2022.10

SWEN90006 Security & Software Testing

ASSIGNMENT 1

TEAM 18 - JIACHEN PING, TIAN HUI, TIANDE YE, ZHEN CAI

1 TABLE OF CONTENT

2 INTRODUCTION

The FotBot server supports 12 commands:

1. USER, and
2. PASS logins;
3. DPIN uses stronger authentication with MFA based on a simulated telecom service;
4. REGU registers a new user;
5. AMFA add/replaces a device to enable MFA;
6. UPDA updates step data;
7. UPDP updates user password;
8. GETS gets step data;
9. ADDF adds a friend;
10. GETF gets all friends;
11. LOGO logouts of the current account;
12. QUIT terminates a connection.

The protocol is stateful. One of these requests from a client could map one state to another, and the request sequence is what determines the server's response and behaviour in advance.

In this assignment, we systematically employ coverage-guided grey-box fuzzing (CGF) techniques to test the FotBot server and discover any of the following security vulnerabilities:

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

Many CGFs currently in use, like the American Fuzzy Lop (AFL) fuzzer [1], are mutation-based. For them to generate a new input, a seed is taken from an initial corpus then mutated. It is added back to the corpus and utilized as the starting point for further rounds if considered interesting, e.g., has just explored a new branching path[1]. In such case, the fuzzer strives for high code coverage since a high value for this metric denotes that a large proportion of the source code has already been tested. Though CGF cannot prove the absence of bugs like symbolic execution does, the program is believed with a better confidence that it's bug-free.

The original AFL mutation operators work in a less structured way, primarily designed to test stateless programs, though. Lacking knowledge of earlier-sent messages or any state transition information, probably the fuzzer continues modifying non-ideal parts of the protocol message. Pham et al. propose AFLNET extending AFL to mitigate the issues [2]. AFLNET acts as a client of the Server Under Test (SUT) and iterates message variations to broaden the state space coverage. The corpus additionally collects message sequences that result in new state-feedback, their associated transitions are learned using an embedded state machine.

In brief, our tasks are:

1. Choosing appropriate seed inputs and dictionaries to increase both line and branch coverage;
2. Discover 3+ vulnerabilities of the FotBot server.

We discuss them in Section 2 and 3 respectively.

<u>You should be aware that spaces will be added to some of the code when you copy from this PDF file. They might need to be taken off before your run.</u>

[1] AFL injects instrumentation at compilation time, which can capture branch coverage and calculate the branch-taken hit counts.

3 EXPERIEMNT

First clone our GitHub repository. Change to the project root directory, run below to create the required Docker image.

```
> docker build . -t swen90006-assignment2
```

Start a container with:

```
> docker run -it swen90006-assignment2
```

Change directory to **~/fotbot**. Compile all executables as README.md instructs.

3.1 CAPTURING SEED INPUTS (OPTIONAL)

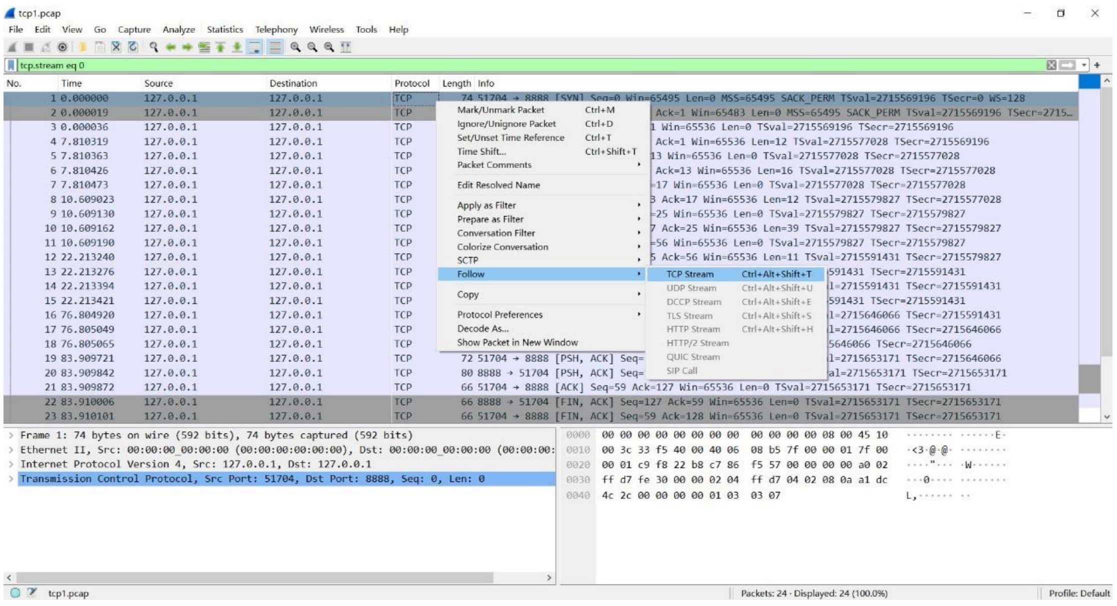
Simulate telecom service at the background. Start the SUT. tcpdump packet analyzer is used to capture traffics between the server and the client. Then start a telnet client to send requests. After client quits, kill tcpdump.

```
~/fotbot$ ./service 127.0.0.1 9999 &
~/fotbot$ ./fotbot 127.0.0.1 8888 127.0.0.1 9999 & echo ubuntu | sudo -S
    tcpdump -w tcp1.pcap -i lo port 8888 &
~/fotbot$ telnet 127.0.0.1 8888
<requests...>
~/fotbot$ sudo pkill tcpdump
```

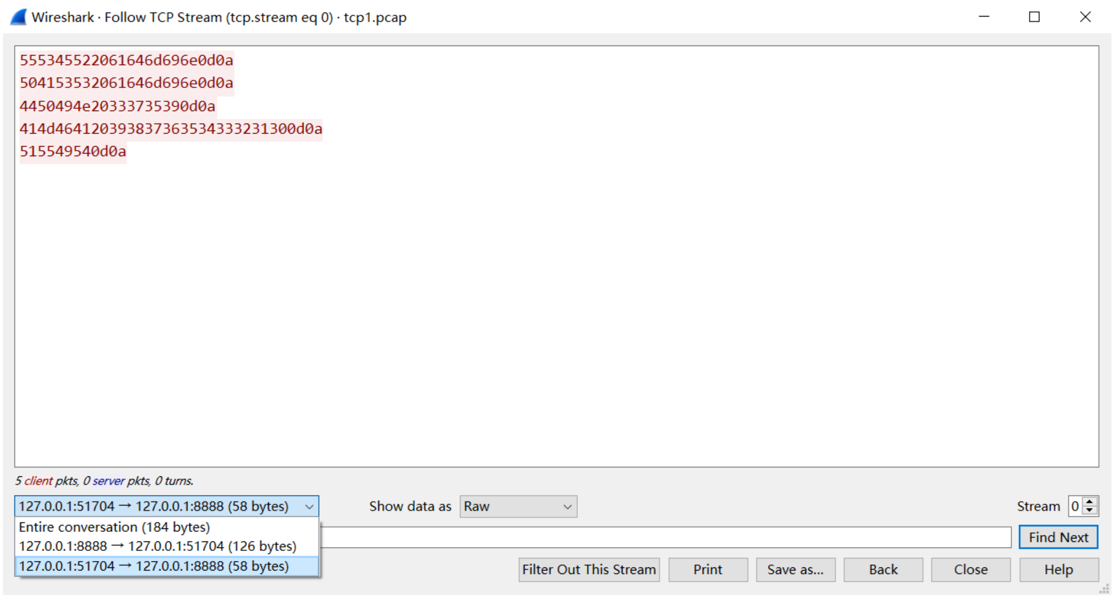
Now we copy the captured file outside the container.

```
> docker cp 8542d0:home/ubuntu/fotbot/tcp1.pcap .
```

Use Wireshark to extract TCP streams of only the requests -> server port 8888, output the request sequence as a seed input for AFLNet, Save the result as a raw data file, named **seed1.raw**. An example of using Wireshark is shown below in the next page.


tcp1.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Mark/Unmark Packet Ctrl+M
Ignore/Unignore Packet Ctrl+D
Set/Unset Time Reference Ctrl+T
Time Shift... Ctrl+Shift+T
Packet Comments
Edit Resolved Name
Apply as Filter
Prepare as Filter
Conversation Filter
Colorize Conversation
SCTP
Follow
Copy
Protocol Preferences
Decode As...
Show Packet in New Window
TCP Stream Ctrl+Alt+Shift+T
UDP Stream Ctrl+Alt+Shift+U
DCCP Stream Ctrl+Alt+Shift+E
TLS Stream Ctrl+Alt+Shift+S
HTTP Stream Ctrl+Alt+Shift+H
HTTP/2 Stream
QUIC Stream
SIP Call

```
tcp.stream eq 0

No.  Time       Source     Destination  Protocol Length Info
   1 0.000000   127.0.0.1  127.0.0.1    TCP      74 51704 → 8888 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2715569196 TSecr=0 WS=128
   2 0.000019   127.0.0.1  127.0.0.1    TCP      Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2715569196 TSecr=2715…
   3 0.000036   127.0.0.1  127.0.0.1    TCP      1 Win=65536 Len=0 TSval=2715569196 TSecr=2715569196
   4 7.810319   127.0.0.1  127.0.0.1    TCP      Ack=1 Win=65536 Len=12 TSval=2715577028 TSecr=2715569196
   5 7.810363   127.0.0.1  127.0.0.1    TCP      13 Win=65536 Len=0 TSval=2715577028 TSecr=2715577028
   6 7.810426   127.0.0.1  127.0.0.1    TCP      Ack=13 Win=65536 Len=16 TSval=2715577028 TSecr=2715577028
   7 7.810473   127.0.0.1  127.0.0.1    TCP      =17 Win=65536 Len=0 TSval=2715577028 TSecr=2715577028
   8 10.609023  127.0.0.1  127.0.0.1    TCP      3 Ack=17 Win=65536 Len=12 TSval=2715579827 TSecr=2715577028
   9 10.609130  127.0.0.1  127.0.0.1    TCP      =25 Win=65536 Len=0 TSval=2715579827 TSecr=2715579827
  10 10.609162  127.0.0.1  127.0.0.1    TCP      7 Ack=25 Win=65536 Len=39 TSval=2715579827 TSecr=2715579827
  11 10.609190  127.0.0.1  127.0.0.1    TCP      =56 Win=65536 Len=0 TSval=2715579827 TSecr=2715579827
  12 22.213240  127.0.0.1  127.0.0.1    TCP      5 Ack=56 Win=65536 Len=11 TSval=2715591431 TSecr=2715579827
  13 22.213276  127.0.0.1  127.0.0.1    TCP      591431 TSecr=2715591431
  14 22.213394  127.0.0.1  127.0.0.1    TCP      l=2715591431 TSecr=2715591431
  15 22.213421  127.0.0.1  127.0.0.1    TCP      591431 TSecr=2715591431
  16 76.804920  127.0.0.1  127.0.0.1    TCP      l=2715646066 TSecr=2715591431
  17 76.805049  127.0.0.1  127.0.0.1    TCP      l=2715646066 TSecr=2715646066
  18 76.805065  127.0.0.1  127.0.0.1    TCP      5646066 TSecr=2715646066
  19 83.909721  127.0.0.1  127.0.0.1    TCP      72 51704 → 8888 [PSH, ACK] Seq=   l=2715653171 TSecr=2715646066
  20 83.909842  127.0.0.1  127.0.0.1    TCP      80 8888 → 51704 [PSH, ACK] Seq=   al=2715653171 TSecr=2715653171
  21 83.909872  127.0.0.1  127.0.0.1    TCP      66 51704 → 8888 [ACK] Seq=59 Ack=127 Win=65536 Len=0 TSval=2715653171 TSecr=2715653171
  22 83.910006  127.0.0.1  127.0.0.1    TCP      66 8888 → 51704 [FIN, ACK] Seq=127 Ack=59 Win=65536 Len=0 TSval=2715653171 TSecr=2715653171
  23 83.910101  127.0.0.1  127.0.0.1    TCP      66 51704 → 8888 [FIN, ACK] Seq=59 Ack=128 Win=65536 Len=0 TSval=2715653171 TSecr=2715653171

> Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 51704, Dst Port: 8888, Seq: 0, Len: 0

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 3c 33 f5 40 00 40 06  08 b5 7f 00 00 01 7f 00   ·<3·@·@· ········
0020  00 01 c9 f8 22 b8 c7 86  f5 57 00 00 00 00 a0 02   ····"··· ·W······
0030  ff d7 fe 30 00 00 02 04  ff d7 04 02 08 0a a1 dc   ···0···· ········
0040  4c 2c 00 00 00 00 01 03  03 07                     L,······ ··

tcp1.pcap        Packets: 24 · Displayed: 24 (100.0%)        Profile: Default
```



```
555345522061646d696e0d0a
504153532061646d696e0d0a
4450494e20333735390d0a
414d4641203938373635343332313300d0a
515549540d0a
```

5 client pkts, 0 server pkts, 0 turns.

```
127.0.0.1:51704 → 127.0.0.1:8888 (58 bytes)
Entire conversation (184 bytes)
127.0.0.1:8888 → 127.0.0.1:51704 (126 bytes)
127.0.0.1:51704 → 127.0.0.1:8888 (58 bytes)
```

Finally, copy **seed1.raw** back into the container.

> docker cp ./seed1.raw 8542d0:home/ubuntu/results/seed_corpus/seed1.raw

Since we have done all these steps, you can skip them safely. The seed is already present at **~/results/seed_corpus/** folder. You can also find it in Appendix 1.

3.2 Dictionaries

AFLNet gives the option of more structured descriptions of the underlying syntax elements [3]. The grammar seeds a syntax-aware fuzzing process through defined keywords, magic headers, or other special tokens associated with the targeted data type.

We write the FotBot command prefixes in the dictionary file, it can be found at **~/results/others/fotbot.dict**.

3.3 Fuzzing Process

Run below. The ALFNET outputs in **~/results/others/out-fotbot**.[2] The file **reboot.sh** restarts the service between iterations.

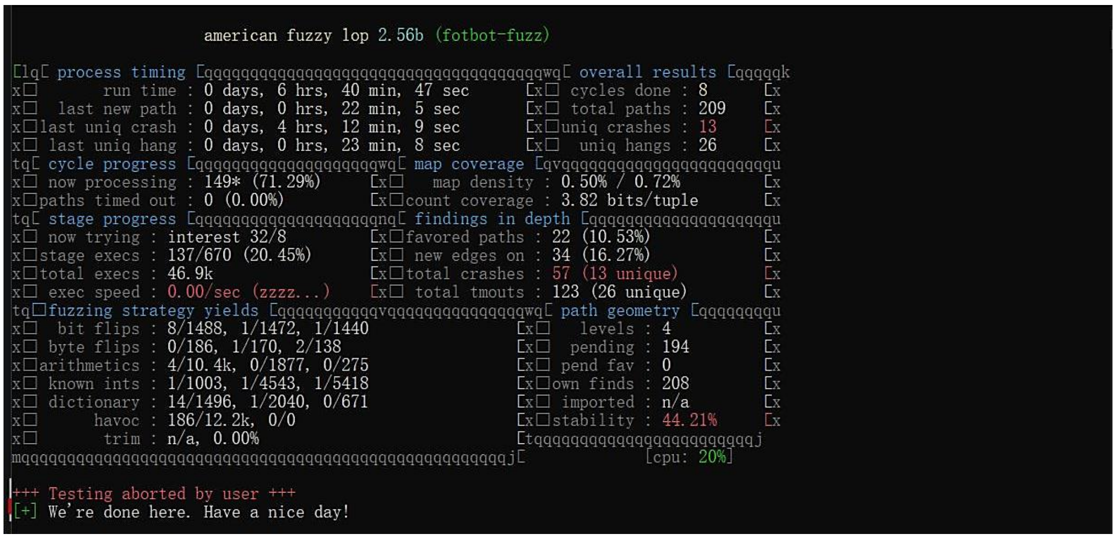
```
~/fotbot$ chmod +x ~/results/others/reboot.sh && chmod -R +r
    ~/results/seed_corpus
```

```
~/fotbot$ afl-fuzz -c ~/results/others/reboot.sh -i ~/results/seed_corpus -
    o ~/results/others/out-fotbot -N tcp://127.0.0.1/8888 -x
    ~/results/others/fotbot.dict -P FOTBOT -D 10000 -q 3 -s 3 -
    EKRF ./fotbot-fuzz 127.0.0.1 8888 127.0.0.1 9999
```

We've made one slight change in **~/fotbot/fotbot.c** source code to control the randomness. Nondeterministic behaviours are not good for AFLNet. In the function **int generatePIN(int N)**, we replace Line 169 with **return 3759;**. Consequently, DPIN is fixed at 3759. We use 3759 in seed corpus so the AFLNet won't face difficulty verifying the MFA (DPIN).

Below is a screenshot of AFLNet. We run it for 6 hours 40 minutes. Have done 8 cycles, went through 209 total (state) paths, and encountered 13 unique crashes. We also put all non-crashing inputs generated by AFLNet in **~/results/generated_corpus**.

[2] We zip this folder before pushing it to our GitHub repository because some AFLNet result files names are not compatible with Windows file system. Unzip it in your Docker container if you want to examine or use it in Section 2.4 and 2.5.



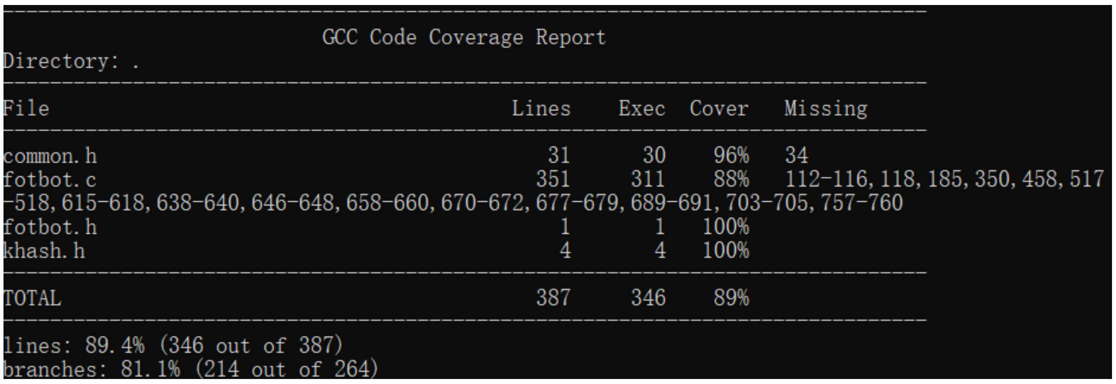
```
                     american fuzzy lop 2.56b (fotbot-fuzz)

[lq[ process timing [qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqwq[ overall results [qqqqqk
x□        run time : 0 days, 6 hrs, 40 min, 47 sec       [x□ cycles done : 8      [x
x□  last new path : 0 days, 0 hrs, 22 min, 5 sec         [x□ total paths : 209    [x
x□last uniq crash : 0 days, 4 hrs, 12 min, 9 sec         [x□uniq crashes : 13     [x
x□ last uniq hang : 0 days, 0 hrs, 23 min, 8 sec         [x□  uniq hangs : 26     [x
tq[ cycle progress [qqqqqqqqqqqqqqqqqqqqqwq[ map coverage [qvqqqqqqqqqqqqqqqqqqqqqqqqu
x□ now processing : 149* (71.29%)      [x□   map density : 0.50% / 0.72%          [x
x□paths timed out : 0 (0.00%)          [x□count coverage : 3.82 bits/tuple        [x
tq[ stage progress [qqqqqqqqqqqqqqqqqqqqqqnq[ findings in depth [qqqqqqqqqqqqqqqqqqqqqu
x□ now trying : interest 32/8          [x□favored paths : 22 (10.53%)            [x
x□stage execs : 137/670 (20.45%)       [x□ new edges on : 34 (16.27%)            [x
x□total execs : 46.9k                  [x□total crashes : 57 (13 unique)         [x
x□ exec speed : 0.00/sec (zzzz...)     [x□ total tmouts : 123 (26 unique)        [x
tq□fuzzing strategy yields [qqqqqqqqqqqvqqqqqqqqqqqqqqqqwq[ path geometry [qqqqqqqqu
x□  bit flips : 8/1488, 1/1472, 1/1440                 [x□   levels : 4          [x
x□ byte flips : 0/186, 1/170, 2/138                    [x□  pending : 194        [x
x□arithmetics : 4/10.4k, 0/1877, 0/275                 [x□ pend fav : 0          [x
x□ known ints : 1/1003, 1/4543, 1/5418                 [x□own finds : 208        [x
x□ dictionary : 14/1496, 1/2040, 0/671                 [x□ imported : n/a        [x
x□      havoc : 186/12.2k, 0/0                         [x□stability : 44.21%     [x
x□       trim : n/a, 0.00%                             [tqqqqqqqqqqqqqqqqqqqqqqqqqj
mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj[             [cpu: 20%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

3.4 CODE COVERAGE

We compute the code coverage with the gcovr tool. We write a shell script file **calc_covr.sh** (Appendix 2) for calculating the cumulative code coverage of all our generated inputs. It takes one command-line argument, the directory name that store the inputs. Please remind to first compile executables, unzip the compressed file, and make sure currently you are in **~/fotbot** directory!

```
~/fotbot$ chmod +x ~/results/others/calc_covr.sh
~/fotbot$ ~/results/others/calc_covr.sh ~/results/others/out-
      fotbot/replayable-queue/
~/fotbot$ gcovr -r . -s
```

We achieve 89.4% line coverage and 81.1% branch coverage. The percentage not covered are mostly branches for handling issues like socket connection refuse.



```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
common.h                                      31      30    96%   34
fotbot.c                                     351     311    88%   112-116,118,185,350,458,517
-518,615-618,638-640,646-648,658-660,670-672,677-679,689-691,703-705,757-760
fotbot.h                                       1       1   100%
khash.h                                        4       4   100%
------------------------------------------------------------------------------
TOTAL                                        387     346    89%
------------------------------------------------------------------------------
lines: 89.4% (346 out of 387)
branches: 81.1% (214 out of 264)
```

4 VULNERABILITIES

4.1 CRASHES

Several crashes are detected through the fuzzing process. We observe that the server will crash when a user attempts to call UPDP twice to update their password for a second time.

To find the root cause of this vulnerability, we write the shell script **asan.sh** (Appendix 3, and see how we replay it) to **afl-replay** the crashes inputs with the ASan address sanitizer. You should follow these instructions:

```
~/fotbot$ sudo apt-get update && sudo apt install llvm -y
~/fotbot$ chmod +x ~/results/others/asan.sh
~/fotbot$ ~/results/others/asan.sh ~/results/others/out-fotbot/replayable-
    crashes/ ~/results/others/out-asan-crash/
```

asan.sh takes two command-line arguments, the first one is where inputs are, the second one is the output directory. We set ASAN_OPTIONS=detect_leaks=0 to ignore the "benign" memory leaks. LLVM is employed to show the exact source line number of the fault. And it shows us that there is a "double free" at Line 326 in the source code (See Appendix 4 example). With this knowledge, we study the source code and start to understand this bug.

The BUG is:

When updating the user's password, the code first frees the memory of the old password on Line 326 by calling **free()** on the pointer **user->password**, and then assigns the new password by letting the pointer **user->password** point to **tokens[0]** on Line 327, which contains the new password. The problem is that, after these two steps, **tokens[0]** is freed by the function **freeTokens()** on Line 333, which means the memory given to the pointer **user->password** is already freed now. So the next time when we update the user's password, the code will again first try to free the memory of the old password by calling **free()** on the pointer **user->password**, but here this pointer is already freed. Hence the "double free" problem and the server crashes.

We upload inputs that trigger a crash **~/results/pocs/crash.zip**, and sanitizer output **~/results/others/out-asan-crash.zip** to our GitHub repository.

4.2 HEAP-USE-AFTER-FREE

We also found a critical memory fault, Use-After-Free, on Line 89 of the source code. This vulnerability was also found using the address sanitizer when we continue to look for vulnerabilities by again replaying with address sanitizer the generated inputs in the **replayable-queue** folder generated by the fuzzing process. The vulnerability is identified in **id:000123,src:000096,op:havoc,rep:16** (See Appendix 5).

```
~/fotbot$ sudo apt-get update && sudo apt install llvm -y
~/fotbot$ chmod +x ~/results/others/asan.sh
~/fotbot$ ~/results/others/asan.sh ~/results/others/out-fotbot/replayable-
    queue/ ~/results/others/out-asan-noncrash/
```

The BUG is:

The code tries to use the password pointer on Line 89. But in the case that a user has updated their password once (not twice so server will not crash), the password pointer is already freed as in the discussion in Section 3.1, therefore triggering a Use-After-Free memory fault when the pointer is referenced again on Line 89.

The impact of this memory fault is non-trivial. The purpose of the function using this pointer on Line 89 is to check if the password given by a user is correct, which will be invoked when a user logs in. Now since the password pointer is already freed, the server loses the content that the pointer is pointing to. This means, when a user updates their password, logs out, and tries to log in back, it may not be successful.

To reproduce this one: replay inputs in **~/results/pocs/use-after-free.zip**. And we upload **~/results/others/out-asan-noncrash.zip** to our GitHub repository.

4.3 UNAUTHORIZED ACCESS & DATA STEALING

C’s **free()** method deallocates the space of a block previously allocated, and freeing it alters the contents of the block [4]. After that, the part of memory can be allocated to other variables. Section 3.2’s use-after-free led us to consider the possibility that this fault could be exploited by an attacker to acquire unauthorized access. We are also aware that any user can add a friend many times even if that user already has been added previously. In Line 485, **strdup(char*)** function allocates memory for the name of the newly added friend. In case there exists one user with a short name, for example "B", the attacker could repeatedly send "ADDF B" to force the server to allocate a lot of memory that is short in length. If some user, let’s call the person bobby, have just updated the password and logged out, then that part of memory to which **bobby->pointer** points will remain uninitialized and ready to be reused. A series of "ADDF B" requests are very likely to allocate string **“B\0”** on that memory address, then **bobby->pointer** points to "B" so the attacker could use it to log in user bobby's account if MFA is not enabled. This is very dangerous - simple "GETS B" will steal all bobby's step data.

Now let’s try to reproduce this vulnerability. Depending on the compiler, and hardware, and platform, our method may fail sometimes in theory. Anyway, just give it a trial, it works well and really steals others’ information on our machines.

Please replay the input **~/results/pocs/unauthorized_access.replay**.

The input follows:

```
USER admin
PASS admin
DPIN 3759
REGU b,passwordb
REGU evil,evil
LOGO
USER b
PASS passwordb
UPDA 1,2,3,4,5
UPDP passwordbnew,passwordbnew
LOGO
USER evil
PASS evil
```

```
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
ADDF b
LOGO
USER b
PASS b
GETS b
QUIT
```

5 REFLECTION & CONCLUSION

In conclusion, we learned a lot in this assignment. From learning to set up the fuzzer, to continuously making improvements to cover more code and find more vulnerabilities, we not only gained more knowledge and experience on using fuzzing, but also improved collaboration and communication skills.

In recap, we made following improvements from the initial setup to find more vulnerabilities: (1) we fix the value of MFA PIN in the source code so that the fuzzer does not need to guess a random number. (2) we use a dictionary containing the commands for the fuzzer to randomly insert, otherwise it is hard for the fuzzer to guess the correct command names. (3) we use a single longer seed input containing more commands, rather than several seed inputs covering different commands. (4) We have successfully discovered ALL 4 types of security vulnerabilities mentioned in **README.md**.

In our experiments, all the aforementioned three points work pretty well to our favour. In particular, fixing the pin really saves a lot of time as we expect. To our surprise, what does not work so well is (2) without (3). We discovered that even with a dictionary in (2), it is still difficult or it takes a long time for the fuzzer to generate inputs that reach interesting states by randomly inserting commands from dictionary. With (3) it works much better and helps us discover more vulnerabilities.

Finally, one clear advantage of the chosen fuzzer, AFLNet, is that it uses the states to guide the fuzzing, making it ideal in this setting compared with other fuzzers which do not consider states of the program. However, just like other CGFs, AFLNet is still not proving the absence of bugs like symbolic execution does.

Some possible features that we think may improve the experience of the user is:

1. Sometimes multiple crashes are for the same reason/bug, if the fuzzer gives 100 crashes after a long compaign, it is hard to work through each one of the crash to see if they are all from the same bug or if there’s any other bug

in some of them. It would be great if there could be some methods to cluster them in a way that crashes for the same reason are classified in the same cluster.

2. As we experienced in (2), simply using dictionaries may still be difficult to help the fuzzer reaching interesting states. One possible reason is that the fuzzer still uses a portion of time to do byte level or other kinds of mutations. To each application scenario, the testing team may wish the fuzzer to spend different amounts of time to do each kind of mutations. It would be nice if an interface would be given such that the user can just pass some parameters to give weights to how many proportions of time they want to spend on different kinds of mutations.

6 REFERENCES

[1] “American fuzzy lop (2.52b).” https://lcamtuf.coredump.cx/afl/. [Accessed: 7-Oct-2022].

[2] V.-T. Pham, M. Bohme, and A. Roychoudhury, “AFLNET: A greybox fuzzer for network protocols,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 460–465, 2020.

[3] “Fuzzing with AFL-fuzz.” https://afl-1.readthedocs.io/en/latest/fuzzing.html. [Accessed: 8-Oct-2022].

[4] “Freeing after malloc (the GNU C library).” https://www.gnu.org/software/libc/manual/html_node/Freeing-after-Malloc.html. [Accessed: 20-Oct-2022].

7 APPENDIX

7.1 ~/RESULTS/SEED_CORPUS/SEED1.RAW

```
USER admin
PASS admin
DPIN 3759
REGU andy andy
ADDF andy
REGU alfred,alfred
ADDF alfred
GETF
GETS
GETS admin
UPDP admin,new
UPDP new,new
LOGO
USER andy
PASS andy
UPDA 30,50,100,100
GETS andy
AMFA 1234567890
LOGO
USER andy
PASS andy
DPIN 3759
LOGO
QUIT
```

7.2 ~/RESULTS/OTHERS/CALC_COVR.SH

```
#!/bin/bash

cd $WORKDIR/fotbot

rm -f fotbot.gcda fotbot.gcno

make clean all > /dev/null 2>&1

for file in $1/*
do
    echo 'Running' $(basename $file)
    {
    pkill -9 service
    ./service 127.0.0.1 9999 &
    ./fotbot-gcov 127.0.0.1 8888 127.0.0.1 9999 &
    aflnet-replay $file FOTBOT 8888
    } > /dev/null 2>&1
done
```

7.3 ~/RESULTS/OTHERS/ASAN.SH

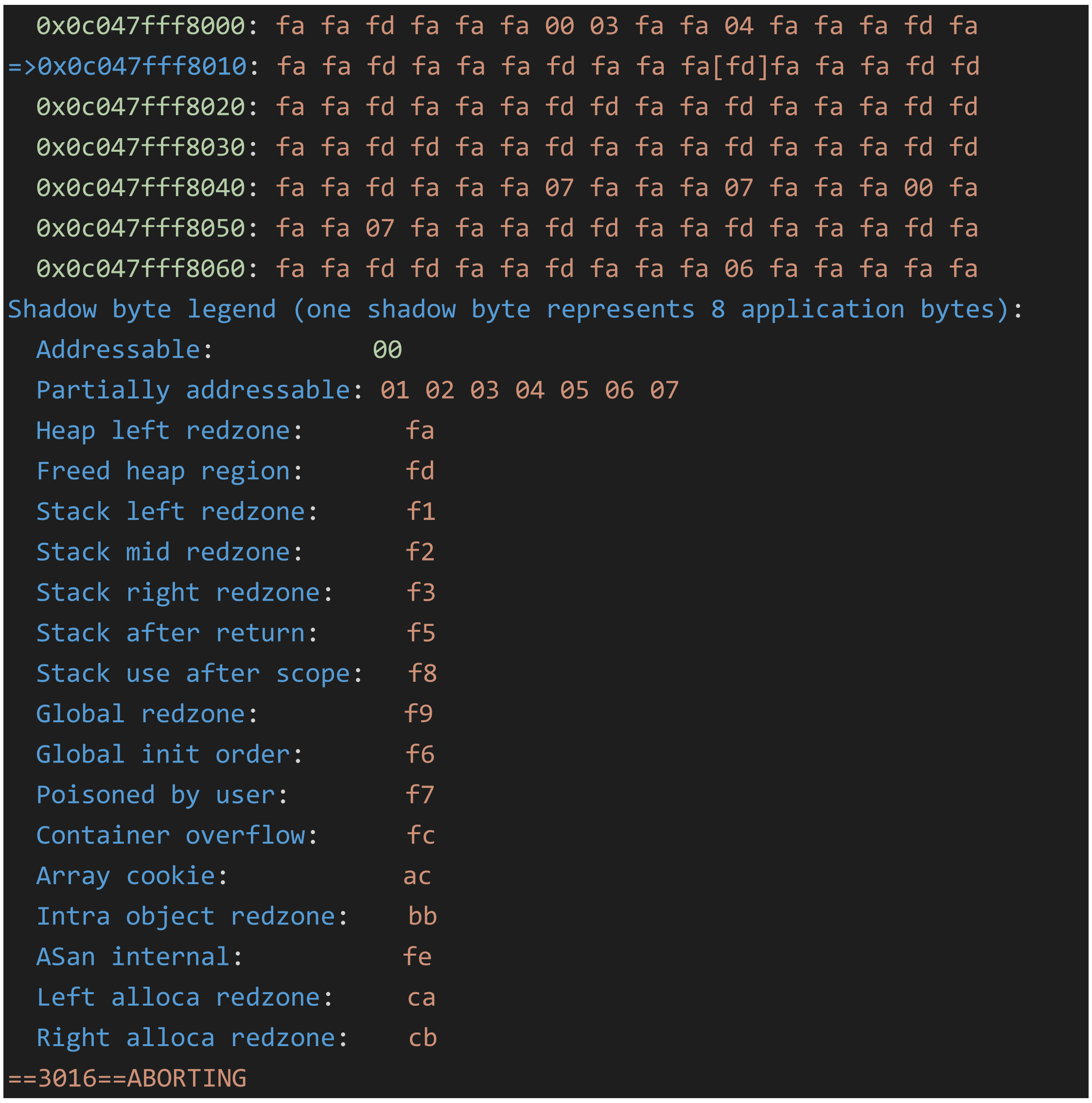
```
#!/bin/bash

cd $WORKDIR/fotbot

rm -rf $2
mkdir $2

for file in $1/id*
do
    echo 'Running' $(basename $file)
    pkill -9 service
    ./service 127.0.0.1 9999 > /dev/null 2>&1 &
    ASAN_OPTIONS=detect_leaks=0 ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-
symbolizer ./fotbot-asan 127.0.0.1 8888 127.0.0.1 9999 > tmp.log 2>&1 &
    pid=$!
    aflnet-replay $file FOTBOT 8888 > /dev/null 2>&1

    wait $pid
    status=$?

    if [ ! $status -eq 0 ]
    then
        cat tmp.log > $2/$(basename $file)
    fi
done

rm -f tmp.log
```

7.4 ~/RESULTS/OTHERS/OUT-ASAN-CRASH/ID:000000,SIG:06,SRC:000096,OP:HAVOC,REP:16

```
=================================================================
==5939==ERROR: AddressSanitizer: attempting double-free on
0x6020000000d0 in thread T0:
    #0 0x4d9d40 in __interceptor_free.localalias.0
(/home/ubuntu/fotbot/fotbot-asan+0x4d9d40)
    #1 0x51588b in fbUPDP /home/ubuntu/fotbot/fotbot.c:326:7
    #2 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #3 0x7fa30bb31c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)
    #4 0x41a059 in _start (/home/ubuntu/fotbot/fotbot-asan+0x41a059)

0x6020000000d0 is located 0 bytes inside of 4-byte region
[0x6020000000d0,0x6020000000d4)
freed by thread T0 here:
    #0 0x4d9d40 in __interceptor_free.localalias.0
(/home/ubuntu/fotbot/fotbot-asan+0x4d9d40)
    #1 0x514bb2 in freeTokens /home/ubuntu/fotbot/fotbot.c:214:5
    #2 0x515a2b in fbUPDP /home/ubuntu/fotbot/fotbot.c:333:5
    #3 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #4 0x7fa30bb31c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)

previously allocated by thread T0 here:
    #0 0x436730 in __interceptor_strdup (/home/ubuntu/fotbot/fotbot-
asan+0x436730)
    #1 0x51276a in strSplit /home/ubuntu/fotbot/./common.h:83:22
    #2 0x5155d3 in fbUPDP /home/ubuntu/fotbot/fotbot.c:316:14
    #3 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #4 0x7fa30bb31c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)

SUMMARY: AddressSanitizer: double-free (/home/ubuntu/fotbot/fotbot-
asan+0x4d9d40) in __interceptor_free.localalias.0
==5939==ABORTING
```

7.5 ~/RESULTS/OTHERS/OUT-ASAN-NONCRASH/ID:000123,SRC:000096,OP:HAVOC,REP:16

```
=================================================================
==3016==ERROR: AddressSanitizer: heap-use-after-free on address
0x6020000000d0 at pc 0x0000004af947 bp 0x7ffc16eb3570 sp 0x7ffc16eb2d20
READ of size 1 at 0x6020000000d0 thread T0
    #0 0x4af946 in __interceptor_strcmp.part.253
(/home/ubuntu/fotbot/fotbot-asan+0x4af946)
    #1 0x513246 in isPasswordCorrect /home/ubuntu/fotbot/fotbot.c:89:12
    #2 0x514eab in fbPASS /home/ubuntu/fotbot/fotbot.c:257:10
    #3 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #4 0x7fb2f73e8c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)
    #5 0x41a059 in _start (/home/ubuntu/fotbot/fotbot-asan+0x41a059)

0x6020000000d0 is located 0 bytes inside of 4-byte region
[0x6020000000d0,0x6020000000d4)
freed by thread T0 here:
    #0 0x4d9d40 in __interceptor_free.localalias.0
(/home/ubuntu/fotbot/fotbot-asan+0x4d9d40)
    #1 0x514bb2 in freeTokens /home/ubuntu/fotbot/fotbot.c:214:5
    #2 0x515a2b in fbUPDP /home/ubuntu/fotbot/fotbot.c:333:5
    #3 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #4 0x7fb2f73e8c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)

previously allocated by thread T0 here:
    #0 0x436730 in __interceptor_strdup (/home/ubuntu/fotbot/fotbot-
asan+0x436730)
    #1 0x51276a in strSplit /home/ubuntu/fotbot/./common.h:83:22
    #2 0x5155d3 in fbUPDP /home/ubuntu/fotbot/fotbot.c:316:14
    #3 0x51b225 in main /home/ubuntu/fotbot/fotbot.c:740:14
    #4 0x7fb2f73e8c86 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21c86)

SUMMARY: AddressSanitizer: heap-use-after-free
(/home/ubuntu/fotbot/fotbot-asan+0x4af946) in
__interceptor_strcmp.part.253
Shadow bytes around the buggy address:
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```



```
  0x0c047fff8000: fa fa fd fa fa fa 00 03 fa fa 04 fa fa fa fd fa
=>0x0c047fff8010: fa fa fd fa fa fa fd fa fa fa[fd]fa fa fa fd fd
  0x0c047fff8020: fa fa fd fa fa fa fd fd fa fa fd fa fa fa fd fd
  0x0c047fff8030: fa fa fd fd fa fa fd fa fa fa fd fa fa fa fd fd
  0x0c047fff8040: fa fa fd fa fa fa 07 fa fa fa 07 fa fa fa 00 fa
  0x0c047fff8050: fa fa 07 fa fa fa fd fd fa fa fd fa fa fa fd fa
  0x0c047fff8060: fa fa fd fd fa fa fd fa fa fa 06 fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==3016==ABORTING
```