🕐Timing Attacks in Authentication

What is a Timing Attack?

A **timing attack** exploits differences in execution time between different code paths. Attackers can measure these differences to infer sensitive information, such as whether a user account exists.

Problem in Current Implementation

Code Snippet (simplified)

```
async getUserByCredentials(identifier: string):
Promise<UserAuthenticationCredentialsModel> {
  const userCredentialsModel = await
this.userRepositoryService.findUserByCredentials(identifier);
  if (!userCredentialsModel) throw new HttpException('INVALID_CREDENTIALS',
HttpStatus.NOT_FOUND);
  return userCredentialsModel;
}
```

Timing Difference

- **User does not exist:** Query returns quickly (\~5ms), exception thrown immediately.
- **User exists:** Query returns user, builds model (\~20ms).

➡️Attackers can measure this difference and **enumerate valid users**.

Exploitation Scenario

```
const emails = ['admin@company.com','john@company.com','fake@test.com'];

for (const email of emails) {
  const start = Date.now();
  try {
    await fetch('/api/auth/signin', { method: 'POST', body: JSON.stringify({
identifier: email, password: 'wrong' }) });
  } catch (e) {}
  console.log(`${email}: ${Date.now() - start}ms`);
}
```

Example Results

- `admin@company.com`: 18ms → **real user**
- `john@company.com`: 19ms → **real user**
- `fake@test.com`: 5ms → **not a user**

Why This is Dangerous

- **Reconnaissance**: Attackers know which accounts exist.
- **Brute force optimization**: Focus on real accounts only.
- **Social engineering**: Target real users.
- **Privacy leak**: Reveals who uses your service.

The Solution: Constant-Time Responses

Ensure **both code paths take the same time**, regardless of user existence.

Fixed Implementation

```
async getUserByCredentials(identifier: string):
Promise<UserAuthenticationCredentialsModel> {
  const startTime = Date.now();

  const userCredentialsModel = await
this.userRepositoryService.findUserByCredentials(identifier);

  if (!userCredentialsModel) {
    await this.performDummyWork();
    await this.ensureMinimumDelay(startTime, 50);
    throw new HttpException('INVALID_CREDENTIALS', HttpStatus.NOT_FOUND);
  }

  await this.ensureMinimumDelay(startTime, 50);
  return userCredentialsModel;
}

private async performDummyWork(): Promise<void> {
  await bcrypt.hash('dummy', 10);
}

private async ensureMinimumDelay(startTime: number, minMs: number):
Promise<void> {
  const elapsed = Date.now() - startTime;
```

```
  if (elapsed < minMs) {
    await new Promise(resolve => setTimeout(resolve, minMs - elapsed));
  }
}
```

Result After Fix

- ✅Real user: \~50ms (consistent)
- ✅Fake user: \~50ms (consistent)
- ❌No more information leakage

Additional Recommendations

1. **Use constant-time password comparison** (`bcrypt.compare` already provides this).
2. **Standardize error messages** – always return the same error (`INVALID_CREDENTIALS`) regardless of root cause.
3. **Rate limiting** – prevent brute force by adding per-IP or per-identifier throttling.
4. **Account lockout / monitoring** – detect repeated failed attempts.
5. **Audit logging** – log failed authentication attempts for analysis.

🕐Key Takeaway

Security principle: Never let response time reveal sensitive information.\ All authentication-related operations must run in **constant time**, independent of input or outcome.