MANUAL PENETRATION TESTING OF

BEGINNER REPORT

Table of Contents

8. **Conclusion**

9. **References**

Introduction

Welcome to a beginner-friendly manual penetration testing walkthrough — no automated scanners, no fancy scripts, just good old-fashioned hacker intuition and a browser.

This document explores some common web vulnerabilities such as:

- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Insecure Direct Object References (IDOR)
- Session Fixation

Our playground? **Altoro Mutual**, a purposely vulnerable banking application designed by IBM for training and testing purposes. You can access the site at:
http://demo.testfire.net

⚠️ **Disclaimer**:
Altoro Mutual is a **deliberately insecure demo site** provided by IBM. **Do not enter any real credentials or sensitive information** on this site. It's meant for learning, not banking — unless you're trying to transfer funds from "Account 800004" to "800005" just for experimenting.
Also, don't create an account thinking this is a real bank — trust me, you'll be disappointed when your imaginary mortgage doesn't get approved.

Throughout this report, each finding includes:

- A **description** of the vulnerability
- A **reproduction walkthrough**
- **Screenshots** to support the process
- Suggested **mitigations** to defend against it

The purpose of this documentation is to:

- Help **beginners** understand web application security by **manually** discovering and exploiting vulnerabilities
- Serve as a **step-by-step guide** that clearly explains the "what," "where," "how," and "so what" of each issue
- Emphasize how easily misconfigurations or insecure inputs can lead to **critical breaches**

Let's peel back the layers of this vulnerable banking app — no scripts, just skills.

Prerequisite Knowledge

To fully understand the findings and steps documented in this report, the reader is expected to have a foundational understanding of the following:

- Basic Web Technologies: Understanding of how websites function, including HTTP/HTTPS protocols, cookies, forms, and client-server communication.
- HTML & JavaScript: Comfort with reading and writing simple HTML and recognizing how scripts can manipulate web behavior.
- Linux Terminal Usage (Kali Linux Preferred): Experience with basic terminal commands, file manipulation, and running local servers using tools like python3 -m http.server.
- Browser Developer Tools: Proficiency in using Inspect Element, Network tab, and Application tab to observe and manipulate web activity.
- OWASP Top 10 Awareness: Familiarity with common vulnerabilities such as SQL Injection, XSS, CSRF, IDOR, and Session Fixation.
- Security Best Practices: A mindset for ethical hacking and the ability to analyze vulnerabilities with the intent to report and educate.

Important Terminologies

Vulnerability
A weakness in a system that can be exploited to compromise its confidentiality, integrity, or availability.

Exploit
A piece of code or technique used to take advantage of a vulnerability.

Payload
The specific input or code used to trigger a vulnerability.

Authentication
The process of verifying the identity of a user, system, or entity.

Authorization
Determines what resources a user is allowed to access after authentication.

Session Token / Session ID
A unique identifier used to maintain a user's session with a web application.

HTTP Request
A communication sent from a client (usually a browser) to a server, often containing methods like GET, POST, etc.

Manual Testing
Testing performed by a human without automation tools to identify vulnerabilities by observing responses and behavior.

PoC (Proof of Concept)
A demonstration that shows how a vulnerability can be exploited.

Penetration Testing
The practice of simulating attacks on a system to discover exploitable vulnerabilities.

Scope & Objectives

Scope:
This penetration test was conducted on the Altoro Mutual demo site (https://testfire.net), a deliberately vulnerable web application maintained by IBM for educational and training purposes. This environment was selected specifically because it is legally safe to test against and provides classic examples of common web vulnerabilities.

Objectives:

- To identify and exploit common web application vulnerabilities manually, without the use of automated tools.
- To gain a deeper understanding of how these vulnerabilities work and how attackers leverage them.
- To demonstrate proof-of-concept attacks in a legal environment using minimal tools (primarily browser-based).
- To suggest basic mitigations that would prevent these vulnerabilities in real-world applications.

Tools & Environment

For the entirety of this manual penetration test, the following tools and environment were used:

- **Operating System**: Kali Linux (2023.4)
- **Web Browser**: Chromium (pre-installed on Kali)
- **Text Editor**: Mousepad (used to craft HTML payloads for CSRF)
- **Local Server**: Python 3's built-in HTTP server (`python3 -m http.server`)
- **Developer Tools**: Chromium's built-in Inspect Element & Network tab
- **Testing Platform**: Altoro Mutual Demo Site

This setup was intentionally kept lightweight and beginner-friendly to demonstrate that **manual testing does not require a heavy toolkit** — just sharp eyes, curiosity, and a decent keyboard.

Methodology

The testing process followed a basic manual pentesting methodology aimed at clarity and learning, not stealth or advanced exploitation:

1. **Reconnaissance:**
   Explored available pages and forms within Altoro Mutual to understand the app's structure and functionality.

2. **Input Field Testing:**
   Manually tested form fields for weak input validation using simple payloads.

3. **Authentication Testing:**
   Assessed login mechanisms for SQL injection and session-related vulnerabilities.

4. **Request/Response Observation:**
   Used browser developer tools (Inspect > Network tab) to monitor HTTP requests, parameters, and session handling behavior.

5. **HTML Injection & CSRF Crafting:**
   Created and executed custom `.html` payloads for testing CSRF and verifying session-dependent actions.

6. **No Automation:**
   All testing was conducted without automated scanners (e.g., Burp Suite, OWASP ZAP, SQLMap) to emphasize manual techniques.

Let's get hacking — *ethically!*

Vulnerabilities Discovered

SQL Injection

What is SQL Injection?

SQL Injection is a code injection technique that exploits vulnerabilities in a web application's database layer. It occurs when an attacker is able to insert or manipulate SQL queries through user input. This vulnerability allows attackers to view, modify, delete, or even execute arbitrary commands on the database, potentially compromising sensitive data and application functionality.

SQL Injection can lead to:

- Unauthorized access to sensitive data (usernames, passwords, personal information)
- Data manipulation or deletion
- Bypassing authentication
- Gaining administrative access to the database or the application

Where Can SQL Injection Be Found?

SQL Injection vulnerabilities are often found in web applications that improperly validate or sanitize user input, especially when the input is used in SQL queries. Common locations include:

- Login forms (username and password inputs)
- Search fields (where the query string is passed to a database)
- URL parameters used to filter or sort data
- Any place where user input is directly used in database queries

Proof of Concept (PoC)

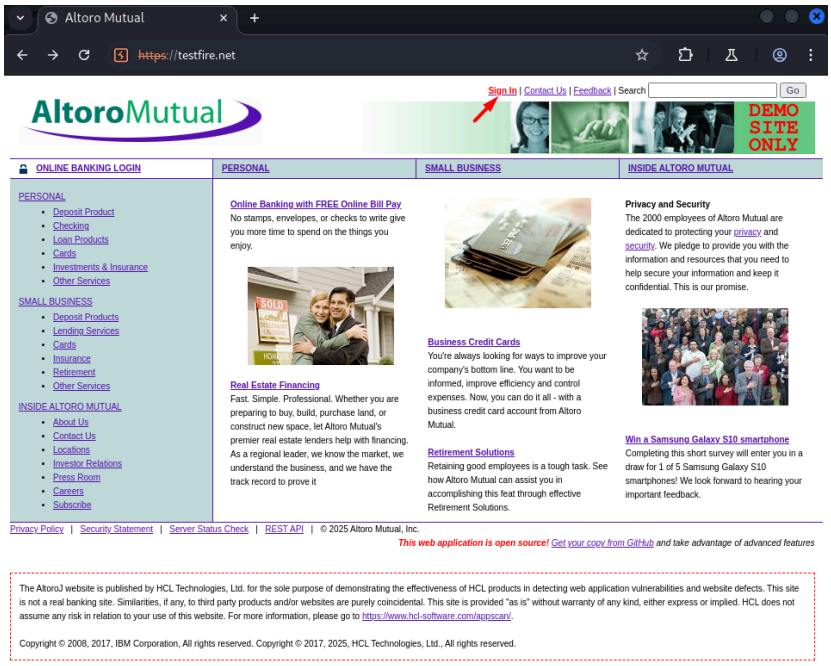
Target Page

URL: https://testfire.net/login.jsp
This page allows users to log in by entering a username and password.

Step-by-Step Manual Test Using SQL Injection:

Step 1: Navigate to the Login Page
Go to:
https://testfire.net/login.jsp

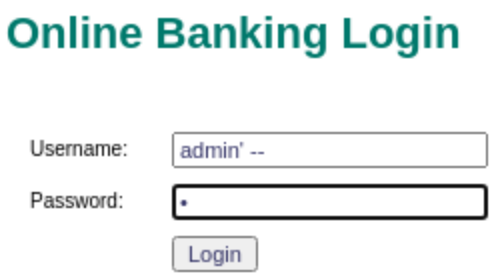
And Click “Sign In”



Step 2: Input SQL Injection Payload

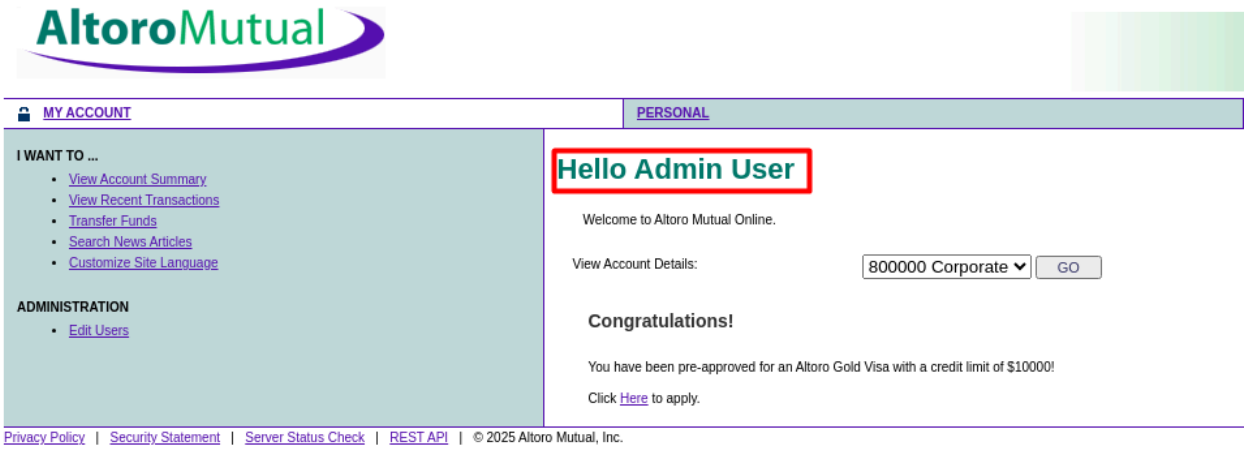
- Username:
  admin' --

- Password:
  a



Step 3: Submit the Form
Click the **Login** button.

Step 4: Observe the Result
If the login is successful and you are redirected to the admin dashboard or logged-in page, this confirms an **SQL Injection vulnerability**.



What Happened?

The payload admin' -- is designed to terminate the SQL query early and comment out the rest of it. Here's what the query likely becomes:

SELECT * FROM users WHERE username = 'admin' --' AND password = 'a'

The -- comment operator tells the SQL engine to ignore the rest of the line, including the password check. However, **in Altoro Mutual**, even though the injection is syntactically valid, the application still performs

an additional **password validation check**. So instead of granting access, it returns an alert or error like "Please enter a valid password."

This shows that while the app is **vulnerable to SQL injection**, it also includes basic client/server-side checks that prevent a successful login with this specific payload.

Mitigations

To prevent SQL Injection:

- Use **prepared statements** or **parameterized queries**.
- Never directly concatenate user input into SQL queries.
- Employ input validation and escaping.
- Use ORM (Object-Relational Mapping) tools that handle SQL safely.
- Regularly scan and test your application for injection vulnerabilities.

*Note: While the SQL Injection payload used here (`admin'--`) works easily on the Altoro Mutual demo site, real-world applications are rarely this vulnerable or obvious. Modern systems typically have input validation, parameterized queries, and other defenses in place. Attempting this on real systems without authorization is illegal and unethical — and definitely **not worth trying.** You might even get laughed at.*

Cross-Site Scripting (XSS)

What is Cross-Site Scripting?

Cross-Site Scripting is a web security vulnerability that allows an attacker to inject malicious scripts into web pages viewed by other users. When a website does not properly validate or encode user input, it may include that input directly in the HTML output. This can lead to the execution of arbitrary JavaScript in a victim's browser.

XSS can be used to:

- Steal cookies or session tokens
- Deface websites
- Redirect users to malicious sites
- Log keystrokes
- Impersonate users or hijack sessions

Where Can XSS Be Found?

XSS vulnerabilities are typically found in areas where user input is reflected or stored by the web application. Common places include:

- Search bars
- Comment sections
- Contact or feedback forms
- Profile update fields
- URL parameters
- Error messages
- Chat applications

There are three main types of XSS:

- **Reflected XSS**: The malicious script is reflected off the web server and executed immediately (often through URL parameters or form inputs).
- **Stored XSS**: The script is stored on the server (e.g., in a database or file) and served to users who later view that content.
- **DOM-based XSS**: The vulnerability exists in client-side JavaScript that handles user input insecurely within the browser's Document Object Model (DOM).

In this document, we will demonstrate **reflected XSS** using the search feature of Altoro Mutual.

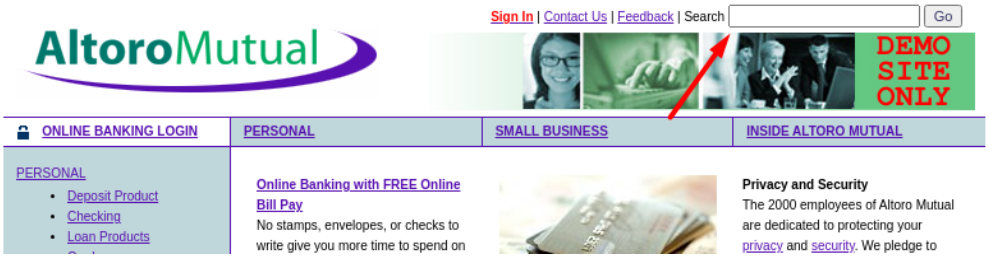
Proof of Concept (PoC)

Target Page

URL: http://demo.testfire.net/search.jsp
This page allows users to search for content using a search input box.

Step-by-Step Manual Testing using XSS:

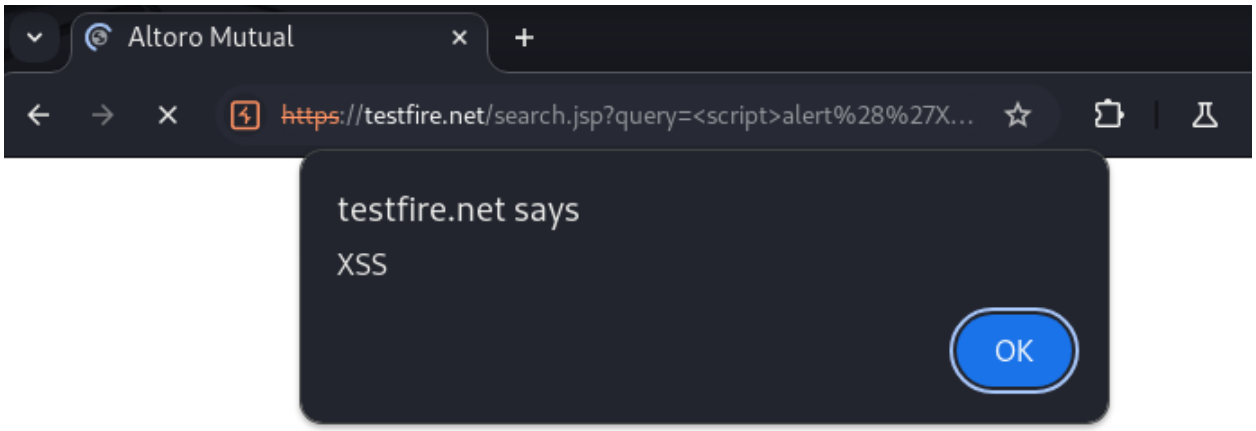
Step 1: Navigate to the Search Field



Step 2: Inject XSS Payload in the Search Field
Enter this payload into the search input box:
<script>alert('XSS')</script

And then click ‘Go’

Step 3: Observe the Alert
An alert box appears displaying the text XSS.



What Happened

The payload you entered was reflected directly in the search results without any sanitization or encoding. Since the input was interpreted as JavaScript, it executed in the browser — confirming a reflected XSS vulnerability.

Mitigations

Escape or encode user inputs before rendering them into the HTML output. Use frameworks or libraries that provide automatic sanitization, and consider implementing a Content Security Policy (CSP) to prevent execution of inline scripts.

CSRF (Cross-Site Request Forgery)

What is Cross-Site Request Forgery?

Cross-Site Request Forgery (CSRF) is a type of vulnerability where an attacker tricks a logged-in user into unknowingly submitting a request to a web application in which they are authenticated. Because browsers automatically include cookies with each request, the application processes the request as if it was a legitimate user action — even though it was initiated by an attacker. This can lead to unauthorized actions such as changing passwords, updating profile information, or initiating financial transactions.

Where This Vulnerability Can Be Found

CSRF vulnerabilities are commonly found in features that allow users to perform state-changing operations, including:

- Password change forms
- Account profile updates
- Email or address change forms
- Financial transactions

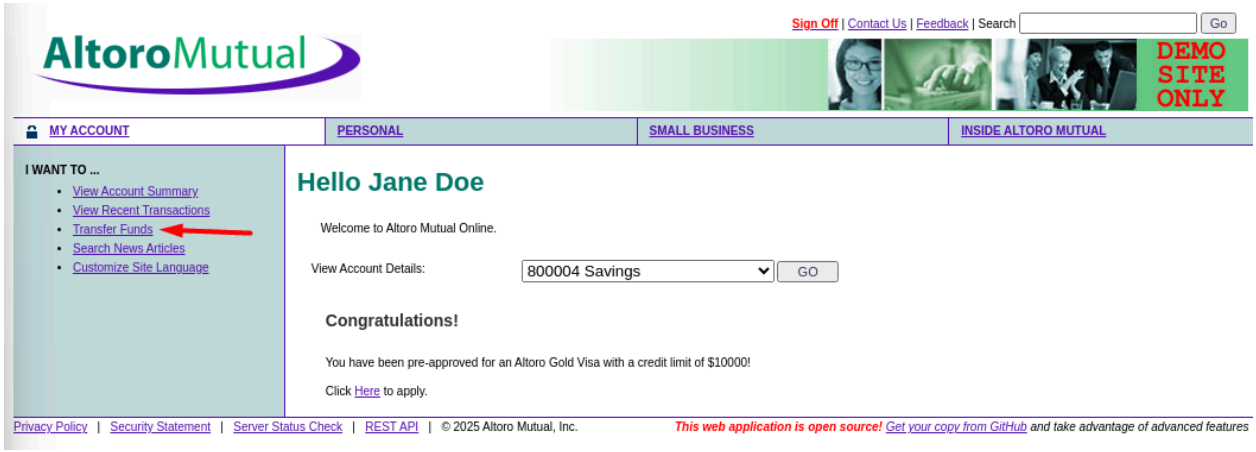
Proof of Concept (PoC)

Step-by-Step Manual testing using CSRF:

Step 1: Log in to the Application

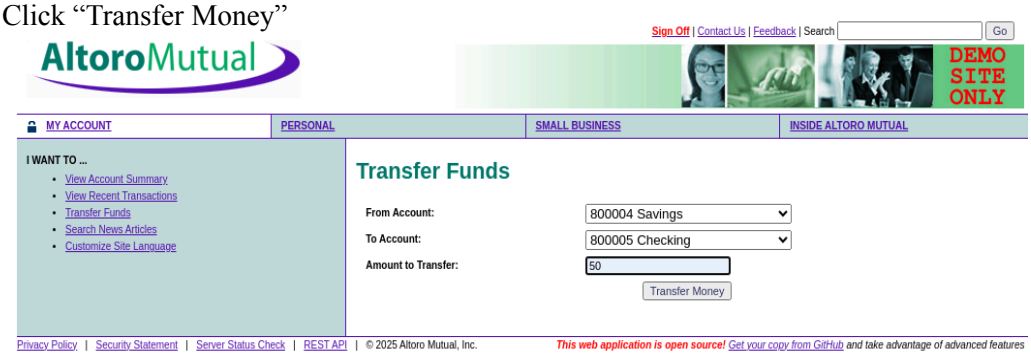
- Go to: https://testfire.net/login.jsp
- Username: `jdoe`
- Password: `demo1234`

After logging in, click on "Transfer Funds"

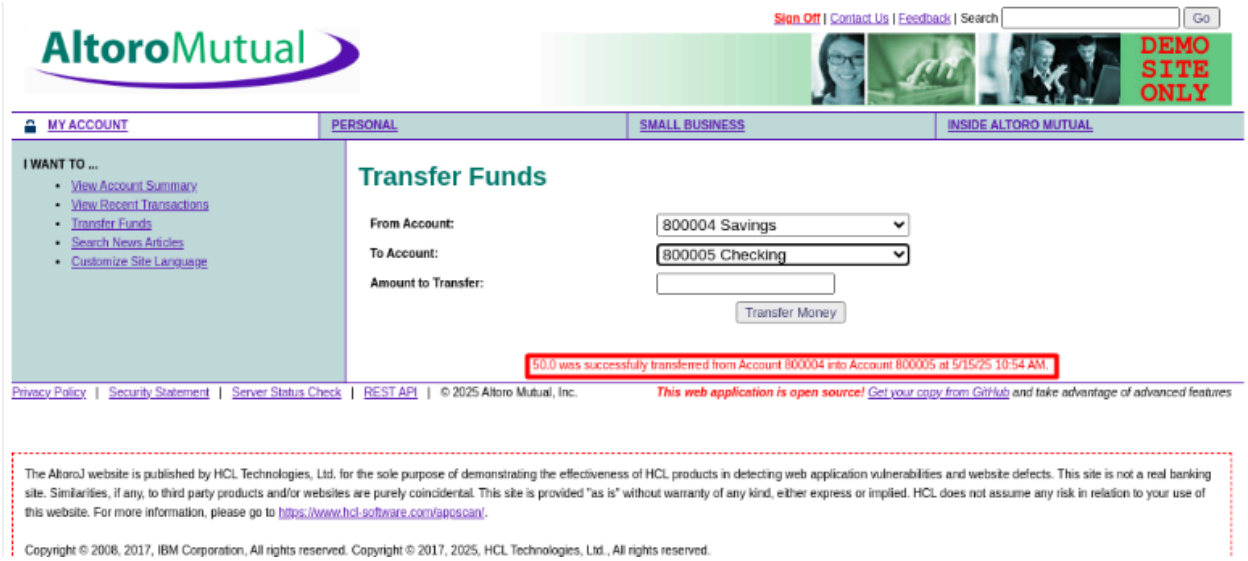


Step 2: Perform a Legitimate Transfer

- From Account: 800004
- To Account: 800005
- Amount: 50
- Click “Transfer Money”



After you click “Transfer Money”, you should see a message that appears when the transfer is successful



Step 3: Capture the POST Request

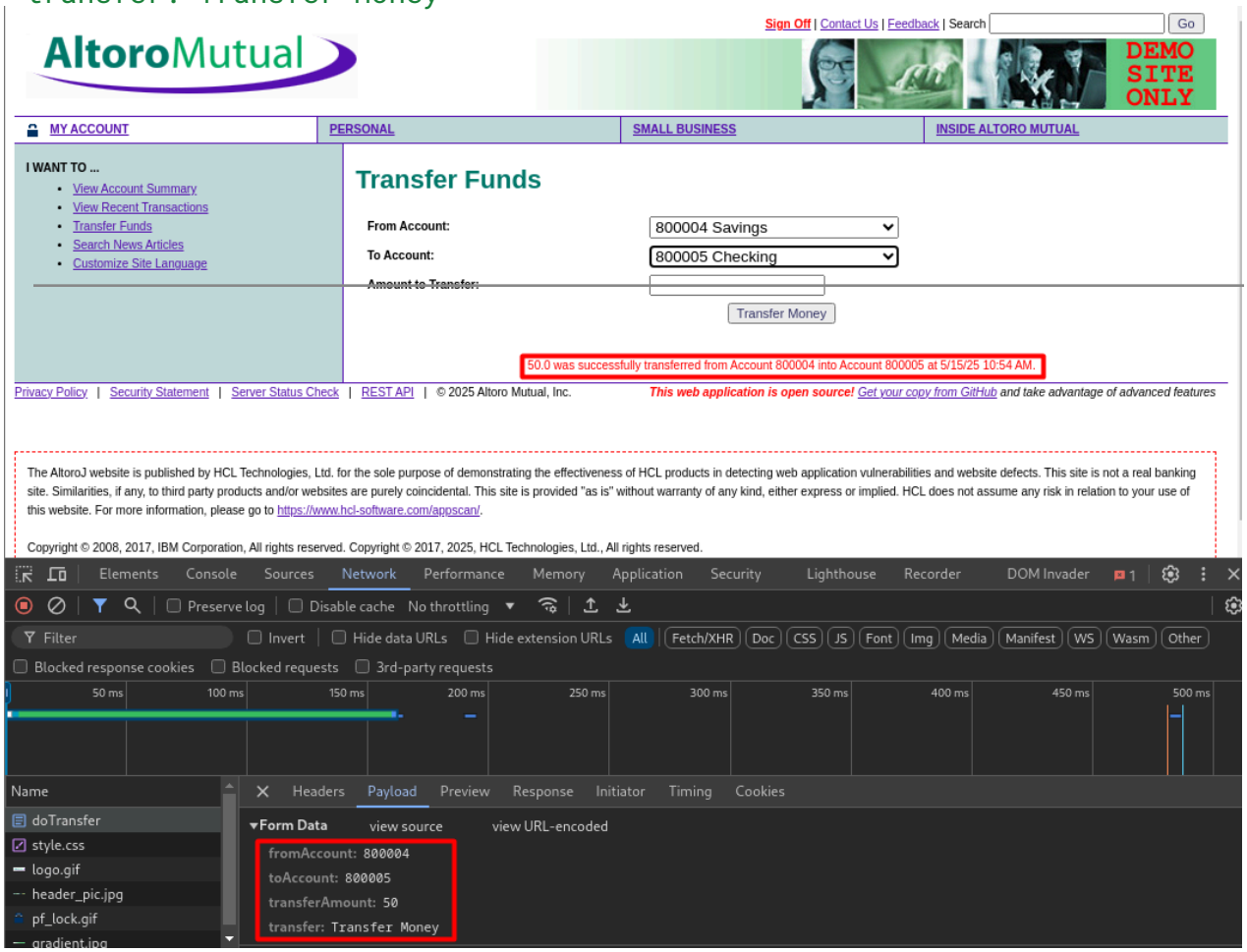
- Right-click anywhere on the page and choose **Inspect**.
- Go to the **Network** tab.
- Perform the transfer again.
- Click on the `doTransfer` request and view the **Payload**.
  You should see:

```
fromAccount: 800004

toAccount: 800005

transferAmount: 1000

transfer: Transfer Money
```



Step 4: Create the Malicious HTML Page

Open a text editor like Mousepad (Linux) and paste the following:

```
<html>
  <body onload="document.forms[0].submit()">
    <form action="https://testfire.net/bank/doTransfer" method="POST">
      <input type="hidden" name="fromAccount" value="800004">
      <input type="hidden" name="toAccount" value="800005">
      <input type="hidden" name="transferAmount" value="1000">
      <input type="hidden" name="transfer" value="Transfer Money">
    </form>
  </body>
</html>
```

Save this file as: `csrf_poc.html`

Step 5: Open the PoC File

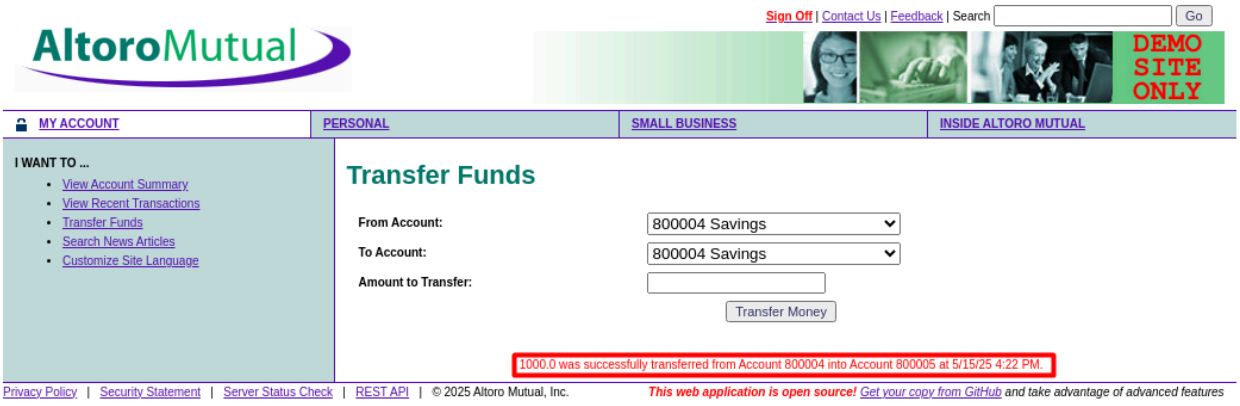
- Open a terminal in Kali Linux.
- Navigate to the folder containing your `csrf_poc.html`.
- Start a local server using:

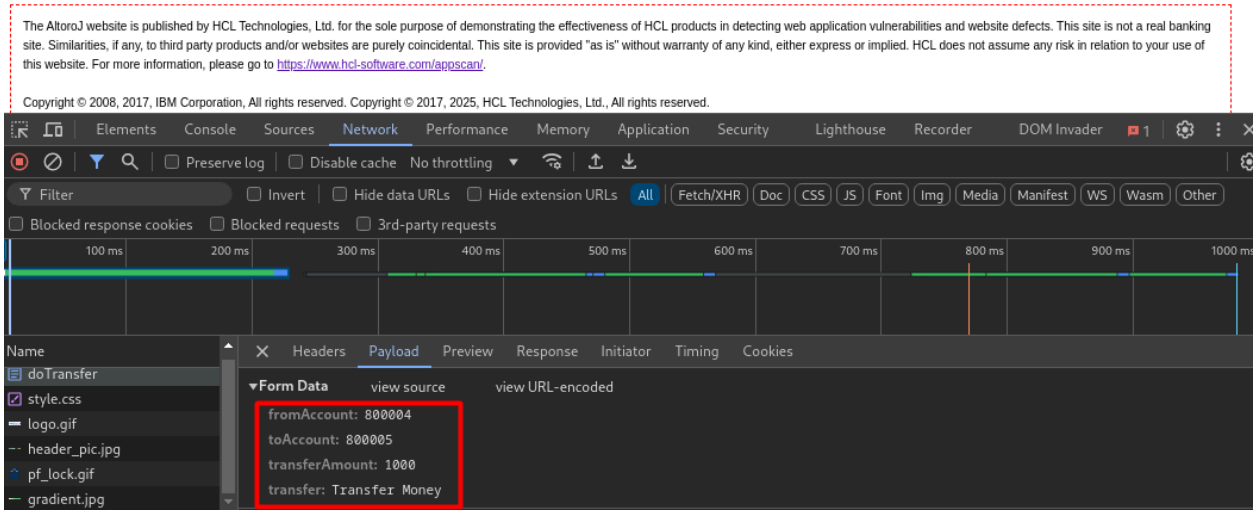
```
python3 -m http.server 8000
```

- Drag and drop the file directly into the browser URL bar.

Step 6: Observe the Behavior

If successful, the transfer will occur without any interaction from the logged-in user, you will see the success message appear on the screen.





What Happened?

By tricking an authenticated user into executing a crafted HTML page (without their knowledge), the attacker was able to initiate an unauthorized funds transfer from the victim's account. Since the application did not validate the origin of the request (e.g., via CSRF tokens or referer checks), it blindly trusted the request as if it came directly from the user.

This demonstrates a classic Cross-Site Request Forgery vulnerability, where user actions are hijacked through forged requests — all while they remain logged in and unsuspecting.

Mitigations

- Implement anti-CSRF tokens in all state-changing forms.
- Enforce SameSite cookie attributes.
- Validate HTTP Referer and Origin headers.
- Require re-authentication for critical operations.

IDOR(Insecure Direct Object References)

What is Insecure Direct Object References?

Insecure Direct Object References occur when an attacker can access or modify objects (files, records, etc.) by manipulating input parameters that reference these objects. This vulnerability is often a result of poor access control mechanisms and can allow attackers to view, modify, or delete data that should be inaccessible to them.

Where Can IDOR Be Found?

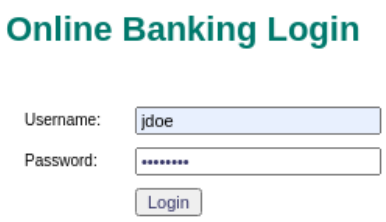
IDOR vulnerabilities can be found in web applications that use user-controlled input to directly reference sensitive objects. Common examples include:

- Access control lists that are not properly enforced
- URL parameters that reference object identifiers (e.g., /profile?id=123)
- File paths or resource names exposed in URLs

Proof of Concept (PoC)

Step-by-Step Manual Testing for IDOR:

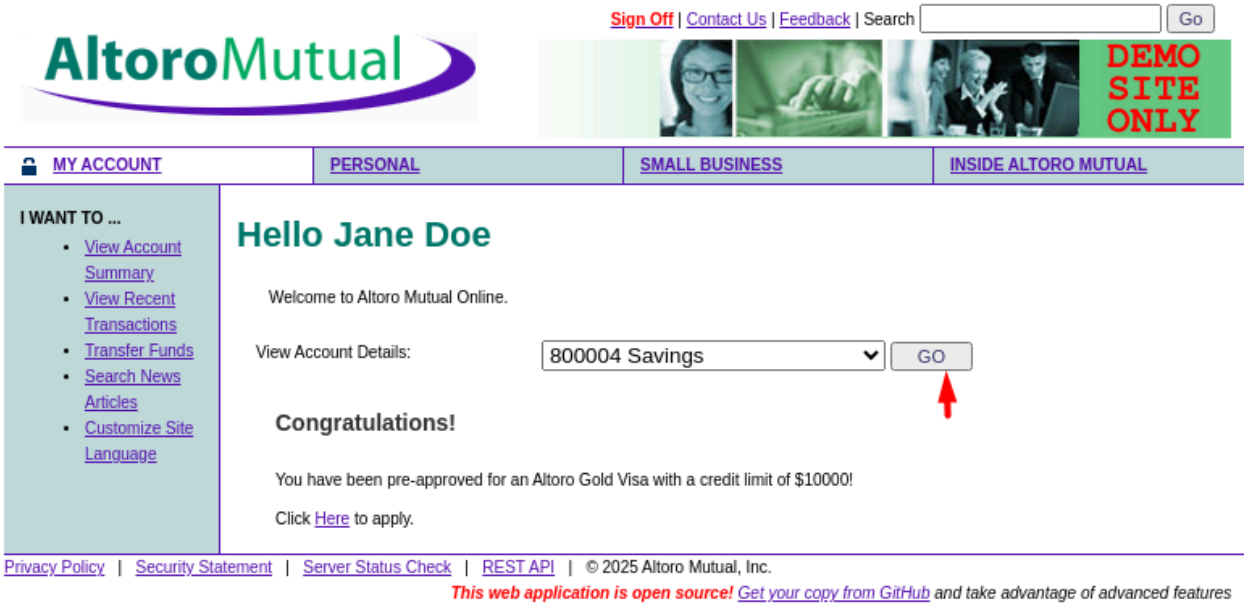
Step 1: Sign In to the dummy account
Go to: https://testfire.net/login.jsp



Username: jdoe

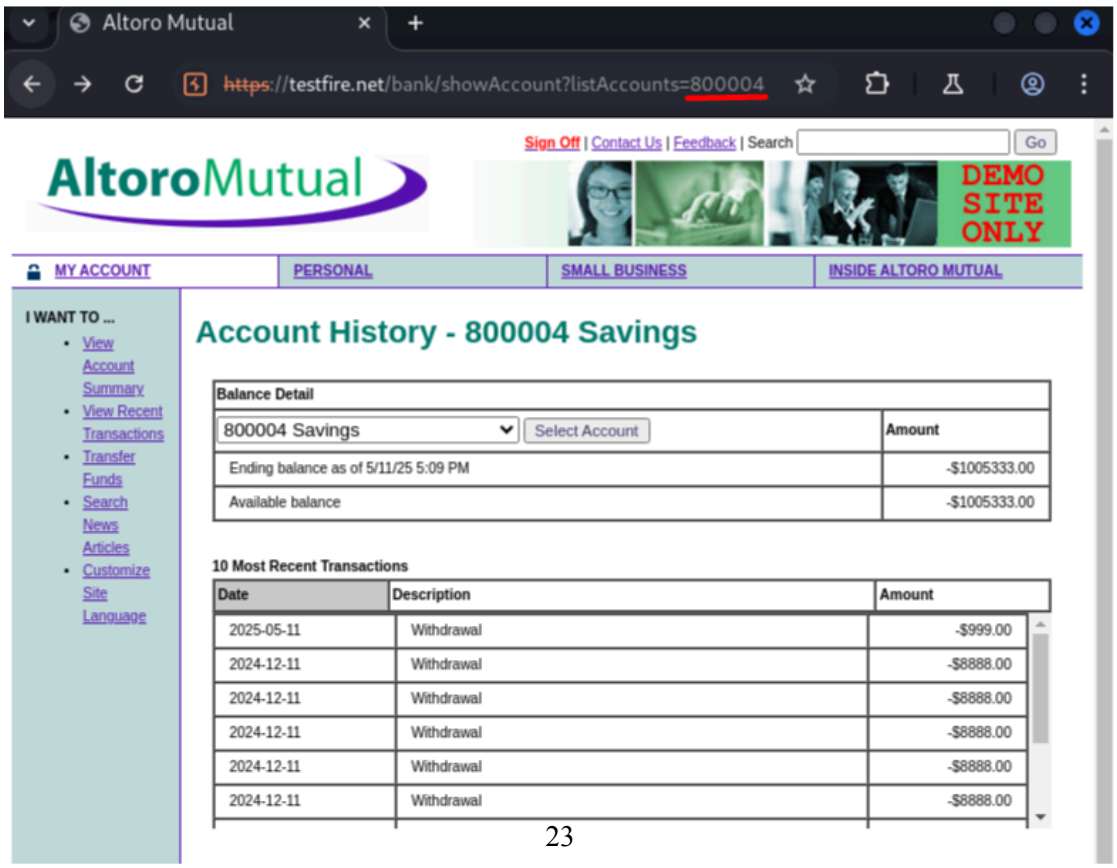
Password: demo1234

Step 2: Once Logged into the account, click on the ‘Go’



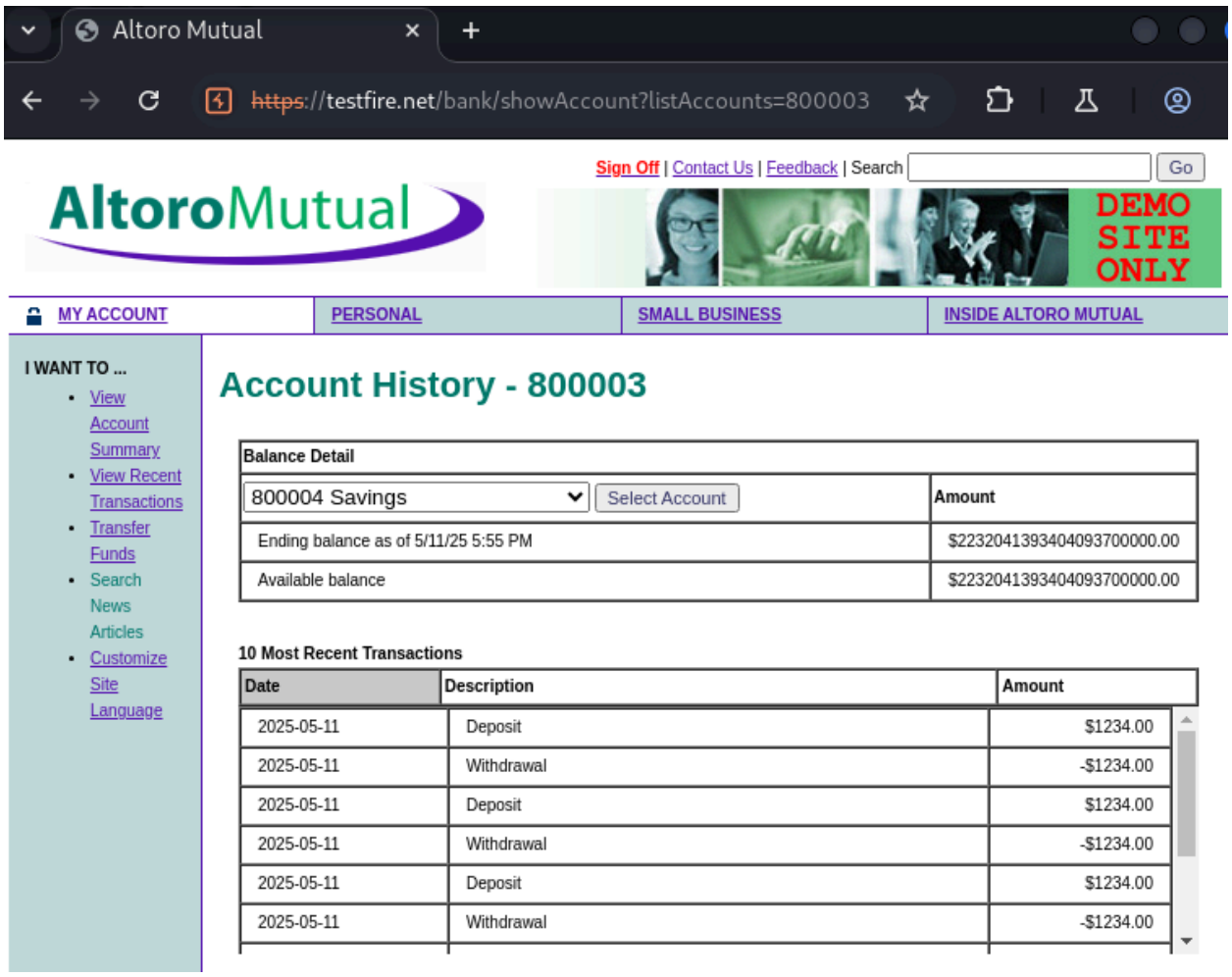
Step 3: Modify the URL Parameter

Change the listAccounts=800004 to listAccounts=800003 in the URL:
https://testfire.net/bank/showAccount?listAccounts=800004



Step 4: Observe the Result

If the account information for a different user is displayed, this confirms an **IDOR vulnerability**.



Note:

You can increment or decrement the value of the listAccounts parameter in the URL

(https://testfire.net/bank/showAccount?listAccounts=800004)

to view other users' account information.

For example, changing 800004 to 800003 or 800005 may expose different account details — indicating a lack of proper access control.

What Happened?

By manipulating the *listAccounts* parameter in the URL, you were able to access another user's account details, demonstrating a lack of proper access control.

Mitigations

To prevent IDOR:

- Ensure proper access control mechanisms are in place for every resource.
- Use indirect object references (e.g., session tokens or randomized IDs) instead of predictable identifiers.
- Validate and verify that users can only access their own resources.

Session Fixation

What is Session Fixation?

Session Fixation occurs when an attacker sets a user's session ID before they authenticate. By fixing a session ID, the attacker can later use that same ID to impersonate the victim. This vulnerability exploits poorly managed session handling mechanisms and can allow attackers to hijack sessions, leading to unauthorized access.

Where Can Session Fixation Be Found?

Session Fixation vulnerabilities are commonly found in web applications that do not properly manage session IDs after authentication. Common locations include:

- Session IDs stored in URLs or cookies that can be controlled by attackers
- Web applications that do not regenerate session IDs upon login
- Applications that allow session ID manipulation via URL parameters or cookies

Proof of Concept (PoC)

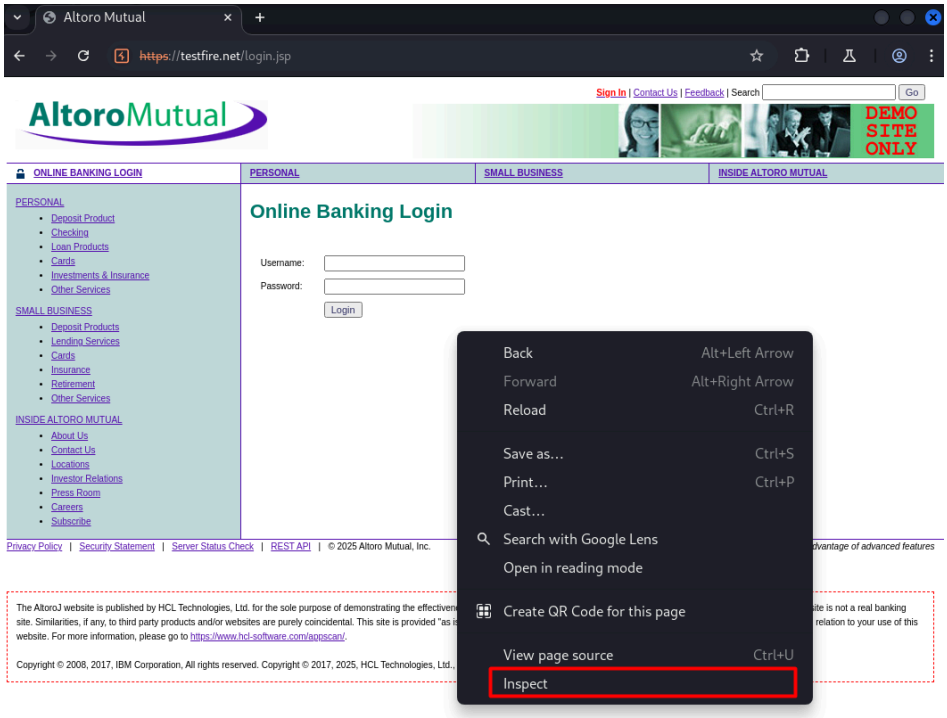
Target Page

URL: https://testfire.net/login.jsp

Step-by-Step: Testing for Session Fixation

Step 1: Inspect the Session ID Before Login

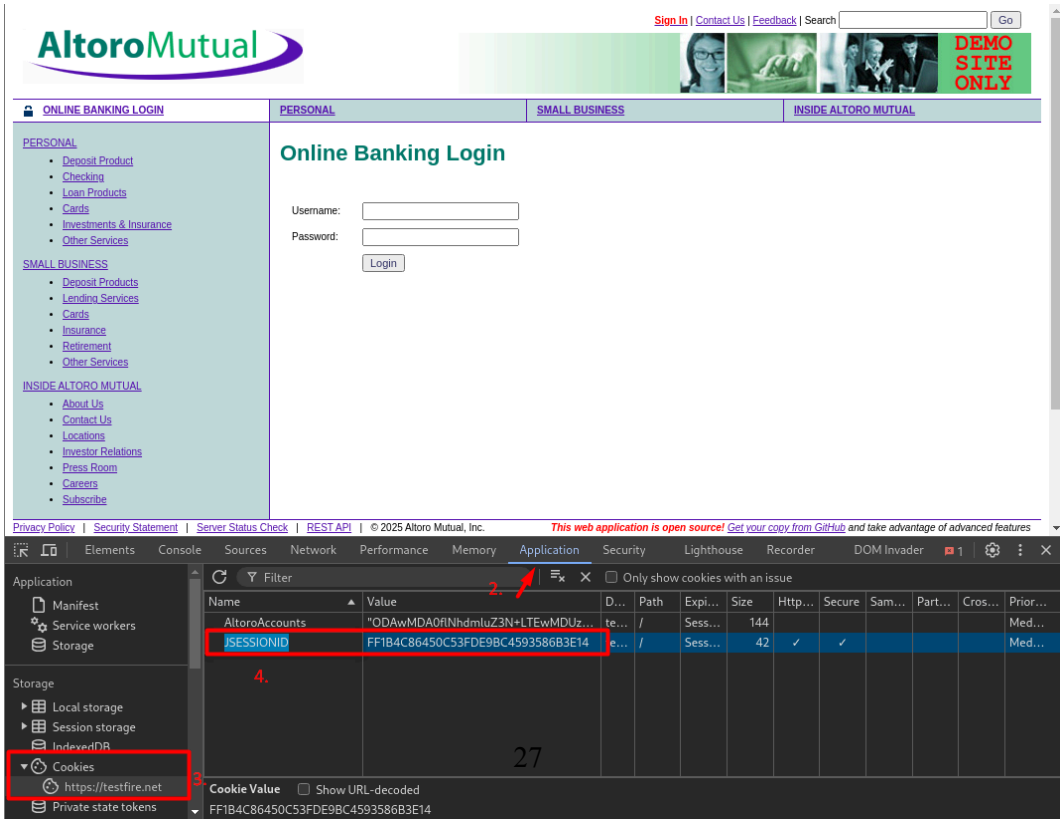
1. Right-click the page and select **Inspect**



2. Navigate to the **Application** tab

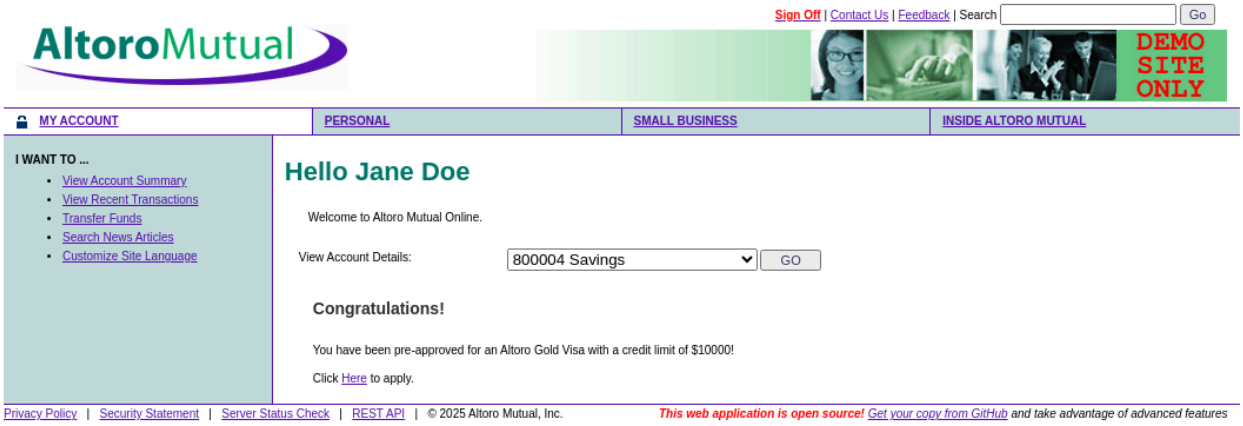
3. On the left panel, expand **Cookies** → click https://testfire.net

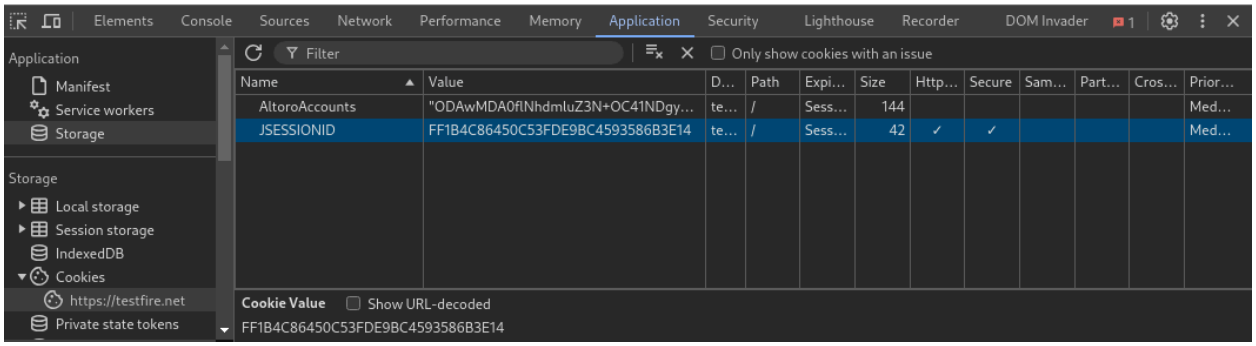
4. Note the session cookie (e.g., JSESSIONID) and its value



Step 2: Log In with Valid Credentials
Use the following test credentials:
Username: jdoe
Password: demo1234

After logging in, go back to the **Application > Cookies** panel and check if the session ID value has changed.





Step 3: Observe the Result

The session ID (FF1B4C86450C53FDE9BC4593586B3E14) remains the same after logging in, confirming a Session Fixation vulnerability. This means an attacker could pre-set a session ID and then trick the victim into logging in with that session, allowing the attacker to hijack the session afterward.

What Happened?

By observing the session ID before and after authentication, we can determine whether the session is being reinitialized. If it isn't, an attacker could pre-set the session ID and hijack the account once the victim logs in.

Mitigations

To prevent Session Fixation:

- Regenerate session IDs upon login to prevent attackers from fixing them beforehand.
- Ensure that session IDs are stored securely in cookies with the HttpOnly and Secure flags.
- Implement proper session timeout and termination upon logout.

Conclusion

Hopefully, you had fun walking through this hands-on manual penetration test of the Altoro Mutual demo site. We explored several key web vulnerabilities—SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Insecure Direct Object References (IDOR), and Session Fixation—without relying on automated tools like Burp Suite or SQLMap.

This was done entirely using a Kali Linux environment and a Chromium browser, to emphasize the importance of understanding the *how* and *why* behind each attack. Automation is powerful, but comprehension is critical.

But let's be clear—this by itself doesn't make anyone a hacker. If you're tempted to declare yourself an expert now… relax. The road to becoming proficient in penetration testing is long, technical, and full of edge cases that don't show up in demo environments. This project was just one small (but exciting) step toward that goal.

There's much more to come. In future walkthroughs, we'll dive into Burp Suite, SQLMap, and more advanced tools and testing strategies. We'll also target other legal platforms designed for training and ethical hacking.

Stay curious, stay ethical—and always hack with permission.

References

OWASP Foundation. (n.d.). OWASP Top Ten. https://owasp.org/www-project-top-ten/

Altoro Mutual Demo Site. (n.d.). https://testfire.net/

PortSwigger. (n.d.). Web Security Academy. https://portswigger.net/web-security

Kali Linux Documentation. (n.d.). https://www.kali.org/docs/

Oracle VirtualBox – https://www.virtualbox.org/manual/UserManual.html

Python HTTP Server Documentation – https://docs.python.org/3/library/http.server.html

Manual Testing Methodologies – https://portswigger.net/web-security