CYB220 Final exam – Fall 2024

Name: Jake Gendreau

Due: Wed, Dec 11th, 2024, 11:59 pm

Turn in: (1) The vulnerable code version (.c or .cpp file) with the good code left in the comments.

(2) The answered exam file (this file with your answers in it).

Points: 100 pts

Objective: In this final exam, you will show what you learned this semester, especially writing more secure code and using static and dynamic analysis tools to find/trigger bugs and vulnerabilities in the program.

Overview:

The exam has four tasks:

(1) (50 pts) Write a c/c++ program based on the requirement.
(2) (15 pts) Use a static analysis tool and a dynamic analysis tool to analyze/test your program.
(3) (20 pts) Inject five vulnerabilities based on the 2024 CWE top 25 software weaknesses.
(4) (15 pts) Use the same static and dynamic analysis tools to analyze/test the vulnerable version of the program to see what they can find.

(50 pts) Task 1: Design and implement a simple program that analyzes user input.

Create a C/C++ program that reads user input and analyzes them.

- (8 pts)The program continues asking for input until the user enters "exit" to terminate the program.
- Please use character arrays (NUL-terminated) to store the user input. The use of the string library is not allowed this time.
- For each round, just read in no more than 999 characters. If the user enters more than 999 characters, cut them off and only keep the first 999 characters. You should add a NUL-terminator at the end of the 999th character.
- After the user enters input, your program should analyze it, modify it, and print out the results one by one automatically (See detailed requirement below).
- Then, ask for another user input until "exit" is received.

In each round, after the user enters input, do these analysis and modifications, and display the results for each of them one by one.

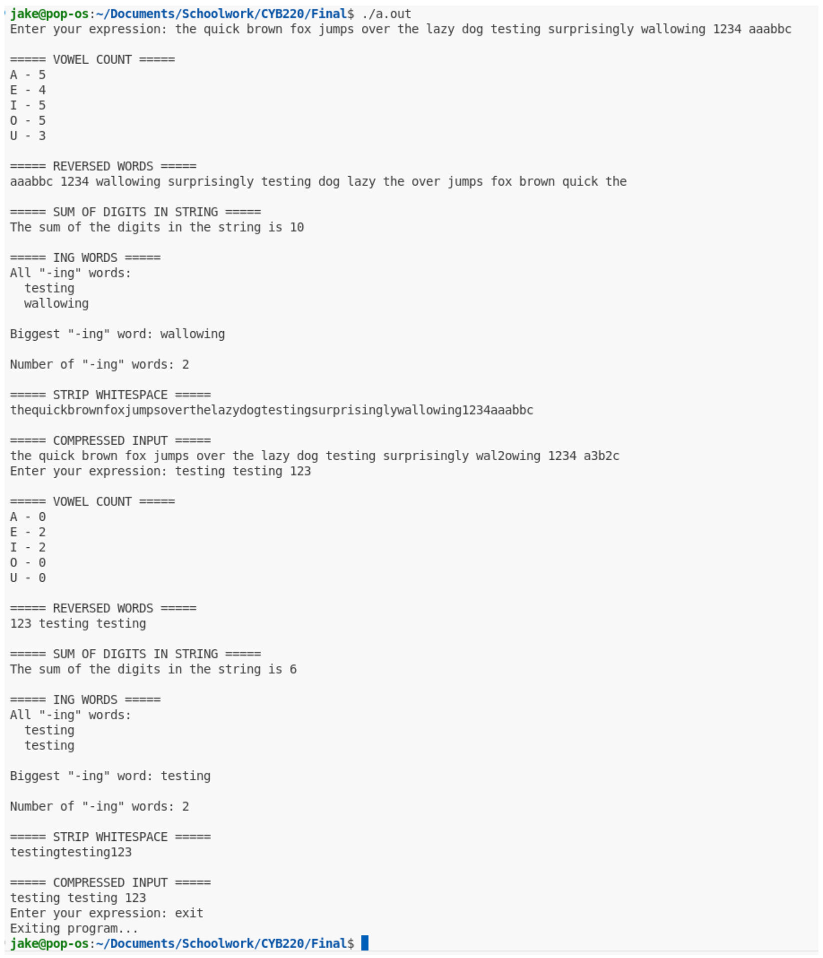
1. (7 pts) Implement a function to count the number of vowels in the user input and print the numbers of each of the vowels separately. Such as a-123, e-328, i-100…
2. (7 pts) Write a function that takes a character array as input, reverses the string, and prints them.
   - E.g., Hey! Are you ready for the winter break? → break? winter the for ready you Are Hey!

3. (7 pts) If the input has one or more numbers in it, print the numbers and the sum of them.
4. (7 pts) Use a pointer to step through the array elements to search for words ending with "ing", print them, and also the count of these -ing words! In addition, print the one with the most characters.
   - Note: only words ending with ing, such as "reading" and "working". The words "surprisingly" and "single" do not count!
   - Print the one with the most characters → If you have "reading", "lying", "sleeping", "surprising", the one word with the most characters is "surprising" (10 chars).

5. (7 pts) Create a function that removes all the white spaces in the input. Print the modified user input.
6. (7 pts) Design a function that compresses the user input by replacing consecutive duplicate characters with a single character followed by its count (For example, "aaabbc" becomes "a3b2c").

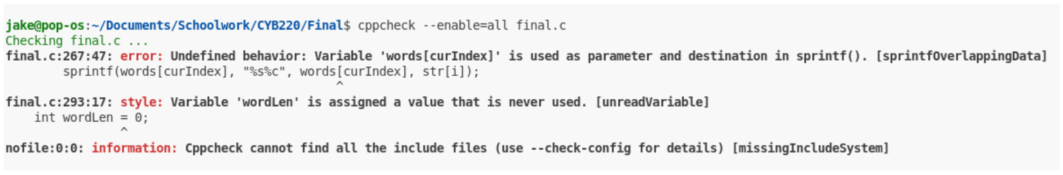
(15 pts) Task 2: After you have your program written:

1. Test your program yourself. Screenshot the output to prove your program is running correctly.



```
jake@pop-os:~/Documents/Schoolwork/CYB220/Final$ ./a.out
Enter your expression: the quick brown fox jumps over the lazy dog testing surprisingly wallowing 1234 aaabbc

===== VOWEL COUNT =====
A - 5
E - 4
I - 5
O - 5
U - 3

===== REVERSED WORDS =====
aaabbc 1234 wallowing surprisingly testing dog lazy the over jumps fox brown quick the

===== SUM OF DIGITS IN STRING =====
The sum of the digits in the string is 10

===== ING WORDS =====
All "-ing" words:
  testing
  wallowing

Biggest "-ing" word: wallowing

Number of "-ing" words: 2

===== STRIP WHITESPACE =====
thequickbrownfoxjumpsoverthelazydogtestingsurprisinglywallowing1234aaabbc

===== COMPRESSED INPUT =====
the quick brown fox jumps over the lazy dog testing surprisingly wal2owing 1234 a3b2c
Enter your expression: testing testing 123

===== VOWEL COUNT =====
A - 0
E - 2
I - 2
O - 0
U - 0

===== REVERSED WORDS =====
123 testing testing

===== SUM OF DIGITS IN STRING =====
The sum of the digits in the string is 6

===== ING WORDS =====
All "-ing" words:
  testing
  testing

Biggest "-ing" word: testing

Number of "-ing" words: 2

===== STRIP WHITESPACE =====
testingtesting123

===== COMPRESSED INPUT =====
testing testing 123
Enter your expression: exit
Exiting program...
jake@pop-os:~/Documents/Schoolwork/CYB220/Final$
```

2. Run the static analysis tool, cppcheck, to see what bugs it can find (if any).



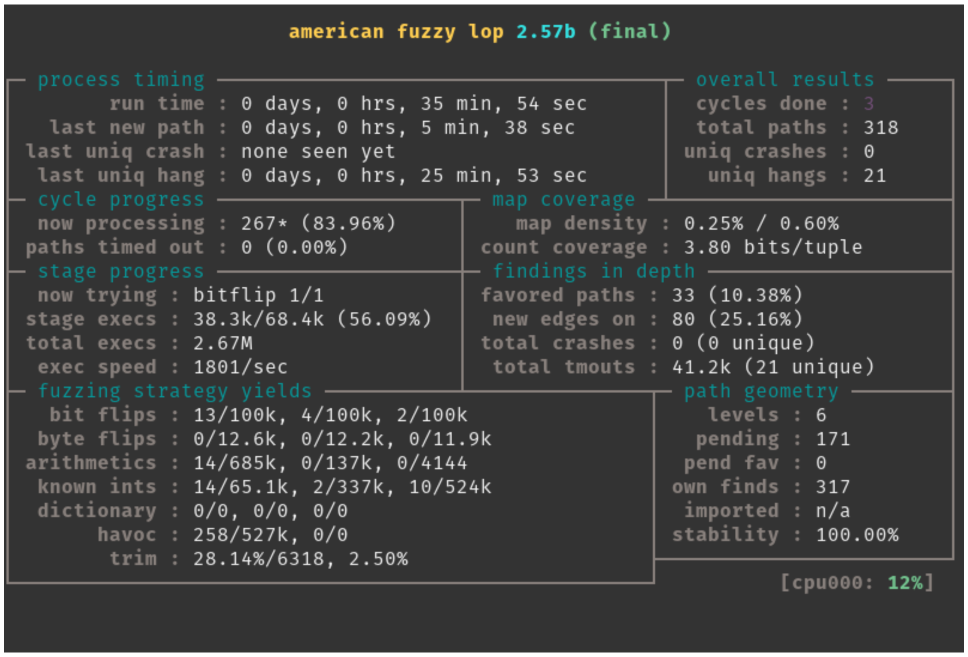
```
jake@pop-os:~/Documents/Schoolwork/CYB220/Final$ cppcheck --enable=all final.c
Checking final.c ...
final.c:267:47: error: Undefined behavior: Variable 'words[curIndex]' is used as parameter and destination in sprintf(). [sprintfOverlappingData]
        sprintf(words[curIndex], "%s%c", words[curIndex], str[i]);
                                              ^
final.c:293:17: style: Variable 'wordLen' is assigned a value that is never used. [unreadVariable]
    int wordLen = 0;
                ^
nofile:0:0: information: Cppcheck cannot find all the include files (use --check-config for details) [missingIncludeSystem]
```

The only error that I have is that my sprintf() function uses one string as both the destination and a source. This is not an issue since I am only appending to the string, so I overwrite the contents of the string with the existing contents.

the style warning is about initializing a variable to an unused value, since it gets reset within the loop. Again, this isn't an issue since it's better to have an initialized, unused value than an uninitialized value.

the last warning is about included files. I do not know how to make it go away.

3. Run the dynamic analysis tool, AFL, to fuzz your program for 30 minutes.



```
                      american fuzzy lop 2.57b (final)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 35 min, 54 sec      │  cycles done : 3      │
│   last new path : 0 days, 0 hrs, 5 min, 38 sec       │  total paths : 318    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 25 min, 53 sec      │   uniq hangs : 21     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 267* (83.96%)     │    map density : 0.25% / 0.60%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.80 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : bitflip 1/1           │ favored paths : 33 (10.38%)           │
│ stage execs : 38.3k/68.4k (56.09%)  │  new edges on : 80 (25.16%)           │
│ total execs : 2.67M                 │ total crashes : 0 (0 unique)          │
│  exec speed : 1801/sec              │  total tmouts : 41.2k (21 unique)     │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 13/100k, 4/100k, 2/100k               │    levels : 6          │
│  byte flips : 0/12.6k, 0/12.2k, 0/11.9k             │   pending : 171        │
│ arithmetics : 14/685k, 0/137k, 0/4144               │  pend fav : 0          │
│  known ints : 14/65.1k, 2/337k, 10/524k             │ own finds : 317        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 258/527k, 0/0                         │ stability : 100.00%    │
│        trim : 28.14%/6318, 2.50%                    ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 12%]
```

I had no crashes using AFL for 35 minutes.

4. If any bugs are found by these tools, make sure you fix them before moving to task 3.

(20 pts) Task 3: Take a look at the 2024 CWE Top 25 Most Dangerous Software Weaknesses (**https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html**)

Identify 5 CWEs from the top 25, and inject them into your code. **4 pts each.**

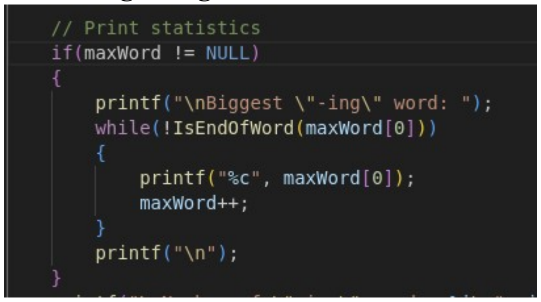
Note: simply comment out your good code and inject the vulnerable code after it (in the same file).

----------- 1. CWE ??? -----------

ID and title: CWE-476: NULL Pointer Dereference

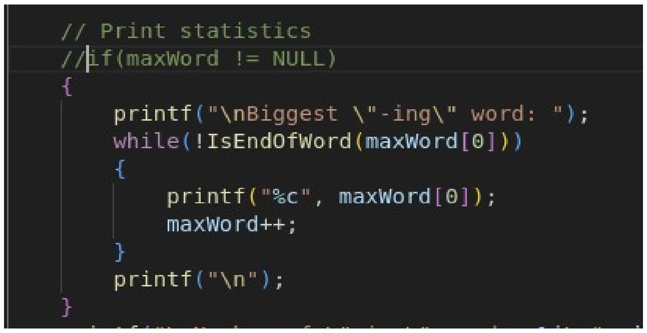
CWE Description: The product dereferences a pointer that it expects to be valid but is NULL.

Your original /good code:



```
// Print statistics
if(maxWord != NULL)
{
    printf("\nBiggest \"-ing\" word: ");
    while(!IsEndOfWord(maxWord[0]))
    {
        printf("%c", maxWord[0]);
        maxWord++;
    }
    printf("\n");
}
```

Your injected vulnerable code:



```
// Print statistics
//if(maxWord != NULL)
{
    printf("\nBiggest \"-ing\" word: ");
    while(!IsEndOfWord(maxWord[0]))
    {
        printf("%c", maxWord[0]);
        maxWord++;
    }
    printf("\n");
}
```

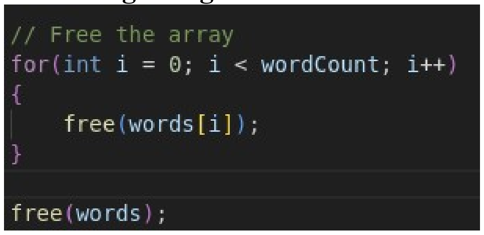
A brief explanation of your vulnerable code: by not checking for a null value of maxWord, it will attempt to dereference maxWord, and cause a crash.

----------- 2. CWE ??? -----------

ID and title: CWE-416: Use After Free

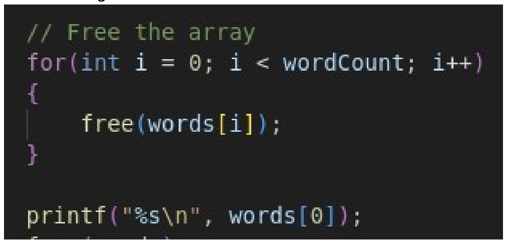
CWE Description: The product reuses or references memory after it has been freed. At some point afterward, the memory may be allocated again and saved in another pointer, while the original pointer references a location somewhere within the new allocation. Any operations using the original pointer are no longer valid because the memory "belongs" to the code that operates on the new pointer.

Your original /good code:



```
// Free the array
for(int i = 0; i < wordCount; i++)
{
    free(words[i]);
}

free(words);
```

Your injected vulnerable code:



```
// Free the array
for(int i = 0; i < wordCount; i++)
{
    free(words[i]);
}

printf("%s\n", words[0]);
```

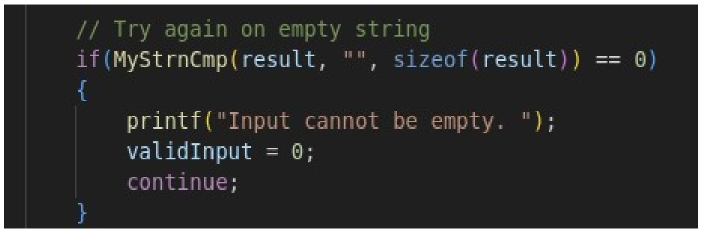
A brief explanation of your vulnerable code: By using words[0] after free, it is referencing memory which is no longer allocated, and may be used by something else. This may cause the program to crash.

----------- 3. CWE ??? -----------

ID and title: CWE-20: Improper Input Validation

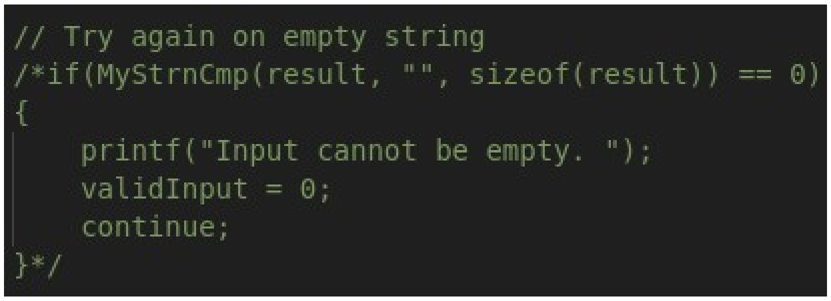
CWE Description: The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.

Your original /good code:



```
    // Try again on empty string
    if(MyStrnCmp(result, "", sizeof(result)) == 0)
    {
        printf("Input cannot be empty. ");
        validInput = 0;
        continue;
    }
```

Your injected vulnerable code:



```
// Try again on empty string
/*if(MyStrnCmp(result, "", sizeof(result)) == 0)
{
    printf("Input cannot be empty. ");
    validInput = 0;
    continue;
}*/
```

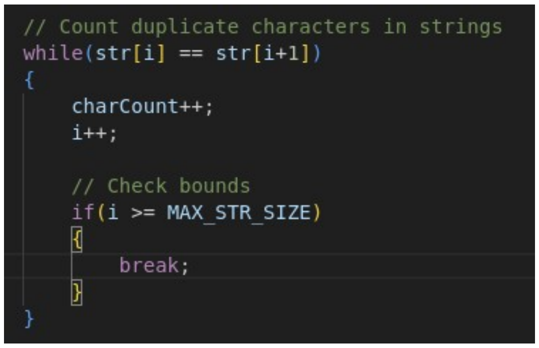
A brief explanation of your vulnerable code: When the possibility for an empty input is present, the processing functions are made unsafe and can cause crashes. By removing the empty checker, the program is made dangerous.

----------- 4. CWE ??? -----------

ID and title: CWE-125: Out-of-bounds Read

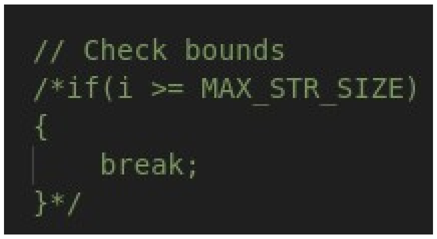
CWE Description: The product reads data past the end, or before the beginning, of the intended buffer.

Your original /good code:



```
// Count duplicate characters in strings
while(str[i] == str[i+1])
{
    charCount++;
    i++;

    // Check bounds
    if(i >= MAX_STR_SIZE)
    {
        break;
    }
}
```

Your injected vulnerable code:



```
// Check bounds
/*if(i >= MAX_STR_SIZE)
{
    break;
}*/
```

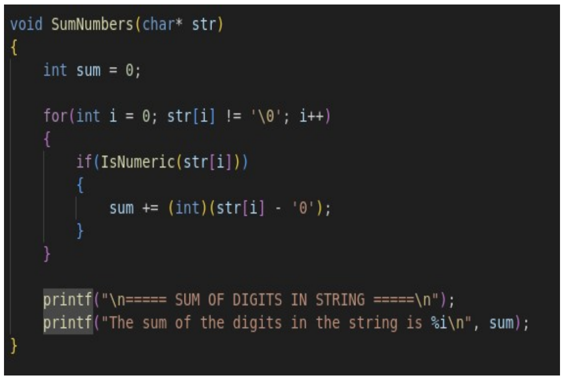
A brief explanation of your vulnerable code: by removing the bounds check, the program can read out of the bounds of the string and can cause a segfault.

----------- 5. CWE ??? -----------

ID and title: CWE-190: Integer Overflow or Wraparound

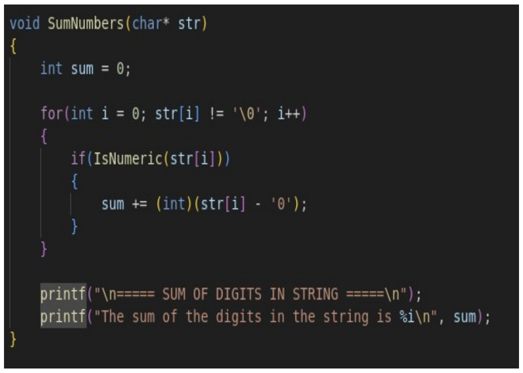
CWE Description: The product performs a calculation that can produce an integer overflow or wraparound when the logic assumes that the resulting value will always be larger than the original value. This occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may become a very small or negative number.

Your original /good code:



```
void SumNumbers(char* str)
{
    int sum = 0;

    for(int i = 0; str[i] != '\0'; i++)
    {
        if(IsNumeric(str[i]))
        {
            sum += (int)(str[i] - '0');
        }
    }

    printf("\n===== SUM OF DIGITS IN STRING =====\n");
    printf("The sum of the digits in the string is %i\n", sum);
}
```

Your injected vulnerable code:



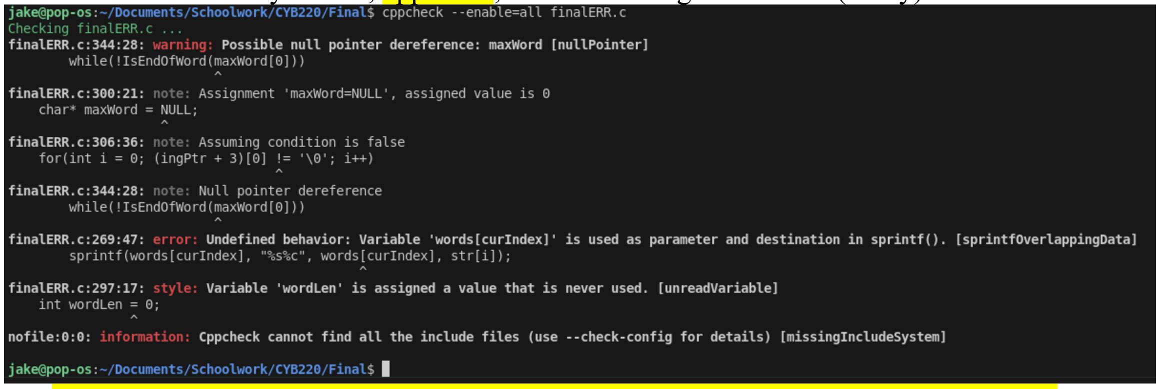
```
void SumNumbers(char* str)
{
    int sum = 0;

    for(int i = 0; str[i] != '\0'; i++)
    {
        if(IsNumeric(str[i]))
        {
            sum += (int)(str[i] - '0');
        }
    }

    printf("\n===== SUM OF DIGITS IN STRING =====\n");
    printf("The sum of the digits in the string is %i\n", sum);
}
```

A brief explanation of your vulnerable code: My code does not check for integer overflow, so it is inherently vulnerable, if the size of the input string were to be changed. At the moment, 999 characters in the maximum. Since it only reads one digit at a time, 9 * 999 = 8991, which is well

within the bounds of an integer. As it stands, the code is not vulnerable, but it could be if something were to change.

(15 pts) Task 4: After you inject the five vulnerable codes into your program:

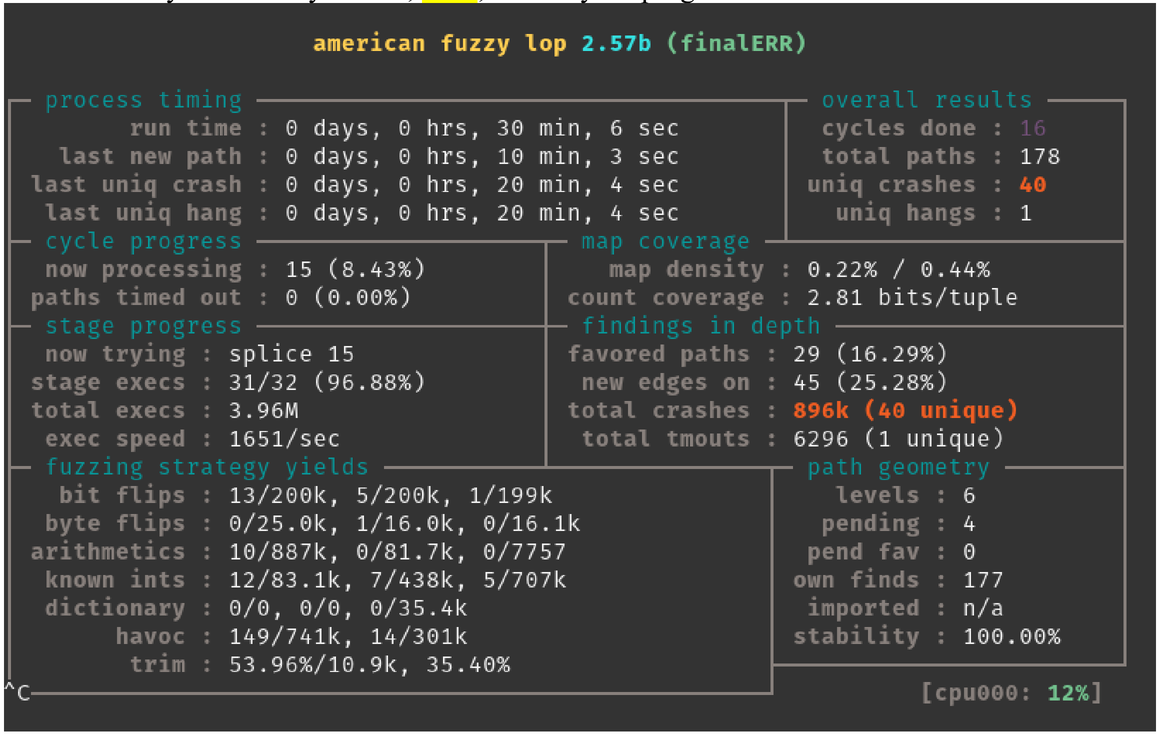
1. Run the static analysis tool, cppcheck, to see what bugs it can find (if any).



```
jake@pop-os:~/Documents/Schoolwork/CYB220/Final$ cppcheck --enable=all finalERR.c
Checking finalERR.c ...
finalERR.c:344:28: warning: Possible null pointer dereference: maxWord [nullPointer]
        while(!IsEndOfWord(maxWord[0]))
                           ^

finalERR.c:300:21: note: Assignment 'maxWord=NULL', assigned value is 0
    char* maxWord = NULL;
                    ^

finalERR.c:306:36: note: Assuming condition is false
    for(int i = 0; (ingPtr + 3)[0] != '\0'; i++)
                                   ^

finalERR.c:344:28: note: Null pointer dereference
        while(!IsEndOfWord(maxWord[0]))
                           ^

finalERR.c:269:47: error: Undefined behavior: Variable 'words[curIndex]' is used as parameter and destination in sprintf(). [sprintfOverlappingData]
        sprintf(words[curIndex], "%s%c", words[curIndex], str[i]);
                                              ^

finalERR.c:297:17: style: Variable 'wordLen' is assigned a value that is never used. [unreadVariable]
    int wordLen = 0;
                ^

nofile:0:0: information: Cppcheck cannot find all the include files (use --check-config for details) [missingIncludeSystem]

jake@pop-os:~/Documents/Schoolwork/CYB220/Final$ 
```

cppcheck was able to find the null dereference, but that is all. This warning is correct.

2. Run the dynamic analysis tool, AFL, to fuzz your program for 30 minutes.



```
                     american fuzzy lop 2.57b (finalERR)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 30 min, 6 sec       │  cycles done : 16     │
│   last new path : 0 days, 0 hrs, 10 min, 3 sec       │  total paths : 178    │
│ last uniq crash : 0 days, 0 hrs, 20 min, 4 sec       │ uniq crashes : 40     │
│  last uniq hang : 0 days, 0 hrs, 20 min, 4 sec       │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 15 (8.43%)        │    map density : 0.22% / 0.44%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.81 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 15             │ favored paths : 29 (16.29%)            │
│ stage execs : 31/32 (96.88%)        │  new edges on : 45 (25.28%)            │
│ total execs : 3.96M                 │ total crashes : 896k (40 unique)       │
│  exec speed : 1651/sec              │  total tmouts : 6296 (1 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 13/200k, 5/200k, 1/199k               │    levels : 6          │
│  byte flips : 0/25.0k, 1/16.0k, 0/16.1k             │   pending : 4          │
│ arithmetics : 10/887k, 0/81.7k, 0/7757              │  pend fav : 0          │
│  known ints : 12/83.1k, 7/438k, 5/707k              │ own finds : 177        │
│  dictionary : 0/0, 0/0, 0/35.4k                     │  imported : n/a        │
│       havoc : 149/741k, 14/301k                     │ stability : 100.00%    │
│        trim : 53.96%/10.9k, 35.40%                  ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 12%]
```

Most of the 40 crashes are due to the null dereference in my FindIngWords() function. If the phrase doesn't include an ing word, the program will crash. Additionally, there is a use after free that will almost always cause an error as well. In every crash report that I checked, one of these two was the cause of the issue.

Now, you have finished all the tasks for the final exam.

Turn in: (1) The vulnerable code version (.c or .cpp file) with the good code left in the comments.

(2) The answered exam file (this file with your screenshots and answers in it).

Make sure you double-check your submission to make sure you upload the correct files.