**Michael Tse**

**10/22/19**

**Application Security – Assignment 2 – Fall 2019**

**Github Repository -** <https://github.com/mt1836/assignment1_week2>

**INTRODUCTION**

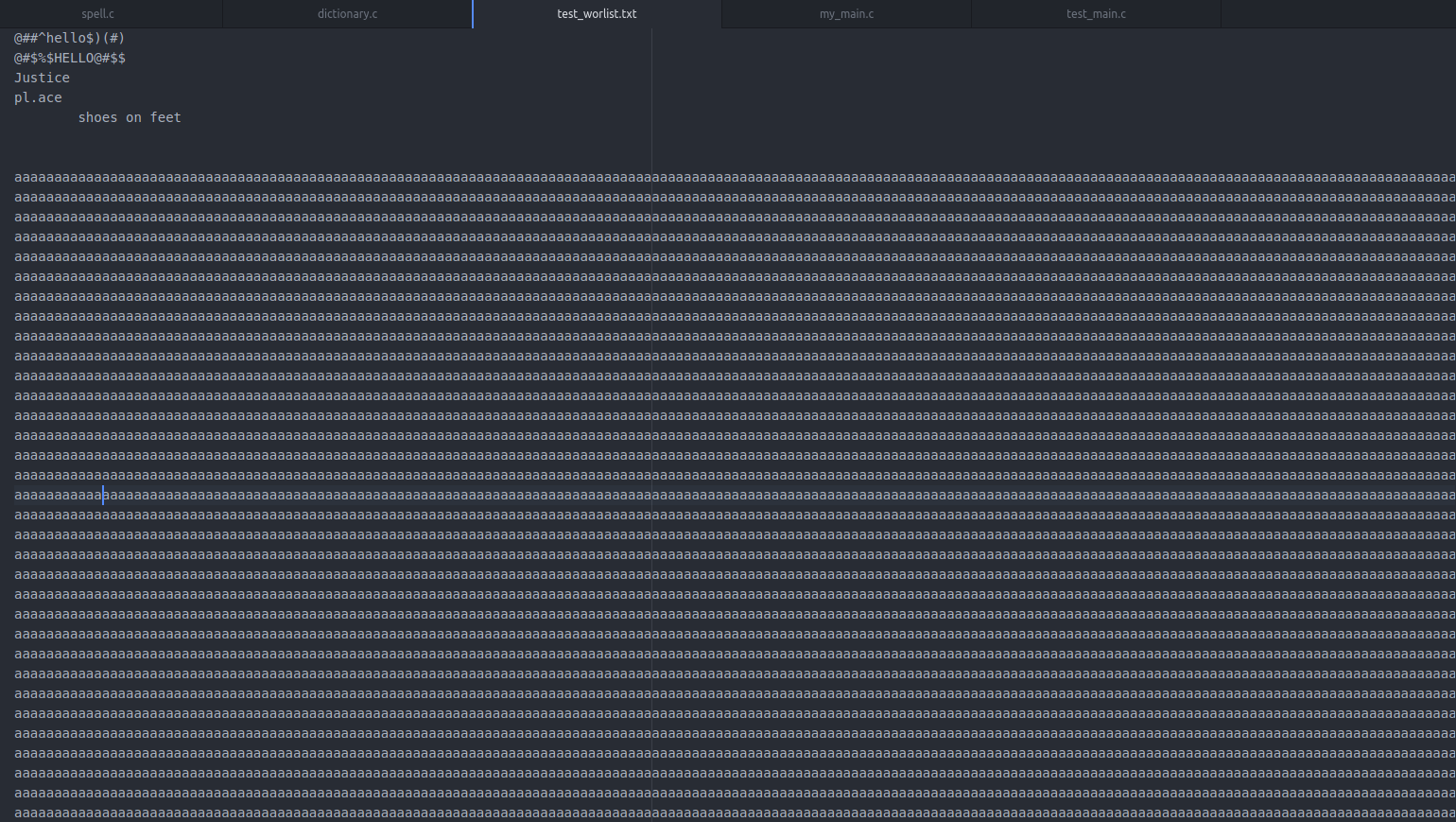
In this assignment we were asked to take the spell checker created in Assignment 1 and turn it into a web application. This web app would be run on localhost:5000 and would store all necessary data in memory. There is no database in scope for this assignment. Once the web app is functioning, we create unit tests for functionality and identify security vulnerabilities and how they can be addressed.

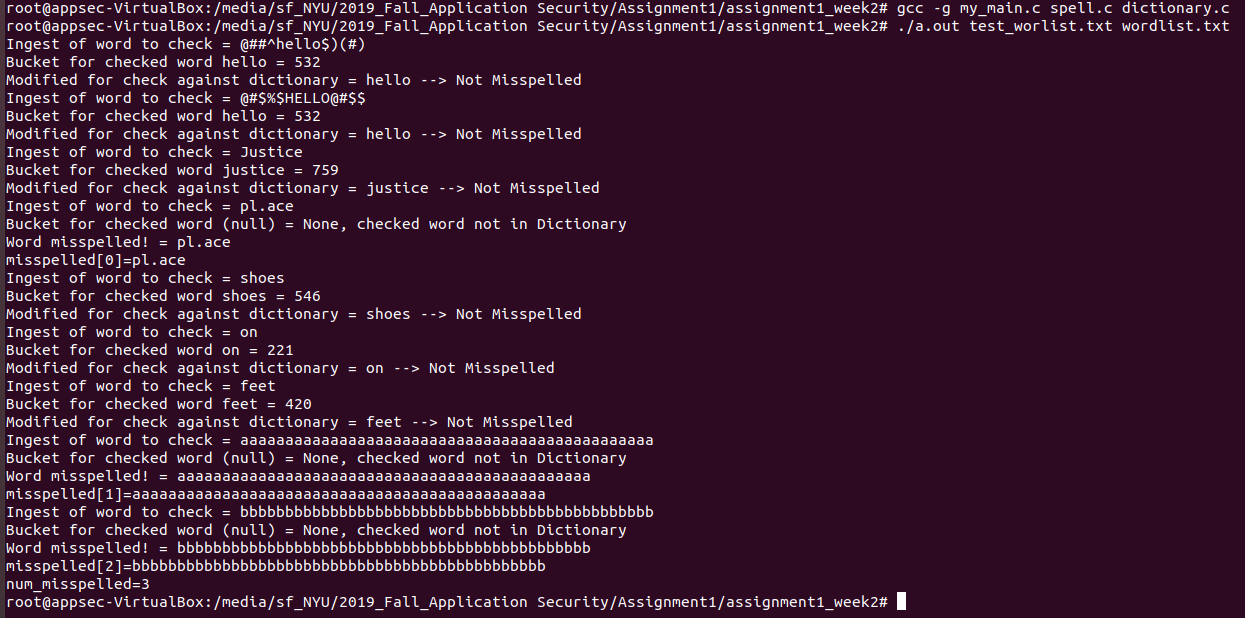
**WEB APPLICATION CREATION**

The web app was created using a combination of python for the logic, flask for the web application framework, wtf for form management and jinja for html templating. The following three html pages were created to support the web app:

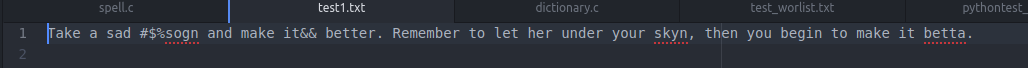
* Register.html: registers new users for the spell check service using 3 fields (username – required, password – required, phone number – optional). Username/password requirements are between 1-20 characters with phone number as optional. If supplied this will be used to mimic two-factor authentication.
* Login.html: registered users are able to login via the login.html page to gain access to the spell check web application.
* Spell\_check.html: once access is granted via the login page, users are able to enter a string of text in a form field and submit the data to the web application. The web app stores the input in both a string and a new text file and reflects it to the user along with the words that are misspelled. The text file is used as input when the spell check binary is called by a method called subprocess.check\_output.

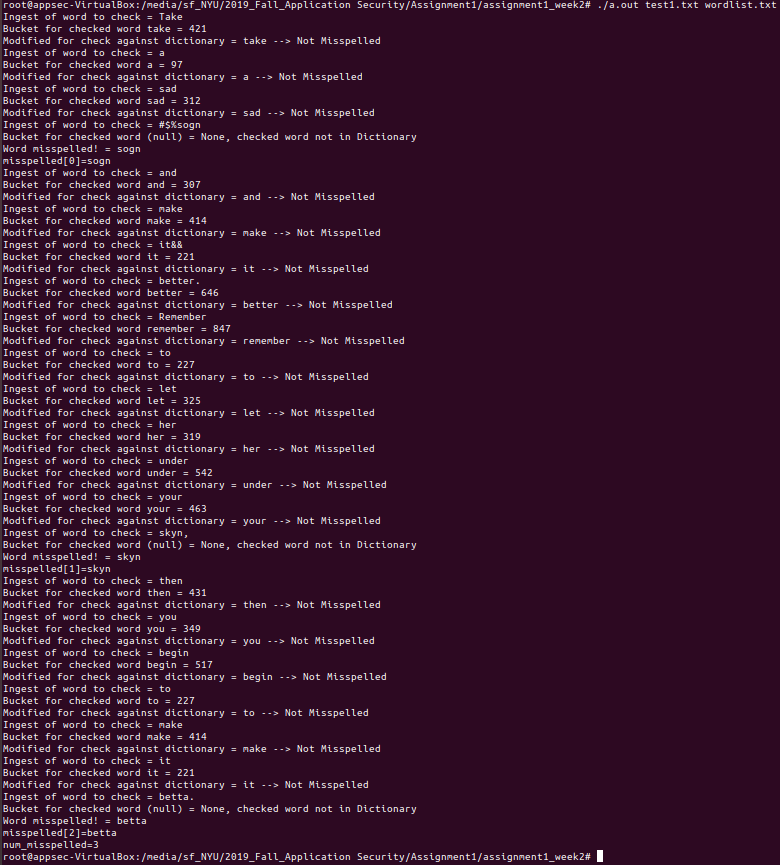
**UNIT TESTING**

The intent for this project was to unit test via the check framework. I had difficulties getting my spell.c to work properly with the tests defined in test\_main.c utilizing the check framework as the frameworks tests called the check\_word function directly. Because of this, my results would fail as I have my lower\_case and remove\_punctuation functions called by check\_words. If check\_word is called directly the ingest would still have capitals and punctuation and never be found as correctly spelled in the hashtable. To address this I spent a significant amount of time (40 hours) moving my lower\_case and remove\_punctuation functions inside check\_word. The new spell.c program worked fine locally for all test cases but I would fail gradescope and I continued to fail test\_main.c. For these reasons, I decided to revert to my original spell.c program (with lower\_case and remove\_punctuation called in check\_words) which was passing Gradescope. The consequence of this decision meant that I could not unit test using the test\_main.c which used the check framework. As an alternative to show that my program addresses the basic functionality as well as edge cases, I modified my spell.c to include printf statements that show output providing proof that it functions properly in all tested scenarios. Below is a screenshot of the input I was testing against (testworlist.txt) and the output in the second screenshot. The first four lines in “testworlist.txt” simulate the tests provided by the professor in the test\_main.c file. As you can see from the output in the second screenshot “@##^hello$)(#)” and “@#$%$HELLO@#$$” have been lower cased and had the punctuation removed from the beginning and end of the word hello. The word “hello” was then checked against the hashtable and found and the result returned was the word was not misspelled. Subsequently the word “Justice” was provided to once again test the lower\_case function was working properly. It ingested “Justice” and checked “justice” against the hashtable and returned the result that it was spelled correctly. The word “pl.ace” was then ingested. No lowercase or punctuation changes were made so “pl.ace” was checked against the hashtable and was not found. “pl.ace” was determined to be misspelled and placed in array position 0 of the misspelled array. Lines 5-7 of “testworlist.txt” was used to test that the program handles tabs and empty lines with carriage returns properly. The spell.c program seems to handle this properly as well, ignoring the tabs and carriage returns and just checking the words “shoes”, “on”, “feet”. All of which were spelled correctly and not entered into the misspelled array. The final test performed was a buffer overflow test. A python script was created to generate 400,000 “a” characters and 20,000 “b” characters. Spell.c did not crash when checking these words as it checked the input and limited the size of the word it would check to 45 characters (the largest word in the English dictionary). If the first 45 characters were read and there were more characters (checked by using the C isspace() function) it would ignore it and continue reading until isspace() returned true, at which point it would terminate the string with a “\0” and check it against the hashtable. In this instance, it found that both the a’s and b’s were misspelled and put them appropriately in the misspelled array.



The following screenshots show the test1.txt file and the associated output. As expected there were three misspellings “sogn”, “skyn”, “betta” which were added to the misspelled array.





**UNIT TESTING - VALGRIND**

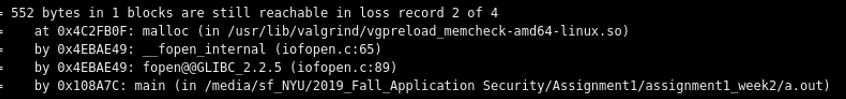
As the functionality of spell.c was demonstrated to work as expected, I then ran the program against Valgrind to check for memory leaks. By using the command line input below I was able to discover the following memory leaks:

Valgrind –tool-memcheck –leak-check=full –show-leak-kinds=all ./a.out testworlist.txt wordlist.txt

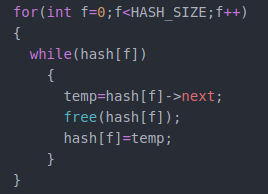
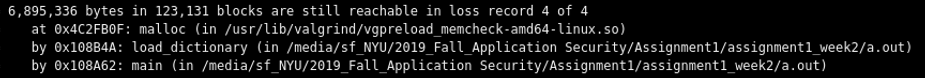
root@appsec-VirtualBox:/media/sf_NYU/201g Fall_Application Security/Assignment1/assignment1 
Memcheck, a memory error detector 
copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al. 
Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info 
Command: 
./a.out testl. txt wordlist.txt 
3066% 
num_misspelled=3 
HEAP SUMMARY: 
in use at exit: 6,896,788 bytes in 123, 152 blocks 
total heap usage: 123, 159 allocs, 7 frees, 6,914,883 bytes allocated 
45 bytes in 1 blocks are definitely lost in loss record 1 of 4 
at Ox4C2FBßF: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64- linux.so) 
week2# 
valg rind 
-tool=memcheck 
- -leak-check—full 
-show-leak-kinds—all 
./a.out testl.txt wordlist. txt 
by ex1ß8CE7: check words (in /media/sf_NYU/201g Fall_Application Security/Assignment1/assignment1_week2/a.out) 
by ex1ß8A92: main (in /media/sf_NYU/2ß19 Fall_Application Security/Assignment1/assignment1_week2/a.out) 
552 bytes in 1 blocks are still reachable in loss record 2 of 4 
at ex4C2FBßF: 
by ex4E8AE49: 
by ex4E8AE49: 
by ex1ß8A7C: 
855 bytes in 19 
at 
by 
by 
6,895, 
at 
by 
by 
ex4C2FBßF : 
Oxlß8E94 : 
ex1ß8A92 : 
336 bytes 
Ox4C2FBßF : 
Oxlß8B4A : 
ex1ß8A62 : 
malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64- Linux. so) 
fopen_internal (iofopen.c:65) 
2.2.5 (iofopen.c:89) 
main (in /media/sf_NYU/2ß19 Fall_Application Security/Assignment1/assignment1_week2/a.out) 
blocks are definitely lost in loss record 3 of 4 
malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64- linux.so) 
check words (in /media/sf_NYU/201g Fall_Application Security/Assignment1/assignment1_week2/a.out) 
main (in /media/sf_NYU/2ß19 Fall_Application Security/Assignment1/assignment1_week2/a.out) 
in 123,131 blocks are still reachable in loss record 4 of 4 
malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64- linux.so) 
load_dictionary (in /media/sf_NYU/201g Fall_Application Security/Assignment1/assignment1_week2/a.out) 
main (in /media/sf_NYU/2ß19 Fall_Application Security/Assignment1/assignment1_week2/a.out) 
LEAK SUMMARY: 
definitely lost: 
indirectly lost: 
possibly lost: 
still reachable: 
supp ressed: 
900 bytes in 20 blocks 
0 bytes in blocks 
0 bytes in blocks 
bytes in 123,132 blocks 
0 bytes in blocks 
For counts of detected and suppressed errors, rerun with: 
ERROR SU%ARY: 2 errors from 2 contexts (suppressed: ß from 0) 
root@appsec-VirtualBox:/media/sf_NYU/201g Fall_Application Security/Assignment1/assignment1_week2# 

There were four specific issues (3 unique) reported by Valgrind:

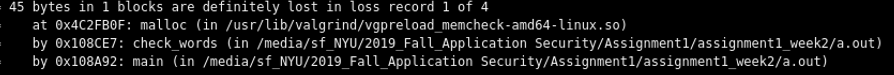
1. This issue was due to the fact that I was closing the input file I was reading from within check\_words when it was opened by my\_main.c. Once I removed the fclose(fp) command out of check\_words and into my\_main.c the valgrind error disappeared.

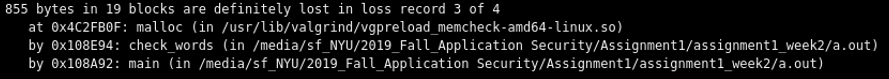


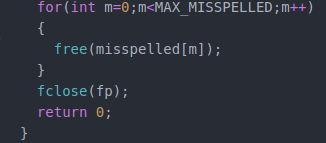
1. This issue was due to the fact that I was allocating memory for my nodes so that I could properly load my dictionary hashtable but I never freed the memory. Once I added the code below to my\_main.c I was able to free all the memory allocated to the nodes used to populate the hashtable.



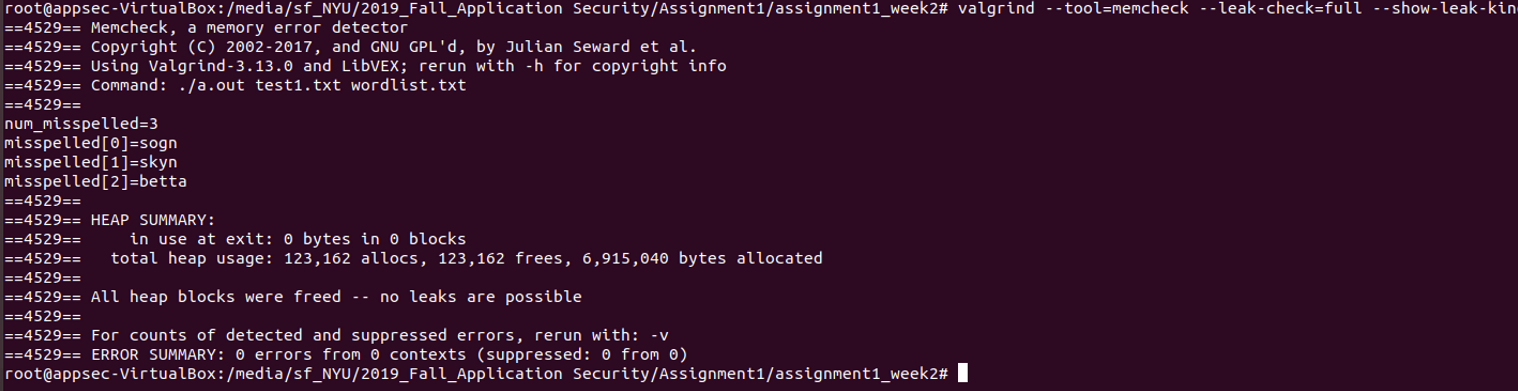
1. The final 2 Valgrind errors were due to allocation of memory for words I was ingesting from the input file. I freed all memory associated with correct spelling in my check\_words function by inserting “free(word);” anytime I found a word that was spelled correctly in check\_words. Since I need to return misspelled array, I cannot free those allocated spaces until I am in my\_main.c which was done with the following code:





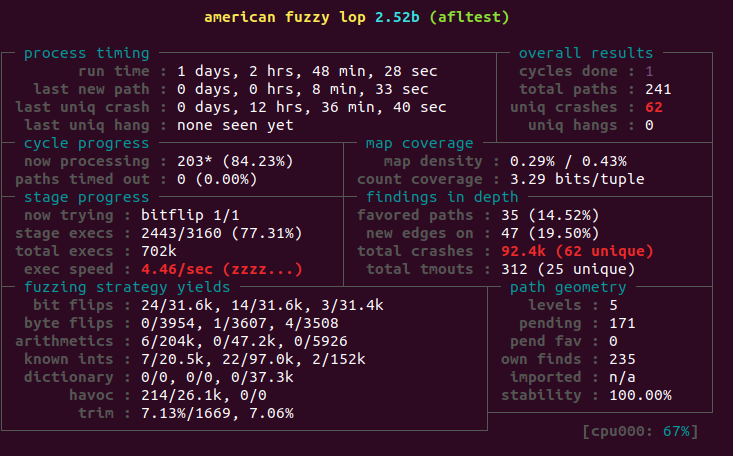


These fixes resulted in a final clean Valgrind output shown below:

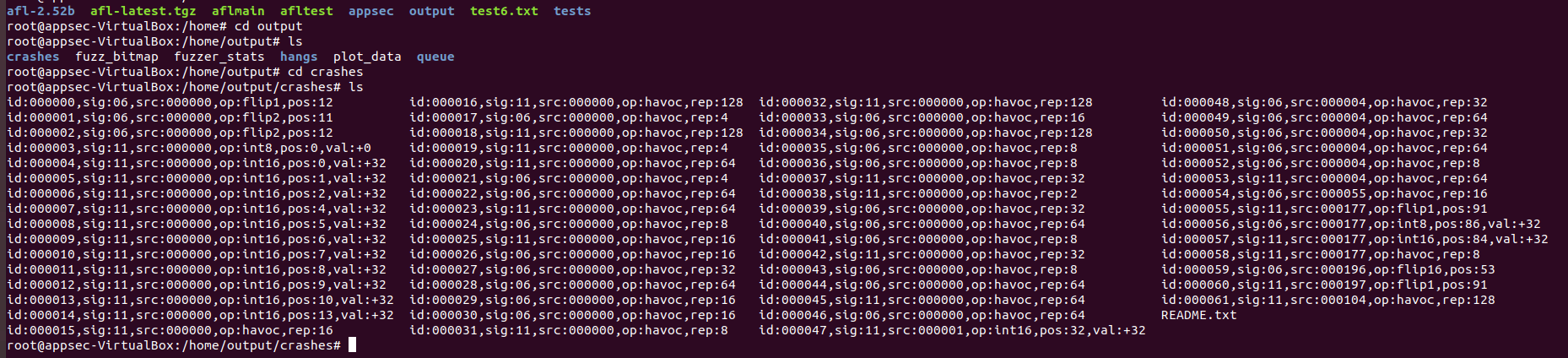


**FUZZ TESTING – AFL**

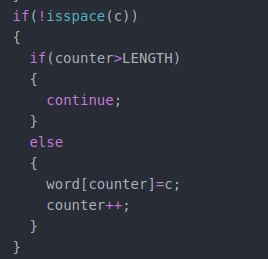
Fuzzing takes the input and mutates it by bit flipping to produce input that takes different paths that a programmer may not have addressed in their code. I used the AFL fuzzer to test my compiled spell checker code. The test ran for 1 day and 2 hours and produced 62 unique crashes as shown in the screenshot below:



The output of the fuzz tests are shown here:



The 63 crashes found in AFL are distributed across two error categories “sig 6 and sig 11”. The sig 6 crashes have an abort code and the sig 11 crashes are a result of a segmentation fault. Upon checking the output files generated by the AFL fuzz tests, they contained non ASCII characters which seem to be the cause of the crashes. I would need to modify the code below to explicitly check for alpha numeric characters through the isalpha() function instead of !isspace(). If it isn’t alpha numeric I would discard and not add it into my word array to be checked by check\_word.



**GIT/GITHUB/TRAVIS CI**

Despite not being able to utilize the tests in test\_main.c with the check framework, I was able to successfully integrate my local git to github and auto build my code through a CI/CD pipeline with Travis CI. The makefile found on my github page builds the required object files and compiled files and creates the appropriate dependencies to implement testing through test\_main. As discussed earlier the make test was not functioning for me as the tests called check\_word directly thus circumventing my lower\_case and remove\_punctuation functions. When I integrated those functions into check\_word I was able to get it to pass the tests locally but not through make test or through gradescope.

**TIME SPENT ON ASSIGNMENT**

I spent 90-100 hours on this assignment. I would estimate 5 hours to write the report, 50 hours to build the spell check code and get it to a functioning state locally that passed gradescope and valgrind. I spend the next 40+ hours figuring out why my code would not pass tests in test\_main through the make test command, then integrating the lower\_case and remove\_punctuation functions into check\_words and troubleshooting why my gradescope no longer passed afterwards. In the end I reverted to the original code and mimicked the test\_main tests using my\_main.c. This was a great assignment, I truly learned a lot and enjoyed the github and Travis CI integration experience as well. I just wish it didn’t take me nearly as long to troubleshoot gradescope and check framework when I had the code working locally. I don’t feel that provided me much benefit and would preferred to have used those 40+ hours to develop additional unit test cases and work more with AFL.