**CS6008 Cryptography and Network Security.**

**Assignment No.5**

**Name:** G Vignesh **Reg No:**2018103078

**Date:**4.6.2022

**Module-4:** NETWORK EXPLOITS

**External Learning-** Using libfuzzer and AFL to fuzz your own C/C++ implementations

**Aim:**

To Fuzz a c/c++ program which is programmed to perform some usual computation tasks. Using Fuzzer tools such as **Libfuzzer/ AFL** to compile and run the c/c++ program executable files and find what are all the inputs that crash the program and creates a log file that shows those crashing input values that our program is vulnerable.

**Tools-used:**

1.Linux Terminal

2.GCC/G++ compiler.

3.Libfuzzer(Clang )

4.AFL(American Fuzzy LOP)

5. Git

**Description:**

Fuzzing is not a special technique its just generates automated inputs files to the given c/c++ programs and analyzes how the program behaves to those inputs, if there are any error such as segmentation fault, integer overflow, buffer overflow, stack crashing etc. It makes a note of it. Finally provides us with a log file containing containing input values that causes the programs to crash along with the reason. Fuzzer is the tool/engine used to generate automated inputs and feed to the program and generate results for us to analyze how to alter our code to overcome the vulnerability.

**Using AFL(American Fuzzy LOP)**

**File.cpp**

#include <fstream>

#include <iostream>

const char \*uridecode(const char \*s) {

    static char ret[100];

    for(auto \*p=ret;\*s;++s) {

        if (\*s=='%') {

            auto const a = \*++s;

            auto const b = \*++s;

            \*p++ = (a<='9' ? a-'0' : a-'a') \* 16 + (b<='9' ? b-'0' : b-'a');

        } else if (\*s=='+') {

            \*p++ = ' ';

        } else {

            \*p++ = \*s;

        }

    }

  return ret;

}

int main() {

    auto const uri = std::string(

        std::istreambuf\_iterator<char>(std::cin),

        std::istreambuf\_iterator<char>()

    );

    std::cout << uridecode(uri.c\_str());

}

This is the sample c++ program which is buggy in nature is used to as target program for fuzzing using AFL.

About the program deliverables:

\* URI decoder.

 \* Translates %xx (xx = two hex digits) to the character with the

 \* appropriate ASCII code. Translates '+' into space. Leaves all

 \* other characters unchanged.

 \*

 \* Example:

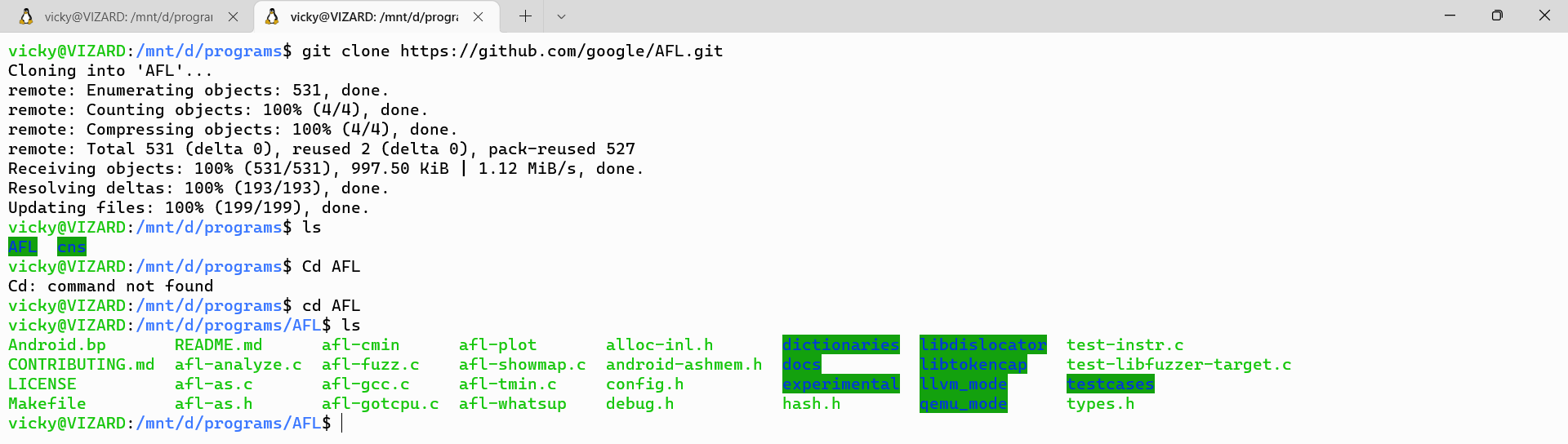
 \* - In:  "Hello+world%21"

 \* - Out: "Hello world!"

**Execution:**

**Step-1:**

To install AFL in our linux system we must git clone the afl official repository. After that we will get folder AFL containing compiler for Fuzzing, change directory to.



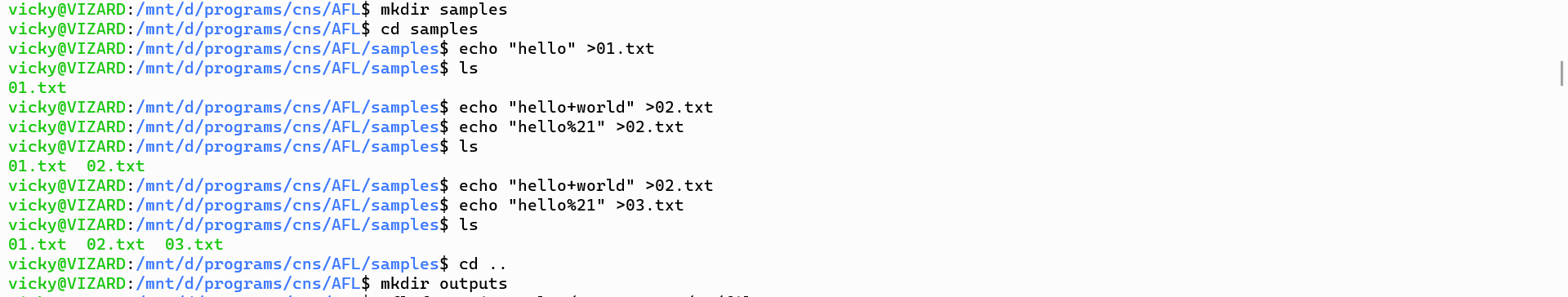
We have to place our sample files inside it.



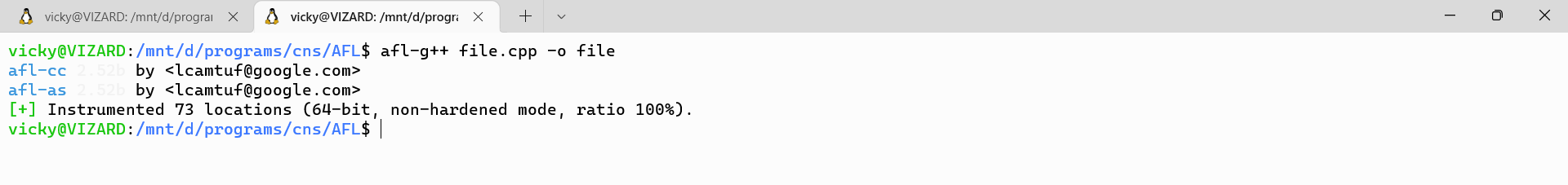
The file.cpp is created and compiled using g++ and we can expect the output as shown.

**Step-2:**

Sample folder is created to place sample inputs to our program and a outputs folder is created to store the output of the AFL tool. AFL tool requires sample files because it automates inputs for our program by alter the input that we have given.



Sample folder contains files 1.txt,2.txt,3.txt contains strings such as “hello”,”hello+world”,”hello%21”.

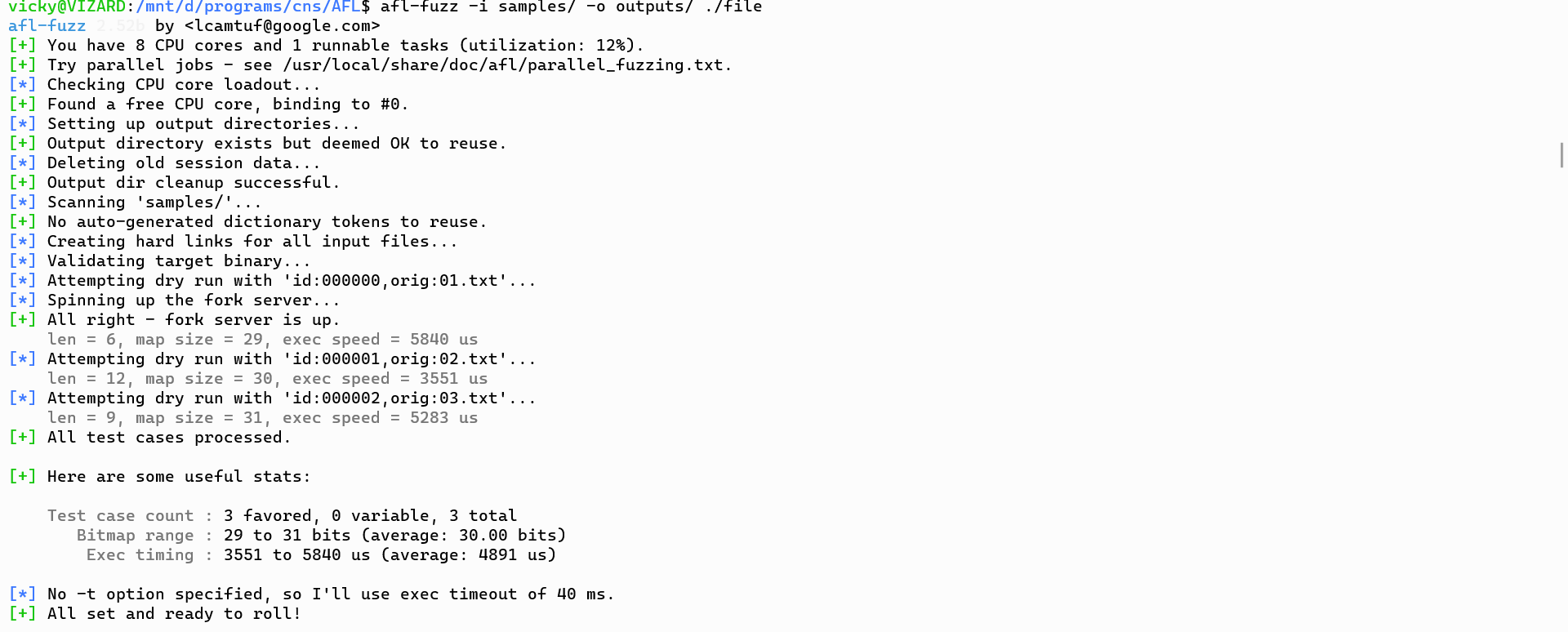


File.cpp is compiled using afl-g++ compiler. And executable binary ./file is created

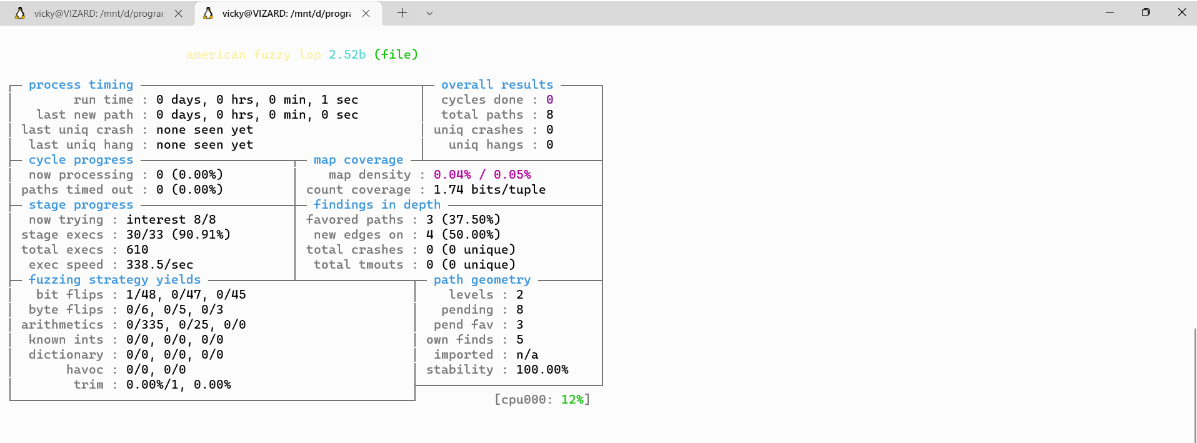
**Step-3:**

**AFL** fuzzer is started to execute by using the command

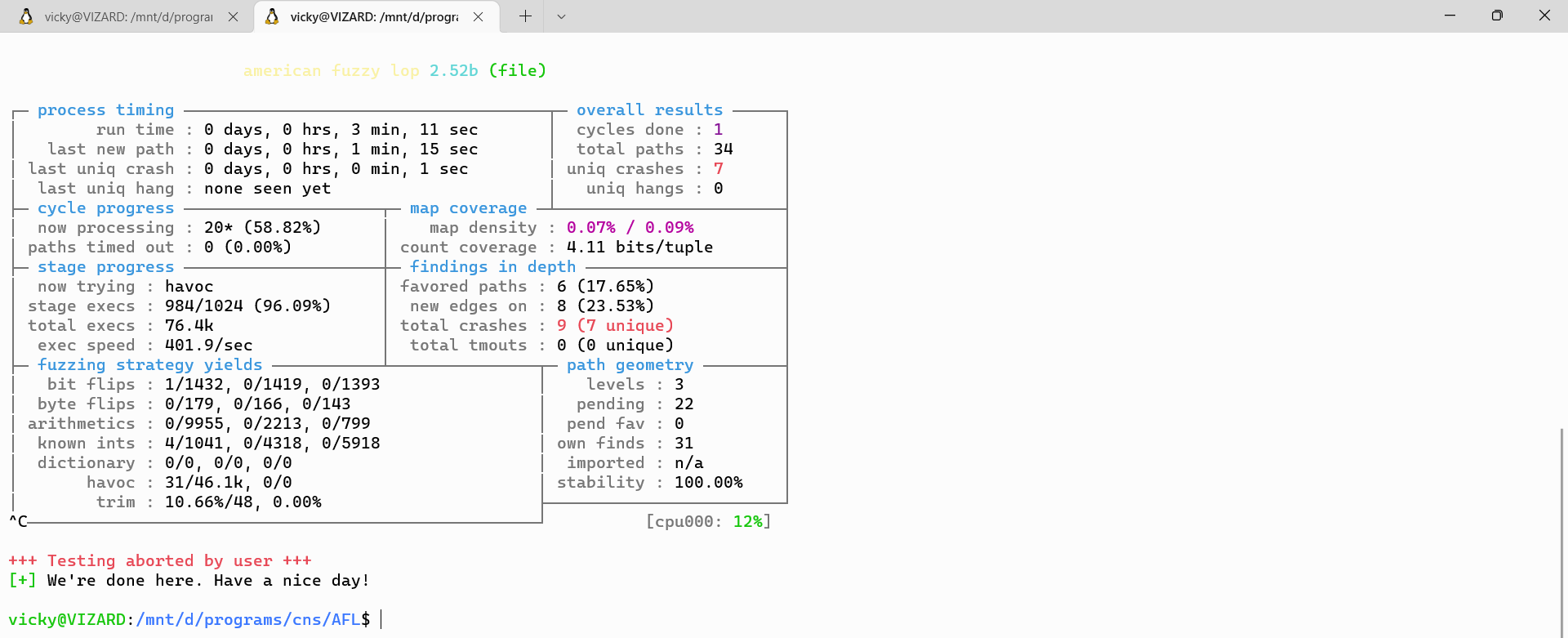
**afl-fuzz –i samples/ -o outputs/ ./file.**

****

**Fuzzer** starts to generate automated inputs and feed to our program which runs over time, showing results.



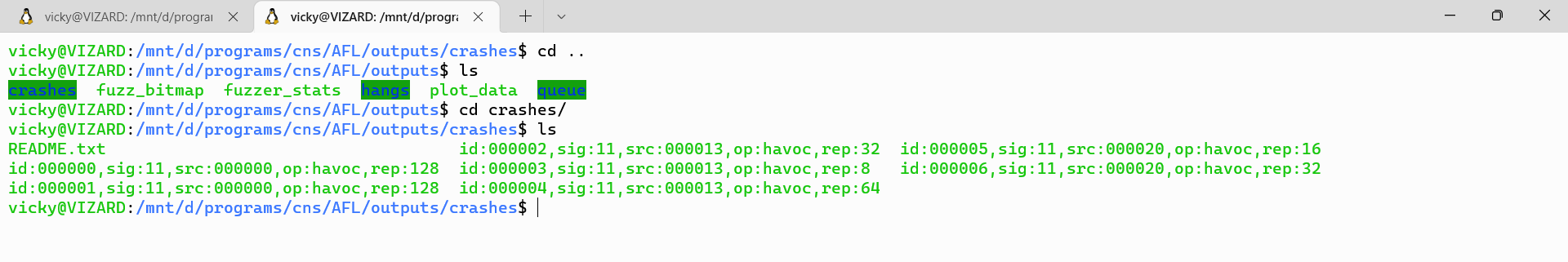
Fuzzer afl keeps and running and finds several possible crashes, we can use **CTRL+C**  to stop if needed.



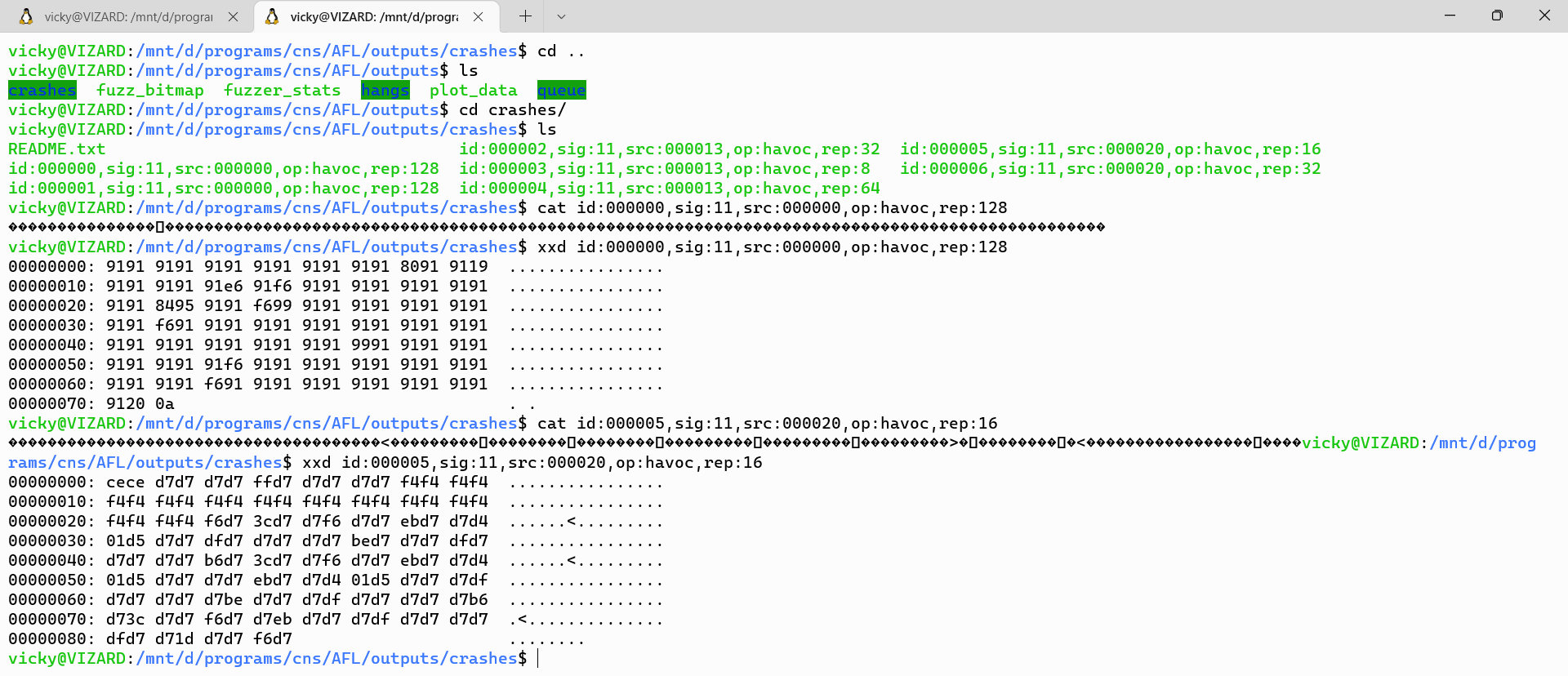
AFL shows us details while running like no.of unique crashes, duration of running, no.of paths covered etc.

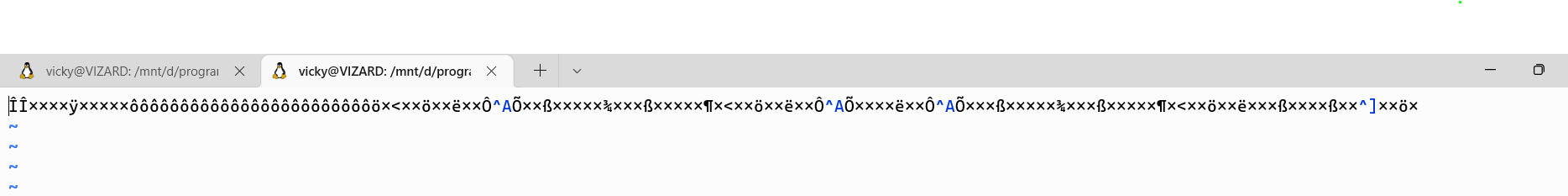
**Step-4:**

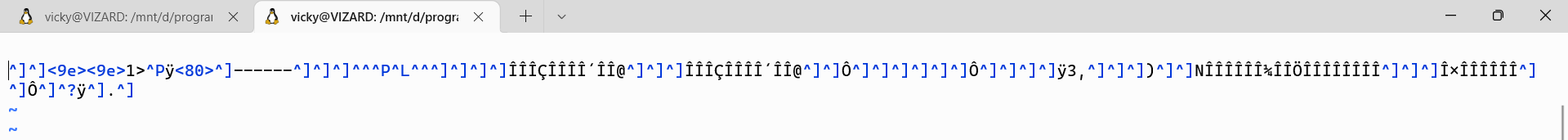
Since we have redirected the outputs of our output log files to a directory called **outputs**  it contains the log files and crash reports produced by the fuzzer AFL.



We can examine what all the inputs that caused crashes, inside the crashes folder.by viewing each crashes separately

 values that cause crashes are shown in hexadecimal value. Vim command can be used to view those values.





These are the crash inputs when given to a program its produces changes in its behavior or the program is vulnerable to these kind of inputs. This information is used by attacker which helps them to identify vulnerabilities from the executable file even without viewing the code.

**Using Libfuzzer for fuzzing.**

Implementing libfuzzer is it’s an inprocess fuzzing technique, it requires c/c++ program in a specific format The first step in using libFuzzer on a library is to implement a *fuzz target* – a function that accepts an array of bytes and does something interesting with these bytes using the API under test. Like this:

*// fuzz\_target.cc*

**extern** "C" int LLVMFuzzerTestOneInput(**const** uint8\_t \*Data, size\_t Size) {

DoSomethingInterestingWithMyAPI(Data, Size);

**return** 0; *// Non-zero return values are reserved for future use.*

}

**Step-1:**

We have to change our c program somehow without modifying the functionality of the actual code.

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<stdint.h>

struct Image

{

    char header[4];

    int width;

    int height;

    char data[10];

};

int LLVMFuzzerTestOneInput(const uint8\_t\* data, size\_t size){

if(size<=12){

return 0;

}

    struct Image \*img;

    img = (struct Image\*)data;

        int size1 = img->width + img->height;

        char\* buff1=(char\*)malloc(size1);

        memcpy(buff1,img->data,sizeof(img->data));

        free(buff1);

        if (size1/3==0){

            free(buff1);

        }

        else{

            //use after free

            if(size1/20 == 0){

                buff1[0]='a';

            }

        }

        int size2 = img->width - img->height+100;

        char\* buff2=(char\*)malloc(size2);

        memcpy(buff2,img->data,sizeof(img->data));

        int size3= img->width/img->height;

        char buff3[10];

        char\* buff4 =(char\*)malloc(size3);

        memcpy(buff4,img->data,sizeof(img->data));

        char OOBR\_stack = buff3[size3];

        char OOBR\_heap = buff4[size1];

        buff3[size3]='c';

        buff4[size1]='c';

        if(size3/5==0){

            buff4=0;

        }

        else{

            free(buff4);

        }

        free(buff2);

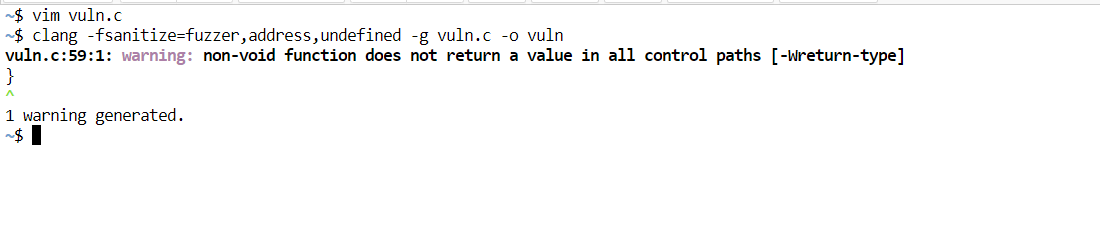
}

This is the c program modified in order to be accepted by libfuzzer fuzzer engine.

**Step-2:**

We need to compile the program using clang compiler which provides libfuzzer mode.

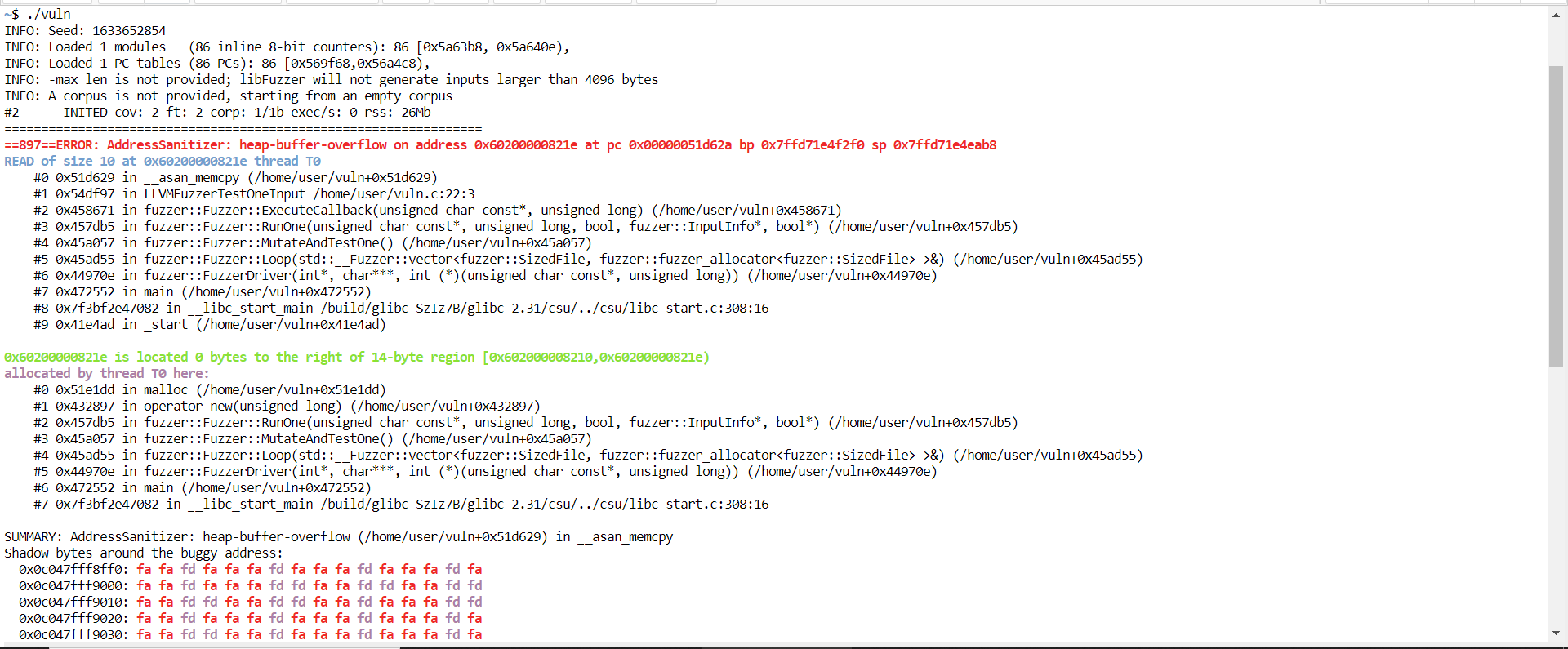
**clang -fsanitize=fuzzer,address,undefined -g inputfile -o outputfile**

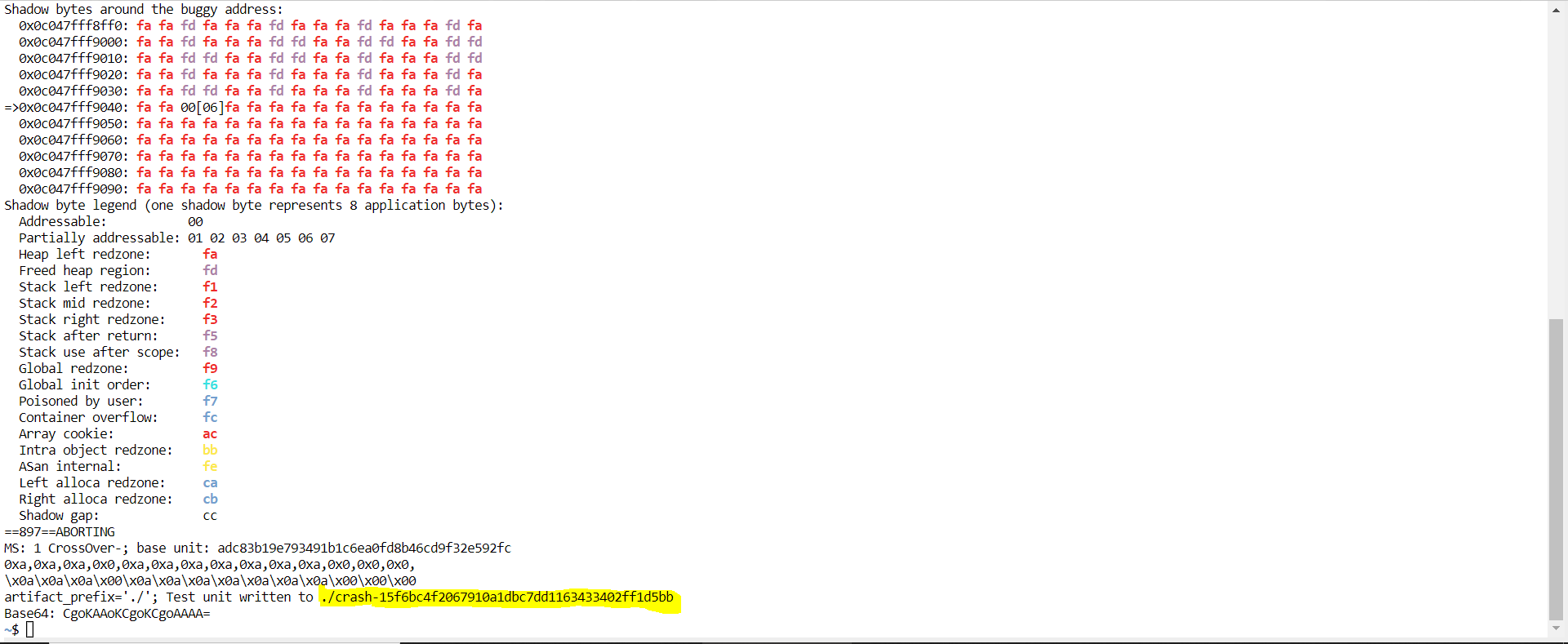


The program compiled using clang in fuzzer mode , usually shows some warning its common because we have modified the program for in process fuzzing, by changing function name as mentioned in libfuzzer documentation.

**Step-3:**

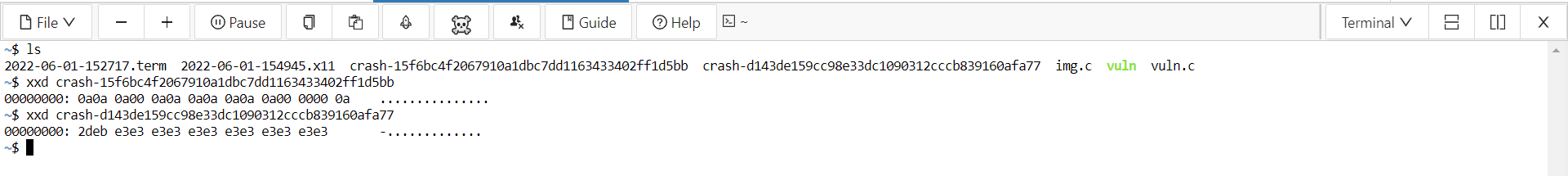
Then we need to run the executable file created by clang, by doing doing this we start fuzzing our code with random inputs to find crashes.



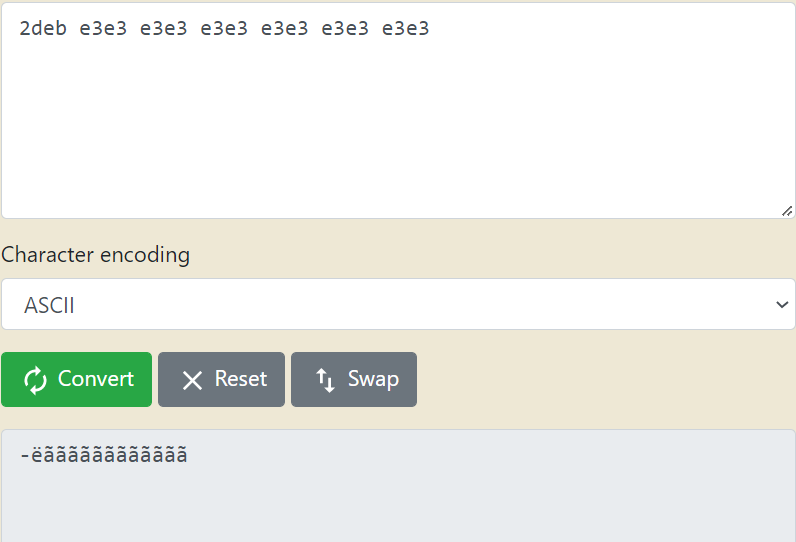


It had identified a heap overflow error for some input value, detailed of those crash are stored in a log as mentioned in the last line of execution, we can examine the values which caused the crash.

**Step-4:**

****

The output is in hexadecimal format, we convert we can obtain the original data which when passed to the function will crash.



We can even pass the crash input file to the executable and see the error reported by the code.



Now we get the details about each crash where it happened, what kind of error is it shown when we pass the crash file created by libfuzzer to our program.

**Conclusion:**

To discover vulnerabilities and fix them in advance, researchers have proposed several techniques, among which fuzzing is the most widely used one. In recent years, fuzzing solutions, like AFL, Libfuzzer, win afl have made great improvements in vulnerability discovery . which also used by attackers to check vulnerabilities and our code to plan the attack, so developers must first use various fuzzer to check and rectify the vulnerabilities so that we can avoid attack by hackers.