**USING LIBFUZZER AND AFL TO FUZZ YOUR OWN C/C++ IMPLEMENTATION**

**SUBJECT NAME: CRYPTOGRAPHY AND NETWORK SECURITY**

**SUBJECT CODE: CS6008**

**MODULE: 4**

|  |  |
| --- | --- |
| **NAME** | BHUVANESHWAR S |
| **REG.NO** | 2019103513 |
| **DATE** | 05/04/2022 |

**USING LIBFUZZER AND AFL TO FUZZ YOUR OWN C/C++ IMPLEMENTATION**

**AIM :**

To implement a c/c++ code using fuzzer tools.

**TOOLS INVOLVED:**

* LIBFUZZER
* AFL FUZZER
* VISUAL STUDIO CODE
* KALI LINUX TERMINAL (WSL)
* WINDOWS (OPERATION SYSTEM)

**PROBLEM DESCRIPTION:**

**Fuzzing** is to try massive numbers of random inputs to code in order to trigger a vulnerability. You create a testbench for the code of interest, pair it with fuzzing engine that generates random data, and launch it on some server somewhere. Hours days, or weeks later – if your testbench is solid , it comes back with a set of inputs that cause the code to crash. This process may be accelerated by:

* Using a sanitizer: compiler-supported sanitizers instrument binaries with extra code to check for illegal conditions, such as out-of-bounds memory accesses, that may not cause an immediate crash. This makes the code under test more likely to fail and thus reduces the fuzzer running time.
* Using coverage-driven fuzzing: fuzzers can monitor program states reached under different inputs and guide the inputs and guide the inputs in a way that tends to produce new ones.

**libFuzzer** can be checked out from LLVM’s Subversion repository and built using their directions. You supply a test driver as a function called LLVMFuzzerTestOneInput with C linkage. The result is a standalone program that exercises the code inside that function. It uses some Clang compiler-supplied instrumentation, via the -fsanitize-coverage option, to monitor which paths are exercised, so gcc is not an option. It must be compiled with -fsanitize=memory to ensure no initialization is missed.

AFL is a standalone tool that uses binary rewriting to instrument the code being tested. It supplies wrapper compilers that call either Clang or gcc as necessary. The test driver is written as a main program that takes the random string from standard input, which means each run is a separate process. However, if you use Clang, there is a special “fast” mode that instruments your code as a compiler pass, rather than a final object code rewrite. This means the instrumentation itself can be optimized, producing faster binaries. So clang-fast is used.

Also, AFL is more mature and has more sophisticated mutation algorithms, and though its one-process-per-test approach is slower, the special Clang support compensates. Address sanitizing seems much faster than memory sanitizing, and you can always re-run all the (unique path) test cases afterwards.

**INPUT:**

Getting an input from a files.

**OUTPUT:**

Given input string is to check whether it is a palindrome or not.

**SCREENSHOT:**

**LIBFUZZER**

**FILENAME : libfuzz.cc**

#include <stdint.h>

#include <stddef.h>

bool fuzz(const uint8\_t \*data, size\_t s){

    for(int i = 0 ; i <= s/2 ; i++){

        if(data[i] != data[s-i-1])

            return false;

    }

    return true;

}

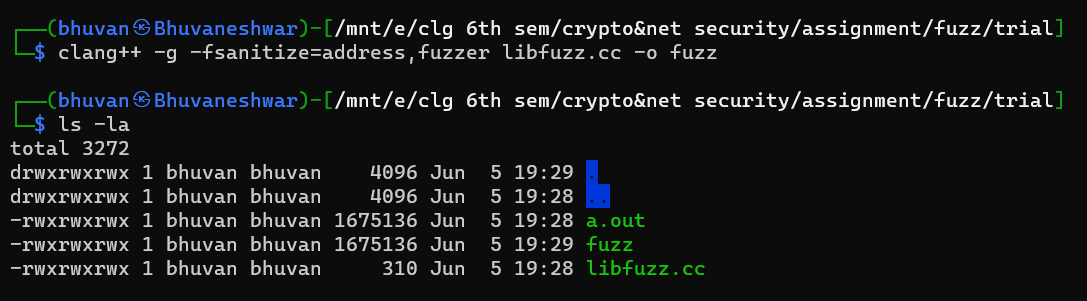
extern "C" int LLVMFuzzerTestOneInput(const uint8\_t \*data, size\_t s){

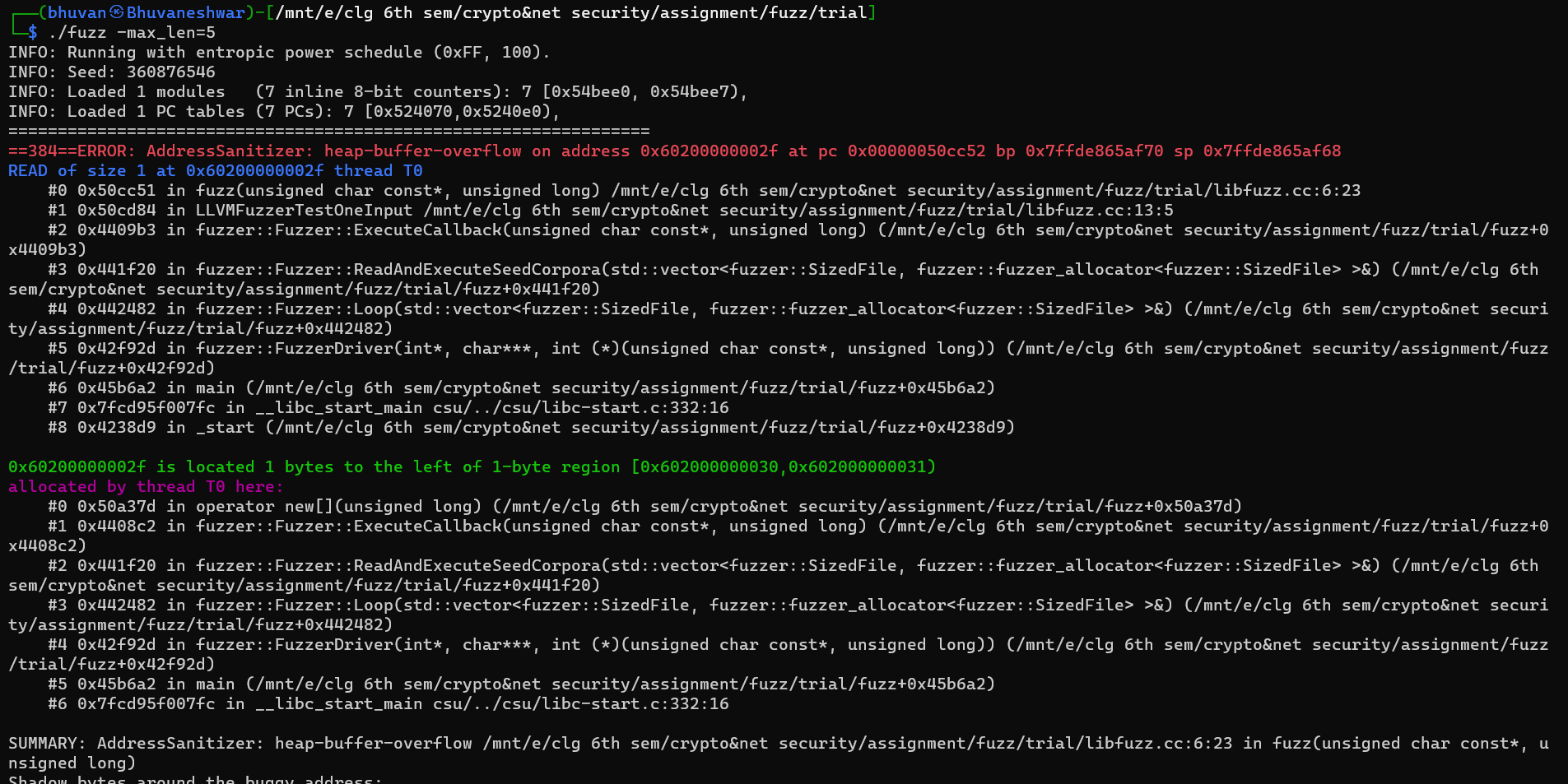
    fuzz(data,s);

    return 0;

}

In the function “fuzz”, in the for loop the value of i is equal to half of value of size, it will give error because it not condition for palindrome. Clang+ is used to add libfuzzer to our program. LLVMFuzzerTestOneInput is the driver of libfuzzer to test fuzz() function. We will compile and see what happens. clang++ -g –fsanitize=address,fuzzer filename is used to compile source file for fuzzing. Max\_len=20 is used to run fuzzer with max input size of 5.





Here the error says that heap-overflow has occurred at address 0x6020000002f at instruction pc 0x000000050xx52 bp 0x7ffde865af70 bp 0x7ffde865af68.

**Program after removal of discovered bug:**

#include <stdint.h>

#include <stddef.h>

#include <signal.h>

bool isPalindrome(const uint8\_t \*data, size\_t s){

    for(int i = 0 ; i < s/2 ; i++){

        if(data[i] != data[s-i-1])

            return false;

    }

    return true;

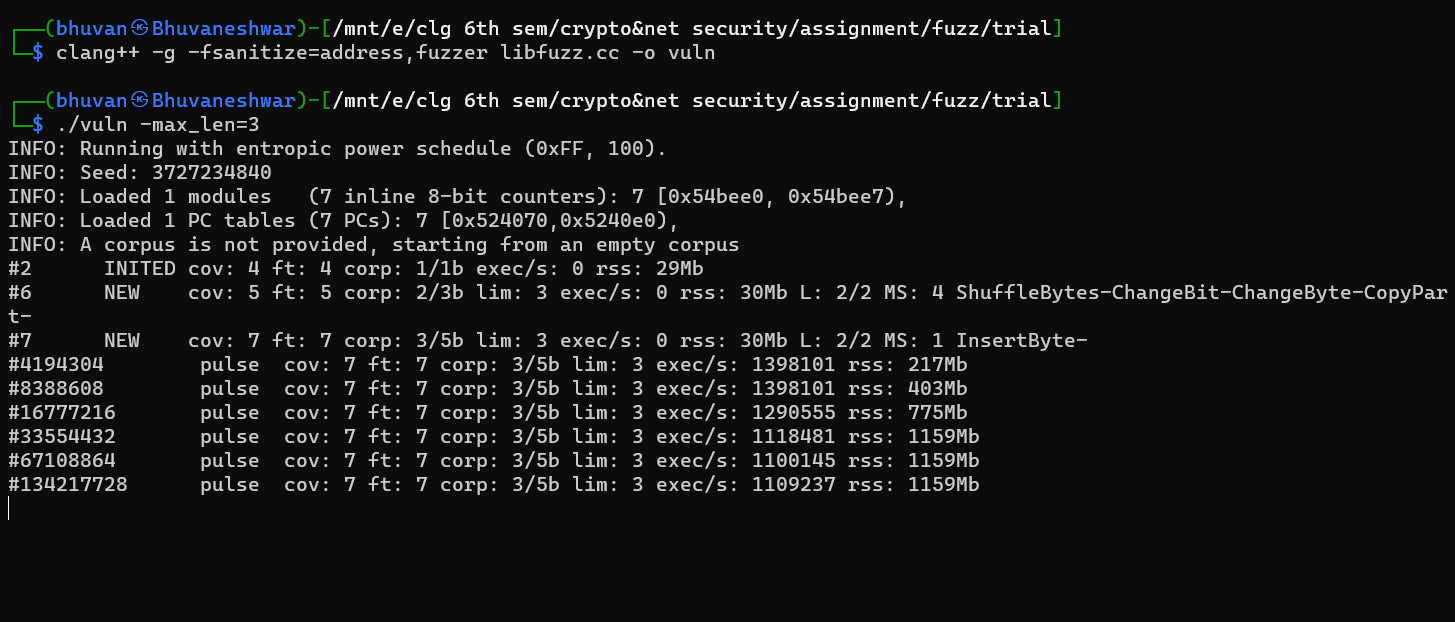
}

extern "C" int LLVMFuzzerTestOneInput(const uint8\_t \*data, size\_t s){

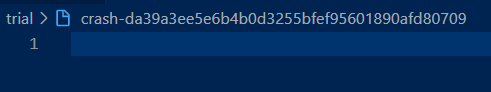
    isPalindrome(data,s);

    return 0;

}



Crash reported at when the input string is null.



AFL

Palindrome.cpp

#include <iostream>

#include <bits/stdc++.h>

#include <fstream>

using namespace std;

bool isPalindrome(string data, int s)

{

    for (int i = 0; i < s / 2; i++)

    {

        if (data[i] != data[s - i - 1])

            return false;

    }

    return true;

}

void parse\_file(char \*filename,vector<string>&input){

    string i;

    ifstream input\_file(filename);

    if(!input\_file.is\_open()){

        cerr << "Could not open the file - '"<< filename << "'" << endl;

        exit(-1);

    }

    while(getline(input\_file,i)){

        input.push\_back(i);

    }

    input\_file.close();

}

int main(int argc, char \*\*argv)

{

    string data;

    if (argc < 2)

    {

        cout << "enter filename";

        return -1;

    }

    vector<string>input;

    parse\_file(argv[1],input);

    for(auto i : input){

        if(isPalindrome(i,i.size())){

            cout<<i<<"\t -->  TRUE\n";

        }

        else{

            cout<<i<<"\t -->  FALSE\n";

        }

    }

    return 0;

}

Compile and run the program using afl-g++

