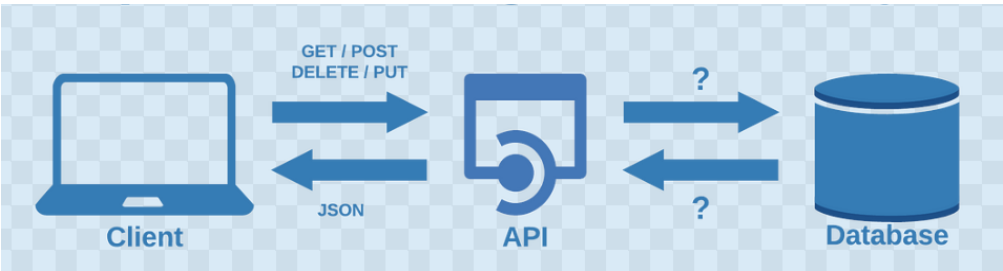
API Anti-Patterns and Vulnerabilities

https://github.com/matt1600/APIAntiPatterns

Author: Matt Walravens

The adoption of APIs has been accelerating as businesses use them for both internal and external operations. As API usage continues, it's important that certain security flaws and vulnerabilities are addressed. This article will dive into a couple security flaws that exist in a simple Python API.



The tech stack includes containerization with Docker Compose, Python with Flask to implement the web API, and Redis database. Docker Compose serves as a simple and portable way to containerize applications that can run on most machines with limited configuration. With Python and Flask, little code is required to implement the API, and the code is english-like and easy to read. In addition, Python comes with a lot of extensions and libraries. However, this is a fun sized example, and Flask is not recommended for large projects that require high scalability. The same goes for Redis. In this example, Redis is perfect because it is an open source, in-memory database that is easily configued with Python and Docker.

First mispattern: Assigning identification sequentially

The first anti-pattern existing in this example is the sequential assigning of identification numbers. In this if statement, user IDs are assigned and stored within Redis numerically.

```
if request.method == 'POST':
    #each time post is called, increment the user count
    redis.incr('posthits')
    postCounter = str(redis.get('posthits'),'utf-8')
    user = "user" + postCounter
```

This approach will give an attacker the chance to guess a low number for a user Id and potentially gain access to that user's data. Not to mention that this method could result in similar ID numbers existing in a distributed environment with multiple databases, and merging DBs would result in duplicate ID numbers.

The solution here is to avoid sequential identification and use either a Universal Unique Identifier(UUID) or a Globally Unique Identifier(GUID). This ensures uniqueness and easy

identification across different systems. There is a 99.99% chance that a group of $3.26*10^{16}$ will not contain any duplicates, making it extremely hard for attackers to guess a potential ID. This solution is easily implemented with a Python library import.

```
import UUID
    if request.method == 'POST':
    iD = uuid.uuid4()
    user = "user" + iD
```

Another common API flaw that exists in this example is not protecting against injection. SQL and other types of injection have the ability to destroy your database, and it is one of the most common web hacking techniques. It works by placing malicious code in an SQL statement, via web page input.

There exists certain client input that, if entered correctly, can access and alter the database. Depending on what the SQL statement is, this can be very dangerous. For example, say a web page is asking for a userID, and then that userID gets inserted into a query. The attacker enters *105 or 1=1* into the webpage and then this query compiles and executes.

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

Since *or 1=1* is always true, an attacker could get access to all the usernames and passwords in that database.

The best way to prevent this is with parameterized queries. Instead of allowing the user to give direct input, we prepare the query structure first and use placeholder for user input.

```
// Vulnerable SQL statement
$vuln = "SELECT Id FROM Users WHERE Username='$username' AND Password='$password'";
$result = $mysqli->query($vuln);

// Safe prepared statement
$stmt = $mysqli->prepare("SELECT Id FROM Users WHERE Username=? AND Password=?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();

// Unsafe prepared statement
$stmt = $mysqli->prepare("SELECT Id FROM Users WHERE Username='$username' AND Password='$
    password'");
$stmt->execute();
```

In the safe prepared statement here, we use “?” as a placeholder. First, this line will be compiled without user input, so there is no risk of injection. Then after, the username and password are added in as pure data, and the code is executed. It’s important to note that prepared statements need to be used correctly in order to be effective. The last SQL statement in this snippet is still vulnerable. User input must be added after the compilation phase to prevent attacks.