Pennsylvania State University

CMPSC 311

Honors Option

Author:
Quinn Butcher *(qdb5021)*

Lecturer:
Prof. Syed Hussian

November 14, 2023

1 Set-up

To set up AFL, we must first clone the AFL repo with the command at:

$$git\ clone\ https://github.com/AFLplusplus/AFLplusplus.git \quad (1)$$

To use the AFL fuzzer, we must first *make* AFL on our home computer, which is provided below [if you have any errors please reference the $README.md$ provided through the GitHub link]. We are using AFL++ because AFL has not been updated in many years, and such there are many improvements that have been made in fuzzing. AFL++ is specifically a fork that provided additional enhancements and features that can be helpful to us, all of which has been continued to be improved.

1.1 Prerequisites

Although running $AFL++$ does not need root access, to build $AFL++$ we need root-access to download the many prerequisites that AFL has such as llvm, clang, etc in order to run properly and efficiently. Such, to run $AFL++$, I am using a VM that is running the Ubuntu 18.04 LTS OS. The version of C that I am using is 7.5.0 which is standard on this OS.

1.2 Download

First download the prerequisites with $sudo\ ap-get\ install$ and then cd into your $AFL++$ directory. Then run:

$$make\ distrib \quad (2)$$

$$sudo\ make\ install \quad (3)$$

in your terminal.

1.3 Overall Outlook of Report

1. We will first delve into how to setup AFL to run on our programs
2. Introduce our 3 different code programs that we will run AFL on and the time that is required to run a fuzzing campaign on each
3. One each code page, the format will follow such as:
   (a) Code outline

(b) Outline of AFL output

(c) Fixes

2 Running AFL

To run AFL, we use $afl - gcc$ and $afl - gcc - clang$ to generate the fuzz output files that we are looking for. We first need to set an input directory and an output directory with the following commands:

$$\text{mkdir output} \tag{4}$$

$$\text{mkdir input} \tag{5}$$

NOTE: We must initialize some test with *echo "<input-text>" > input/1.txt* such that we can have an initial tester seed if there needs input. AFL will figure the rest out, such this input text does not need to be complex. We will detail the inputs that are chosen for each AFL fuzz.

After making the input and output directories, we must make our program. First, we create a *MakeFile* that links back to $afl - gcc$, such we make a *MakeFile* in which we include we can include either of the following, based on what our initial C function that we are testing:

```
CC=afl-gcc  #compiler
TARGET=<fileName> #target file name
all:
        AFL_HARDEN=1 $(CC) -fsanitize=address -fno-omit-frame-pointer -
    fsanitize=undefined <fileName>.c -o $(TARGET)

clean:
        rm $(TARGET)
```

OR

```
CC=afl-gcc-clang        #compiler
CFLAGS=-c -Wall -I. -fpic -g -fbounds-check -Werror
LDFLAGS=-L.
LIBS=-lcrypto

OBJS=tester.o util.o mdadm.o

%.o:  %.c %.h
  $(CC) $(CFLAGS) $< -o $@

tester: $(OBJS) jbod.o
  $(CC) $(LDFLAGS) -o $@ $^ $(LIBS)

clean:
  rm -f $(OBJS) tester
```

Then we can make our program. To do this we run:

$$\text{make all} \tag{6}$$

$$\text{make} \tag{7}$$

NOTE If we use the first *Makefile*, we use the first command and if we use the seconed *Makefile*, we use the second command.

Finally we run the afl-fuzzer with the following commands where we replace *filename* with our desired file-name for our C file.

afl-gcc -g -fsanitize=address ***filename***.c -o ***filename*** (8)

afl-fuzz -i input -o output -m none – ./***filename*** @@ (9)

NOTE: We do not need to do the first step (step (8)) if we use the second *Makefile*.

To run our crashes/hangs, we must execute the *.o* file that we are testing by running

./<pathtofile> id:0000<rest-of-file> (10)

3 Code 1

3.1 Code

```
/*
Author: Hardik Shah
Email: hardik05@gmail.com
Web: http://hardik05.wordpress.com
*/

//a vulnerable c program to explain common vulnerability types
//fuzz with AFL

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct Image
{
  char header[4];
  int width;
  int height;
  char data[10];
};

void stack_operation(){
  char buff[0x1000];
  while(1){
    stack_operation();
  }
}

int ProcessImage(char* filename){
  FILE *fp;
  struct Image img;

  fp = fopen(filename,"r");            //Statement   1

  if(fp == NULL)
  {
    printf("\nCan't open file or file doesn't exist.\r\n");
    exit(0);
  }


  while(fread(&img,sizeof(img),1,fp)>0)
  {
    //if(strcmp(img.header,"IMG")==0)
    //{
      printf("\n\tHeader\twidth\theight\tdata\t\r\n");

      printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);


      //integer overflow 0x7FFFFFFF+1=0
      //0x7FFFFFFF+2 = 1
      //will cause very large/small memory allocation.
      int size1 = img.width + img.height;
      char* buff1=(char*)malloc(size1);

      //heap buffer overflow
      memcpy(buff1,img.data,sizeof(img.data));
      free(buff1);
      //double free
      if (size1/2==0){
        free(buff1);
      }
      else{
        //use after free
        if(size1/3 == 0){
          buff1[0]='a';
        }
      }

```

```
71
72        //integer underflow 0-1=-1
73        //negative so will cause very large memory allocation
74        int size2 = img.width - img.height+100;
75        //printf("Size1:%d",size1);
76        char* buff2=(char*)malloc(size2);
77
78        //heap buffer overflow
79        memcpy(buff2,img.data,sizeof(img.data));
80
81        //divide by zero
82        int size3= img.width/img.height;
83        //printf("Size2:%d",size3);
84
85        char buff3[10];
86        char* buff4 =(char*)malloc(size3);
87        memcpy(buff4,img.data,sizeof(img.data));
88
89        //OOBR read bytes past stack/heap buffer
90        char OOBR = buff3[size3];
91        char OOBR_heap = buff4[size3];
92
93        //OOBW write bytes past stack/heap buffer
94        buff3[size3]='c';
95        buff4[size3]='c';
96
97        if(size3>10){
98          //memory leak here
99          buff4=0;
100       }
101       else{
102         free(buff4);
103       }
104       int size4 = img.width * img.height;
105       if(size4/2==0){
106         //stack exhaustion here
107         stack_operation();
108       }
109       else{
110         //heap exhaustion here
111         char *buff5;
112         do{
113         buff5 = (char*)malloc(size4);
114         }while(buff5);
115       }
116       free(buff2);
117     //}
118     //else
119     //  printf("invalid header\r\n");
120
121   }
122   fclose(fp);
123   return 0;
124 }
125
126 int main(int argc,char **argv)
127 {
128   if (argc < 2) {
129         fprintf(stderr, "no input file\n");
130         exit(-1);
131     }
132   ProcessImage(argv[1]);
133   return 0;
134 }
```

3.2 Description of code

This is a coding example by Hardik Shah that includes many different crashes in C to help demonstrate and test to make sure that it is working as intended. The main file that we are testing for is *imgRead.c*. The

input file in this case includes just the word "test," which demonstartes how *AFL++* can generate input cases.

3.3 Fuzzing Campaign

We ran a 1 minute fuzzing campaign on this program, which resulted with some of the following bugs:

1. *AddressSanitizer failed to allocate 0xfffffffffffff7fe bytes*
2. *runtime error: signed integer overflow: 1296911693 + 1296911793 cannot be represented in type 'int'*
3. *runtime error: signed integer overflow: 2122219134 + 2122219134 cannot be represented in type 'int'*

There were a total of 10 crashes found in this code, of which the output file entire folder with the *AFL-fuzzer* stats and crashes reports.

3.4 Fixes

Since this was a test run, there are no fixes that are needed to be made. If the reader is curious, linked with the project folder, a fixed version is included titled *imgRead_patched.c*.

4 Code 3 [Assignment 3]

4.1 Code

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <err.h>
#include <assert.h>

#include "jbod.h"
#include "mdadm.h"
#include "util.h"
#include "tester.h"

#define TESTER_ARGUMENTS "hw:"
#define USAGE                                                \
  "USAGE: test [-h] [-w workload-file]\n"                    \
  "\n"                                                       \
  "where:\n"                                                 \
  "    -h - help mode (display this message)\n"              \
  "\n"                                                       \


/* Utility functions. */
char *stringify(const uint8_t *buf, int length) {
  char *p = (char *)malloc(length * 6);
  for (int i = 0, n = 0; i < length; ++i) {
    if (i && i % 16 == 0)
      n += sprintf(p + n, "\n");
    n += sprintf(p + n, "0x%02x ", buf[i]);
  }
  return p;
}

int run_workload(char *workload);

int main(int argc, char *argv[])
{
  char *workload = NULL;

  workload = argv[1];
  run_workload(workload);
  return 0;
}


int equals(const char *s1, const char *s2) {
  return strncmp(s1, s2, strlen(s2)) == 0;
}


int run_workload(char *workload) {
  char line[256], cmd[32];
  uint8_t buf[MAX_IO_SIZE];
  uint32_t addr, len, ch;
  int rc;

  memset(buf, 0, MAX_IO_SIZE);

  FILE *f = fopen(workload, "r");
  if (!f)
    err(1, "Cannot open workload file %s", workload);

  int line_num = 0;
  while (fgets(line, 256, f)) {
    ++line_num;
```

```
71      line[strlen(line)-1] = '\0';
72      if (equals(line, "MOUNT")) {
73        rc = mdadm_mount();
74      } else if (equals(line, "UNMOUNT")) {
75        rc = mdadm_unmount();
76      } else if (equals(line, "WRITE_PERMIT")) {
77        rc = mdadm_write_permission();
78      }else if (equals(line, "WRITE_PERMIT_REVOKE")) {
79        rc = mdadm_revoke_write_permission();
80      } else if (equals(line, "SIGNALL")) {
81        for (int i = 0; i < JBOD_NUM_DISKS; ++i)
82          for (int j = 0; j < JBOD_NUM_BLOCKS_PER_DISK; ++j)
83            jbod_sign_block(i, j);
84      } else {
85        if (sscanf(line, "%7s %7u %4u %3u", cmd, &addr, &len, &ch) != 4)
86          errx(1, "Failed to parse command: [%s\n], aborting.", line);
87        if (equals(cmd, "READ")) {
88          rc = mdadm_read(addr, len, buf);
89        } else if (equals(cmd, "WRITE")) {
90          memset(buf, ch, len);
91          rc = mdadm_write(addr, len, buf);
92        } else {
93          errx(1, "Unknown command [%s] on line %d, aborting.", line, line_num);
94        }
95      }
96
97      if (rc == -1)
98        errx(1, "tester failed when processing command [%s] on line %d", line,
        line_num);
99    }
100   fclose(f);
101   return 0;
102 }
```

4.2 Description of Code

This is the code that opens the input files and runs them (we include simple-input). We only need to include simple-input because AFL generates new test cases for us. The code that will be tested for assignment 3 will consist of the code that I submitted originally and an updated version of the code to see which code has many fixes. I have labeled **Fa23-assignment3** as the assignemtn code that will have my original code that I submitted along with the fuzzed commands. I will then have another folder labeled as **FIXED-fa23-assignment3** in which my fixed code will be and I will run a shorter fuzzing campaign.

4.3 Fuzzing Campaign 1

We ran a 2 hour and 15 minute fuzzing campaign which resulted in some of the following bugs:

1. *Buffer Overflow* detected
2. *Buffer Overflow* detected
3. *Buffer Overflow* detected

NOTE: Each buffer overflow detection is a unique crash, which can be seen in the attached output file crashes.
The crash statistics show that a total of 19 unique crashes were found (all being buffer overflows), while a total of over 20 thousand crashes were found.

4.4 Fuzzing Campaign 2

We ran a 1 hour and 15 minute fuzzing campaign in which we used *afl-gcc-clang* instead of *afl-gcc* [afl-gcc is outdated and such the clang version runs much faster] which resulted in some of the following bugs:

1. *Buffer Overflow* detected
2. *Buffer Overflow* detected
3. *Buffer Overflow* detected

NOTE: Although I have got the same crashes, there were much fewer buffer overflows which signifies that the code was much more efficient. There were a total of 4 less unique crashes, which is very telling of the improvements.

4.5 Fixes

Although we got crashes in both of these fuzzing campaigns, the crashes involved buffer overflows which is likely from the large amount of input commands. The one thing that I found in my revised code was a incorrect validation term where it didn't pass all the validation that *mdadm.c* needed to pass, such causing buffer overflows when the JBOD size was too small for a bigger *WRITE* command. I ending up fixing this and the crashes went down when I re-ran it by a total of 3-4 unique crashes [not included in the GitHub files]. I found this buffer overflow with *gdb*.

5 Code 4 [Assignment 4]

5.1 Code

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <err.h>
#include <assert.h>

#include "jbod.h"
#include "mdadm.h"
#include "util.h"
#include "tester.h"

#define TESTER_ARGUMENTS "hw:s:"
#define USAGE                                                                  \
  "USAGE: test [-h] [-w workload-file] [-s cache_size]\n"                      \
  "\n"                                                                         \
  "where:\n"                                                                   \
  "    -h - help mode (display this message)\n"                                \
  "\n"


/* Utility functions. */
char *stringify(const uint8_t *buf, int length) {
  char *p = (char *)malloc(length * 6);
  for (int i = 0, n = 0; i < length; ++i) {
    if (i && i % 16 == 0)
      n += sprintf(p + n, "\n");
    n += sprintf(p + n, "0x%02x ", buf[i]);
  }
  return p;
}

int run_workload(char *workload, int cache_size);

int main(int argc, char *argv[])
{
  int ch, cache_size = 0;
  char *workload = NULL;

  workload = arg[1];
  cache_size = 1024;

  if (workload) {
    run_workload(workload, cache_size);
    return 0;
  }

}

int equals(const char *s1, const char *s2) {
  return strncmp(s1, s2, strlen(s2)) == 0;
}

int run_workload(char *workload, int cache_size) {
  char line[256], cmd[32];
  uint8_t buf[MAX_IO_SIZE];
  uint32_t addr, len, ch;
  int rc;

  memset(buf, 0, MAX_IO_SIZE);

  FILE *f = fopen(workload, "r");
  if (!f)
    err(1, "Cannot open workload file %s", workload);
```

```
71
72    if (cache_size) {
73      rc = cache_create(cache_size);
74      if (rc != 1)
75        errx(1, "Failed to create cache.");
76    }
77
78    int line_num = 0;
79    while (fgets(line, 256, f)) {
80      ++line_num;
81      line[strlen(line)-1] = '\0';
82      if (equals(line, "MOUNT")) {
83        rc = mdadm_mount();
84      } else if (equals(line, "UNMOUNT")) {
85        rc = mdadm_unmount();
86      } else if (equals(line, "WRITE_PERMIT")) {
87        rc = mdadm_write_permission();
88      } else if (equals(line, "WRITE_PERMIT_REVOKE")) {
89        rc = mdadm_revoke_write_permission();
90      } else if (equals(line, "SIGNALL")) {
91        for (int i = 0; i < JBOD_NUM_DISKS; ++i)
92          for (int j = 0; j < JBOD_NUM_BLOCKS_PER_DISK; ++j)
93            jbod_sign_block(i, j);
94      } else {
95        if (sscanf(line, "%7s %7u %4u %3u", cmd, &addr, &len, &ch) != 4)
96          errx(1, "Failed to parse command: [%s\n], aborting.", line);
97        if (equals(cmd, "READ")) {
98          rc = mdadm_read(addr, len, buf);
99        } else if (equals(cmd, "WRITE")) {
100         memset(buf, ch, len);
101         rc = mdadm_write(addr, len, buf);
102       } else {
103         errx(1, "Unknown command [%s] on line %d, aborting.", line, line_num);
104       }
105     }
106
107     if (rc == -1)
108       errx(1, "tester failed when processing command [%s] on line %d", line,
      line_num);
109   }
110   fclose(f);
111
112   if (cache_size)
113     cache_destroy();
114
115   jbod_print_cost();
116   cache_print_hit_rate();
117
118   return 0;
119 }
```

5.2 Description of Code

This is the code that opens input files and runs them with a cache (we include simple-input). The cache size is a default 1024. This fuzzing campaign also consists of two different projects, the original assignment 4 that I submitted and a FIXED assignment 4. The original assignment is detailed in the first fuzzing campaign and the updated assignment 4 is detailed in the second fuzzing campaign.

5.3 Fuzzing Campaign 1

We ran a 2 hour and 20 minute fuzzing campaign [also with *afl-clang*] which resulted in some of the following bugs:

1. *runtime error: signed integer overflow: 4294967295 + 4294967295 cannot be represented in type 'int'*
2. *Buffer Overflow* detected
3. *Buffer Overflow* detected

5.4 Fixes

I first made a false bug in my code to have an error that is not a buffer overflow. I think that all these buffer overflows are coming from the way the *jbod_disk* is configured, such the only fix that could be made is to change how we are accessing the data structure entirely. I think that maybe using a **hash-map** instead of this huge array will be a lot more beneficial and help reduce all of these bugs.

5.5 Fuzzing Campaign 2

We ran a 1 hour 20 minute fuzzing campaign [also with *afl-clang*] which resulted in some of the following bugs:

1. *runtime error: signed integer overflow: 4294967295 + 4294967295 cannot be represented in type 'int'*
2. *Buffer Overflow* detected
3. *Buffer Overflow* detected

5.6 Fixes

Although there are similar bugs in this fuzzing campaign, the total cycles w/o finds increased by 9, which is a big increase based on the large input file that we worked with. This is telling to show that although there are more unique bugs (which resulted from improper *JBOD* operation validation which I found after fuzzing), the total cycles that were valid worked more frequently than the initial code. One of the big fixes that I made from my first fuzzing campaign was how *JBOD* write worked as it was causing many buffer overflows by stretching over disks in weird ways and adding too much information to the temporary buffer. This was also found with *gdb* and fixed from my second fuzzing campaign.

6 Changes to AFL

6.1 User-Changes

I would first make it easier to detect buffer overflows with *AFL*, specifically so that I wouldn't have to use *gdb* to see which my buffer is overflowing. This was extremely prevalent in my assignment 3, such I think this is an important fix to make. Another fix that I think that should be made to *AFL* is an easier way to track which bugs happen instead of having to run the bugs again through the object file again to see what went wrong in my code.

6.2 Perspective of AFL++

I think that *AFL++* is much better than *AFL* and has made great strides in speeding up the fuzzing process. I also think that it also reaches a higher bitmap coverage percentage, which is very important in finding bugs in code by increases the number of new branches being taken from the input.