments, the first one is a pointer to the word we want to check for misspellings and the second argument is the hash table with the dictionary file already loaded. Once the arguments are passed in, the function will check the word for punctuation at the beginning and/or at the end of the word. If there is punctuation at these points, the function removes them by copying only the wanted characters that we also turn into lower case into a temporary variable that its allocated in memory.

The word is then turned into all lower case, the hash value is computed and saved in a variable. A new node is created as a cursor with the values of the node at the position calculated. A while loop will check if the word stored in the cursor node is the same as the one we are checking. If it's the same then the word is properly spelled and the function returns true. If not, then the cursor is assigned the next node of the current one until the function reaches a NULL. If a NULL is reached, the while loop stops and "false" will be returned meaning the word was misspelled.

3. **check_words function:**
    a. Function definition: **int check_words(FILE* fp, hashmap_t hashtable[], char* misspelled[]);**

This function helps put the other two functions above together and drives the flow of the program. The function takes three arguments, the first one is the file pointer of the file it is checking in read only mode, the second argument is the hashtable with the dictionary already loaded, the third argument is a pointer to an array where the function will store all of the misspelled words. This function will also return a number with the count of all misspelled words.

To start this function creates two variables, one for the misspelled word count and one that will act as a buffer for each line being read from the file. A while loop will go through each line in the file and does the following:

- Check if the input line of the file is more than 1024 bytes, if so, it will split it into two strings. The new line characters will be removed from the file and replaced with an end of null character.
- Split the line by the spaces and save the first word in a pointer with a next pointer being the next word after a space until there are no more words.
- A while loop that goes through each word after it has been separated until there are no more words.

Inside of the while loop the functions do the following:

- Remove punctuation or numbers from the beginning and/or end of the word and copy only the necessary characters into a temporary variable.
- The function then calls on the check_word function and tests to see if the word was misspelled or not. If the word was not misspelled then the function continues to the next word. If the word was misspelled then the function allocates some memory space for the word and saves it into a variable which is then put in the misspelled array and the counter of misspelled words is updated.

After the function goes through the entire file to be checked, the file pointer is closed and the number of misspelled words is returned.

Valgrind

Once the script was working as I intended, I installed Valgrind into my machine to see what kind of memory errors my code had. Valgrind is a set of "tools that can automatically detect many memory management and threading bugs, and profile your programs in detail"(Valgrind.org). I ran Valgrind with the --leak-check=full and --track-origins=yes options to get a comprehensive look into the execution of the script. While looking at the output I noticed I was running into a lot of "Conditional jump or move depends on uninitialised value(s)" and "Uninitialised value was created by a heap allocation" errors as well as "blocks are definitely lost in loss" errors.

- Conditional jump or move depends on uninitialised value(s)
  - These errors are due to using variables that have not been initialized in executions like strlen and strcmp. Valgrind highly suggests against using this since the variable has not been initialized, the results of those executions might change depending on the variable assignment.
  - To fix this I made sure to initialize every variable after it's creation.
  - Example: I was calculating the length of temp_word in check_word without anything being copied to it or initialized. To fix this, I copied the word being passed in the arguments to temp_word to initialize it to ensure strlen would execute properly.
- Uninitialised value was created by a heap allocation
  - This error is similar to the above since variables created by malloc were not being explicitly initialized before passing it to functions which could end up causing memory corruption and unexpected results.

    - To fix this I made sure to initialize every variable created by malloc after it's creation.
    - Example: I created the temp_word pointer with malloc but did not initialize it. I made sure to copy the word being passed in the argument to temp_word after it's creating and before passing it to other functions like hash_function.
- Blocks are definitely lost in loss record ….
    - This error is due to memory not being free at the end of execution after being allocated with malloc. As explained in the manual "The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program. Such cases should be fixed by the programmer"(http://valgrind.org/docs/manual/mc-manual.html).
    - I am still trying to find a way of freeing memory without causing other errors.

Valgrind also reported that some memory was indirectly lost meaning that "the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost"(http://valgrind.org/docs/manual/mc-manual.html) and some memory is still reachable meaning "A start-pointer or chain of start-pointers to the block is found"(http://valgrind.org/docs/manual/mc-manual.html). After fixing the "unitialized" errors I ran Valgrind again and received the following output only showing the Heap summary errors.

Note: I am still in the process of fixing memory errors due to not being able to free memory without causing issues during execution.

Week 3

Now that I have a working spell checker with most memory issues taken care of, I moved on to testing the application. There are two main ways I will be testing this application, the first one if by writing my own test in a test_main.c file and by Fuzzing.

Writing my own unit and integration tests

By writing my own test I can make sure most of the code I wrote is working properly. I can write different types of tests to examine specific parts of the program, like functions. A test can be written to make sure the load_function executed properly and so on. These types of tests will check common cases and potential edge cases the functions might encounter. The following tests were created for the application:

- test_dictionary_normal - Unit Test
    - Function tested: load_dictionary();
    - Ensures the dictionary file was loaded properly in the hashtable and certain words are located in certain buckets.
- test_dictitonary_abnormal - Unit Test
    - Function tested: load_dictionary();
    - Checks if passing a non existing file will return false.
- test_dictionary_overflow - Unit Test
    - Function tested: load_dictionary();
    - Checks if words that are too long for dictionary are handled properly.
- test_check_word_normal - Integration Test
    - Function tested: check_word(); and load_dictionary();
    - Ensures hashtable is loaded properly and checks if some words are in the hashtable or not.
- test_check_word_overflow - Integration Test
    - Function tested: check_word(); and load_dictionary();
    - Checks if long words are handled properly and returns false when once is passed in.
- test_check_words_normal - Integration Test
    - Function tested: check_words(); check_word(); and load_dictionary();
    - Ensures the entire program is working fine by checking if the load_dictionary function is working properly, then checking the spelling in a test file and check if the misspelled words are the ones we predicted.
- test_check_words_abnormal - Integration Test
    - Function tested: check_words(); check_word(); and load_dictionary();

    - Ensures the entire program is working fine by checking if the load_dictionary function is working properly, then checking the spelling in a test file and check if the misspelled words are the ones we predicted. In this test, we are using words that have multiple punctuation and numbers at the beginning and/or end of the word.
- test_check_words_overflow - Integration Test
    - Function tested: check_words(); check_word(); and load_dictionary();
    - Ensures the entire program is working fine by checking if the load_dictionary function is working properly, then check if the long word in a test file is handled properly.
- test_check_words_max _misspelled - Integration Test
    - Function tested: check_words(); check_word(); and load_dictionary();
    - Ensures the entire program is working fine by checking if the load_dictionary function is working properly and that the misspelled array does pass the max capacity.

While working on these tests I found a couple of bugs that I needed to fix. For example, the application would only remove one character of punctuation at the beginning or end or the word. I fix this so that it would also remove numbers as well as punctuation from the beginning and/or end of the word. This will also take care of removing multiple punctuation or numbers by checking where the first and last alphabetic character is and copy that information into a temporary word to be checked.

I was also experiencing some issues when testing for an overflow when testing check_words since lines that were too long were not being handled properly and I kept getting segmentation faults. It turned out that when I copied the word after it was trimmed of punctuation or numbers, I was not adding a '\0' or terminator character in the string to properly signify when the string ended.

Another requirement that I had not noticed I was missing, was that there should not be more than 1000 misspelled words in the misspelled array. I changed the while loop that reads the words after they are split by spaces and added a check to make sure the program ends if the number of misspelled words maxes out.

Fuzzing test

After creating tests for possible problems I could think of, I moved on to Fuzzing. Fuzzing is an automated way of testing software that makes it easier to find unexpected issues and hard to find bugs by providing random, invalid, or unexpected data into your program. For this part of the assignment I used AFL-Fuzzer to test my spell checker application and find other issues I have not tested for. I downloaded the compressed file from their site and installed it in

my machine, I then ran some tests again on my application and received output showing some errors.

> **Command ran:** fl-fuzzer -i {path to afl}/test_cases -o afl_out/ ./spell_check @@ @@
> **AFL-Fuzzer output:** https://github.com/ng1968/Application-Security/tree/master/afl_out

In the output, I found that my application was not able to handle the reading of non human readable files. For example, the fuzzer tried to pass multiple files that contained binary information that made my program crash. I could fix this by ensuring that binary data is processed properly by skipping this information. Since some binary files also contain human readable words, it would be helpful to be able to spell check those words while skipping all other characters that are not alphabetic. I would also have to add the "b" mode for binary files to be able to read the information data properly. Since right now, my code is trimming characters that are not alphabetic at the beginning and/or end of the word, input that is all non-alphabetic will cause issues since the program will remove all of the characters in the word which should be skipped from spell check.

To finalize this project for now, I added a status image to my GitHub repository to show if the build and test have been completed properly. To do this, I followed the instructions on https://docs.travis-ci.com/user/status-images/ and added the markdown text the image in the README.md file. When the build and test complete successfully, GitHub will show the status as the following.

Resources

- AFL-Fuzzer: http://lcamtuf.coredump.cx/afl/
- AFL-Fuzzer output: https://github.com/ng1968/Application-Security/tree/master/afl_out
- Assignment Guidelines: https://drive.google.com/open?id=1ywZOL_hvGhvqLkC9eFYpaXTSHaoOcUhJ
- Github Repository: https://github.com/ng1968/Application-Security
- Git Signing Your Work: https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work
- Github Adding a new ssh key: https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account
- Github Hashmap: https://github.com/kcg295/AppSecAssignment1
- How to use strtok: https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm
- Travis-CL: https://travis-ci.org
- Travis-CL C Guide: https://docs.travis-ci.com/user/languages/c/
- Travis-CL Tutorial: https://docs.travis-ci.com/user/tutorial/
- Valgrind: http://www.valgrind.org/
- Valgrind manual: http://valgrind.org/docs/manual/mc-manual.html
- Valgrind output 1: https://drive.google.com/open?id=1jFHh8guzQjzj84IaHXjffScEY03kJ9jI
- Valgrind output 2: https://drive.google.com/open?id=1pRJockSF2zLiO9hYoDO2AzXiQRHry5Ud