Complexity Analysis between Bruteforcing and Blind SQL Injection with LIKE-Based Data Exfiltration

Muhammad Jibril Ibrahim - 13523085[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*mjibrahimcollege@gmail.com*, *13523085@std.stei.itb.ac.id*

***Abstract*—The continuous evolution of cyber threats underscores the critical need to understand the methods employed to compromise password security. This paper investigates and compares two prevalent techniques: brute-forcing and SQL injection with LIKE-based exfiltration. Brute-forcing involves an exhaustive search through all possible password combinations, making it computationally intensive, particularly for robust hashing algorithms. Conversely, SQL injection leverages vulnerabilities in database systems to retrieve sensitive data efficiently, with LIKE-based patterns enabling partial password recovery through targeted queries. By analyzing the algorithmic complexity, hash dependencies, and practical effectiveness of these methods, this study highlights their relative computational demands and implications for system security. The findings aim to inform the development of more resilient security mechanisms and best practices in safeguarding credentials against these attacks.**

***Keywords*—Algorithm Complexity, Bruteforcing, SQL, SQL Injection, Password cracking**

I. Introduction

Data exfiltration is a critical concern in cybersecurity, especially when dealing with poorly secured applications vulnerable to attacks like brute force or SQL injection. Two common approaches to extracting sensitive information, such as password hashes, are brute forcing and blind SQL injection with LIKE-based exfiltration. This paper examines the complexity of these methods, focusing on their computational and algorithmic characteristics.

Brute forcing relies on systematically trying all possible combinations until the correct one is found, making it a straightforward but often resource-intensive technique. On the other hand, blind SQL injection with LIKE-based exfiltration involves querying a database in a manner that reveals data bit by bit or character by character, leveraging feedback from the application to refine guesses. This method is slower but more covert, often evading basic security measures.

The context for this exploration is inspired by a Capture the Flag (CTF) challenge where participants were tasked with extracting a password hash to gain administrative access. Understanding the underlying complexities of these approaches is essential not only for CTF enthusiasts but also for professionals seeking to secure applications against such attacks.

This paper explores the algorithmic complexity of brute-forcing and SQL injection with LIKE-based exfiltration, focusing on their application to password cracking. By examining the computational requirements and practical implications of each method, we aim to provide a nuanced understanding of their relative strengths, weaknesses, and impact on modern cybersecurity practices.

II. Fundamental Theorem

A. Combinatorics

Combinatorics is a branch of mathematics concerned with counting the arrangements of objects without the need to enumerate all possible configurations explicitly. In combinatorics, it is essential to calculate all possible arrangements of objects. Two fundamental principles often applied as calculation methods in combinatorics are the rule of product and the rule of sum.

- Rule of Product
  If two independent experiments are performed, where the first experiment results in p possible outcomes and the second experiment results in q possible outcomes, then performing **both** experiments will yield $p * q$ possible outcomes.

- Rule of Sum
  If two independent experiments are performed, where the first experiment results in p possible outcomes and the second experiment results in q possible outcomes, then performing **either** the first or the second experiment will yield $p + q$ possible outcomes.

Two key concepts in combinatorics are permutation and combination. These concepts help determine the number of ways to select or arrange objects under specific constraints.

- Permutation
  A permutation of r objects chosen from n elements, denoted as $P(n, r)$, refers to the number of possible arrangements of r objects selected from n, where $r \leq n$ and no object is repeated in any arrangement. Permutations are used when the order or position of objects is significant. The formula of permutation is as follows:
  $$P(n, r) = \frac{n!}{(n-r)!}$$

- Combination
  A combination of r objects chosen from n elements, denoted as $C(n, r)$, refers to the number of ways to select r objects from n elements without regard to the order of selection. Combinations are used when the

order or position of objects is not important. The formula of combination is as follows:

$$C(n,r) = \frac{n!}{r!\,(n-r)!}$$

B. Algorithm Complexity

In programming, developing algorithms that are both accurate and efficient is of utmost importance. The efficiency of an algorithm is measured based on the time and space it consumes during execution. An efficient algorithm is one that minimizes both time and memory usage. These measurements are collectively referred to as an algorithm's complexity, which provides insight into how well the algorithm performs under various conditions.

Algorithmic complexity is categorized into two main types: time complexity and space complexity. Time complexity measures the number of computational steps required as a function of the input size (n). It focuses on how the execution time of an algorithm scales with the size of its input. In contrast, space complexity assesses the amount of memory required for an algorithm to run, including memory for variables, data structures, and function calls.

Often, exact details about time complexity are less critical than understanding how an algorithm's runtime grows with increasing input size. This growth is represented using Big-O notation, a mathematical notation that describes the upper bound of an algorithm's computational complexity. Big-O notation provides an asymptotic analysis, focusing on the term with the highest order of growth. For example, an algorithm with a time complexity of $O(f(n))$ indicates that $f(n)$ represents the dominant term governing its performance as the input size increases.

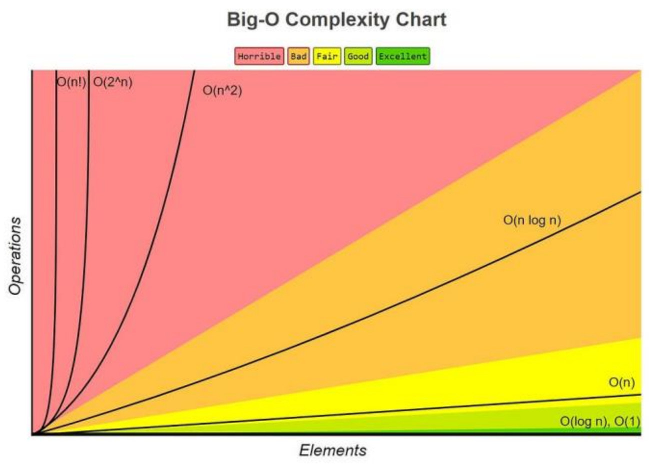


Image 2.1 Big-O complexity chart
(Source: https://www.bigocheatsheet.com/)

Although one may not see or feel the difference of complexity with a small number of input n. When using a huge amount of input, the difference becomes strikingly clear. For example, an input amount of 10^9 when computed with an algorithm of complexity $O(n)$ and assuming a single operation takes about 1 ns or 10^{-9} second then the algorithm will only take 1 seconds. While an algorithm of complexity $O(n \log_2 n)$, when given the same amount of input, would take nearly 30 seconds. That is 30 times longer than the previous algorithm.

C. Hash

Hashing is a fundamental concept in computer science and cryptography, defined as the process of mapping data of arbitrary size to a fixed-size value using a mathematical function known as a hash function. The output of this process, often referred to as a hash value, digest, or checksum, serves as a unique representation of the input data.

The primary purpose of hashing is to facilitate efficient data retrieval, comparison, and integrity verification. Hashing is extensively employed in various applications, including database indexing, cryptographic protocols, digital signatures, and password storage.

According to Rogaway and Shrimpton in their work on cryptographic hash functions, an effective hash function exhibits the following 3 essential properties, Preimage Resistance, Second-Preimage Resistance, and Collision Resistance.

Preimage Resistance means it should be computationally infeasible to reverse-engineer the input data (preimage) from its hash value. This property ensures the security of sensitive information that is hashed, such as passwords.

Second-Preimage Resistance means that given a hash value and its corresponding input, it should be infeasible to find a different input that produces the same hash value. This property prevents malicious tampering of data to create identical hash outputs.

Collision Resistance means it should be infeasible to find two distinct inputs that produce the same hash value. This property is crucial for ensuring the uniqueness of hash values, particularly in digital signatures and integrity verification.

As mentioned before, one application of hashing is password storage. In today’s data-driven world, a secure and feasible hashing function is not just an option, it is a necessity. One such hashing function that is used for password storage is *Bcrypt*, a hash based on the Blowfish cipher.

D. Brute Force

A brute-force algorithm is a fundamental computational approach that systematically enumerates all possible solutions to a problem to identify the correct one. This method does not rely on advanced strategies but instead exhaustively tests each potential solution. The algorithm guarantees that if a solution exists, it will eventually find it, given enough time and resources.

However, brute-force algorithms are highly inefficient for large problem spaces due to their exhaustive nature. As the size of the problem grows, the number of possible combinations increases exponentially, making the algorithm increasingly slow and resource-intensive. Despite this inefficiency, brute-force algorithms are simple to implement and can be relied upon when no better optimization techniques are available.

In practical applications, brute-force attacks are commonly used in cybersecurity for password cracking, where an attacker attempts every possible password until the correct one is found. This guarantees success, but the process can be extremely time-consuming, especially for long or complex passwords.

E. SQL Injection

An SQL injection attack involves embedding or "injecting" malicious SQL queries into input fields provided by a client application. This attack allows unauthorized commands to be executed on the backend database. If successful, SQL injection exploits can lead to the extraction of sensitive information, unauthorized data manipulation (insert, update, delete), administrative operations on the database (e.g., shutting down the database management system), and even file retrieval from the DBMS file system. In certain cases, attackers may execute operating system commands via the database. As a subset of injection attacks, SQL injection specifically targets the execution of unintended SQL commands by manipulating input data.

The root cause of SQL injection vulnerabilities simple and well-known, insufficient validation of user inputs. Developers have proposed numerous coding guidelines to address this issue, emphasizing defensive programming practices, such as proper input encoding and rigorous input validation. While these techniques are effective when applied systematically, they rely heavily on human effort, making them prone to errors. Addressing SQL injection vulnerabilities in legacy codebases is particularly challenging, as it requires significant manual effort to review and fix potentially insecure code. As a result, preventing SQL injection vulnerabilities demands a combination of rigorous coding standards, automated tools for input validation, and regular security audits.

Consider a web application with a login form that accepts a username and password. The application then executes the following SQL query to authenticate the user:

```
  SELECT * FROM users WHERE username =
'$username' AND password = '$password';
```

In a vulnerable application, that is, if the application does not properly validate user input, an attacker could exploit this by entering the following values into the username and password fields:

- Username: `Admin`
- Password: `' OR '1'='1`

Given this input, The resulting query to the database would be:

```
  SELECT * FROM users WHERE username =
'Admin' AND password = '' OR '1'='1';
```

Because of the `OR '1'='1'` part, the password filter would always equal to `TRUE` and would log you in as the admin. This is of course a major security vulnerability. Where its not just limited to logging in as another user, It could lead data manipulation and extraction.

F. Blind SQL Injection

Blind SQL injection is a subtype of SQL injection where an attacker exploits a vulnerability without directly observing the database's response. Unlike traditional SQL injection, where error messages or query results are visible, blind SQL injection relies on indirect clues, such as variations in page behavior or response times, to infer information about the database.

There are 2 types of Blind SQL Injection, Boolean-based and Time-based. In Boolean-Based Blind SQL Injection, attackers inject SQL queries that return either a true or false result, causing changes in the application's behavior based on the query's outcome. By analyzing these behavioral differences, attackers can infer details about the database structure or its contents. On the other hand, Time-Based Blind SQL Injection involves injecting SQL commands that intentionally cause delays with function such as SLEEP. By observing the response time, attackers can determine whether specific conditions hold true, enabling them to systematically extract data.

III. Analysis

A. Basic Premise

The problem stated here will be the premise to be solved by the algorithm method, brute forcing and like-based SQL Injection. The premise is as follows:

Given a software application with a login system that connects to a database. The app saves the password data in the database after hashing it with Bcrypt hash. The hash has a length of 60 characters that consist of alphanumeric letters, uppercase and lowercase, and 3 special characters ($, . , /). The goal of this premise is to exfiltrate the password hash of the admin. The correctness of the inputted answer can be tested by the SQL query for the login function. The normal SQL login query is as such,

```
  SELECT * FROM users WHERE username =
'$username' AND password = '$password';
```

Where it checks if the user exist and has the correct password.

B. Brute force

The brute force method is by simply trying every possible combination of the password hash. Given the hash has a length of 60 characters and consist of alphanumeric letters and 3 special character, on a single character of the hash it will have 65 possible characters.

Remember that the goal is to exfiltrate the password hash. Logging in will merely be our Boolean-based Blind SQL Injection which would check if our input is correct or not. The SQL Injection for this method is as follows:

- Username: `admin’ AND pwhash = ‘<Input>’--`
- Password: `random`

On a correct inputted hash, we would then be logged in. while if our inputted is incorrect, we would not be logged in. the password can be anything because in the Username input we give “`--`“ which means to ignore the command after the symbol so it will ignore the password input. This is the basis of the SQL Injection. We now just need to calculate the complexity.

Using combinatorics rule of product, we can find the amount of all possible combinations of the password hash. With 65 possible character with 60 character length, we can calculate that the amount of possible combination is $65^{60} \approx 8 * 10^{107}$. This is a huge number. If someone really try to brute force this, with a computer that can calculate a single operation in 1 ns. It would take them about $8 * 10^{98}$ seconds. To put in perspective, the age of the universe is estimated to be $4.3 * 10^{17}$ seconds.

This shows that Bruteforcing on big numbers is unfeasible and practically impossible with our current computational technology. Giving it another perspective, it means that our security system is secure enough as long as it is implemented correctly

C. LIKE-Based SQL Injection

The LIKE command in SQL is used to perform pattern matching within string data. It is commonly used in WHERE clauses to filter rows based on whether a column's value matches a specified pattern. This pattern matching supports wildcard characters to define patterns.

One such wildcard character is "`%`" which is essential for this method. This wildcard matches for zero or more characters. For example, querying with "`name LIKE 'A%'` " would match for "Anna", "Andrew", "Agus", and so on.

As is in the brute forcing method, Logging in will merely be our Boolean-based Blind SQL Injection which would check if our input is correct or not. The SQL Injection for this method is as follows:

- Username: `admin' AND pwhash LIKE '<Input>%'--`
- Password: `random`

Unlike in the brute forcing method, we would be logged in not just when our input is fully correct but also if our input matches the first part of the password hash. In another words, we would be logged in if the password hash starts with our input.

This is much, much more efficient than the brute forcing method as we can iteratively check the correctness of our inputted characters. We can use combinatorics to calculate the number of possible combinations. But unlike in the brute forcing method, we can use the rule of sum as now every character is independent of each other. With 65 possible character with 60 character length, we can calculate that the amount of possible combination is $65 * 60 = 3900$ amount of possible combination. This is an astronomical decrease from the brute force method. What would have taken more than the age of the universe can be shorten into as little as 3.9 micro seconds.

IV. Implementation

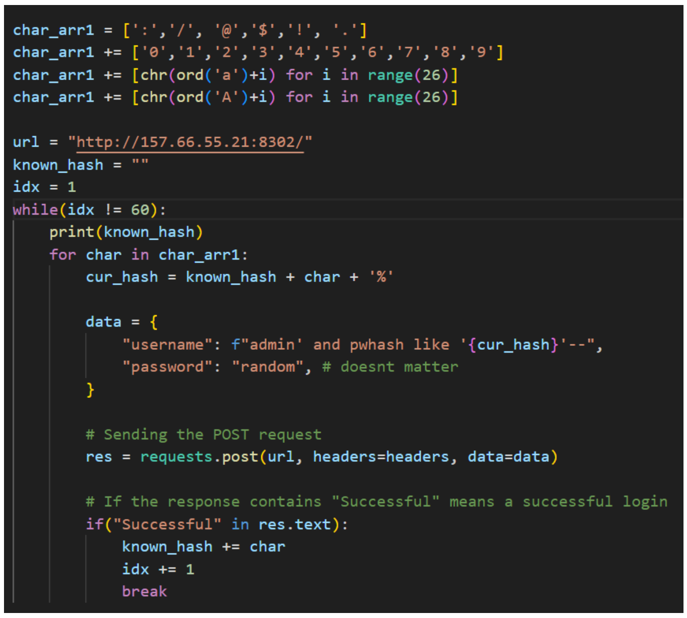
A. Brute force

The Brute force code try for every combination of the 65^{60} possible combination. It would mean that it consist of 60 nested for-loops with each loops is a for-loop with the length of 65. I skipped the implementation of the brute force method as it is not feasible both in theory and practically.

B. LIKE-Based SQL Injection

We first create the list that contains all 65 possible characters that might appear in the hash, including 3 special characters (:, . , /.), digits (0-9), and both lowercase and uppercase letters (a-z, A-Z). The script starts with an empty `known_hash` string and iteratively builds it one character at a time. At each step, it appends a candidate character to the known portion of the hash and uses a wildcard (%) to account for the unknown remaining characters.

The injection is crafted within the username parameter of the login payload. The SQL query checks if the hashed password (`pwhash`) in the database starts with the guessed hash (`cur_hash`).



```python
char_arr1 = [':','/', '@','$','!', '.']
char_arr1 += ['0','1','2','3','4','5','6','7','8','9']
char_arr1 += [chr(ord('a')+i) for i in range(26)]
char_arr1 += [chr(ord('A')+i) for i in range(26)]

url = "http://157.66.55.21:8302/"
known_hash = ""
idx = 1
while(idx != 60):
    print(known_hash)
    for char in char_arr1:
        cur_hash = known_hash + char + '%'

        data = {
            "username": f"admin' and pwhash like '{cur_hash}'--",
            "password": "random", # doesnt matter
        }

        # Sending the POST request
        res = requests.post(url, headers=headers, data=data)

        # If the response contains "Successful" means a successful login
        if("Successful" in res.text):
            known_hash += char
            idx += 1
            break
```

We continue this injection with the guessed `cur_hash` character by character until we have all 60 characters of the password hash. We have successfully exfiltrate the password hash data in an efficient and quick method.

V. Conclusion

This analysis highlights the stark contrast between brute forcing and LIKE-based SQL injection in terms of efficiency and feasibility for extracting password hashes. Brute forcing, while conceptually straightforward, is computationally impractical for modern hashing algorithms like Bcrypt. The exponential growth of possible combinations, combined with the immense computational time required, renders brute-force attacks infeasible, underscoring the robustness of secure hashing practices when implemented correctly.

In contrast, LIKE-based SQL injection demonstrates the power of leveraging database query capabilities for efficient data exfiltration. By iteratively testing each character of the hash and using the database's response as feedback, this method drastically reduces the number of required operations. The shift from exponential to linear complexity makes this approach not only feasible but also highly effective in practice.

The study of these techniques highlights the importance of robust input validation, parameterized queries, and other secure coding practices to mitigate vulnerabilities. While brute forcing relies on sheer computational power, LIKE-based SQL injection exploits logical weaknesses in application design. Together, they emphasize the necessity of a multifaceted approach to security, combining strong cryptographic measures with rigorous application-level protections.

By understanding the relative strengths and weaknesses of these approaches, developers and security professionals can better defend against attacks while appreciating the computational challenges faced by attackers. This underscores the ongoing importance of both cryptographic advancements and secure software design in safeguarding sensitive information.

VI. ACKNOWLEDGMENT

The author expresses heartfelt gratitude to God Almighty, Allah Subhanahu wa Ta'ala, for His blessings and guidance, enabling the completion of this paper titled "Complexity Analysis between Bruteforcing and SQL Injection with LIKE-Based Exfiltration on Password Cracking" in a timely manner. The author extends sincere appreciation to their parents and friends for their unwavering support and encouragement, particularly in providing mental motivation throughout the writing process. Special thanks go to Dr. Rinaldi Munir, the lecturer for Discrete Mathematics K1 in the 2024/2025 academic year, for sharing valuable knowledge and providing guidance during the learning process. The author is also deeply grateful to Dr. Rila Mandala, one of the lecturers for the same course, for delivering extensive materials and references that were instrumental both during the lectures and in the preparation of this paper. Lastly, the author would like to thank all other individuals and parties who contributed to the completion of this paper.

REFERENCES

[1] G. O. Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance." Fast Software Encryption, Lecture Notes in Computer Science, vol. 3017, pp. 371–388, Springer, 2004.

[2] A. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[3] P. G. J. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE), Arlington, VA, USA, Mar. 2006, pp. 13–15.

[4] OWASP. (20225, January 7). SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection

[5] Portswigger. (20225, January 7). Blind SQL Injection. https://portswigger.net/web-security/sql-injection/blind

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Januari 2025

[illegible]

Muhammad Jibril Ibrahim
13523085