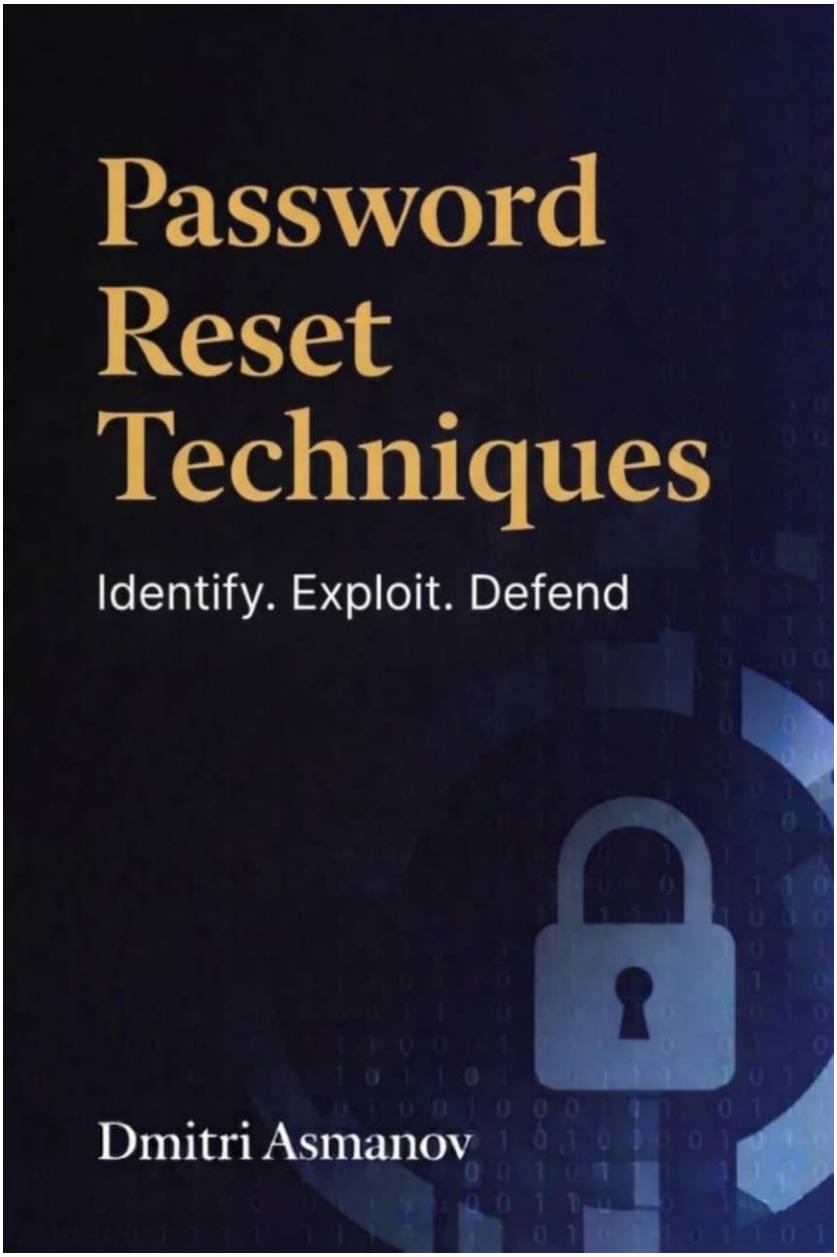
Password
Reset
Techniques
Identify. Exploit. Defend
Dmitri Asmanov

Password Reset Techniques

Identify. Eploit. Defend

Introduction: The Illusion of Sequence

When developers write code, they envision users interacting with their application in a polite, chronological order: *Step 1, Step 2, Step 3*. They write a password reset flow assuming a user will request a single token, read a single email, and submit a single password change request.

Bug bounty hunters do not interact politely.

By manipulating the asynchronous nature of web servers, attackers can bend time. We can submit fifty identical "Step 3" requests in the exact same millisecond. If the backend database does not explicitly lock the user's row during processing, the server will evaluate all fifty requests simultaneously. This is known as a **Time-of-Check to Time-of-Use (TOCTOU)** vulnerability, or a Race Condition.

In password recovery, race conditions are catastrophic.

Excerpt from Chapter 10: Racing the Database (TOCTOU Attacks)

Let's examine a standard OTP (One-Time Password) verification flow used during a password reset.

The developer implements a strict rate limit: *“If the user enters the wrong 6-digit OTP three times, lock the account.”*

The backend logic looks like this:

1. **Check:** Has the user failed 3 attempts? (If yes, block).
2. **Check:** Does the submitted OTP match the database?
3. **Action:** If wrong, increment the failed attempt counter by 1.

The Exploitation Phase

If an attacker sends 100 incorrect OTP guesses sequentially, the system blocks them after the third try. But what if the attacker sends 100 incorrect OTP guesses *in a single, concurrent burst*?

Because web servers handle requests using multi-threading, all 100 threads hit **Step 1** at the exact same millisecond.

- **Thread 1 asks:** "Has the user failed 3 attempts?" The database says: "No, 0 failures."
- **Thread 50 asks:** "Has the user failed 3 attempts?" The database says: "No, 0 failures."
- **Thread 100 asks:** "Has the user failed 3 attempts?" The database says: "No, 0 failures."

All 100 requests bypass the security check before the database has time to update the counter in **Step 3**. The attacker has effectively bypassed the rate limit entirely, allowing them to brute-force the 6-digit OTP and achieve full Account Takeover.

⚔⚔ **The Attacker's Toolkit (Offense):** Finding these microsecond windows manually is impossible. In the Tier 2 Specialist Toolkit, we use **File 95:** `otp_race_bruteforcer.py`. This engine utilizes Python's asynchronous thread pooling to synchronize the final byte of the HTTP requests, guaranteeing all payloads strike the target application's API gateway at the exact same microsecond.

The Defense: Atomic Locks

How do you defend against an attack that happens faster than the server can count? You stop relying on application-level logic and push the security down to the database layer.

To prevent race conditions, the backend must implement a **Pessimistic Row Lock**. When the first thread begins evaluating the OTP, it must lock the specific database row associated with that user.

Instead of a standard SQL `SELECT` statement, the developer must use: `SELECT otp_code, failed_attempts FROM users WHERE id = ? FOR UPDATE;`

The `FOR UPDATE` command tells the database: *"Do not let any other thread read or write to this row until I am completely finished with my transaction."* The other 99 malicious

threads are forced to wait in line. By the time Thread 2 is allowed to read the row, Thread 1 has already incremented the counter, and the logic holds firm.

🛡️ **The Defender's Toolkit (Defense):** Developers often struggle to implement database locks correctly across complex ORMs (Object-Relational Mappers). Refer to **File 37:** `atomic_toctou_defender.sql` and **File 50:** `sql_row_locker.py` in your Tier 2 archive for drop-in, framework-agnostic examples of secure, atomic transaction processing.

Ready to break the flows?

This excerpt scratches the surface of asynchronous logic flaws. In the full edition of *Password Reset Techniques*, we cover 17 chapters of advanced exploitation, including:

- Bypassing CDN caching rules to leak active reset tokens.
- OAuth state manipulation to downgrade high-assurance passkeys.
- Session purgatory: Maintaining access long after the victim thinks they've locked you out.

Stop banging your head against hardened login portals. Learn how to break the recovery logic.

[Link to purchase the Full Standard Edition or upgrade to the Tier 2 Specialist Toolkit Edition]