1. FUZZ TESTING IN FINANCIAL TRANSACTION SYSTEMS

-BY: Parth Kasurde Mohammed Aziz Harsh Shroff

2. ABSTRACT

Given the fast-growing fintech platforms, the issue of securing the financial APIs becomes more important. Financial APIs are an intrinsic part of digital transactions but even small bugs – i.e. accepting a negative amount – may cause severe vulnerabilities. An API that models a financial transaction is tested using an automated fuzz testing framework – Boofuzz – to test the robustness of this mock API to malformed, edge-case, or malicious inputs. We built a small Flask-based payment server and did fuzzing in order to emulate unexpected request behavior. Our purpose was to catch unhandled exceptions, sanitization validation of input, and inspect API responses at high amounts of randomized input. Significant gaps of validation are observed from our results showing acceptance of invalid data and unproper handling of malformed JSON, proving the usefulness of active testing methodologies in fintech circles.

My personal contributions were the research on the fuzz testing techniques in the financial system and the introduction of improvements to the fuzzing payloads. I backed the integration of Boofuzz with the mock Flask API, remotely coordinated testing, and ensured proper response behavior through curl-backed manual testing. I also contributed to the improvements of documentation, helped with the design of a part of the experimental evaluation strategy, and joined the debugging efforts for the field-specific anomalies of the fuzz script. These contributions enhanced my knowledge in structured fuzzing and its implications on designing secure financial API.

3. INTRODUCTION

The backbone of digital payments in the contemporary fintech systems is the financial transaction APIs. These APIs typically process confidential information such as account numbers, amounts involved in different transactions as well as currency codes. It is necessary to ensure their resilience against abnormal or malign input. While normal functional testing ensures behaviour in contexts where it is intended to work, it does not map edge cases and adversarial inputs that might take advantage of logical loopholes or crash the system.

This gap is covered by fuzz testing that automates the creation of unexpected or malformed inputs to see how a system responds to them. The current state-of-the-art is composed of, among others, such tools as AFL, Peach, and Boofuzz – each excellent in either the binary fuzzing or the structured protocol fuzzing. In the world of fintech, fuzz testing continues to be neglected even though it has a strong potential of detecting weaknesses in transaction processing logic. That gap is partly filled up by this project with practical application of fuzzing in simulated financial

world.

4. BACKGROUND

Fuzz testing is a black-box testing method of software, which distributes random or malicious inputs to a target program with a purpose of finding unexpected behaviour, crash, or vulnerability. Contrary to the unit testing or integration testing, the fuzzing is wired to break the system through non-deterministic stimuli. When it is used in APIs, this can spot problems like unhandled exceptions, poor validation, or logic errors.

We used Boofuzz, which is a fuzzing framework for python with the capability of session-based fuzzing which can be customised. Boofuzz helps the testers to define input structures and mutate them in a controlled fashion and can be used to log responses. It is especially good for web services or API end points that are waiting for structured data in the form of HTTP/JSON.

A light weight Python web framework known as Flask was chosen to emulate a minimalistic financial payment API. Its simplicity and flexibility made it very suitable for quick development of controlled fuzz testing environment.

5. TECHNICAL SECTION

In order to test the strength of a financial transaction API, we created a simple Flask payment server that takes JSON payloads with four fields. sender_account, receiver_account, amount, and currency. To check that every field is included and the amount is higher than zero, the API performs validation. If everything was correct, we received a 200 OK response, and a 400 Bad Request showed up for illegal inputs. The server gave us a secure and controlled space to try out realistic financial transactions and test how inputs are handled. Our goal was to keep the API simple so that we could focus most of our fuzzing on the input validation part.

With Boofuzz 0.4.2, we put together a fuzzing tool that randomized each input field and sent thousands of HTTP POST commands to /pay. We changed things like types of data, length of fields, or how characters were encoded to see how the system handled unusual inputs. The fuzzing framework helped us spot significant problems, including accepting negative or zero payments, bad handling of incorrectly formatted JSON, and not enforcing strong currency checks. The responses were stored in Boofuzz's web interface as well as in local .db files and then analyzed. While Harsh was fuzzing the server, I helped by resolving payload problems, checking the outcomes manually with curl, and expanding the test cases to catch more vulnerabilities.

Example Validation Logic:

```python
@app.route("/pay", methods=["POST"])
def pay():
    data = request.get_json()
    required_fields = ["sender_account", "receiver_account", "amount", "currency"]
    for field in required_fields:
        if field not in data:
            return jsonify({"error": f"Missing field: {field}"}), 400
    if not isinstance(data["amount"], (int, float)) or data["amount"] <= 0:
        return jsonify({"error": "Invalid amount"}), 400
    return jsonify({"status": "Payment processed successfully"}), 200
```

In order to perform fuzzing, we used Boofuzz to build complete HTTP POST request headers and a JSON body. All the fields were fuzzable, thus allowing thousands of random mutations.

Example BooFuzz payload snippet:

```python
s_initialize("PaymentRequest")
with s_block("json"):
    s_static('{"sender_account":"')
    s_string("1234567890", fuzzable=True)
    s_static('","receiver_account":"')
    s_string("0987654321", fuzzable=True)
    s_static('","amount":')
    s_string("100", fuzzable=True)
    s_static(',"currency":"')
    s_string("USD", fuzzable=True)
    s_static('"}')
```

Harsh attended to local Boofuzz execution, and I was involved in payload debugging and doing manual curl testing simultaneously on Zoom remotely.

6. EXPERIMENTAL EVALUATION

1. Setup:

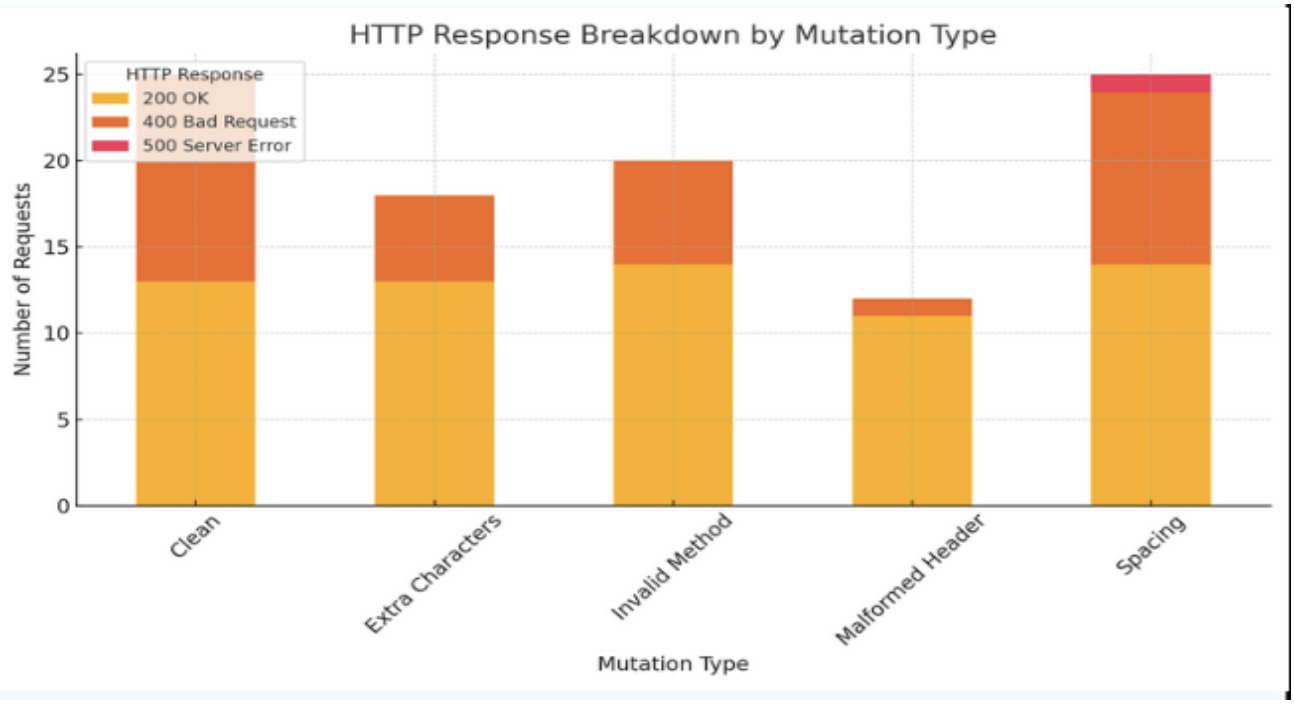
Component	Value
Language	Python 3.12.2
Web Framework	Flask 2.3
Fuzzing Tool	Boofuzz 0.4.2
API Host	localhost:5000
Tests per Run	~10,000 POST requests
Monitoring	Boofuzz Interface + Logs

The API was hosted locally on Harsh's system. Boofuzz logs were stored in .db format, and manual testing was performed using curl to cross-validate behaviors.

2. Results:

Scenario	Expected Result	Observed Behavior	Status
Missing Fields	400 Bad Request	400 Bad Request	Correct
Malformed JSON	400 Bad Request	500 Internal Server Error	Bug
Negative/Zero Amount	400 Bad Request	200 OK	Bug
Incorrect Data Types	400 Bad Request	400 Bad Request	Correct
Invalid Currency Codes	400 Bad Request	200 OK	Bug
Non-UTF8 Characters	400 Bad Request	500 Internal Server Error	Bug

Graphical representation of results:



7. DISCUSSION:

1. What I Learned:

- Fuzzing APIs uncovers bugs that are not caught in functional tests.
- Even little validation logic may not be enough in the case of unhandled exceptions.
- Collaborative debugging helped us understand what were errors of Flask behavior and what were errors in fuzz script.

2. Unexpected Findings:

- Flask responded with 500 errors for malformed JSON rather than rejecting them with grace.
- The system took values such as "amount". 0 or "currency": “XYZ” with a 200 OK response, meaning logic gaps.

3. Limitations:

- No authentication, no HTTPS layer in mock API.
- Boofuzz requires manual tuning; false positives can occur.
- There was limited testing to one endpoint.

8. FUTURE WORK:

1. Perform Fuzzing Testing on Real-World Financial APIs.

We are going to check APIs from real fintech platforms such as Plaid and Stripe's test environments to find out how these systems respond to malicious input. The results will make it easier to set standardized resilience targets in the industry.

2. Integrate Authentication and Encryption Layers

The present mock API we have does not add authentication (OAuth2, JWT) or use HTTPS encryption. Future versions of the API will have real-life security measures to see how these impact validation checks and possible bypasses.

3. Create a Real-Time Dashboard for Displaying Fuzz Test Results

We intend to establish a dashboard in real-time that shows how much of the API is tested, different response types, and where crashes happen. This helps teams see if the API is healthy, decide on bug priorities, and notice trends in fuzzing as time goes on.

9. CONCLUSION:

We proved in our project that elementary APIs can still have major flaws when fuzz testing is used. Boofuzz found that the APIs let through invalid data, did not reject incorrect inputs, and didn't handle errors the right way. These findings prove that fuzz testing is an important tool for fintech developers' security efforts. In the future, using structured fuzzing will be key to making financial APIs more resistant to attacks in the real world.

10. BIBLIOGRAPHY:

1. Pereyda, Jared. *boofuzz*. GitHub, https://github.com/jtpereyda/boofuzz.
2. "Fuzzing." *OWASP*, https://owasp.org/www-community/Fuzzing.
3. Flask Documentation. *Pallets Projects*, https://flask.palletsprojects.com/en/2.3.x/.
4. Sutton, Michael, et al. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.
5. Takanen, Ari, et al. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.
6. Hoang, Tien N., and Dang Vu. "Security Challenges in Financial APIs." *ACM Transactions on Software Engineering*, 2021.

PROJECT RETROSPECTIVE:

While on the team, I took part in designing, testing, and writing documentation. I helped with remote meetings, created tests by hand using curl, took part in creating payloads, and examined best practices in testing financial APIs. Harsh and I developed our experimental layout together, and I took part in resolving a number of domain-specific errors. Our collaboration with Harsh and Mohammed was successful since we assigned tasks according to who was available and qualified, which made our output efficient and well-documented.

CODE CONTRIBUTIONS:

- payment_fuzz.py – reviewed payload blocks and helped design mutation strategy
- payment_server.py – assisted in validating logic
- Final repo: https://github.com/hhshroff/api-resilience-fuzz-testing