Assignment 1

Group 18
Firas Saleh **Henrique Luz** **Marina Toledo**

February 8, 2026

1 Report

Describe your solution, addressing the following questions

- How did you approach the assignment?
- What did you try?
- What worked and what did not work?
- What have you discovered when you decrypted the file?
- What is the security model in this assignment? What is the threat model? What security requirements are being violated?
- What have you learned from this exercise?
- Any other insights?

2 Introduction

SQL injection remains one of the most prevalent and dangerous vulnerabilities in web applications. This assignment investigates a blind SQL injection attack against a password reset service implemented in Node.js with Express and sqlite3. Unlike traditional SQL injection attacks where query results are directly visible, blind SQL injection requires attackers to infer information through observable side effects, such as application behavior or response timing.

The target application accepts a username and queries the database using string concatenation without parameterization, creating an SQL injection vector. The service always returns the same message regardless of query success, eliminating content-based information disclosure. However, by leveraging timing differences induced by database operations, we can extract sensitive data character by character.

This report details our methodology, optimizations, and theoretical analysis of this timing-based attack.

3 Vulnerability Analysis

3.1 Attack Surface

The vulnerable endpoint constructs SQL queries through direct string concatenation:

```
"select * from users where username='" + user + "'"
```

This construction allows arbitrary SQL code injection when the `user` variable contains malicious input. The absence of input validation, parameterized queries, or prepared statements creates a critical security vulnerability.

3.2 Database Schema

The database contains a `users` table with three columns: `username`, `passwordhash`, and `key`. The `key` column stores the user's private PGP decryption key in ASCII format.

3.3 Attack Objective

The goal is to retrieve the private key stored in the `key` field for user `admin`. The key follows the standard PGP private key block format, beginning with `-----BEGIN PGP PRIVATE KEY BLOCK-----` and ending with `-----END PGP PRIVATE KEY BLOCK-----`. Since the application provides no direct output channel for query results, we must employ an inference-based approach.

4 Methodology

4.1 Timing-Based Boolean Inference

Our approach is inspired by Meer and Slaviero's work on timing-based attacks. The fundamental technique involves constructing SQL queries that conditionally introduce time delays based on boolean predicates. By

measuring response times, we can determine whether specific conditions are true or false. We inject the following payload into the username field:

```
admin' AND condition AND sleep(sleeptime) --
```

Our custom sleep function creates a SQL condition that forces the database to generate a large random blob. The function randomblob(n) in SQLite3 generates random data of size n bytes. By requesting a large blob (with size controlled by the sleeptime parameter), we force the database to perform significant computational work. We calibrated the blob size to produce delays of approximately sleeptime seconds. The sleep condition compares this blob to a ranodm value (e.g., randomblob(sleeptime*100000000) = '1234'), which evaluates to false, but the computational cost lies in the blob generation itself, not the comparison.

Due to short-circuit evaluation of the AND operator in our injection payload (condition AND sleep(sleeptime)), if condition evaluates to false, the database returns immediately without generating the blob. If condition is true, the database must generate the blob before evaluating the comparison, introducing a measurable delay.

4.2 Initial Approach: Brute Force Character Matching

Our initial approach involved testing each character position against all possible ASCII characters (0-127). For each position i, we would inject:

```
substr(key, i, 1) = 'c'
```

for each candidate character c. With uniform distribution of characters, this requires an average of 64 requests per character (128/2 expected comparisons until finding the match). Assuming the target key has length n characters, and denoting the baseline response time as t_{base} and the delay time as t_{delay}, the expected total time for brute force is:

$$T_{brute} = n \times (63 \cdot t_{base} + t_{delay})$$

4.3 Optimization: Binary Search

To significantly reduce the number of requests, we implemented a binary search algorithm. Instead of testing each character value, we test whether the character's Unicode value is less than or equal to a midpoint:

```
unicode(substr(key, i, 1)) <= mid
```

The algorithm maintains bounds $[low, high]$ initialized to $[0, 127]$ for ASCII characters. At each iteration, we test the midpoint $mid = (low + high)/2$. If the condition is true (indicated by a long response), we update $high = mid$; otherwise, $low = mid + 1$. The search terminates when $low == high$.

This reduces the worst-case number of queries per character to

$$\lceil \log_2(128) \rceil = 7,$$

with an average of approximately

$$\log_2(128) - 1 = 6$$

queries. The expected time becomes:

$$T_{\text{binary}} = L \cdot 6 \cdot t_{\text{base}} \quad \text{(assuming ideal conditions)}$$

However, this assumes we never encounter delayed responses. In practice, we must account for true positive responses and verification overhead.

4.4 Challenges and Solutions

4.4.1 Network Variability

Network latency and server load create non-deterministic response times. Even when the condition is false, response time may occasionally exceed our threshold $t_{\text{threshold}}$, causing false positives. If we denote the probability of such spurious delays as p, then without mitigation, our results would be unreliable.

4.4.2 Initial Solution: Retry on True

Our first mitigation strategy involved retrying the same condition n times whenever we detect a delayed response. If all n retries also show delays, we conclude the condition is truly true. This reduces the false positive probability to p^n.

However, this approach severely degrades performance. In binary search, we expect approximately 3 true responses per character on average (half of the 6 queries return true). Each true response now requires $n + 1$ delayed responses (the initial detection plus n retries), giving expected time per character:

$$T_{\text{char}} = 3 \cdot t_{\text{base}} + 3 \cdot (n + 1) \cdot t_{\text{delay}}$$

With $n = 3$ retries and $t_{\text{delay}} = 2$ seconds, this becomes approximately 24 seconds per character, making the optimization counterproductive.

4.4.3 Improved Solution: Negative Condition Validation

We developed a more efficient validation strategy: when detecting a potential true response, immediately test the negation of the condition. We inject:

```
admin' AND NOT(condition) AND randomblob(2*sleeptime*100000000)='1' --
```

If the original condition is truly true, the negation must be false, resulting in a fast response. If we observe a fast response to the negation, we confirm the original was genuinely true. If the negation also produces a delay, the original detection was a false positive caused by network issues.

Note that we double the sleep time in the negation test ($2 \cdot \text{sleeptime}$). This accounts for potentially persistent network congestion—if delays are due to network issues rather than query logic, the doubled sleep ensures we still observe a timeout, preventing false confirmations.

4.4.4 Adaptive Sleep Time

Network conditions change dynamically. We implemented adaptive threshold adjustment: when detecting false positives (negation test fails), we double the `sleeptime` parameter. When conditions stabilize (several consecutive tests without false positives), we gradually reduce `sleeptime` to maintain efficiency. This balances robustness against varying network conditions with minimal performance overhead.

References