CS-GY9163: Assignment 1

Author: Murtaza Munaim <mm10118>
Date: September 29th, 2019

Week 2

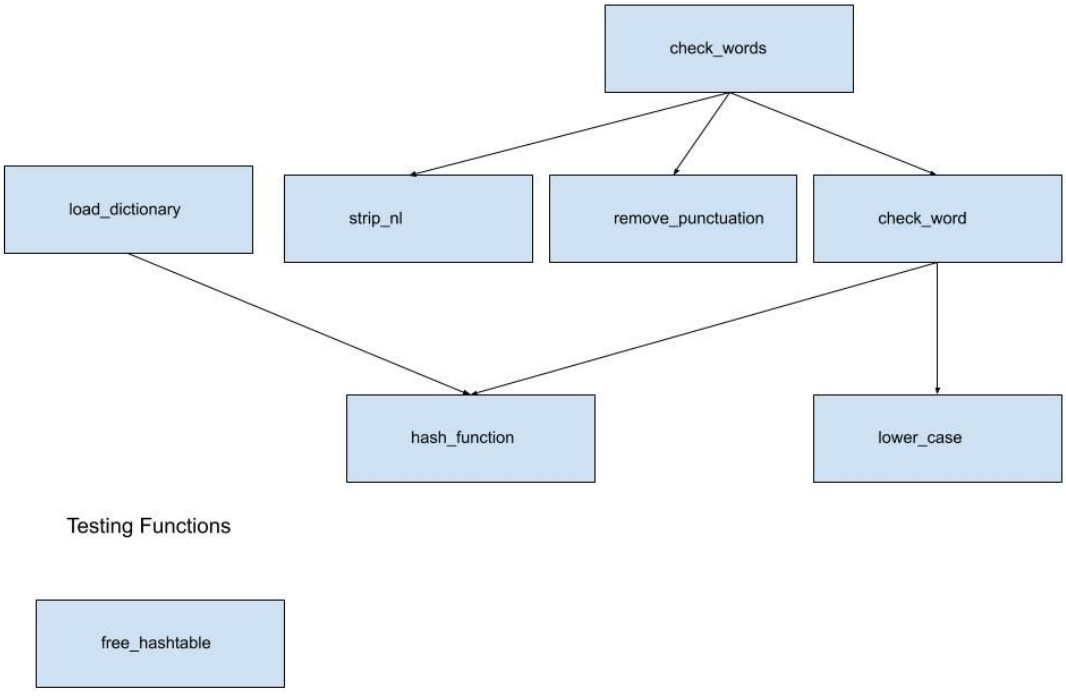
Repository Link:

https://github.com/manizzle/AppSecAssignment1

Program Outline:

The program has 2 main C files, with 1 file being the meatiest, **spell.c,** and helper code in **dictionary.c**

The general flow of the code is as such:



Valgrind Output:

```
==5363== Memcheck, a memory error detector
==5363== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5363== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright
info
==5363== Command: ./spell_check hashmap/test1.txt hashmap/wordlist.txt
==5363==
==5363==
==5363== HEAP SUMMARY:
==5363==     in use at exit: 25,021 bytes in 169 blocks
==5363==   total heap usage: 227,017 allocs, 226,848 frees, 8,476,115 bytes
allocated
==5363==
==5363== LEAK SUMMARY:
==5363==    definitely lost: 0 bytes in 0 blocks
==5363==    indirectly lost: 0 bytes in 0 blocks
==5363==      possibly lost: 8,840 bytes in 7 blocks
==5363==    still reachable: 16,181 bytes in 162 blocks
==5363==         suppressed: 0 bytes in 0 blocks
==5363== Rerun with --leak-check=full to see details of leaked memory
==5363==
==5363== Use --track-origins=yes to see where uninitialised values come
from
==5363== For lists of detected and suppressed errors, rerun with: -s
==5363== ERROR SUMMARY: 95 errors from 4 contexts (suppressed: 398 from 10)
```

What bugs you expect may exist in your code

- Bug #1: For the most part I believe bugs such as heap overflows and stack overflows might exist
    - Reason #1: These bugs will occur because there are many fixed buffers in the code in which code is copied into but the user input comes from file input which is not being explicitly limited in the input specification
    - Mitigation #1: By having specific guards for these functions in the code whenever we copy user input into fixed buffers, we can ensure that we are not overwriting any buffers, heap or stack ones.
- Bug #2: Possible memory leaks exist which are lost by the program

- Reason #2: Many heap allocations are performed within the code base for holding string data from files.
- Mitigation #2: By trying to identify where heap allocations are performed and then subsequently free()'ing these allocations systematically you can try to avoid not cleaning up memory allocations

All of the bugs found by your tests

The test case did not find bugs, the more important thing to see here is that the code coverage report led to writing test cases that exercised all paths of the code base which allowed for better test coverage of code.

```
~/School/appsec/a1/AppSecAssignment1 master xcrun llvm-cov report
./spell_check -instr-profile=testcov.profdata spell.c
Filename                                                 Regions
Missed Regions     Cover   Functions  Missed Functions  Executed
Lines      Missed Lines     Cover
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
/Users/mmunaim/School/appsec/a1/AppSecAssignment1/spell.c         104
0   100.00%           9                 0   100.00%         177
0   100.00%
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
TOTAL                                                             104
0   100.00%           9                 0   100.00%         177
0   100.00%
~/School/appsec/a1/AppSecAssignment1 master
```

All of the bugs found by fuzzing

```
~/School/appsec/a1/AppSecAssignment1 master * git diff
diff --git a/spell.c b/spell.c
index 85c30b3..38721c2 100644
--- a/spell.c
+++ b/spell.c
@@ -119,7 +119,7 @@ bool check_word(const char* word, hashmap_t
hashtable[]) {
```

```
 int strip_nl(char* word) {
        int word_len = strlen(word);
        for (int i = 0; i < word_len; i++) {
-               if (word[i] == '\n') {
+               if (!isprint(word[i])) {
                        word[i] = 0;
                        return i;
                }
```

Bug #1:

- Cause #1: When a word contains bytes which are not alphanumeric, because we assume signed character bytes in the function `int hash_function(const char* word)` , which cannot be changed by the user, it is possible to generate negative indexes for the hashtable. Negative indexing can cause SIGSEGV crashes.
- Fix #1: Do not allow non-printable characters in tokens as the wordlist won't contain this
  - This fix solved all 30 crashes found by AFL
- Fix #2 (Future): Not possible, but the hash function can output only unsigned values.

Fuzzer Output

```
~/School/appsec/a1/AppSecAssignment1 master * ~/Development/afL/afl-whatsup
-s findings/
status check tool for afl-fuzz by <lcamtuf@google.com>

Summary stats
=============

       Fuzzers alive : 4
      Total run time : 1 days, 10 hours
         Total execs : 0 million
    Cumulative speed : 0 execs/sec
       Pending paths : 1 faves, 161 total
  Pending per fuzzer : 0 faves, 40 total (on average)
       Crashes found : 30 locally unique

~/School/appsec/a1/AppSecAssignment1 master *
```