CSE 543
Computer Security
Milestone 2
MQTT Fuzzing using AFLNET
11/29/2020

Team - 3
Sneha Suhitha Galiveeti - ssg163@psu.edu
Pranitha Malae - pzm5372@psu.edu

GOAL:

The goal of the proposed system is to identify the vulnerabilities present in the MQTT protocol by modifying the AFLNET fuzzer.

Design Overview:

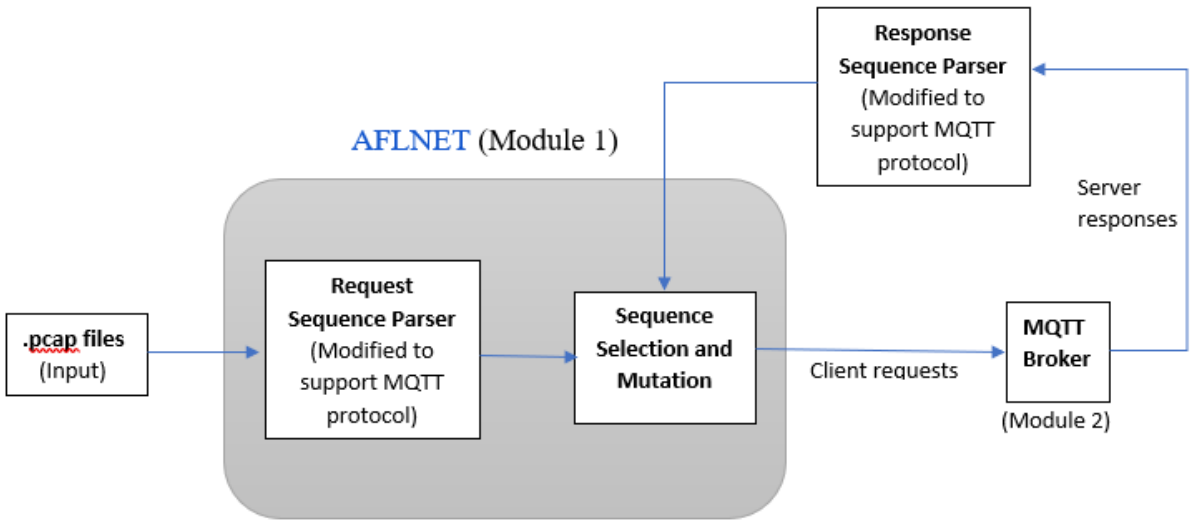

Figure 1: Overview of AFLNET

Figure 1 gives the high-level overview of the AFLNET[1] Fuzzer with the modifications proposed.

Following are the major components of the AFLNET Fuzzer:

1) **Input files:** AFLNET takes .pcap files as input, which capture the traffic packet information exchanged between the client and the server.
2) **Request Sequence Parser:** This component of AFLNET extracts the request packet information from the traffic and creates corpus of sequences to send as requests to server. The proposed system, targets to modify this component to recognize the MQTT message packets and parse them accordingly for the AFLNET to support MQTT client requests.
3) **Sequence Selection and Mutation:** This component mutates the existing request sequences and chooses the next request sequence to be sent to the server.
4) **MQTT Broker:** This is the target server which is the MQTT broker. The server just receives the requests from AFLNET (client) and sends out the responses back to AFLNET.
5) **Response Sequence Parser:** This component gets the responses from the server, parses the packets received and sends the information to the state machine learning which learns about the server state, based on the information received. The proposed system, targets to modify this component to support MQTT broker responses.

Implementation Details and Evaluation Setup:

The proposed system is implemented on a X-64bit system with Ubuntu16.04 and 4GB RAM.

The proposed system is built using two softwares which are Mosquitto 1.5.5[2] and the AFLNET Fuzzer[1].

As mentioned in the figure 1, Module 1 is the AFLNET software which has all the components described above. The components which were modified are the Request Sequence Parser and Response Sequence Parser. These were modified by implementing two new functions ‘extarct_requests_mqtt()’ and ‘extract_response_codes_mqtt()’. Both these functions are introduced to parse the buffer and extract requests and responses, respectively to support MQTT protocol. Mosquitto 1.5.5 is the implementation of Module 2. This is the target on which fuzzing is done.

The setup for these two softwares is clearly explained in the README file of this [3] Github repository.

The evaluation metrics for the proposed system are number of hangs, number of crashes and code coverage.

We do not have any baseline for evaluation metrics but some vulnerabilities in MQTT protocol were detected by using MQTT specific fuzzers in these papers [4], [5].

Evaluation Results:

The results to the proposed system are the ones generated after the fuzzing of MQTT protocol. This was implemented in two ways. One way is by fuzzing the MQTT protocol in stateless mode and the other way is by fuzzing the MQTT protocol considering the states of the server.

Result 1:

In the communication between Publisher and the broker, the broker will not maintain the state and thereby MQTT protocol is stateless with one type of client (publisher). So, the proposed system also does fuzzing in stateless mode. Figure 2 shows the results generated from this type of fuzzing.

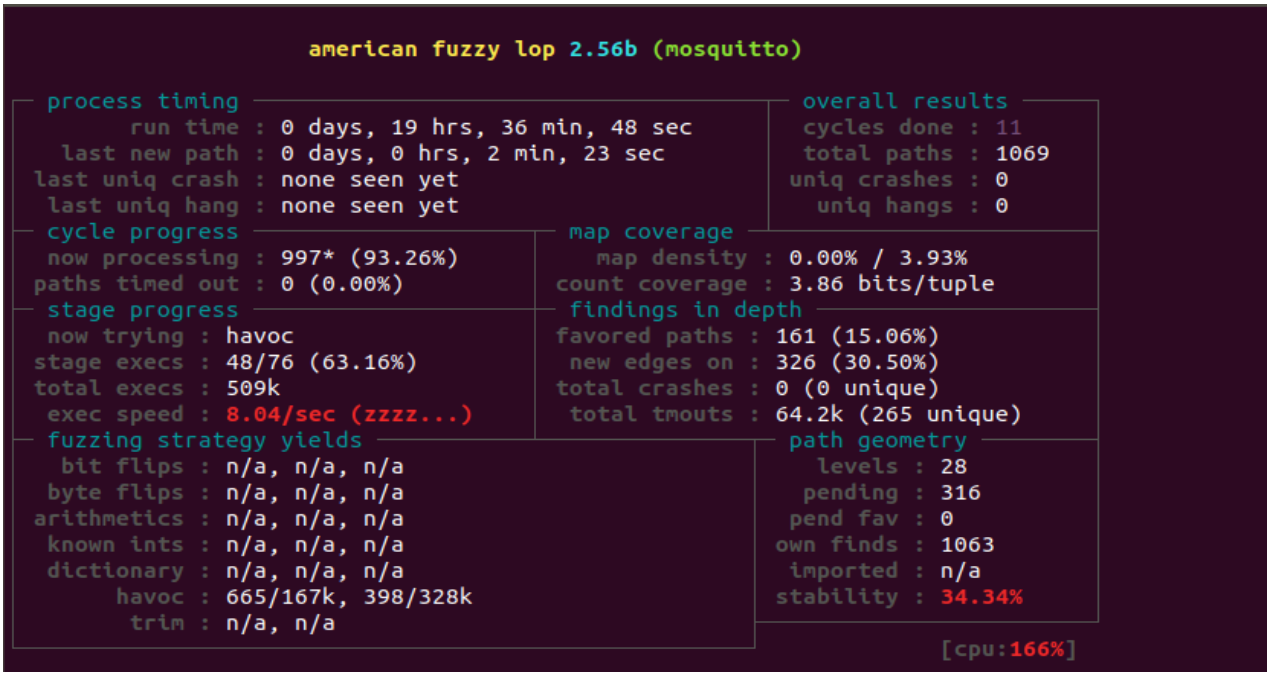

Figure 2: Results from stateless fuzzing

As shown in the figure 2, the fuzzer executed for 19 hours and 36 mins. No crashes and hangs were detected. The code coverage was 3.93%.

Result 2:

According to the MQTT protocol, in the other way of communication i.e., between subscriber and the broker, the broker maintains the state. Therefore, fuzzing on the Mosquitto broker by considering states is also implemented. Figure 3 shows the results from this fuzzing.

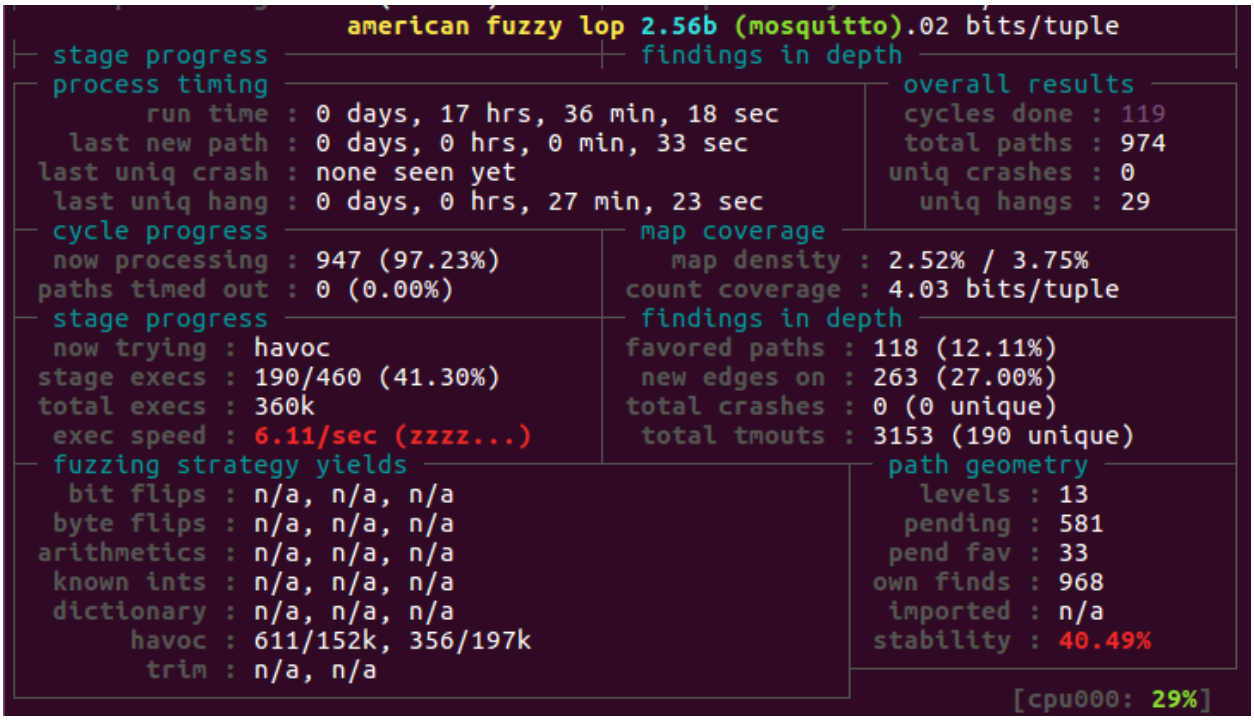

Figure 3: Results from fuzzing considering the states of broker

As shown in the figure 3, the fuzzer executed for 17 hours and 36 mins. 29 unique hangs were detected with zero crashes. The code coverage was 3.75%.

Analysis of results 1 and 2:

- The .pcap files which were generated using tcpdump did not capture all the packets exchanged between the clients and the broker:
    - Traffic between multiple connections was not registered. For example, if one publisher and one subscriber were connected to the broker, only one of the communications is registered.
    - Traffic between multiple connections of same type were also not registered. For example, with two subscribers and one broker.
    - Acknowledgement messages by server were also not registered, but we are not sure if tcpdump is unable to capture them or if the broker is not sending the response.
- The code coverage value is very less because of the following:
    - The .pcap files did not cover all the packet types.
    - The architecture of the AFLNET. AFLNET will not exactly behave as the MQTT client, for example, when a subscriber subscribes for a topic, MQTT protocol sends a connect

request immediately followed by subscribe message. This sequence of messages cannot be guaranteed in AFLNET due to the mutations and random selection of client requests.

- According to the MQTT protocol, few message types act as both requests and responses based on the scenario. But, AFLNET can only classify a message type as a request or a response but cannot dynamically change based on the scenario.
- All the detected 29 hangs were due to the one same issue. When the modified AFLNET (client) sends the improper (mutated) requests to the broker, the broker, according to the MQTT protocol, cannot recognize such packets and disconnects with the client. But the AFLNET will not recognize this and keeps sending new requests on the same client name to the broker resulting in a hang.

Result 3:

We have explicitly introduced a bug into the Mosquitto software and executed the fuzzing to test if the modified AFLNET was able to detect the bug. Figure 4 shows the results.

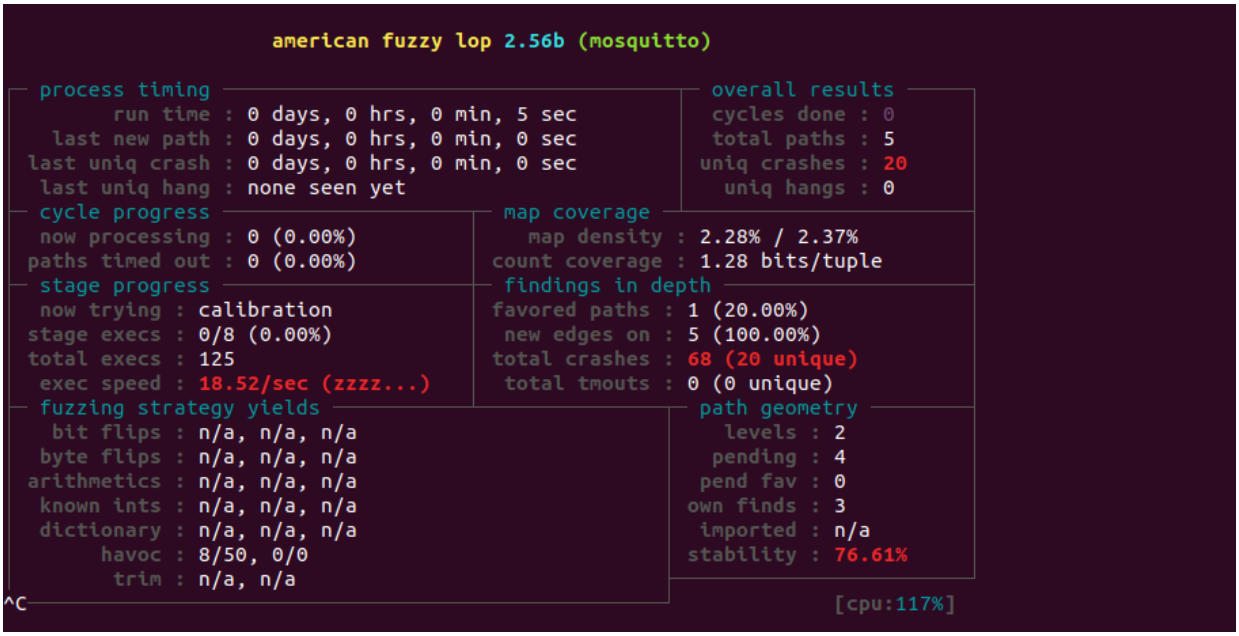

Figure 4: Fuzzing on buggy Mosquitto broker

As shown in the figure 4, the modified AFLNET executed for less than a minute and detected 20 unique crashes, with a very a less code coverage. The bug introduced was a null pointer read, placed at a point where the server disconnects the client due to the improper packet header of the request. This bug was successfully detected by the modified AFLNET.

References:

1. https://github.com/aflnet/aflnet.git
2. https://github.com/eclipse/mosquitto
3. https://github.com/SuhithaG/CSE543-Project3
4. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7570995/pdf/sensors-20-05194.pdf
5. http://downloads.hindawi.com/journals/wcmc/2018/8261746.pdf
6. https://mboehme.github.io/paper/ICST20.AFLNet.pdf
7. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
8. https://www.hivemq.com/mqtt-essentials/