SWEN90006 Assignment2 Report

Zheyuan Wu, Chenyang Wang, Lanye Shao, Zeqian Li

October 30, 2023

1 Introduction

TopStream is a movie service by which users connect to a TopStream server and watch the movies under their account. The accounts are divided into four categories in total: three for customers including FREE_ACCOUNT, BASIC_ACCOUNT and VIP_ACCOUNT and one for administrator called ADMIN_ACCOUNT.

Generally, the server of TopStream provides the following functionalities:

- To maintain a list of user accounts including an administrator account (username: "admin", password: "admin", MFA device id: "0123456789"). For simplicity, all passwords are not encrypted. To increase the security of the system, the admin account uses MFA.
- To add new free user account. Only the admin account should have the capability to add new users to the system using the REGU command, and update their account type (e.g., from a free account to a paid account) using the UPDA command.
- To allow users update their password and add/replace their device id (i.e. a valid 10-digit mobile phone number) to enable MFA if they wish.
- To load movies from files (the admin account is required) using the LOAD command. For simplicity, all movies are stored in text files and the information is not encrypted either. This sample file stores 10 movies. Each line contains the information of one movie including its name, its length in minutes, and its type (2: only VIP accounts can watch, 1: both BASIC and VIP accounts can watch, and 0: all accounts can watch).
- To list all movies using the LIST command.
- To play movies with suitable permissions using the PLAY command.

To be more specific, the server supports the following commands:

- USER & PASS for weak authentication

- `DPIN` for stronger authentication with MFA
- `REGU` register a new free user
- `AMFA` add/replace a device to enable MFA
- `UPDA` update account type
- `UPDP` update user password
- `LIST` list all movies
- `PLAY` play a selected movie
- `LOGO` log out of the current account
- `QUIT` terminate a connection

In this assignment, we are supposed to fuzz test the TopStream server to find out those security vulnerabilities via AFLNet, a greybox fuzzer for network protocols. 5 sorts of faults are taken into consideration for this assignment which are as following:

- Any fault that causes TopStream to crash or hang leading to a denial-of-service attack (e.g., Null pointer dereference (CWE-476)).
- Critical memory faults such as Stack/Heap Buffer Overflow (CWE-121 and CWE-122) and Use-After-Free (CWE-416).
- Logic/functional faults that would allow attackers to gain unauthorized access (e.g., CWE-285)
- Logic/functional faults that would allow attackers to steal users' information or compromise the integrity of users' data.
- Logic/functional faults that would allow attackers to cause financial loss to the owner of the TopStream service.

Following is a brief description of AFLNet and how it functions:

- Greybox Fuzzing: AFLNet adopts a greybox fuzzing approach, which is a hybrid method lying between black-box and white-box fuzzing. Unlike simple command-line tools, servers have a vast state space, making server fuzzing challenging. AFLNet[1], through well-defined sequences of input messages specified [2] in a protocol, can traverse this state space more effectively[3].
- Mutational Approach: It employs a mutational approach, using state-feedback alongside code-coverage feedback to guide the fuzzing process. This means that it alters existing input data to create new test data, rather than generating test data from scratch[1].

- Client Simulation: AFLNet acts as a client to replay variations of original message sequences sent to the server, retaining variations effective[1] at increasing the coverage of code or state space. The server states exercised[3] by a message sequence are identified using the server's response codes.
- Seed Inputs: It takes message sequences as seed inputs[1], which are initially captured from some sample usage scenarios between a sample client and the server[4] under test (SUT).
- Workflow: Initially, communication between the client and the server is captured and stored in a specified file (e.g., rtsp.pcap). Using Wireshark network analyzer, the requests are extracted from this file to be used as seed input for AFLNet[1]. Optionally, modifications can be made to the server code to make it more effectively fuzzable. Finally, the fuzzing process is initiated, with AFLNet continuing to infer the implemented state machine of the SUT, updating a .dot file (ipsm.dot) so users can monitor AFLNet's progress in terms of protocol inferencing[1].
- Performance: According to a benchmark study on the ProFuzzBench benchmark, AFLNet's existing state selection algorithms achieve very similar code coverage, although a variant named AFLNetLegion showed better performance in selected case studies[5].

In brief, our tasks are:

1. To choose appropriate seed inputs and to increase line coverage and branch coverage.
2. To discover more than 3 vulnerabilities of the TopStream server.

2 Instructions and Experiments

2.1 Installation and Compile

To reproduce our project, first clone our Github repository

```
git clone https://github.com/SWEN90006-2023/swen90006-assignment-2-group-28.git
```

Change to the root dictory, create the docker image and run the container with

```
docker build . -t swen90006-assignment2
docker run -it swen90006-assignment2
```

Then change to `~/topstream`, compile executables with `make all`

2.2 Capture Seed Input

We use `tcpdump` to capture traffics between server and client, firstly we need start the server with

```
./service 127.0.0.1 9999
./topstream 127.0.0.1 8888 127.0.0.1 9999
```

and then capture the traffic with

```
sudo tcpdump -w rtsp.pcap -i lo port 8888
```

Finally start a telnet clinet to send request and record

```
telnet 127.0.0.1 8888
```

After capturing enough data, quit client and `tcpdump`

```
sudo pkill tcpdump
```

Now we get a file named `rtsp.pcap`, then copy the file outside of the container

```
docker cp <container id>:/home/ubuntu/topstream/rtsp.pacp <your local file path>
```

Open the file use Wireshark to extract TCP streams from `127.0.0.1 49716`[1] `-> 127.0.0.1 8888`, save the data as a raw file `seed.raw`. Now we could use `seed.raw` as the seed input of AFLNet. Refer to the steps of AFLNet document for detail if necessary. Finally upload `seed.raw` into docker container.

```
docker cp seed.raw[2] <container id>:home/ubuntu/results/seed_corpus
```

We have also upload our seed files into repositary, it could be found in GitHub

2.3 Dictionary

The command prefixes for topstream are recorded in the dictionary file located at `~/results/others/topstream.dict`.

2.4 Fuzzing Process

To start the fuzzing process, run with

```
chmod +x ~/results/others/restart.sh && chmod -R +r ~/results/seed_corpus

afl-fuzz -d -c /home/ubuntu/results/others/restart.sh -i /home/ubuntu/results/seed_corpus
-o /home/ubuntu/results/output-topstream -N tcp://127.0.0.1/8888 -x
/home/ubuntu/results/others/topstream.dict -P TOPSTREAM -D 10000 -q 3 -s 3 -E -K- R
./topstream-fuzz 127.0.0.1 8888 127.0.0.1 9999 ./service 127.0.0.1 9999
```

It could sometimes encounter a problem like cannot connect to the server, try running without `-E`

```
afl-fuzz -d -c /home/ubuntu/results/others/restart.sh -i /home/ubuntu/results/seed_corpus
-o /home/ubuntu/results/output-topstream -N tcp://127.0.0.1/8888 -x
/home/ubuntu/results/others/topstream.dict -P TOPSTREAM -D 10000 -q 3 -s 3 -K- R
./topstream-fuzz 127.0.0.1 8888 127.0.0.1 9999 ./service 127.0.0.1 9999
```

If the server says timeout, rerun the above process to try.

We have run the fuzzing process for 13 hrs 14 mins, totally 32 cycles, 286 paths and encountered 33 unique crashes, Figure 1 deplicts the results.

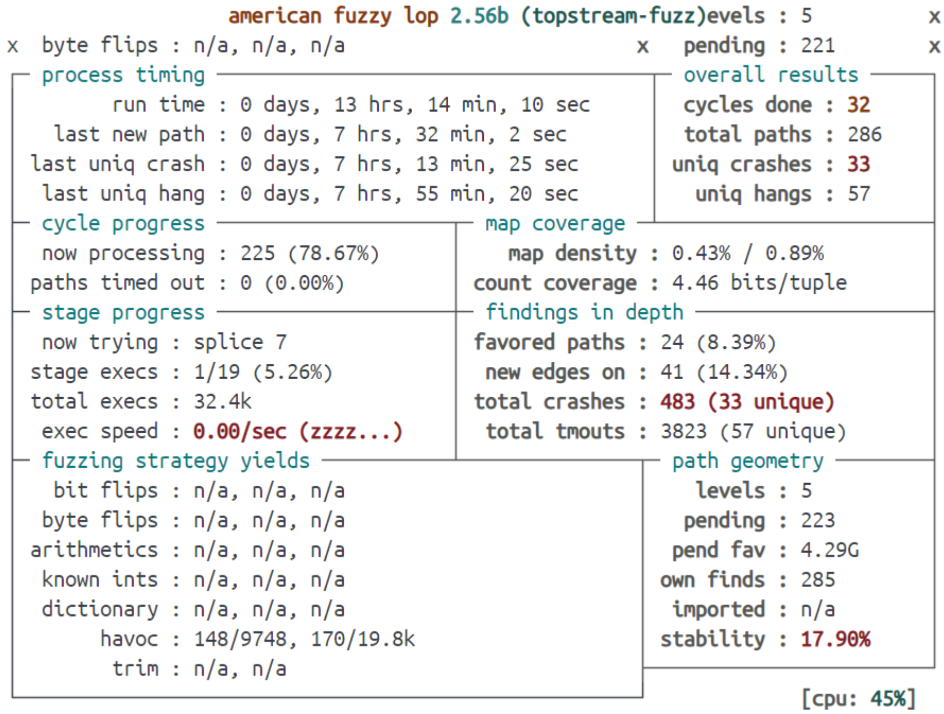


Figure 1: Fuzzing Test Results

This result is the best results we got from all of the experimental trials.

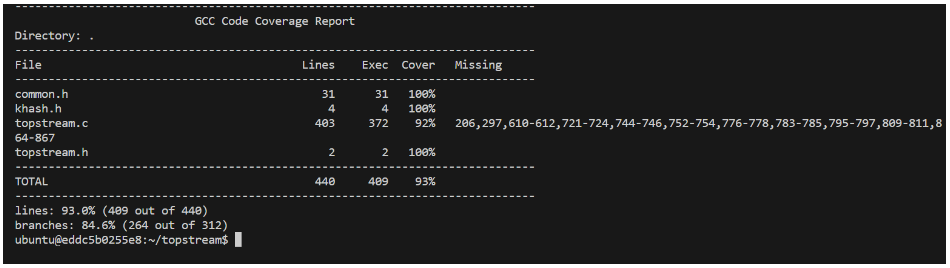
- For code-coverage: We ran our experiments mainly as follows: we first design seeds covers all of the commands in Topstream server, and see what lines are missing, and then we insert into the seed with specific commands to let afl-fuzzing cover the lines. For example, if there is only `UPDA zheyuan,VIP` in our seed, it may only cover `line 503-505`, then we add `UPDA zheyuan,BASIC`, `UPDA zheyuan,FREE` and run `afl-fuzz` for long enough time, it will largely cover `line 499-502`.

- For vulnerabilities: We first run the afl-fuzzing with a very simple seed we can think of randomly and `afl-fuzz` it with very long time and found out that no crash was out. Then our idea is settled: every time we manually find the error in `topstream.c`, we will design seed specifically to let `afl-fuzz` produce the error more easily. For example, when we found that in `line 375` in `topstream.c` (discussed in detail in vulnerability) there is heap buffer overflow error, we will insert command `UPDA` with very long password to let afl-fuzz identify crash more easily and it proved that this method was effective. However, for other vulnerabilities, due to time constraint or flaws in the seed deisgned, it is very hard for afl-fuzz to identify the error.

2.5 Code Coverage

We use the `gcovr` tool to determine code coverage. `cal_cov.sh` is written to compute the overall code coverage from all our produced inputs. This script requires one command-line parameter: the directory where the inputs are saved. Be aware that the replayable-queue is a zip file, please unzip it and put it in the correct directory as below.

```
chmod +x ~/results/others/cal_cov.sh
~/results/others/cal_cov.sh ~/results/output/replayable-queue
gcovr -r . -s
```



```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
common.h                                      31      31   100%
khash.h                                        4       4   100%
topstream.c                                  403     372    92%   206,297,610-612,721-724,744-746,752-754,776-778,783-785,795-797,809-811,8
64-867
topstream.h                                    2       2   100%
------------------------------------------------------------------------------
TOTAL                                        440     409    93%
------------------------------------------------------------------------------
lines: 93.0% (409 out of 440)
branches: 84.6% (264 out of 312)
ubuntu@eddc5b0255e8:~/topstream$
```

Figure 2: Code Coverage

3 Vulnerabilities

3.1 Heap Buffer Overflow

3.1.1 Issue 1

The first vulnerability is heap buffer overflow, which is the most important one. It is triggered by the user updating the original password with a very long enough password. `asa.sh` shell script is written to afl-replay the crash inputs with the ASan(address sanitizer) and stored in `~results/others/asa.sh`. `asa.sh` accepts two arguments, which is the output and input directory. `ASAN\_OPTIONS=detect\_leaks=0` because it is mentioned in the readme in the github security vulnerability does not count memory leaks and benign integer overflows. `llvm-symbolizer` is to show the exact line where the issue occurs. `asa.sh` also shows a summary of the error, for this case is heap buffer overflow.

```
sudo apt-get update
sudo apt install llvm -y
chmod +x ~/results/others/asa.sh
~/results/others/asa.sh ~/results/output/replayable-crashes/ ~/results/crash1/
```

Please remember: if you want to generate crash1 zip again, you may need to delete the output folder and crash1 zip first because otherwise the directory is incorrect.

For this issue, the bug is caused by line 375 in `topstream.c` which is `strcpy(user->password, tokens[0]);` in `topstream.h` the `user_info_t` is defined as:

```
typedef struct {
    char* device_id;
    char password[MAX_PASSWORD_LENGTH + 1];
```

```
    int type; //free, basic, vip
} user_info_t;
```

where MAX_PASSWORD_LENGTH is equal to 20 and memory is allocated for the new user object in the newUser function:

```
user_info_t *newUser() {
    user_info_t *user = (user_info_t *) malloc(sizeof(user_info_t));
    user->password[0] = '\0';
    user->device_id = NULL;
    user->type = FREE_ACCOUNT;
    return user;
}
```

However, in line 375, there is no condition to check the length of the updated password length, which means that the user can input any length of password, if the user input password such as `UPDA aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa,aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`, the length of the updated password will exceed the `MAX_PASSWORD_LENGTH`, (longer than the space allocated to `user->password`), thus may overwrite adjacent memory and cause heap buffer overflow.

Inputs leading to a crash are in `~/results/pocs/buffer-overflow-crash.zip`, and ASan result in `~/results/others/asan-buffer-overflow-crash.zip`.

3.2 unauthorized access & financial loss

3.2.1 Issue 2

We found the issue manually through exploring the code in `topstream.c` but not through afl fuzz. The issue is not that serious but is a way of unauthorized access. The issue is caused by line 596, which is `tokens[2][strlen(tokens[2]) - 1] = '\0';`. As it is not mentioned in the readme in the github **the format of movie.txt**, assume a movie.txt with no newline character at the end of last line in the `movie.txt` and the movie of last is only for `VIP` account type, since there is no newline character(**no extra empty line**), it will instead remove the last character of the token, changing the last token "2" into empty string " ", then in `line 599: m->type = atoi(tokens[2])`, `atoi` converts the empty string into '0' . As it is defined in `line 538~540`

```
if (m->type == 0) {
    strcpy(type, "FREE");
}.....
```

If the admin (evil admin acts as attacker) loads this type of movie.txt without the newline character at the end of the last line, the last VIP movie will be free for any account type to watch. Thus, it is considered as a logic/functional fault that would allow attackers to gain unauthorized access. Due to this logic/functional fault, though not that much, it still causes some financial loss to the owner of Topstream service because paid movies turn into free movies for every account type to play.

First, prepare a `movie.txt` without a newline character at the end of the last line and save it in `~/topstream`. Then follow the input below(assume the last line is the 10th line and should only for VIP account type) (in `./topstream movies4.txt`):

```
USER admin
PASS admin
DPIN 6573
REGU zhe,zhe
LOAD movie4.txt
LOGO
USER zhe
PASS zhe
PLAY 10
LOGO
QUIT
```

Replay the input in `~/results/pocs/missingnewchar.replay`. The expected result is `free` account user zhe can play the `VIP` movie. Line `545~547` in `~/others/topstream1.c` is where we plan to add assert but failed to get any replayable-crashes after afl-fuzz it.

3.2.2 Issue 3

Issue 3 is also related to financial loss, So in `line 653`, which is `if (user->type >= getMovieType(index))` is where the root of the issue lies. This line of code checks whether the index of user type is greater than the movie type index. If true, the movie can be played, if not, the movie cannot be played. However, in `line 538~544`

```
if (m->type == 0) {
    strcpy(type, "FREE");
} else if (m->type == 1) {
    strcpy(type, "BASIC");
} else {
    strcpy(type, "VIP");
  }
}
```

It shows that the `VIP` can be any number except `0,1`, so assume a `movie.txt` looks like below:

```
Good morning, 135, 2
Killer story,140,34
Jack's friend,125,356
```

When User enter `PLAY 1`, because of logic of function `getMovieType`:

```
int getMovieType(int index) {
    kliter_t(lmv) *it;
    it = kl_begin(movies);
    for (int i = 0; i < index; i++) {
        if (it != kl_end(movies)) {
        it = kl_next(it);
         }
    }
if (it == kl_end(movies)) return -1;
movie_info_t *m = kl_val(it);
return m->type;
}
```

It `getMovieType(1)`, which corresponds to the line of movie "Killer Story" and since all vip account `user->type==2`. It will return an error saying that the movie is unable to be watched. However, Killer Story can be watched by Will because according to the logic of the code, any number except except `0,1` is defined as VIP account type, so that may cause great financial loss to the owner of Topstream server if this type of `movie.txt` is loaded, which will let the `VIP` movies cannot be watched anymore.

Unfortunately, we did not make the crash with afl-fuzz because we forgot we can use assertions to make the crash. Please input the following sequence to have a try based on instructions on **.readme** for execution of Topstream server(`./topstream/movie4.txt`):

```
USER admin
PASS admin
DPIN 6573
REGU zhe,zhe
UPDA zhe,VIP
LOAD movies4.txt
LIST
LOGO
USER zhe
PASS zhe
LIST
PLAY 1
```

```
PLAY 2
```

The expected result is VIP user zhe cannot `PLAY 1` nor `PLAY 2`. `Line 654` in `~/others/topstream1.c` is where we plan to add assert but failed to get any replayable crashes after `afl-fuzz` it.

Replay the input in $\tilde{/}$results/pocs/incorrect$_VIP_index.replay$.

3.3 Stack buffer overflow

3.3.1 Issue 4

The issue is caused by line 545 in `topstream.c`, which is

```
sprintf(tmpMovieStr, " %d. %s, %d, %s\r\n", ++index, m->name, m->length, type)
```

In `line 536`, it defines: `char tmpMovieStr[MAX_MOVIE_INFO_LENGTH]` with a fixed size where `MAX_MOVIE_INFO_LENGTH=100` so assume a `movie.txt` has the longest line like this:

```
APerfectPairingAPerfectPairingAPerfectPairingAPerfectPairingAPerfectPairingAaaaa,120,2
```

The total length of character of this line is 97 characters, which is smaller than 100, so when the administrator types in: `LOAD movie.txt`, `movie.txt` can be loaded. However, if administrator types: `LIST`, stack buffer overflow will happen because `sprintf` add more characters into `tmpMovieStr` with blank space, and index and other characters such as '.'. Then the result line will be:

```
1. APerfectPairingAPerfectPairingAPerfectPairingAPerfectPairingAPerfectPairingAaaaa, 120, VIP
```

The length of this line is 104 characters, which is more than 100, exceeding the allocated size and will **lead the telnet service to crash** because of stack buffer overflow.

Unfortunately, we did not successfully make the crash with afl-fuzz. But you can try to input the following sequence based on instructions on **.readme** for execution of Topstream server (`movies5.txt` is in `~/topstream`):

```
USER admin
PASS admin
DPIN 6573
LOAD movies5.txt
LIST
LOGO
```

When we try on `telnet 8888`, the expected result is that it will report `the stack smashing detected` and the server will crash and end the service. However, even though we try to afl-fuzz it for a long time, there are no replayable-crashes but it is indeed a stack buffer overflow error.

Replay the input in `~/results/pocs/stackbufferoverflow.replay`.

4 Reflection and Conclusions

In conclusion, our team learned a lot in assignment 2. We learned to use docker and aflnet fuzz-testing tools and experienced looking for problems that might cause system crashes in a large amount of code. We not only gained more knowledge and experience on fuzzing test, but also improved collaboration and communication skills.

In summary, we've implemented several enhancements to our initial setup to effectively uncover more vulnerabilities:

- MFA PIN Fix: MFA PIN is set to 6573 so that fuzzer does not need to guess a random number.
- Command Dictionary: fuzzing dictionary are 12 prefix command so that fuzzer can fuzz with commands in the fuzzing dictionary instead of large number of random useless commands.
- Unified Seed Input: We use a combined long seed input instead of multiple small seed inputs.

In our experiments, fixing the pin really saved a lot of time. Using only a fuzzy dictionary didn't produce good results. Fortunately, employing with fuzzy dictionary and combined long seed input simultaneously produced relatively good results.

In addition, we chose one long seed, rather than multiple short ones, because longer seeds contain more complex and comprehensive input that helps test deeper program logic. Such test cases can help fuzzy testing more fully cover the different execution paths of a program. Longer seeds are also more likely to contain heuristic input that can lead the fuzzer to more potential problems. This helps to find complex vulnerabilities in the later stages of fuzzing testing. However, sometimes it may lead the server to timeout because it is too big, not in our case, but it is still better to seperate some of the seed input out.
In the above four kinds of preparation, we have successfully discovered ALL 4 types of security vulnerabilities mentioned in README.md.

- UPDP function Topstream.c does not check the length of the updated password.
- LOAD function in topstream.c does not check whether there is newline character at the end of movie file(e.g.movie.txt)
- because of logic faults in PLAY and LIST function, VIP account type users may not watch the movies they can watch.
- function does not check the length of each line of movie file in LIST function.

Some suggestions to aflnet fuzzing: First, it is better for devlopers to cluster all the crashes with same causes together so that it saves much time to go over each crash to see it root cause. As mentioned, relying solely on dictionaries may still pose challenges in guiding the fuzzer towards reaching interesting states. One potential reason for this challenge is that the fuzzer somtimes mutates the command itself and add many useless mutations. It is better for the developers to let users choose the weights of different types of mutations so that it will be more effective to identify the error quickly.

References

[1] "Aflnet: A greybox fuzzer for network protocols," Oct 2023.

[2] "Csdl — ieee computer society," IEEE Computer Society, 2020.

[3] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," *IEEE Xplore*, Oct 2020.

[4] "Aflnet: A greybox fuzzer for network protocols," *Unimelb.edu.au*, 2023.

[5] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," 2023.