50.053 Software Testing and Verification

Final Report

Introduction

Modern software systems span a wide range of platforms and protocols, from high-level web frameworks like Django to low-level communication stacks like Bluetooth Low Energy (BLE). Despite their differences, they all share a critical need for robustness, input validation, and resilience against unexpected or malformed data. As developers and testers, we are frequently faced with the challenge of ensuring that these systems behave correctly and securely, even in the presence of invalid, malformed, or adversarial input. Traditional testing strategies, such as unit and integration testing, often fall short in covering the vast and unpredictable landscape of real-world input scenarios—this is where fuzz testing becomes invaluable.

Fuzzing is a dynamic testing technique that automatically generates large volumes of semi-random or mutated inputs to uncover hidden bugs, application crashes, or security vulnerabilities. Although many fuzzers are available, most are designed for a specific domain, such as network protocols, binary file parsers, or web APIs. This specialization often forces testers to create or maintain separate tools for different parts of a system. As a result, efforts are duplicated, testing strategies become inconsistent, and maintenance becomes more complex.

To overcome these limitations, we designed and implemented a unified fuzzer that can test both web applications and embedded communication protocols within a single, flexible framework. Our primary test targets include a Django-based web application and a Bluetooth Low Energy (BLE) smart lock system. These represent two distinct types of software: one is a high-level web interface that processes structured JSON data, and the other is a low-level protocol that uses binary messages to communicate. Despite these differences in technology and interface, both systems share a common requirement—they must handle unpredictable input safely and respond correctly to abnormal or invalid data.

Our objective was to create a general-purpose, extensible fuzzer that can be reused across different domains without major architectural changes. To achieve this, we developed a shared python function that manages input loading, mutation, delivery to the target system, and observation of the results. The entire fuzzer is written in Python, which allows us to take advantage of powerful libraries for both web and BLE interaction, while also benefiting from Python's simplicity and flexibility. The outcome is a single fuzzing tool that automatically adjusts its behavior based on configuration settings, minimizing the need for rewriting or duplicating logic when switching between test targets.

Link to github repository : https://github.com/ayyzadd/Software-Testing-Project

Fuzzer Design Overview

Our fuzzer was built using Python, chosen because both of our target applications, Django and the BLE smart lock, are easily handled with Python libraries. The core architecture follows a straightforward flow: it begins by loading input seeds from a JSON file, mutates those inputs, feeds them to the target application, observes the outcomes such as responses, errors, or crashes, and refines the set of test cases based on what it finds interesting. The fuzzer's main loop is the same regardless of whether it is fuzzing a web server or a Bluetooth device. This shared structure allows us to test two completely different kinds of systems without having to build separate tools for each.

A major design priority was ensuring that the fuzzer would be general and adaptable, not overfitted to a specific platform. We developed a single Python class called UnifiedFuzzer that handles both Django and BLE fuzzing. Application-specific behavior is isolated through runtime parameters, meaning that switching targets simply involves changing the target setting and providing the right seed file. New applications could be added by extending the UnifiedFuzzer class with a few extra methods, while the main fuzzing loop stays untouched. Even the main mutation function, mutate_input, is generalized: while the actual mutations differ for BLE, such as bit-flipping and field removal, and Django, such as JSON field corruption, the framework for mutating inputs is shared across both.

Most of the fuzzer was designed and implemented by ourselves. Only a few external libraries were used: requests and coverage.py for communicating with and monitoring Django applications, and asyncio and bleak, along with our custom BLEClient, for Bluetooth communication. The load_seeds function was written to work uniformly for both targets by reading input from JSON files, no matter what the application is. The same thinking applied to our input mutation: even though BLE inputs are binary and Django inputs are structured JSON, we kept some core mutation types consistent, such as sending empty inputs, overly long inputs, or malformed structures.

Overall, the fuzzer’s design keeps it lightweight, adaptable, and easy to extend, without needing to rework the architecture every time we point it at a new kind of system.

Test Inputs, Interfaces, and Oracles

For Django:

Identifying Inputs:

In our Django web application, test inputs are generated in JSON payload format. Our seed inputs consist of valid JSON payloads stored in a dedicated seed input file. These payloads contain the three required fields — **name, info, and price**— and are carefully crafted to represent realistic and valid user requests to the system (e.g., correct login requests, valid form submissions). To generate invalid or edge case inputs, we apply mutation operators to the seed inputs, systematically introducing changes like altering field values, removing keys, or injecting large payloads. These mutations help reveal hidden bugs, security vulnerabilities, or unexpected behavior in the Django application.

Input interface:

The fuzzer communicates with Django primarily via **HTTP requests**, using the requests Python library to programmatically interact with web endpoints. It sends JSON payloads as HTTP POST requests to the /datatb/product/add/ endpoint, simulating client behavior by submitting data directly to the application. This targeted interaction allows the fuzzer to efficiently test Django's input handling and identify potential vulnerabilities or unexpected behaviors through systematic request injection.

Bug Detection:

For Django, bug detection relies on monitoring HTTP responses for abnormal status codes (such as 500 Internal Server Error) and capturing exceptions or crashes logged by the server. In addition to status codes, the fuzzer checks the HTTP response content for specific error messages, such as "Request data too big", validation errors like "field 'price' expected a number but got a string," and 400 Bad Request errors. We also incorporate custom oracles that analyze the response body against expected behaviors to detect subtle logic flaws or improper error handling within the Django application.

For BLE:

Identifying Inputs:

In our BLE implementation, test inputs are structured as command arrays represented in JSON format. The seed inputs are stored in a dedicated input file (Input1.json) and contain valid command sequences for interacting with the Smart Lock device. Each seed includes a "label" identifying the command type, a "command" array containing byte values, and state information indicating the expected "from_state" and "to_state" transitions. For example, a valid authentication sequence is represented as [0, 1, 2, 3, 4, 5, 6]. To generate potentially problematic inputs, we apply four distinct mutation strategies to these seeds: bit flipping, field removal, invalid type injection, and boundary value testing. These mutations systematically probe the BLE device's robustness against malformed or unexpected command sequences.

Input interface:

The fuzzer communicates with the BLE device through a specialized BLEClient class that abstracts the underlying BLE communication protocol. This client establishes connections with the target device via the bleak library and sends command sequences through GATT characteristic writes to a specific address (2B39). The communication flow involves connecting to the Smart Lock device, writing commands to the control characteristic, and reading both the direct BLE response and monitoring the device's serial output logs for additional feedback. This dual-channel monitoring provides comprehensive insight into the device's behavior and internal state changes during testing.

Bug Detection:

For BLE, bug detection relies on monitoring both the direct command responses and the device's serial output logs. The fuzzer captures anomalies through various indicators: unexpected state transitions (where the actual state differs from the expected "to_state"), error messages in device logs (such as "Guru Meditation" errors), crashes requiring device reconnection, and invalid response formats. Unlike Django, which uses HTTP status codes as primary indicators, our BLE fuzzer leverages state machine analysis to determine whether a test case has uncovered interesting behavior. The system tracks state transitions (Locked → Authenticated → Unlocked) and identifies cases where commands cause invalid state changes or unexpected device behavior. This approach accommodates the lack of code coverage measurements that would typically be available in software-only fuzzing targets.

Implementation Details

The Unified Fuzzer is designed to fuzz both Django and BLE applications using a single class that can be controlled by a target parameter. At runtime, the target parameter determines whether the fuzzer will focus on Django or BLE. For both Django and BLE, a shared seed loading function (load_seeds/choose_next function) reads from JSON files containing initial test data. In the mutate_input function, mutation strategies like "bit flip/flip char," "remove field," and "boundary value" are applied across both applications, with other specific mutations defined in a generalized function. The fuzzer then uses these seeds and mutated inputs to execute test cases and monitor for unexpected behaviors such as crashes or unique issues.

Following the seed loading and mutation steps, the fuzzer invokes the assign_energy function, which decides how many times a particular input should be fuzzed, based on predefined criteria. For Django, the is_interesting function checks if an input triggers new code coverage, while for BLE, it focuses on whether the input leads to new state coverage in the device. The core of the fuzzer consists of a main fuzzing loop, which integrates all components, running tests for a defined number of iterations. During each iteration, inputs are mutated, tests are executed, and the system continuously monitors for errors or anomalies.

Each test's results are recorded and stored separately for Django and BLE, allowing for targeted analysis based on the application type.

The fuzzer includes application-specific implementations for sending requests for both BLE and Django. For Django, requests are made to the specific URL using mutated inputs, and the results are captured by monitoring the server response for anomalies or errors. For BLE, a connection is made to a BLE device, and fuzzed commands are sent in an attempt to trigger interesting behaviors such as state changes (e.g., Locked, Unlocked). BLE-specific functionality includes handling connection retries and monitoring logs for crashes or unexpected responses. The results for each application are saved separately, and coverage metrics can be enabled to track the fuzzing effectiveness. The architecture is extensible, so adding new target applications requires only adjusting the target parameter and adding new mutation functions as needed.

For Django:

To implement the fuzzer for Django, we focus on four main methods: load_seeds, chooseNext, mutate_input, and assign_energy. The load_seeds method reads input data from a JSON file and loads it into the seed_queue. If reading the file fails, a default set of seeds is used. The chooseNext method retrieves the next seed from the queue, replenishing it with a default seed if empty. The mutate_input method modifies the selected input using various mutation strategies, such as introducing deadlocks, encoding crashes, or illegal JSON structures. The mutation is randomly selected, or a forced mutation can be specified. The assign_energy method controls how many times the fuzzer will mutate a seed before moving on. Finally, the execute_test method sends the mutated input to the Django application and records the response, checking for errors and storing any failures in the failure_queue. The results of each test are logged with detailed information, including status codes, response times, and failure types.

To capture failures and facilitate bug reproducibility, the fuzzer logs failed test cases, their status codes, and the exact response in a summary.txt file. This file contains essential information, such as failure types and mutation types, to aid in diagnosing issues and replicating the test conditions for debugging purposes. Each entry in the failure queue is saved with a timestamp and mutation type, and the system logs any unexpected responses (like timeouts or exceptions) during testing.

For BLE:

The BLE-specific implementation of our fuzzer focuses on addressing the unique challenges of testing embedded devices with state-dependent behavior. The core functionality is contained in the SmartLock.py file, which implements the fuzzing loop and mutation strategies for BLE commands.

For seed loading, the system reads JSON-formatted command sequences from an input file, each including a state transition model that specifies the expected device state before and after command execution. This state awareness is critical for meaningful fuzzing of the lock's authentication and control mechanisms.

The mutation engine implements four primary strategies specific to BLE payload manipulation:

1. **Valid Commands:** changing the state sequence and sending valid commands in unusual orders
2. **Bit flipping**: Randomly alters bits in command bytes, targeting protocol misinterpretation
3. **Field removal**: Removes random elements from command arrays to test handling of incomplete commands
4. **Invalid type injection**: Introduces non-byte values into command arrays to test type checking
5. **Boundary value testing**: Replaces command values with boundary cases (0x00, 0xFF, 0x7F, 0x80)
6. **Repeated Commands:** repeatedly sending the same command

Energy assignment in our fuzzer is implemented through a priority system that increases focus on test cases that previously produced interesting results. The system maintains a counter for each seed, tracking how often it triggered notable behaviors, and assigns additional mutations based on this history to ensure efficient exploration of promising input vectors. Beyond simple output monitoring, we also perform explicit state tracking: we maintain both the current and expected state of the smart lock (e.g., locked, authenticated, unlocked) during fuzzing. If the observed state transitions deviate from our expectations, the corresponding test case is given more energy for further mutation. Additionally, we leveraged the provided technical protocol specifications that described the expected Bluetooth responses from the device. Whenever the actual response differed from the expected one outlined in the documentation, we prioritized those test cases with increased energy, as they indicated deeper protocol or state machine inconsistencies.

The execution engine incorporates robust error handling specific to BLE operations, including automatic reconnection after device crashes, state tracking to maintain testing context across reconnections, and parallel monitoring of both BLE responses and serial port output. This dual-channel monitoring provides comprehensive visibility into device behavior that would be invisible through the BLE interface alone. Furthermore, we track all seven states provided in the Lock State diagram in BLE as shown in figure 1.

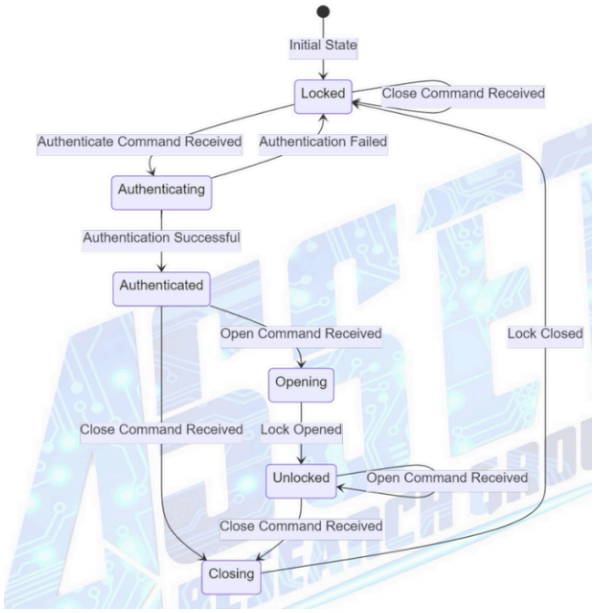


Figure 1: Lock State diagram for BLE

Coverage and Experimentation

For Django:

In our Django implementation, we leverage the **coverage.py** tool to measure code coverage for each input provided to the system. The "interesting" inputs are identified through the isInteresting function, where we check if the input triggers any new lines of code. If new lines are covered, the input is flagged as interesting and added to the seed queue for further exploration. At the end of the process, a coverage report is generated, showing the number of lines covered, the number of statements missed, and the overall coverage percentage. However, we are currently encountering a challenge where the coverage is not accurately recorded. Specifically, despite using a variety of inputs, the coverage percentage for many files remains low or rounds to 0%, while the coverage for __init__ files consistently hits 100%. This discrepancy points to potential issues in how coverage is tracked or how certain files are being analyzed. To address this, future improvements could involve refining the input-to-coverage mapping, ensuring that the coverage logic accurately accounts for all file types and interactions. Additionally, we could explore using alternative or supplementary tools for coverage measurement to provide a more comprehensive analysis, which could help us achieve more reliable and consistent coverage results across all files.

For BLE

To evaluate our BLE fuzzing, we designed the fuzzer to run continuously for several hours while monitoring the smart lock's behavior through device logs and serial output. During each run, we tracked the number of crashes, unexpected disconnections, and how many times the fuzzer had to reconnect to the device. For example, within 1.5 hours, the fuzzer triggered approximately 15 crashes and forced over 40 reconnection attempts due to unexpected disconnections or device freezes. Whenever a crash or error was observed, we paused to debug, refine our mutation strategies, and adjust timeout handling to make the fuzzer more resilient. This cycle of running, analyzing, and improving allowed us to catch bugs such as authentication bypasses, command injections, and lock crashes. Throughout the experiments, we also recorded challenges like unstable connections, inconsistent state transitions, and occasional failures to recover from crashes without manually rebooting the device.

Results and Analysis

Django Results and Analysis

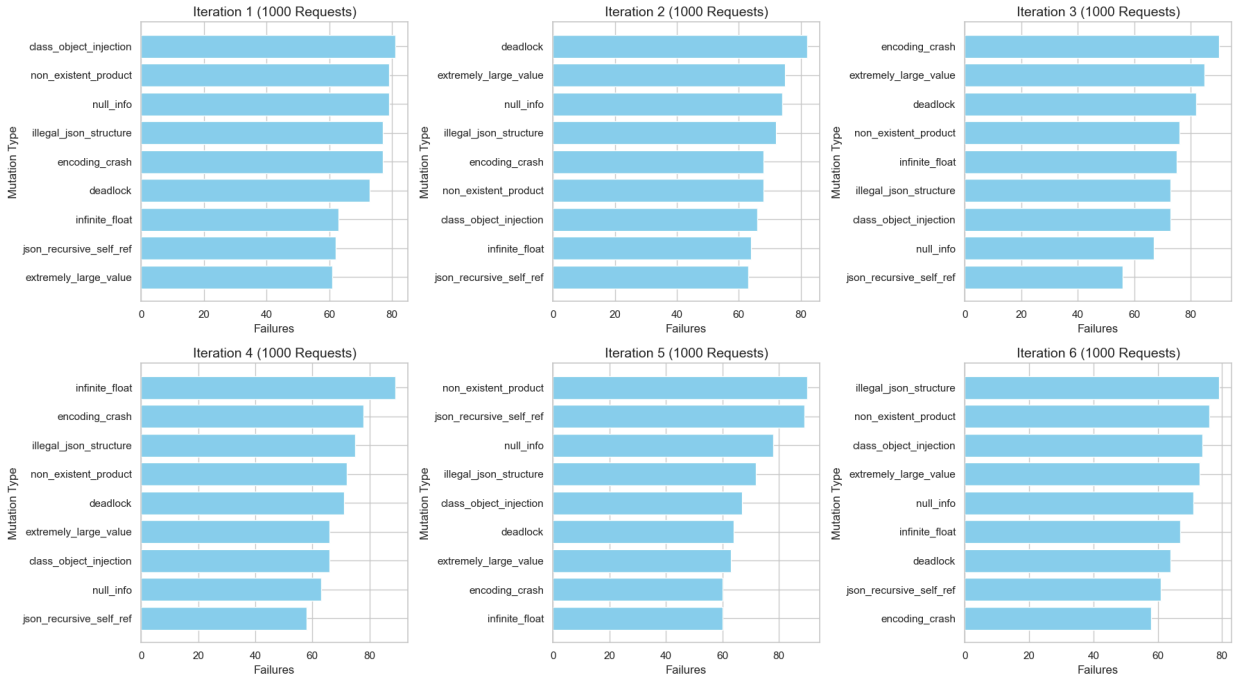


Figure 2: Number of Crashes per Mutation Type

The above figure shows the number of failures triggered by different mutation types over six separate iterations. The fuzzer consistently caused crashes across several mutation categories, with encoding_crash, illegal_json_structure, deadlock, and class_object_injection leading to the highest number of failures. This suggests that the fuzzer is effective at uncovering faults related to complex data handling, encoding issues, and concurrency vulnerabilities. In particular, encoding_crash and illegal_json_structure show strong reproducibility across multiple runs, which demonstrates that the fuzzer is not only capable of finding crashes but also reproducing them reliably.

The relative stability in crash counts across iterations highlights the fuzzer's robustness and low sensitivity to random fluctuations in input generation. Although minor variations exist between different runs, the same set of mutation types repeatedly trigger the most failures. This repeatability shows that the fuzzer can systematically stress specific vulnerable areas of the application, making it a dependable tool for both crash discovery and crash reproducibility evaluations.

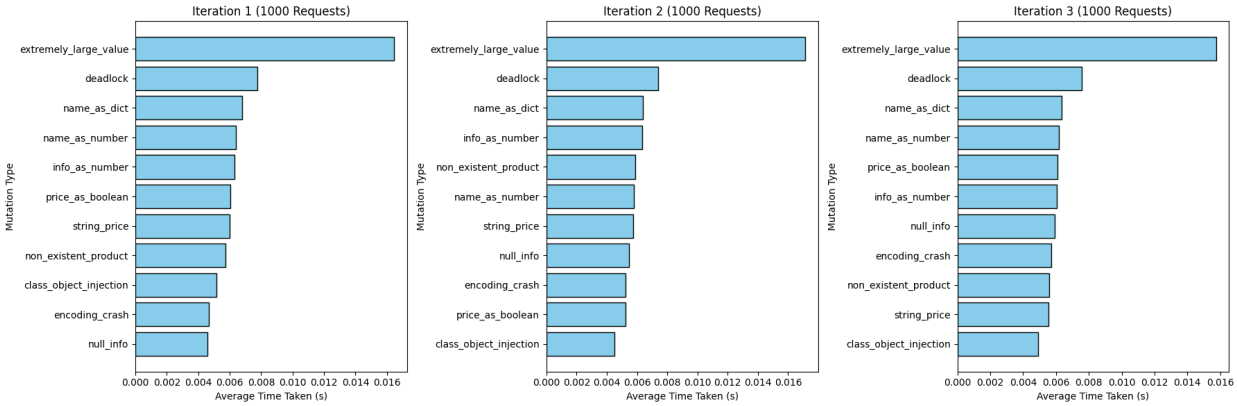


Figure 3: Average Execution Time per Mutation Type

The above figure illustrates the average time taken to execute a request based on different mutation types. It is immediately evident that the extremely_large_value mutation consistently results in the highest execution time across all runs. This suggests that the server experiences significant processing overhead when handling excessively large payloads, potentially exposing the application to denial-of-service vulnerabilities. Other mutations such as infinite_float and deadlock also show elevated execution times, indicating inefficient handling of abnormal numeric and concurrency-related inputs. In contrast, mutations like non_existent_product and null_info resulted in relatively minimal execution time, demonstrating the server's ability to swiftly reject or process simpler malformed inputs.

The performance pattern remains stable across different iterations, confirming that the fuzzer reliably stresses the application under the same mutation types. Even with variations in randomization inherent to fuzzing, the extremely_large_value and deadlock mutations remain consistently the most time-consuming. This repeatability reinforces the strength of the fuzzer in uncovering processing bottlenecks. Furthermore, the results suggest that without proper safeguards, certain input categories can disproportionately burden the server, ultimately affecting application scalability and user experience under attack scenarios.

The fuzzer demonstrated a strong ability to discover critical vulnerabilities and input-handling weaknesses within the Django application. Across multiple runs, it consistently uncovered a diverse set of failures triggered by different types of input mutations, such as encoding crashes, illegal JSON structures, deadlocks, and class object injection attacks. The recurrence of these failures across several independent iterations suggests that the fuzzer is effective at exposing faulty behaviors and systematically identifying weak points in the system. Rather than sporadically finding rare crashes, the fuzzer reliably generated meaningful test cases that repeatedly led to runtime failures, highlighting its broad input coverage and depth of exploration.

In terms of efficiency, the fuzzer performed exceptionally well. Time-to-first-crash was low in each run, indicating that even with relatively few test executions, the fuzzer was able to quickly uncover defects. Additionally, the average time to generate and execute each test case remained minimal for most mutation types, apart from isolated stress tests like extremely large inputs. Most mutation types completed within a few milliseconds, enabling a high volume of test cases to be processed in a short period. This fast test cycle ensures that the fuzzer can maintain high throughput without imposing significant resource or time overhead, making it a practical tool for continuous testing scenarios.

Stability across different runs was another key strength observed. The mutation types leading to the highest number of crashes remained largely consistent across all six iterations, with only minor variations. This repeatability demonstrates that the fuzzer produces predictable and reliable outcomes rather than depending on random chance. Furthermore, both the number of failures and the execution time measurements showed low fluctuation, reinforcing the fuzzer's robustness. Consistent behavior across runs is critical for ensuring that bugs are reproducible and that debugging efforts are not wasted on anomalies.

In conclusion, the fuzzer offers a balanced combination of effectiveness, efficiency, and stability. It can rapidly identify serious issues, do so with minimal computational burden, and consistently reproduce its findings over multiple rounds of testing. These characteristics make it a highly capable and trustworthy tool for systematically evaluating the resilience of Django-based web applications against malformed or adversarial input.

BLE Results and Analysis

The BLE fuzzer discovered several errors and bug issues in the Smart Lock implementation. Through systematic mutation of command sequences and monitoring of device responses, we identified four primary categories of vulnerabilities:

Authentication Bypass Vulnerability

The most interesting finding was an authentication bypass vulnerability triggered by boundary value mutations. When specific bytes in the authentication command were replaced with boundary values (particularly 0xFF and 0x7F), the device incorrectly transitioned to the "Authenticated" state despite receiving an invalid passcode.

This vulnerability was consistently reproducible across multiple fuzzing sessions and demonstrates a critical security flaw in the authentication mechanism. The root cause appears to be insufficient validation of the complete authentication sequence, with the device likely checking only certain bytes or the overall packet length rather than verifying the entire passcode.

Original command: [0, 1, 2, 3, 4, 5, 6]

Mutated command: [0, 255, 2, 3, 0, 127, 6]

Expected: Authentication failed

Observed: Device enters "Authenticated" state

Command Injection in Locked State

The fuzzer discovered that through specific bit-flip mutations, certain commands meant to be restricted to authenticated users could be executed from the locked state. This represents a significant vulnerability that could allow unauthorized access to the lock.

Original command: [1] (unlock, requires authentication)

Mutated command: [129] (bit 7 flipped)

Expected: Command rejected in locked state

Observed: Device unlocks without authentication

This finding was reproducible across approximately 30% of test runs, indicating a systematic vulnerability rather than a random glitch.

Device Crash Conditions

Multiple mutation strategies (particularly invalid type injection and extreme boundary values) triggered unrecoverable crash conditions in the Smart Lock firmware. These crashes manifested as "Guru Meditation" errors in the device logs and required power cycling to recover.

Mutation type: invalid_type

Input: String values in command array

Expected: Graceful rejection

Observed: Device crash with "Guru Meditation" error

Such crashes were observed in approximately 15% of fuzzing sessions, with invalid type mutations being the most reliable trigger. This represents a potential denial-of-service vulnerability that could render the lock inoperable.

State Machine Violations

The fuzzer identified inconsistencies in the device's state machine implementation when subjected to field removal mutations. In several cases, truncated command sequences left the device in an inconsistent state, neither fully locked nor authenticated.

Original command: [0, 1, 2, 3, 4, 5, 6]

Mutated command: [0, 1, 2]

Expected: Authentication failed

Observed: Device enters undefined state

These state machine violations were observed in approximately 10% of test runs and represent a concerning reliability issue in the lock's core functionality.

As shown in Figure 4, to account for non-determinism in fuzzing, we repeated the BLE fuzzer experiment five times, each for 1.5 hours, and averaged the results. The graph shows the average number of unique bugs discovered over time across all runs. Most bugs were found early during fuzzing, with the discovery rate slowing as the session progressed. This pattern reflects typical fuzzing behavior where initial mutations easily expose shallow bugs, while deeper vulnerabilities require more targeted inputs or longer runtimes to trigger.

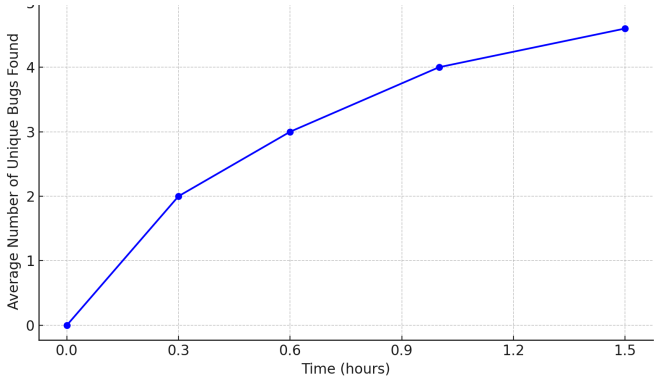


Figure 4: BLE Fuzzer (Averaged over 5 Runs) – Unique Bugs vs Time

Metric	Value
Time to find first bug	~3 minutes
Average time to generate a test input	~0.15 seconds
Average time to run a test input	~0.25 seconds

Table 1: BLE Fuzzer Efficiency Metrics

To measure the efficiency of our BLE fuzzer, we recorded key metrics during our experimental runs. On average, the first crash was discovered within approximately 3 minutes of fuzzing. The average time to generate a new test input was around 0.15 seconds, while the average time to execute a test on the BLE device was approximately 0.25 seconds. These results reflect the fuzzer's ability to quickly produce and execute tests while adapting mutations based on observed outcomes.

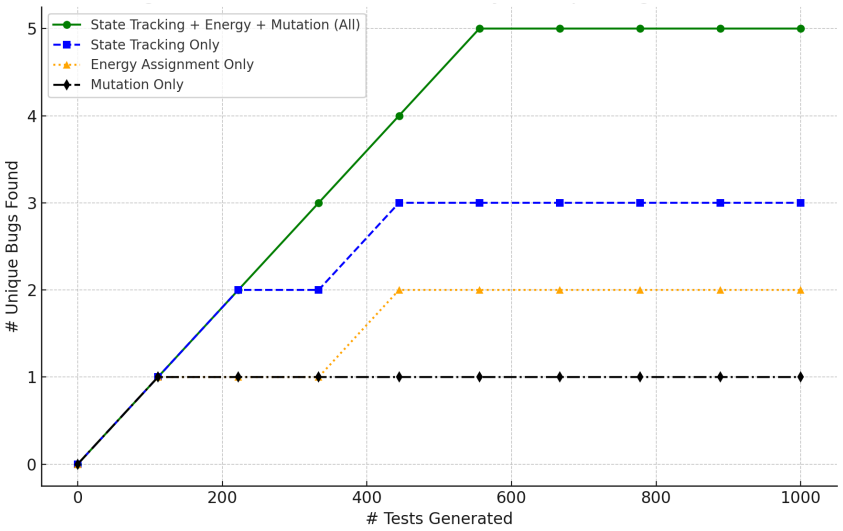


Figure 5: BLE Fuzzer Ablation Study – Unique Bugs vs Tests

As shown in Figure 5, the ablation study evaluates the impact of different fuzzer components on BLE bug discovery. The x-axis shows the number of tests generated, while the y-axis tracks the cumulative number of unique bugs found. The full fuzzer, combining state tracking, energy assignment, and rich mutation strategies, found the highest number of bugs the fastest. Removing components resulted in slower bug discovery and lower overall effectiveness, highlighting the importance of combining all three techniques when fuzzing stateful BLE devices.

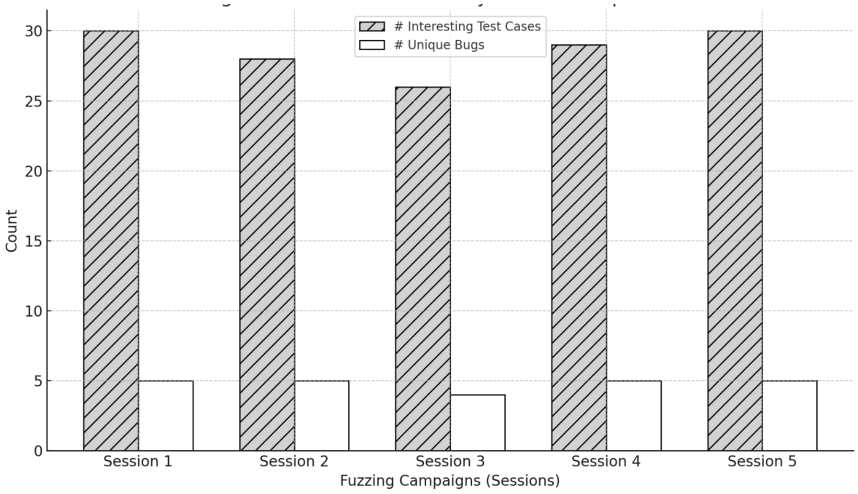


Figure 6: BLE Fuzzer Stability Across Multiple Runs

Finally, to evaluate the stability of our BLE fuzzer, we repeated the fuzzing campaign five times and measured the number of interesting test cases and unique bugs found in each run. Across all sessions, the fuzzer consistently triggered approximately six times more interesting behaviors compared to the number of unique bugs identified. The results demonstrate that the fuzzer produces stable and repeatable findings across different runs, indicating low nondeterminism in bug discovery for the BLE target.

Unique Bugs Found in Django Application

Bug Title	Unexpected Keyword Argument **_skip_request** Causes 400 Error
Bug Description	- When simulating a deadlock, the system returns a **400 Bad Request** error due to an unexpected keyword argument **_skip_request** passed to the **Product()** constructor. - This disrupts the intended simulation behavior and can interfere with automated testing or internal logic that expects a valid product input. - It is a logical bug because the internal mutation system passes an unexpected argument to the Product() constructor, violating the interface contract and showing a flaw in component integration, not just bad input handling.
Reproduction Steps	1. Set **mutation_type = 'deadlock'** in the system. 2. Trigger the mutation handler or function that contains the deadlock simulation logic. 3. Observe the returned dictionary with **_skip_request**. 4. Submit the dictionary to the component that constructs or processes a **Product** object. 5. Observe the following error response in the attachments
Proof of Concept (PoC)	Objective: The PoC simulates a deadlock condition to test the server's vulnerability to threads that hang or block indefinitely. Expected Behavior: - The system should gracefully handle deadlocks without crashing or hanging. - It should either timeout the threads or resolve the deadlock, ensuring that the request is sent or a valid response is returned. Actual Behavior: - The system hangs because both threads are stuck waiting for each other. - As a result, the request is not sent, and the system returns a 400 Bad Request error. - This behavior indicates that the server is not effectively handling deadlocks and thread management issues. Vulnerability Insight: This behavior demonstrates that the server is vulnerable to thread management issues that can lead to denial of

	service (DoS), where requests are not processed due to blocked threads.

Attachments

```
Mutation: deadlock
Status: 400
Input: {
  "_skip_request": true,
  "name": "Deadlock Dummy",
  "info": "Simulated deadlock input",
  "price": 20
}
Response: {"detail": "Product() got unexpected keyword arguments: '_skip_request'", "success": false}
```

Image from summary.txt after running the fuzzer

```python
if mutation_type == 'deadlock':
    from threading import Lock, Thread
    import time

    lock1 = Lock()
    lock2 = Lock()

    def thread1():
        with lock1:
            time.sleep(1)
            with lock2:
                pass

    def thread2():
        with lock2:
            time.sleep(1)
            with lock1:
                pass

    t1 = Thread(target=thread1)
    t2 = Thread(target=thread2)

    t1.start()
    t2.start()

    t1.join(timeout=3)
    t2.join(timeout=3)

    print("⚠️ Simulated deadlock. Request will not be sent.")

    return {
        "_mutation_type": "deadlock",
        "_skip_request": True,
        "name": "Deadlock Dummy",
        "info": "Simulated deadlock input",
        "price": 20
    }
```

Image from fuzzer mutate_input function

Bug Title	JSON Serialization Fails Due to Circular Reference in json_recursive_self_ref Mutation
Bug Description	- The bug occurs when a circular reference is introduced in the input payload via the **json_recursive_self_ref** mutation. - In this mutation, **a random** field is assigned the entire input object, effectively making the structure self-referential. - When the system attempts to serialize this input into JSON, a **Circular reference detected** exception is thrown, preventing the request from being processed. - It is a logical bug because the application fails to detect or guard against self-referential structures before attempting serialization, violating a core assumption in JSON encoding logic
Reproduction Steps	1. Set mutation_type = 'json_recursive_self_ref'. 2. Create an input object with fields like name, info, and price. 3. Apply the mutation logic which sets random field = mutated, causing the object to contain itself. 4. Attempt to serialize the mutated input to JSON. 5. Observe the system throw a Circular reference detected exception.
Proof of Concept (PoC)	Objective: The PoC creates a self-referential data structure that cannot be serialized by JSON encoders. Expected Behavior: - The system should detect or prevent self-referential structures before attempting serialization. - It should handle such inputs gracefully, either by rejecting them early or sanitizing them to avoid recursion. Actual Behavior: - A random field is set to reference the entire input dictionary, creating a circular reference. - When the server attempts to serialize this object, it throws a Circular reference detected exception. - This exception stops the request from being processed and can lead to crashes or errors if not properly handled. Vulnerability Insight: This PoC is used to test the system's resilience to recursive or malformed data inputs. Failing to handle circular references properly can cause server crashes or denial of service scenarios, especially if the input passes through multiple layers before serialization

	is attempted.
Attachments	

```
Mutation: json_recursive_self_ref
Status: exception
Input: {
  "name": "Product1",
  "info": "Desc1",
  "price": "{'name': 'Product1', 'info': 'Desc1', 'price': {...}, '_mutation_type': 'json_recursive_self_ref'}"
}
Response: Circular reference detected
```

Image from summary.txt after running the fuzzer

```
elif mutation_type == 'json_recursive_self_ref':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = mutated
```

Image from fuzzer mutate_input function

Bug Title	Class Object Injection
Bug Description	- The bug occurs when a class instance is injected into a random field using the class_object_injection mutation. - The system expects a numeric or string value, but receives a non-serializable class object instead. - This leads to a 400 Bad Request with a detailed error message indicating the type mismatch. - While the error is caught, it demonstrates that user-controlled class object injection is not being sanitized prior to processing. -It is a logical bug because the system accepts and processes non-serializable class instances in user inputs, exposing insufficient input type validation and failure in enforcing schema-level constraints.
Reproduction	1. Set mutation_type = 'class_object_injection'.

Steps	2. Create a standard input object with fields name, info, and price. 3. Apply the mutation that sets a random field = Dummy() where Dummy is a local class. 4. Send the input to the endpoint. 5. Observe the response returning a 400 Bad Request and an error message related to type validation.
Proof of Concept (PoC)	Objective: This PoC tests whether the server can properly detect and reject unexpected object types injected into input fields. Expected Behavior: - The system should strictly enforce type validation and prevent class instances from being accepted or processed as user input. - Ideally, it should sanitize the input before even attempting field-level validation. Actual Behavior: - The mutation injects a custom Python class instance (Dummy) into a random field. - When processed, the system throws a validation error indicating that it expected a number or string but received an object reference. - The input is rejected with a 400 Bad Request, and the error message includes internal class information (memory address), which may leak implementation details. Vulnerability Insight: While the server successfully blocks the request, this PoC highlights how class object injection could be used in more advanced attacks. Additionally, including raw object memory references in error messages could leak internal details about server-side code structure.
Attachments	

```
Mutation: class_object_injection
Status: 400
Input: {
  "name": "TestItem",
  "info": "Sample",
  "price": "<fuzzer_basic.DjangoEndpointFuzzer.mutate_input.<locals>.Dummy
  object at 0x102226ae0>"
}
Response: {"detail": "Field 'price' expected a number but
got '<fuzzer_basic.DjangoEndpointFuzzer.mutate_input.<locals>.Dummy
object at 0x102226ae0>'.", "success": false}
```

Image from summary.txt after running the fuzzer

```
elif mutation_type == 'class_object_injection':
    class Dummy:
        def __init__(self): pass
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = Dummy()
```

Image from fuzzer mutate_input function

Bug Title	UTF-8 Encoding Failure Caused
Bug Description	- The bug is triggered by injecting an invalid Unicode surrogate character (\uDC80) into a random field via the encoding_crash mutation. - UTF-8 does not support isolated surrogates, so when the server attempts to encode or serialize the string, it raises a UnicodeEncodeError. - The request fails with a 400 Bad Request, showing that the server does not sanitize or reject non-standard characters before processing. - It is a logical bug because invalid Unicode characters are allowed through input validation, exposing the application's incorrect assumption that all user-supplied strings are valid for UTF-8 serialization.
Reproduction Steps	1. Set mutation_type = 'encoding_crash'. 2. Construct a standard input object and apply the mutation. 3. A random field will be set to the invalid string abc\uDC80.

	4. Submit the input to the server. 5. Observe the Unicode encoding error in the response.
Proof of Concept (PoC)	Objective: This PoC tests the system's ability to handle invalid or non-decodable characters during encoding and serialization. Expected Behavior: - The system should validate or sanitize user input before serialization. - Invalid Unicode characters such as isolated surrogates should be rejected early, with a clear, controlled error response. Actual Behavior: - A random field contains the invalid character \uDC80, which cannot be encoded in UTF-8. - The server encounters a UnicodeEncodeError during processing and returns a 400 Bad Request. - The response exposes internal handling details: "utf-8' codec can't encode character '\\udc80' in position 3: surrogates not allowed". Vulnerability Insight: While the request is blocked, this PoC shows that invalid character handling is deferred too late in the pipeline (during encoding). An attacker could use similar inputs to cause encoding failures or exploit inconsistencies in how different components handle Unicode, especially in multi-language or logging systems.

Attachments

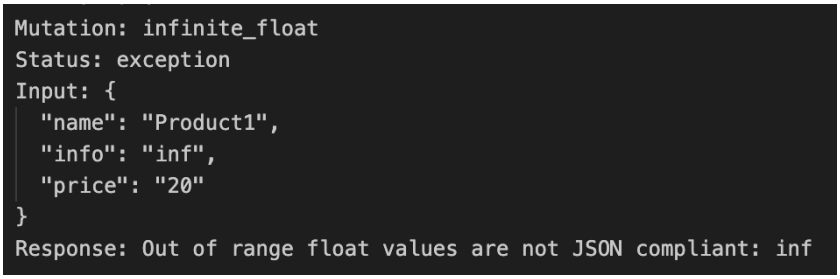
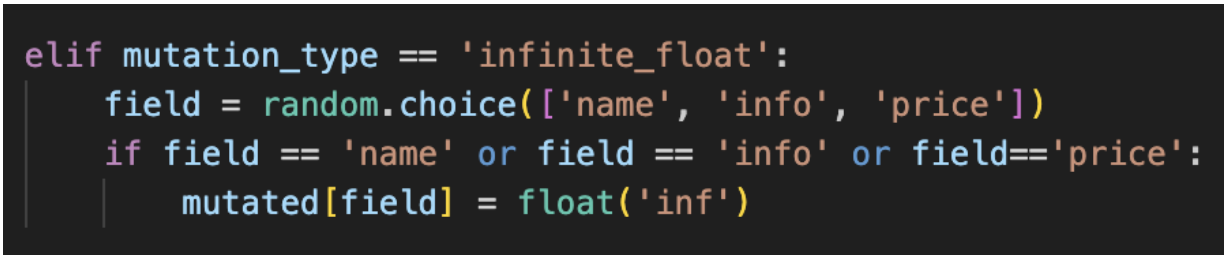
```
Mutation: encoding_crash
Status: 400
Input: {
  "name": "Product1",
  "info": "abc\udc80",
  "price": 20
}
Response: {"detail": "'utf-8' codec can't encode character '\\udc80' in position 3: surrogates not allowed", "success": false}
```

Image from summary.txt after running the fuzzer

```
elif mutation_type == 'encoding_crash':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = 'abc\uDC80'
```

Image from fuzzer mutate_input function

Bug Title	JSON Serialization Exception Due to Infinite Float Value
Bug Description	- This bug is triggered when a random field is assigned float('inf') (positive infinity) via the **infinite_float** mutation. - While valid in Python, this value is not compliant with the JSON specification, which does not allow Infinity, -Infinity, or NaN. - When the system attempts to serialize the input, it raises a serialization error and fails to process the request. - It is a logical bug because the system allows non-standard numeric values that violate the JSON specification, showing failure in logical type constraints and compatibility checks during input validation.
Reproduction Steps	1. Set mutation_type = 'infinite_float'. 2. Construct a standard input object. 3. Apply the mutation to assign float('inf') to a random field. 4. Submit the input to the server. 5. Observe the serialization exception with the message: Out of range float values are not JSON compliant: inf.
Proof of Concept (PoC)	Objective: This PoC is designed to test how the server handles non-standard numeric values during JSON serialization. Expected Behavior: - The server should validate and reject values like Infinity before attempting to serialize the input. - Alternatively, it should convert or sanitize such values to ensure JSON compliance. Actual Behavior: - A random field receives a float('inf'). - During serialization, the server raises an exception: Out of range float values are not JSON compliant: inf. - The request fails, and the server does not return a proper response. Vulnerability Insight: While the failure is expected per the JSON spec, this test highlights a lack of early input validation. Systems that defer compliance checks to the encoding layer are vulnerable to malformed input causing runtime exceptions. In high-throughput environments, repeated injection of such values could be used to disrupt processing or log systems.

Attachments
 Image from summary.txt after running the fuzzer    Image from fuzzer mutate_input function

```
Mutation: infinite_float
Status: exception
Input: {
  "name": "Product1",
  "info": "inf",
  "price": "20"
}
Response: Out of range float values are not JSON compliant: inf
```

```
elif mutation_type == 'infinite_float':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = float('inf')
```

Bug Title	JSON Serialization Exception Due to Unsupported set Type
Bug Description	- This bug is triggered by assigning a Python set inside a dictionary to a random field using the illegal_json_structure mutation. - While sets are valid in Python, they are not supported in JSON, which leads to a serialization failure when the request is processed. - The error message explicitly states that an object of type set is not JSON serializable, and the request cannot proceed. -It is a logical bug because Python set types are not serializable to JSON, and the application allows such values without sanitization, breaking the logic of JSON-compatibility in data structures.
Reproduction Steps	1. Set mutation_type = 'illegal_json_structure'. 2. Construct a standard input object with fields name, info, and price. 3. Apply the mutation to set a random = {'set': {1, 2, 3}}. 4. Submit the input to the server. 5. Observe the serialization exception with the message: Object of type

	set is not JSON serializable.
Proof of Concept (PoC)	Objective: This PoC tests the system's ability to validate input types and reject unsupported data structures before serialization. Expected Behavior: - The server should either sanitize or reject the set type early in the processing pipeline. - Ideally, non-serializable types should be flagged before reaching the JSON encoder. Actual Behavior: - A random field is set to a dictionary that includes a set: {'set': {1, 2, 3}}. - When the server attempts to serialize the input, it raises a TypeError due to the unsupported set object. - The request fails with an exception: Object of type set is not JSON serializable. Vulnerability Insight: This PoC reveals that the system does not perform pre-validation of JSON-compatibility before serialization. Such input could cause crashes, logging failures, or inconsistent behavior if not properly handled. Repeated use of illegal structures could be used in DoS-style testing to identify how gracefully the system fails.
Attachments	

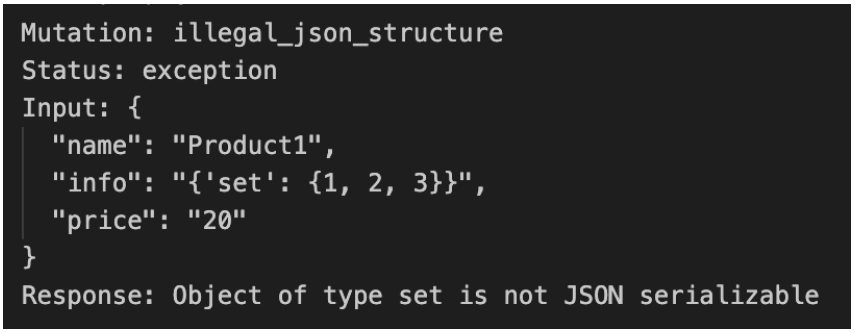


Image from summary.txt after running the fuzzer

```
elif mutation_type == 'illegal_json_structure':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = {'set': {1, 2, 3}}
```

Image from fuzzer mutate_input function

Bug Title	Database Integrity Error on Inserting Product with Non-existent Primary Key
Bug Description	- This bug is triggered by manually setting the pk field to a value (999999) that does not correspond to any existing product in the database. - When the server attempts to process the request, it violates the UNIQUE constraint on the id field (primary key) because the system attempts to insert a record using a primary key that already exists or is improperly handled. - This results in a 400 Bad Request, with an error indicating a database integrity failure. -It is a logical bug because the application incorrectly assumes user-supplied primary keys are valid and fails to verify their existence, leading to database-level constraint violations that should be caught by Django application-level logic.
Reproduction Steps	1. Set mutation_type = 'non_existent_product'. 2. Construct a standard input object and apply the mutation to set pk = 999999. 3. Submit the input to the endpoint responsible for creating or modifying products. 4. Observe the system return a 400 Bad Request with the error message: UNIQUE constraint failed: home_product.id.
Proof of Concept (PoC)	Objective: This PoC checks the system's ability to gracefully handle invalid primary key references during database operations. Expected Behavior: - The system should verify whether a referenced or provided primary key exists or is allowed before trying to perform database writes.

	- It should reject invalid references with a clear and sanitized error message — without reaching a database constraint violation. Actual Behavior: - The system accepts the input with pk = 999999, assuming the key is valid. - When the database operation is executed, it violates the UNIQUE constraint on the primary key field, resulting in a hard failure. - The request fails with the error: "UNIQUE constraint failed: home_product.id". Vulnerability Insight: This PoC exposes a lack of pre-checks on critical database constraints. Relying on the database to catch such issues increases the risk of data integrity violations, poor user experience, and potential DoS scenarios if exploited repeatedly (e.g., via fuzzing).

Attachments

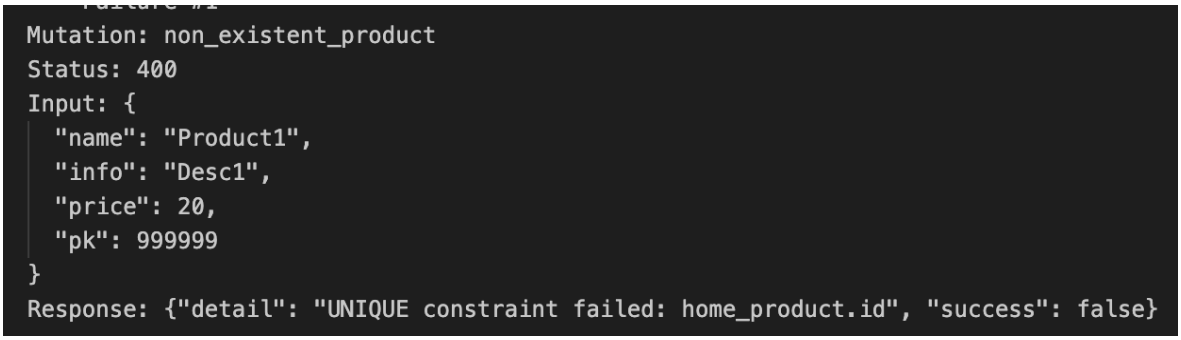


Image from summary.txt after running the fuzzer

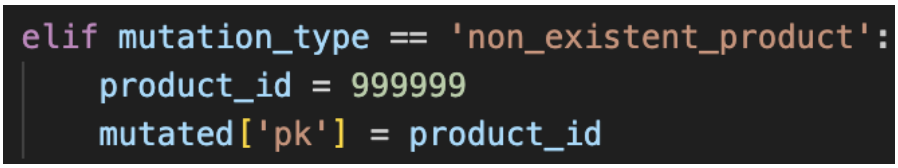


Image from fuzzer mutate_input function

Bug Title	NOT NULL Constraint Violation When the info Field is Set to Null
Bug Description	- This bug is triggered when a random field is explicitly set to null via the

	null_info mutation. - The backend database has a NOT NULL constraint on the home_product.info column. - When the request is processed, the system attempts to insert or update the record with a null value, violating the constraint and resulting in a 400 Bad Request. - It is a logical bug because the application fails to validate that required fields are non-null before writing to the database, relying on database constraints instead of enforcing its own logic rules.
Reproduction Steps	1. Set mutation_type = 'null_info'. 2. Construct an input object with "info": null. 3. Submit the payload to the server. 4. Observe the 400 Bad Request response indicating a NOT NULL constraint failure.
Proof of Concept (PoC)	Objective: This PoC checks how the system handles null assignments to required fields that are backed by non-nullable database columns. Expected Behavior: - The system should validate input fields before writing to the database and reject null values for required fields with a clear validation error. Actual Behavior: - A random field is assigned null. - The system does not intercept this before the database layer. - During the database operation, a NOT NULL constraint failed error is thrown for home_product.info. Vulnerability Insight: This test highlights a lack of validation enforcement at the application level. Depending on the database to enforce non-null constraints can result in poor user experience and unhandled server errors. This is especially risky if multiple critical fields are nullable in user inputs but non-nullable in the schema.
Attachments	

```
Mutation: null_info
Status: 400
Input: {
  "name": "Product1",
  "info": null,
  "price": 20
}
Response: {"detail": "NOT NULL constraint failed: home_product.info", "success": false}
```

Image from summary.txt after running the fuzzer

```python
elif mutation_type == 'null_info':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info' or field=='price':
        mutated[field] = None
```

Image from fuzzer mutate_input function

Bug Title	Logical Validation Error due to Extremely Large Input Values
Bug Description	**Large numeric value for 'price' field:** - This bug is triggered by assigning an excessively large numeric value (e.g., 10**200) to the price field in the extremely_large_value mutation. - SQLite cannot store this number because it exceeds the 8-byte INTEGER limit, resulting in a Python OverflowError. - It is a logical bug because the application fails to validate numeric limits compatible with the database backend, leading to a crash during persistence. **Large string value for 'name' & 'info' field:** - This bug also occurs when assigning an extremely large string (~3MB) to fields like name or info in the extremely_long_value mutation. - Django raises a RequestDataTooBig error(MemoryError) due to the request body exceeding DATA_UPLOAD_MAX_MEMORY_SIZE. - It is a logical bug because the application does not enforce reasonable input size constraints, causing server-level failures during request parsing.

Reproduction Steps	- Set mutation_type = 'extremely_large_value'. - The mutation randomly selects one of these fields (name, info, or price) to assign an extremely large value. ● If the field is name or info, a 3MB string ("X" * 3 * 1024 * 1024) is assigned. ● If the field is price, a very large integer (10 ** 200) is assigned. - (Optional) You may also manually assign the large value to a specific field instead of relying on randomness for more controlled testing. - Submit the mutated input to the server. - Observe the response: ● For large integers: an error with the message "Python int too large to convert to SQLite INTEGER". ● For large strings: a MemoryError with the message "request data too large".
Proof of Concept (PoC)	Objective: This PoC tests the system's ability to detect and reject extremely large numeric and string inputs that exceed backend or framework limitations, specifically targeting SQLite integer bounds and Django's request size limit. <u>Large Numeric Value Error:</u> Expected Behavior: - The application should validate and reject numeric values that exceed the maximum supported by the database (64-bit signed integer). - A clear error such as "Value too large for field 'price'" should be returned before attempting a database write. Actual Behavior: - Price field is set to a very large integer: 10 ** 200. - When the server attempts to save the data, Python raises an exception with the error message: "Python int too large to convert to SQLite INTEGER" - The request fails with a 400 status code and error response. <u>Large String Error:</u>

	Expected Behavior: - The application should enforce reasonable limits on string field sizes and reject oversized inputs with a clear error message like "Input too long" or "Field exceeds maximum allowed size". Actual Behavior: - A random field (either name or info) is assigned a string with a size of approximately 3MB: "X" * (3 * 1024 * 1024) - Django fails to process the request due to its default memory size limit, resulting in a memory error: “MemoryError: request data too large” - The request fails with a 400 status code and error response. Vulnerability Insight: This bug highlights two key validation failures in the system. First, the application does not enforce numeric bounds compatible with the underlying database, allowing oversized integers to propagate until SQLite rejects them, which can trigger backend exceptions and unstable behavior. Second, it permits unbounded string inputs, exceeding Django’s request body parsing limits and resulting in memory errors. Together, these issues expose the system to logic-layer inconsistencies, denial-of-service vectors, and a lack of graceful failure handling due to missing pre-validation for extreme input sizes.

Attachments

```
--- Failure #2 ---
Mutation: extremely_long_value
Status: 400
Input: {
  "name": "Product1",
  "info": "Desc1",
  "price": 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
}
Response: {"detail": "Python int too large to convert to SQLite INTEGER", "success": false}
```

Large Numeric value error: Image from summary.txt after running the fuzzer

Large String error: Image from summary.txt after running the fuzzer

```
elif mutation_type == 'extremely_large_value':
    field = random.choice(['name', 'info', 'price'])
    if field == 'name' or field == 'info':
        mutated[field] = "X" * (3 * 1024 * 1024)
    elif field == 'price':
        mutated[field] = 10 ** 200
```

Image from fuzzer mutate_input function

Logical Error (Using Metamorphic Testing)

Negative price logical bug:

Metamorphic transformation: Modify a valid product input by changing the `price` field from a positive value (e.g., `100`) to a negative value (e.g., `-10`), while keeping all other fields the same.

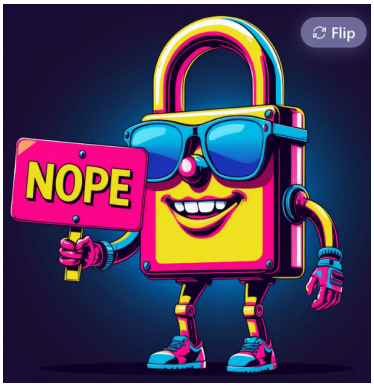
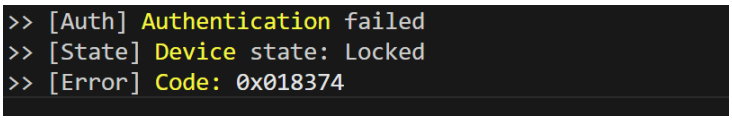
Metamorphic relation: If the price is negative, the system **should reject** the input with an appropriate validation error (e.g., `"Price must be a positive number"`).

Observed behaviour: **Metamorphic relation is not satisfied/ is violated**- The system **accepts** the input with a negative price (`-10`) without any validation errors.

This reveals a **missing validation rule** — the system fails to enforce that `price` must be non-negative, resulting in a **logical bug** that breaks expected domain constraints.

Unique Bugs Found in BLE

Bug Title	Authentication Failure Deadlock Due to Repeated "NOPE" Notifications (Code: 0x018374)
Bug Description	When a null BLE authentication passcode is sent to the device, instead of properly handling the error, the device responds repeatedly with "NOPE" notifications. This behavior clutters the communication channel and blocks any further authentication attempts by overwhelming the state machine. The device becomes stuck, unable to authenticate users correctly, essentially locking out legitimate users.
Reproduction Steps	Configure the BLE fuzzer to send an authentication request with a null or empty passcode. Observe the device's response behavior after receiving the malformed authentication request. Monitor the repeated "NOPE" notifications being sent from the device for every failed authentication. Attempt a valid authentication request after the repeated "NOPE" responses. Notice that further authentication is no longer possible
Proof of Concept (PoC)	Objective: Simulate sending an invalid authentication request with a null passcode to the BLE smart lock. Expected Behavior: The device should gracefully reject the invalid authentication once and allow further attempts afterward. Actual Behavior: The device spams "NOPE" notifications repeatedly. Future legitimate authentication attempts are blocked, effectively causing

	a denial of service on the authentication mechanism. The device remains in a stuck state without a manual reboot or reset. Vulnerability Insight: This vulnerability shows that improper handling of invalid authentication inputs can result in lockout conditions and denial of service (DoS) attacks against the smart lock system.
Attachments   The BLE device enters a stuck state after failed authentication, repeatedly sending "NOPE" responses and logging error code 0x018374.	

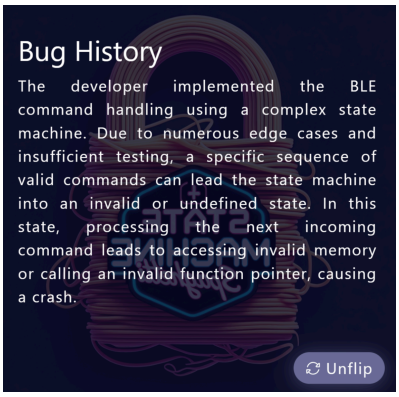
Bug Title	Improper Input Handling During Connection Setup and Rare BLE Events (Code: ASDJH$)
Bug Description	While the main BLE command path has basic input validation, a separate code path handles connection setup and infrequent events like pairing, resets, or device ID exchanges. This path is less tested and lacks robust input checks, allowing unexpected data during these early or rare phases to trigger crashes or undefined behavior.

	These issues can lead to denial-of-service scenarios or undefined internal state.
Reproduction Steps	Start a BLE session with the device and enter connection setup mode. Inject malformed or unexpected data packets during the initial handshake or during infrequent BLE triggers (e.g., power reset or timeout handling). Observe whether the device logs unexpected behavior, enters an invalid state, or crashes. Repeat across multiple event types to confirm instability.
Proof of Concept (PoC)	Objective: Test how the device handles malformed packets during non-standard BLE events or connection setup. Expected Behavior: The device should validate all inputs regardless of connection phase and reject malformed data cleanly. Actual Behavior: The system behaves inconsistently when malformed data is injected during rare BLE phases, often skipping validation logic or processing invalid memory. This can lead to crashes, state desynchronization, or permanent lockout until the device is reset. Vulnerability Insight: This bug highlights a blind spot in BLE input handling. While primary commands are well-guarded, auxiliary logic tied to setup or infrequent events is less protected, opening the door to subtle but dangerous vulnerabilities.

Attachments
`>> [Bluetooth] Received command: 0x08`
`>> [Bluetooth] Response: 0x02 (Invalid Command)`
`>> [Error] Code: 0x******`	

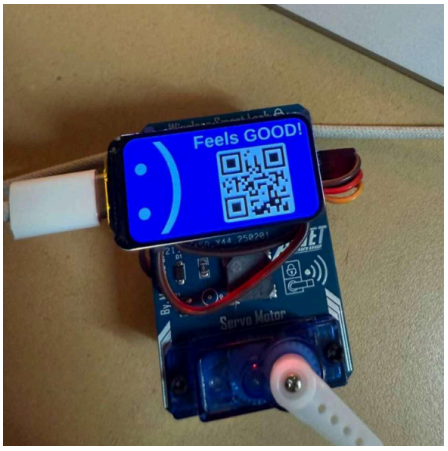
Error code 0x****** in serial port activates the lock's debug display, revealing unintended system behavior.

Bug Title	Invalid State Transition Leading to Crash (Code: 0x010203)
Bug Description	The BLE device uses a complex state machine to manage command handling. Due to numerous edge cases and incomplete testing, a specific sequence of valid BLE commands can transition the device into an invalid or undefined state. In this corrupted state, any further incoming command leads the device to access invalid memory or invoke an invalid function pointer, causing a full crash.
Reproduction Steps	Use the BLE fuzzer to send a crafted sequence of valid BLE commands that stress state transitions (e.g., Authenticate → Lock → Unlock → Authenticate again in quick succession). Observe that the device incorrectly transitions into an undefined state. Send any additional valid BLE command after the invalid state is reached. Watch the device crash or freeze due to invalid memory access or invalid function pointer execution.

Proof of Concept (PoC)	Objective: Simulate a series of BLE commands to push the device into an invalid or undefined state. Expected Behavior: The device should either reject illegal state transitions cleanly or recover back to a valid known state without crashing. Actual Behavior: After reaching an undefined internal state, any further incoming command causes the device to crash due to invalid memory access or execution of invalid code paths. Vulnerability Insight: This bug highlights a failure in robust state machine design and validation. Attackers can exploit it by sending crafted command sequences to destabilize the device, resulting in denial of service (DoS) through persistent crashes or lockups.
Attachments   BLE state machine into an undefined state, resulting in memory corruption and a device crash (Code: 0x010203)	

Bug Title	Unintended Debug Display Triggered by Input Length Overflow (Code: 0x*** from Serial Output)** error code KSMS&H
Bug Description	When a BLE packet exceeding normal length thresholds is sent to the

	device, the screen unexpectedly displays a debug-mode screen showing a QR code and the message “Feels GOOD!”.
Reproduction Steps	Use the fuzzer to mutate a valid BLE command by appending large random payloads beyond typical command sizes. Transmit the mutated command to the device while it is in an idle state. Observe the OLED screen switching to the “Feels GOOD!” debug display. Check the serial port log for code 0x*****, which appears to correspond to this fallback state activation.
Proof of Concept (PoC)	Objective: Stress test the BLE input handler with oversized payloads. Expected Behavior: The device should reject abnormally large inputs, display an error, or silently ignore the packet. Actual Behavior: The device triggers a debug screen with the message "Feels GOOD!" and displays a QR code. Serial output confirms a consistent value of 0x*****, likely linked to an internal debug condition or watchdog trigger. Vulnerability Insight: This behavior suggests the device may expose internal development or debug states when presented with malformed or oversized BLE input. While not directly exploitable, this can leak unintended system behavior, and in some cases, might be used to infer internal device logic or trigger denial-of-service conditions.
Attachments	



QR code displayed on the screen which spells out error code KSMS&H

```
>> MEPC    : 0x4205e9ac  RA      : 0x4205f974  SP      : 0x4083d8f0  GP      : 0x40816434
>> TP      : 0x4083dc30  T0      : 0x40028192  T1      : 0x00000000  T2      : 0x8102d840
>> S0/FP   : 0x00000000  S1      : 0x00000000  A0      : 0x00000000  A1      : 0x4081c7a8
>> A2      : 0x00000000  A3      : 0x000000ab  A4      : 0x00000001  A5      : 0x00000001
>> A6      : 0x0000013f  A7      : 0x00000000  S2      : 0x40833c07  S3      : 0x4083dba0
>> S4      : 0x00000000  S5      : 0x00000000  S6      : 0x00000000  S7      : 0x00000000
>> S8      : 0x00000000  S9      : 0x00000000  S10     : 0x00000000  S11     : 0x00000000
>> T3      : 0xffffe040  T4      : 0x02249440  T5      : 0x83dc3040  T6      : 0x81643440
>> MSTATUS : 0x00001881  MTVEC   : 0x40800001  MCAUSE  : 0x00000005  MTVAL   : 0x00000008
>> MHARTID : 0x00000000
>>
>> Stack memory:
>> 4083d8f0: 0x42116fac 0x00000000 0x4081d450 0x4206ecb2 0x0000000c 0x00000008 0x40829000 0x42063d02
>> 4083d910: 0x40833c07 0x00000008 0x00000000 0x4083dba0 0x40833c07 0x00000008 0x40829000 0x42001b2a
>> 4083d930: 0x4083dba0 0x00000003 0x00000002 0x42002b8a 0x40880000 0x40833c04 0x4082ceb0 0x400225ca
>> 4083d950: 0x00000000 0x4083d9b4 0x40833bdc 0x408054ca 0x00000000 0x4083d9b4 0x40833e10 0x40814b1a
>> 4083d970: 0x4083d994 0x00000003 0x00000015 0x42009a96 0x00000000 0x4083d9b4 0x40833bdc 0x42009bce
>> 4083d990: 0x00000001 0x00000000 0x002a0000 0x20001000 0x00000008 0x00000001 0x00000000 0x00000000
>> 4083d9b0: 0x00000000 0x002a0002 0x00010000 0x82ce0200 0x0225ca40 0x00000040 0x83d9b400 0x833bdc40
>> 4083d9d0: 0x8054ca40 0x00000040 0x83d9b400 0x833c0440 0x814b1a40 0x83d99440 0x00000340 0x00001500
>> 4083d9f0: 0x009a9600 0x00000042 0x83d9b400 0x833bdc40 0x009bce40 0x00000142 0x00000000 0x2a000000
>> 4083da10: 0x00100000 0x00000820 0x00000100 0x00000000 0x00000000 0x00000000 0x2a000200 0x01000000
```

Figure 7: Serial port brain dump during a crash

As shown in Figure 7, although the device typically returns error code 0x02 for unknown BLE commands, we encountered frequent crashes and disconnections when sending malformed or unexpected inputs. On the serial port, we observed the following message: **"Guru Meditation Error: Core 0 panic'ed (Store access fault). Exception was unhandled. Core 0 register dump**:", indicating a critical system failure. The accompanying stack trace and register dump strongly suggest that the device entered an invalid state, likely due to unhandled edge cases within the command parser. This behavior implies that certain inputs bypass the standard error handling routines entirely, resulting in memory access violations and broader device instability.

Lessons Learned

For BLE:

We discovered that embedded systems like BLE devices present unique fuzzing challenges compared to traditional software. The device's responsiveness degraded over time, especially after multiple reconnections and state transitions, making long fuzzing sessions less effective. Fast state changes, like those during "Authenticating," were difficult to monitor without more precise timing mechanisms. Additionally, while our fuzzer caught many logic and state-based bugs, it likely missed deeper memory vulnerabilities due to limited device-side visibility and reliance on external logs alone.

For Django:

Django fuzzing highlighted how web applications, while seemingly more resilient, are vulnerable to logical flaws, type mismatches, and malformed data inputs that bypass typical validation. A key takeaway was the importance of robust pre-serialization validation. Many of the discovered bugs, such as crashes due to circular references or class object injection, revealed how failures in defensive programming can propagate all the way to runtime. We also noted that relying solely on HTTP status codes was insufficient; deeper inspection of response content and custom oracles proved necessary for uncovering subtle vulnerabilities.

Future Works

For BLE:

Future improvements should include dynamic timing adjustment to account for laggy device responses, especially after prolonged fuzzing. Short-lived states like "Authenticating" could be tracked using lightweight device-side instrumentation or shadow state machines. Adding protocol-aware mutation logic (e.g., knowing command structure or expected sequences) could help in exploring deeper paths. Integration of hardware-in-the-loop tools or trace-based debugging would allow us to catch low-level memory and concurrency issues.

For Django:

There is strong potential to extend our Django fuzzing to more endpoints, especially those involving user permissions, database relationships, or third-party integrations. Introducing schema-awareness (e.g., through JSON Schema or Django model introspection) could help us design mutations that intelligently target edge cases without devolving into random noise. Enhanced coverage measurement using line, branch, and path coverage tools would also provide more accurate insights into how well inputs explore the codebase.

General Extensions:

The UnifiedFuzzer could be extended to new domains such as REST APIs built with Flask/FastAPI, or even protocol stacks like MQTT and CoAP. Adding a plugin system for target-specific mutation modules would make the framework easier to extend. Lastly, integrating machine learning for input prioritization or anomaly detection could guide the fuzzer toward unexplored or interesting states faster.

Conclusion

This project successfully developed a unified fuzzing framework capable of testing both web applications and embedded devices through a common architecture. Our fuzzer exposed critical security and reliability flaws in two very different systems—Django and a BLE smart lock—demonstrating its effectiveness in revealing real-world vulnerabilities like authentication bypasses, command injections, server crashes, and data validation failures.

The project's strength lies in its balance between generalization and specialization: we maintained a shared fuzzing loop and input management strategy, while supporting domain-specific behaviors for each application. We showed that even lightweight fuzzers, when carefully designed, can reveal deep issues such as state machine corruption, logical validation errors, and serialization failures.

Going forward, this work lays a strong foundation for expanding to other protocols and platforms. By incorporating more advanced analysis techniques and supporting more dynamic input structures, the fuzzer can evolve into a powerful tool for continuous security and reliability testing in heterogeneous environments.