VULNERABILITY ASSESSMENT AND PENETRATION TESTING REPORT

Target site: https://demo.tesrfire.net

VULNERABILITY #1:

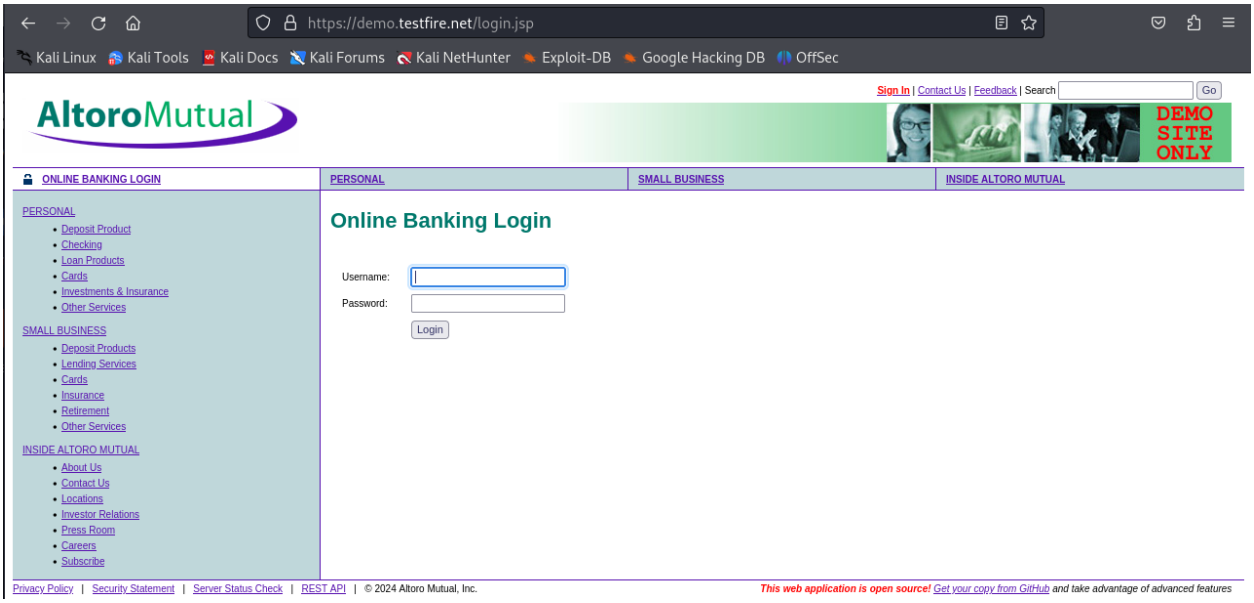
Vulnerability Name: SQL injection (SQLi)

Vulnerability Description: SQL injection is a malicious technique where an attacker exploits vulnerabilities in a web application's input validation mechanisms to inject malicious SQL code into the backend database queries. This unauthorized SQL code, often inserted through user input fields, can manipulate the structure or content of the database, leading to unauthorized access, data disclosure, modification, or deletion. By carefully crafting input that alters the intended SQL queries, attackers can bypass authentication, retrieve sensitive information, or even gain control over the entire database.

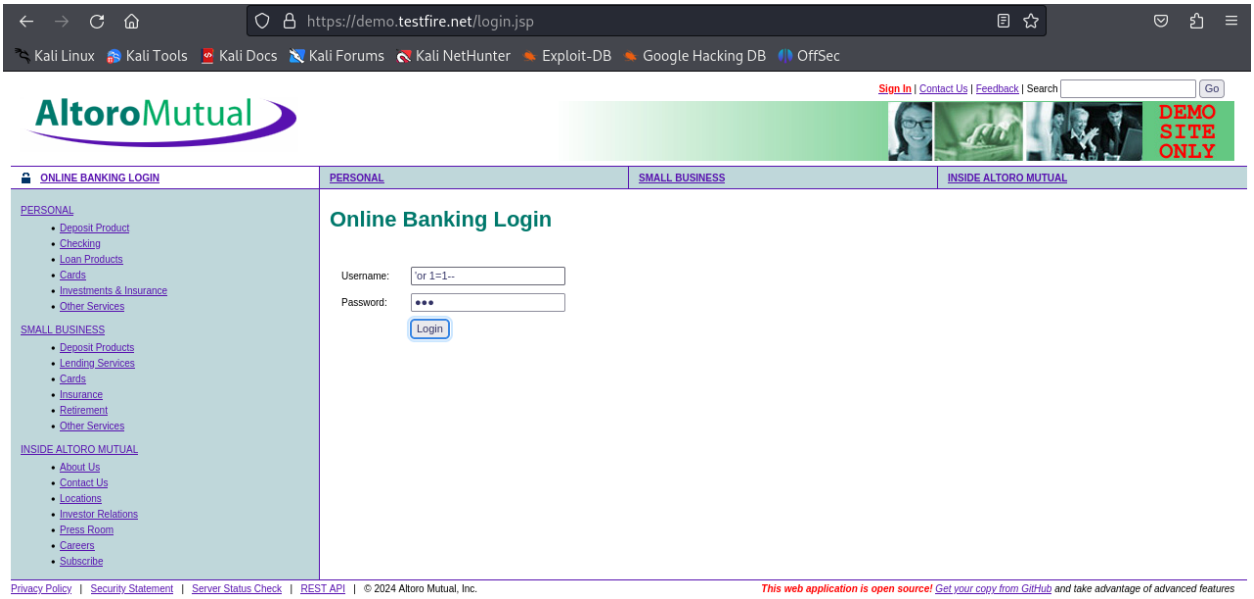
Affected URL: https://demo.testfire.net/login.jsp

Proof of Concept/Steps to reproduce:

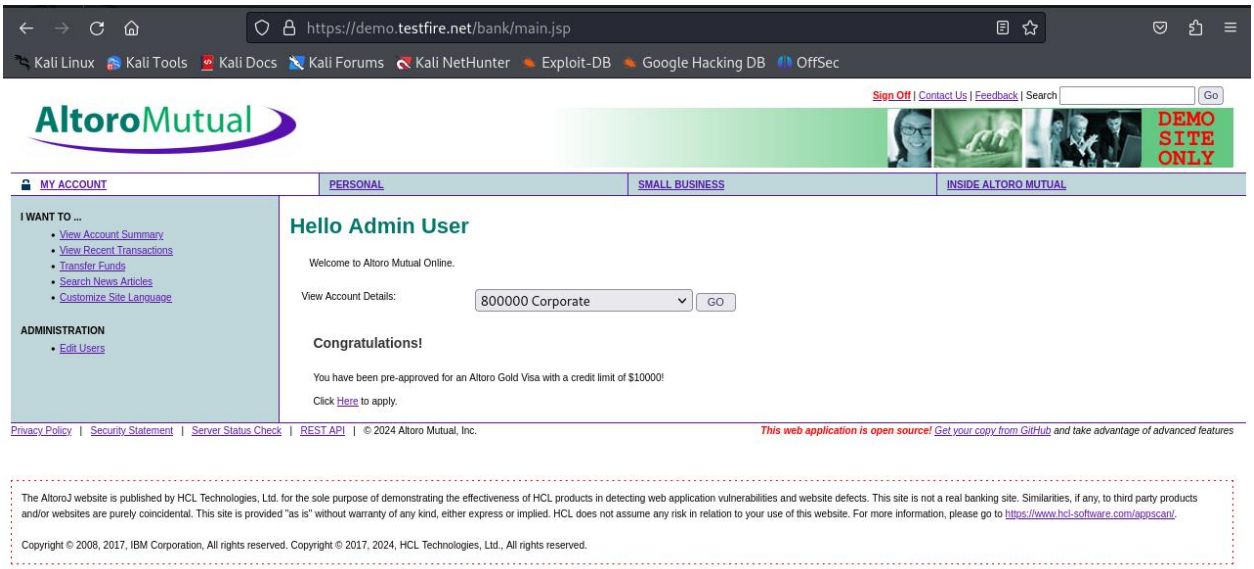
STEP 1: Open the login page of the website by clicking the 'Sign In' button.



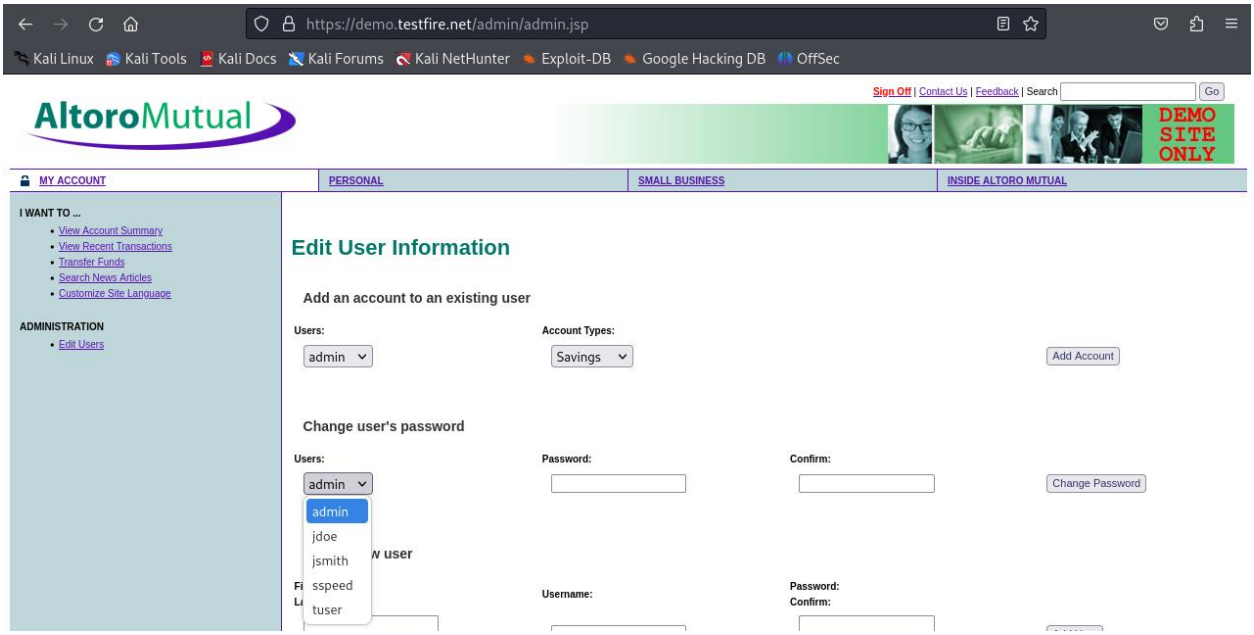
STEP 2: Without prior knowledge of usernames or passwords, a SQL injection attempt was made using the following syntax in the username field: ***'or 1=1--***. Concurrently, random text was input into the password field.



STEP 3: On clicking the login button, unauthorized access was granted, allowing entry as the Admin user. This critical SQL injection vulnerability poses a serious threat, enabling unauthorized access through manipulation of the authentication process.



STEP 4: Upon gaining access as an admin user, promptly identified all other usernames, including *jdoe*, *jsmith*, *sspeed*, *tuser* and privileges associated with administrative roles.



Solution/Prevention:

To prevent the SQL injection vulnerability, consider implementing the following solutions:

- **Use Parameterized Statements or Prepared Statements:** Utilize parameterized queries or prepared statements in your application code. This ensures that user inputs are treated as data and not executable SQL code, preventing injection attacks.
- **Input Validation and Sanitization:** Implement robust input validation and sanitization practices. Validate and sanitize user inputs to ensure they adhere to expected formats, minimizing the risk of malicious input.
- **Least Privilege Principle:** Implement the principle of least privilege by assigning minimal necessary permissions to database accounts used by the application. This restricts the potential impact of a successful SQL injection attack.
- **Web Application Firewall (WAF):** Deploy a Web Application Firewall to monitor and filter HTTP traffic between a web application and the Internet. WAFs can detect and prevent SQL injection attacks by filtering malicious requests.
- **Regular Security Audits and Code Reviews:** Conduct regular security audits and code reviews to identify and rectify potential vulnerabilities. Automated tools and manual reviews can help ensure that the codebase is secure against SQL injection and other common threats.
- **Security Education and Training:** Educate developers, administrators, and other relevant personnel about secure coding practices and the risks associated with SQL injection. Knowledgeable teams are better equipped to prevent and address vulnerabilities.

Tools Used: NA

VULNERABILITY #2:

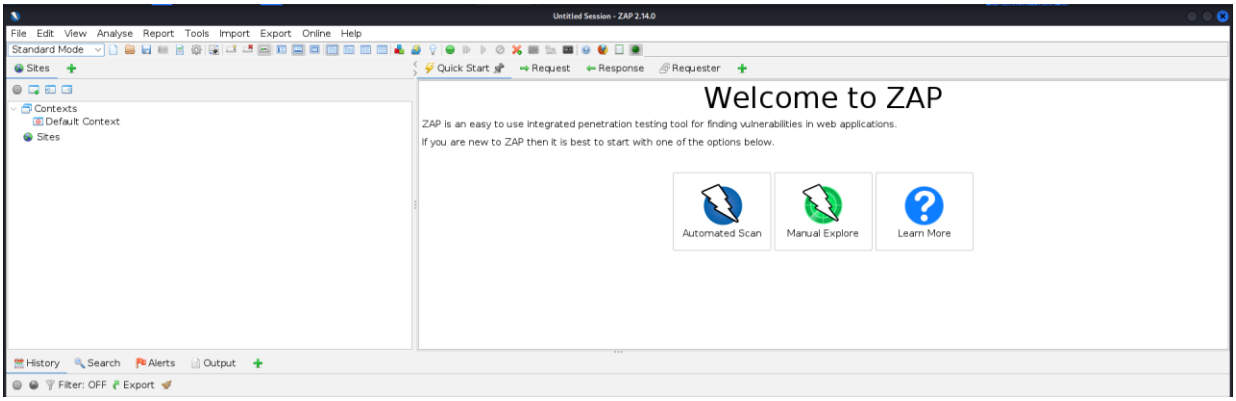
Vulnerability Name: Weak authentication

Vulnerability Description: A weak authentication vulnerability refers to a security weakness in a system or application arising from the use of easily guessable or insecure passwords by users. This vulnerability occurs when passwords lack complexity, are too short, or follow predictable patterns, making them susceptible to brute-force attacks, dictionary attacks, or other password-cracking methods. Common indicators of a weak password vulnerability include passwords that are easily guessable, commonly used words, or easily discoverable information such as names, birthdates, or simple combinations of characters. Exploiting weak password vulnerabilities can lead to unauthorized access, data breaches, and compromised security.

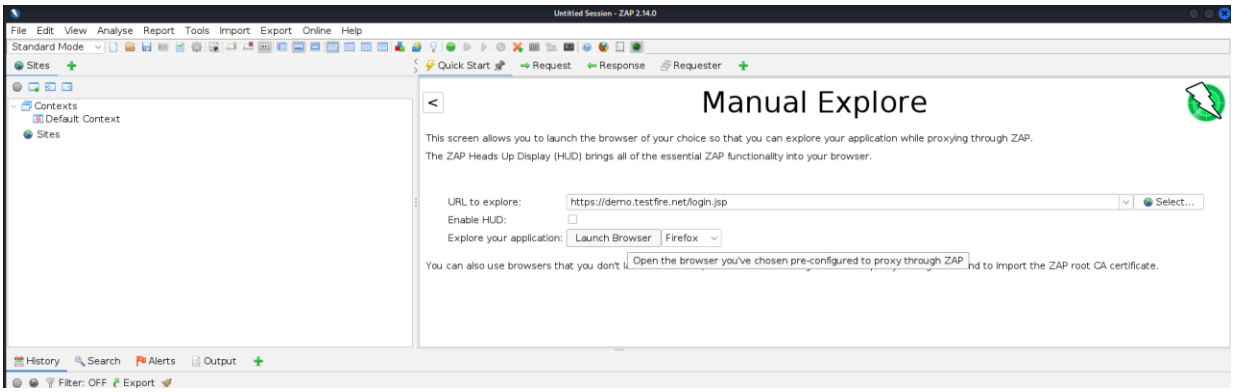
Affected URL: https://demo.testfire.net/login.jsp

Proof of Concept/Steps to reproduce:

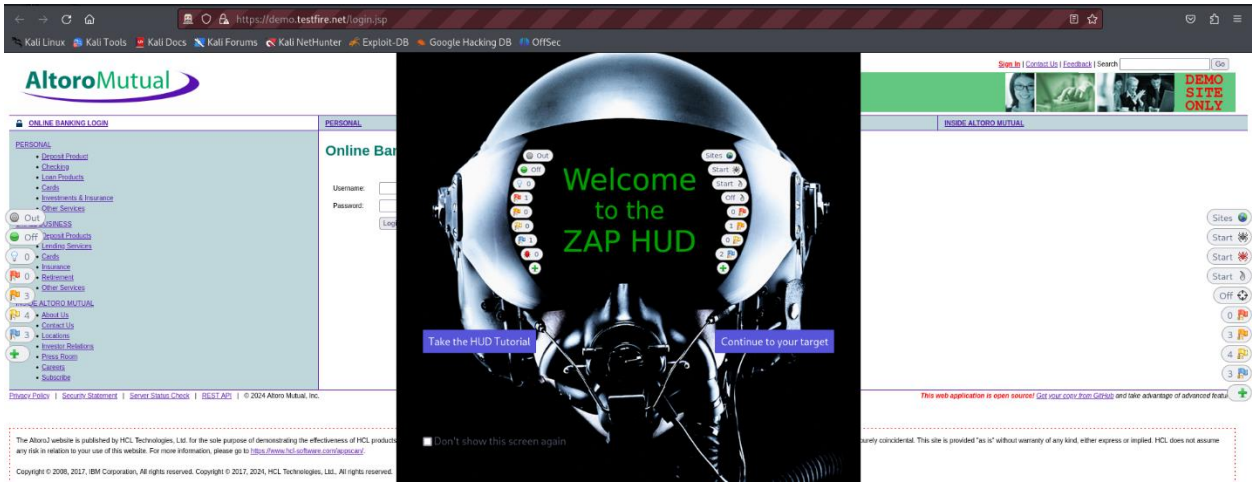
STEP 1: In order to Brute-force the password for the user **Admin**, OWASP ZAP tool is being used. Open the ZAP application.



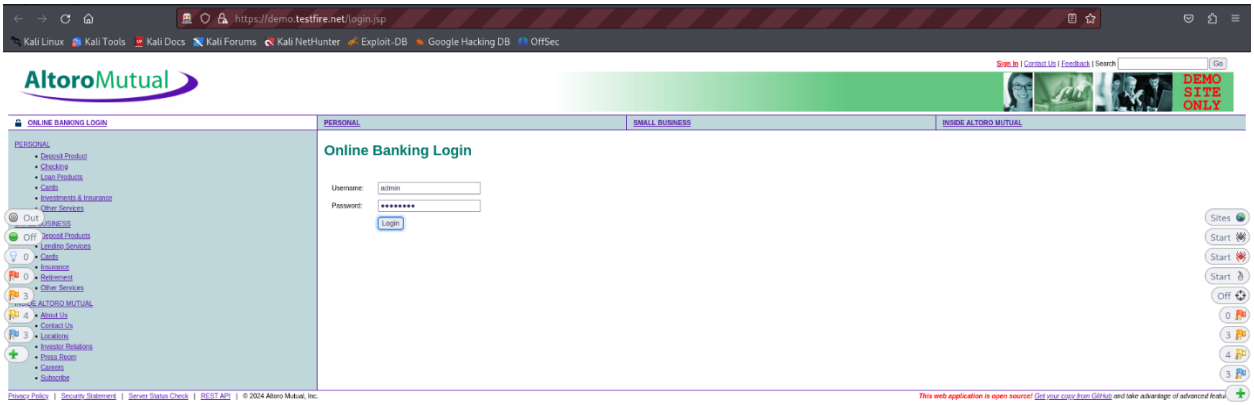
STEP 2: Click on 'Manual Explore' and enter the target URL in the Input field. Then click on 'Launch Browser'.

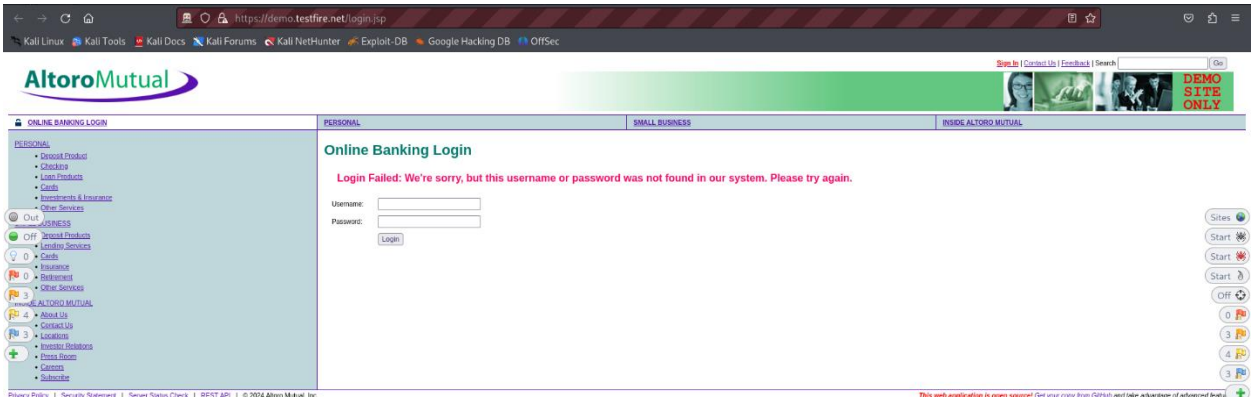


A browser session will be started with ZAP HUD

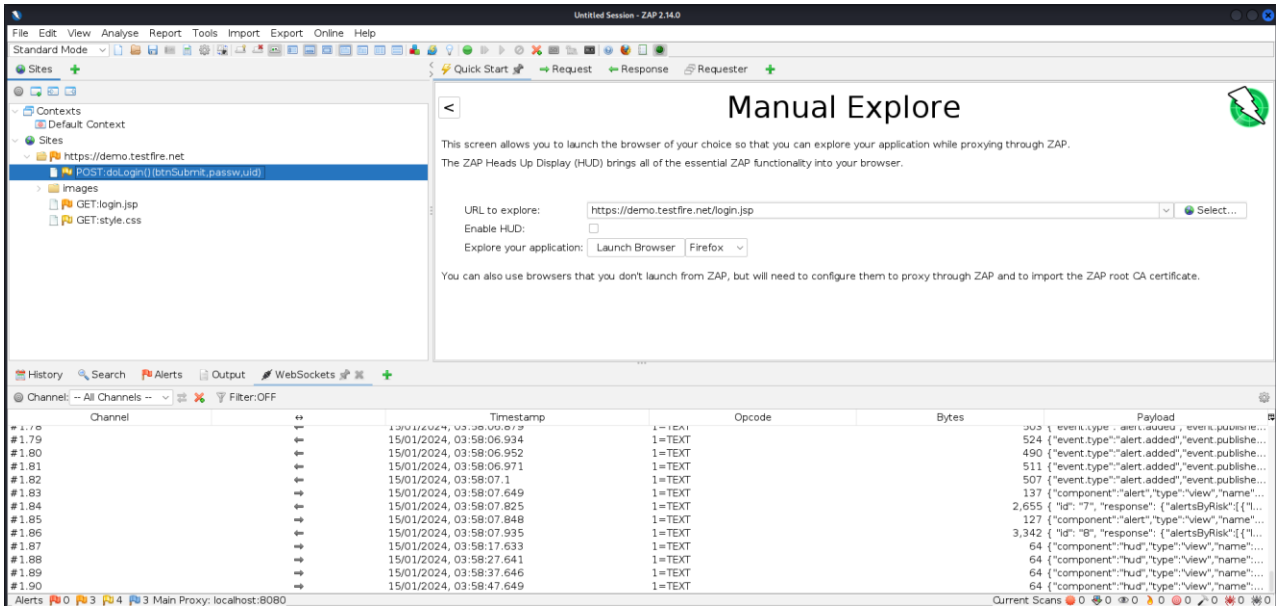


STEP 3: Click on 'Continue to your target' and enter any random string in the password field for the user **Admin** to make a failed login attempt**.**

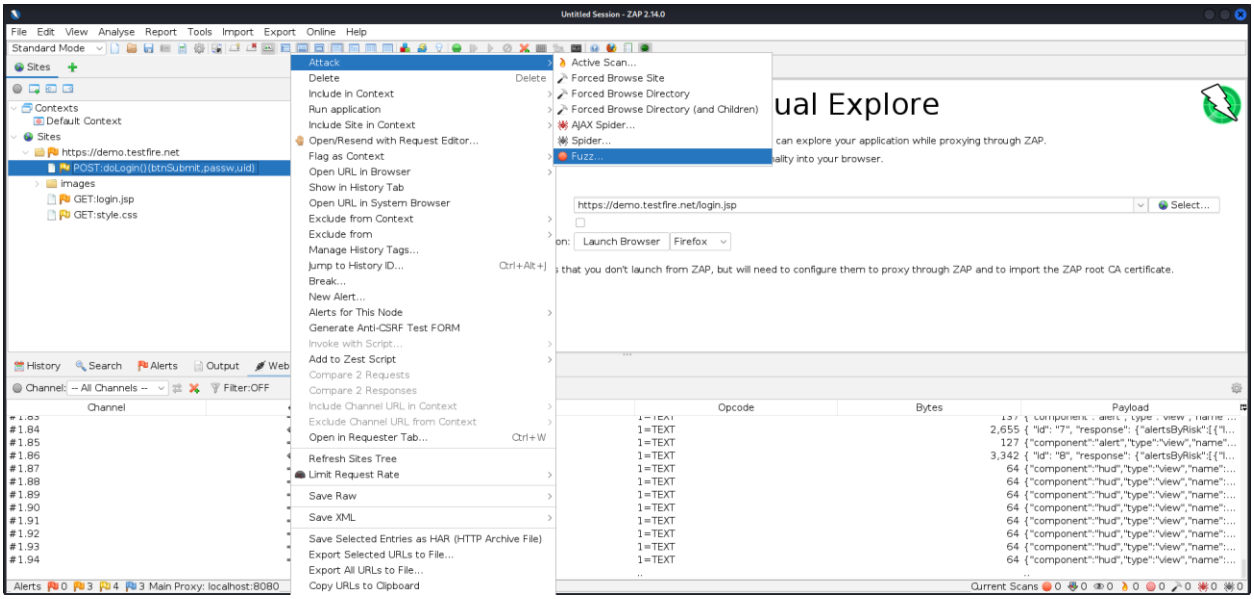




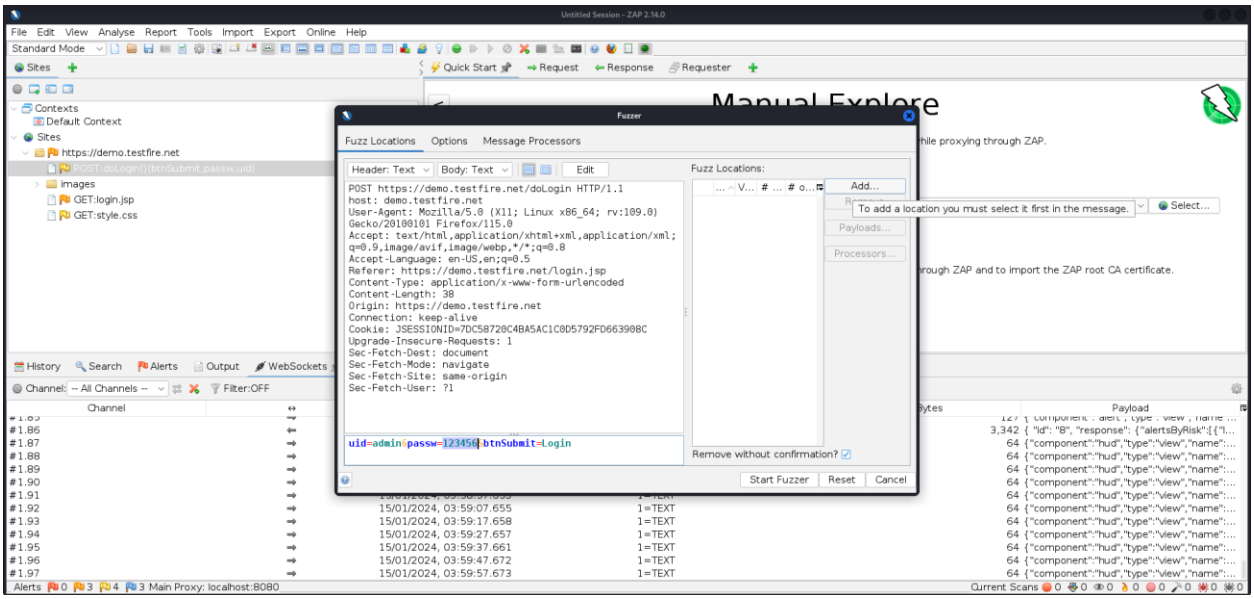
The website and the login page action will be added to the sitemap of ZAP



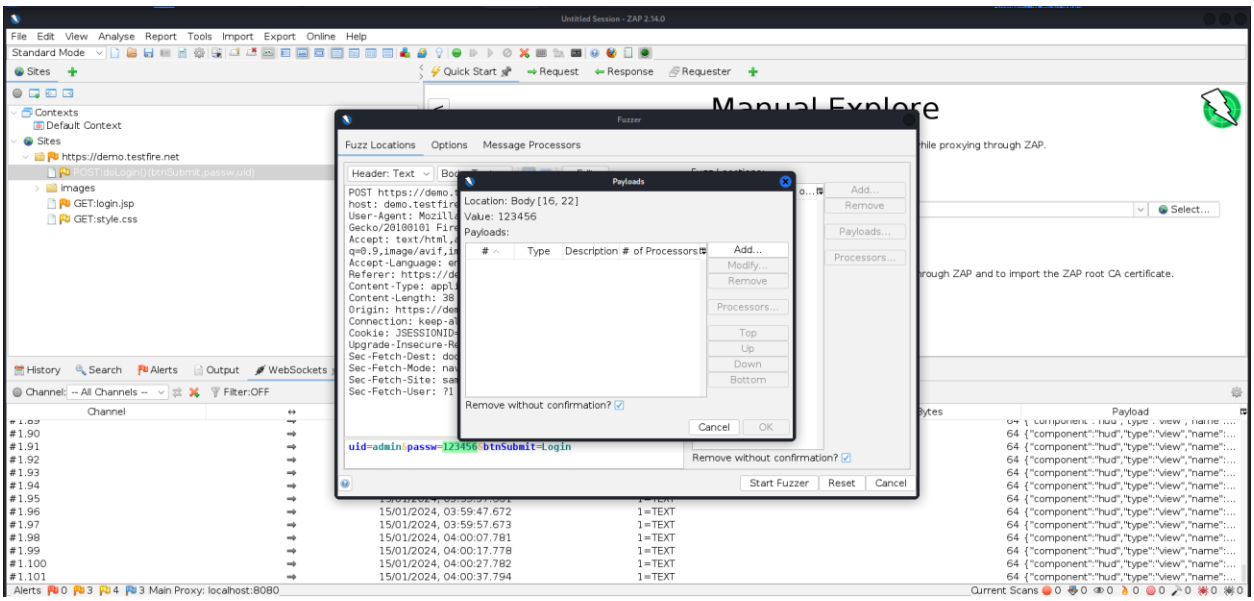
STEP 4: Right click on the POST request in the sitemap, navigate to 'Attack' and then click on 'Fuzz'.



The Fuzzer window will appear. Select the entered incorrect password "123456" and click on the Add button.

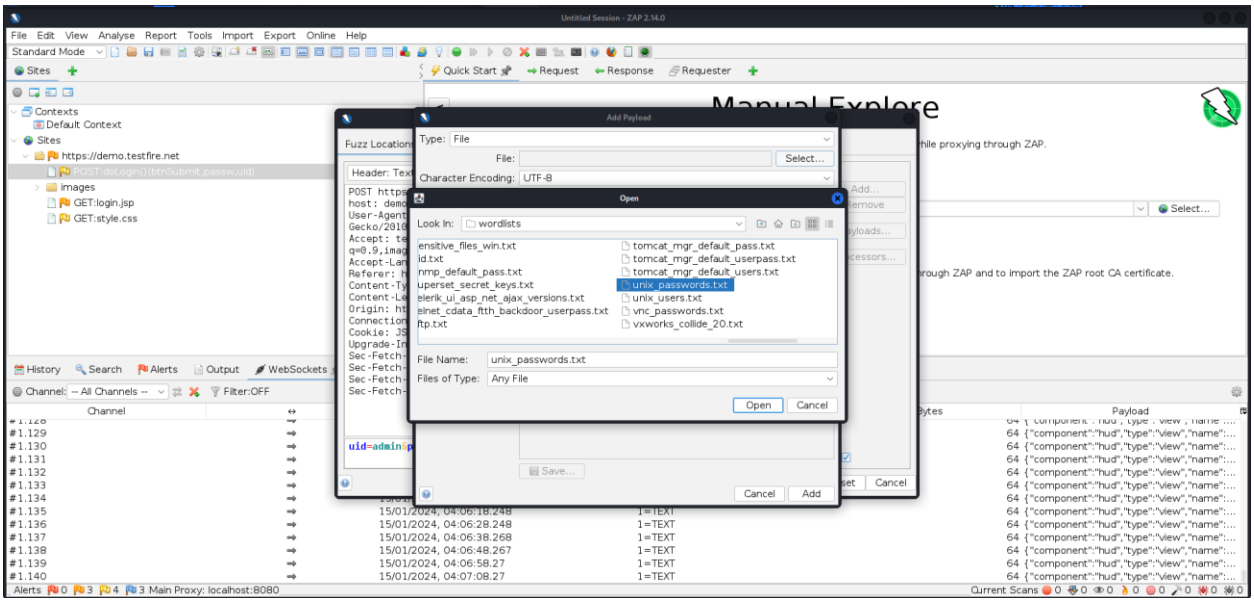


The payloads window will appear.



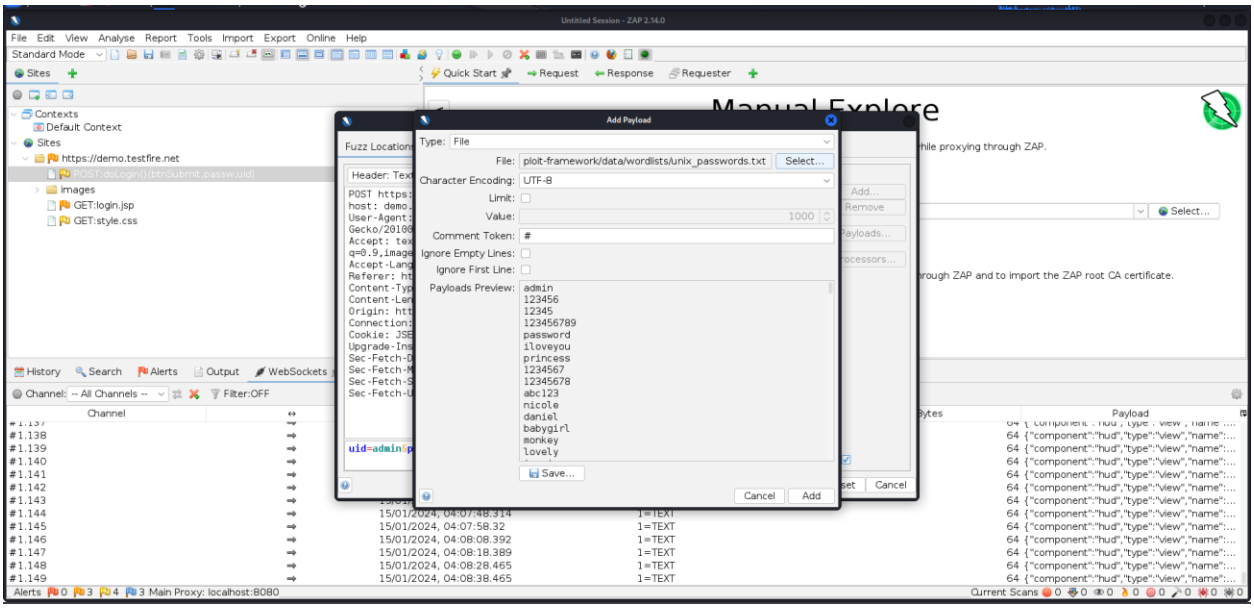
Click on the Add button and select the file for password payload.

Here ‘unix_passwords.txt’ file is selected.

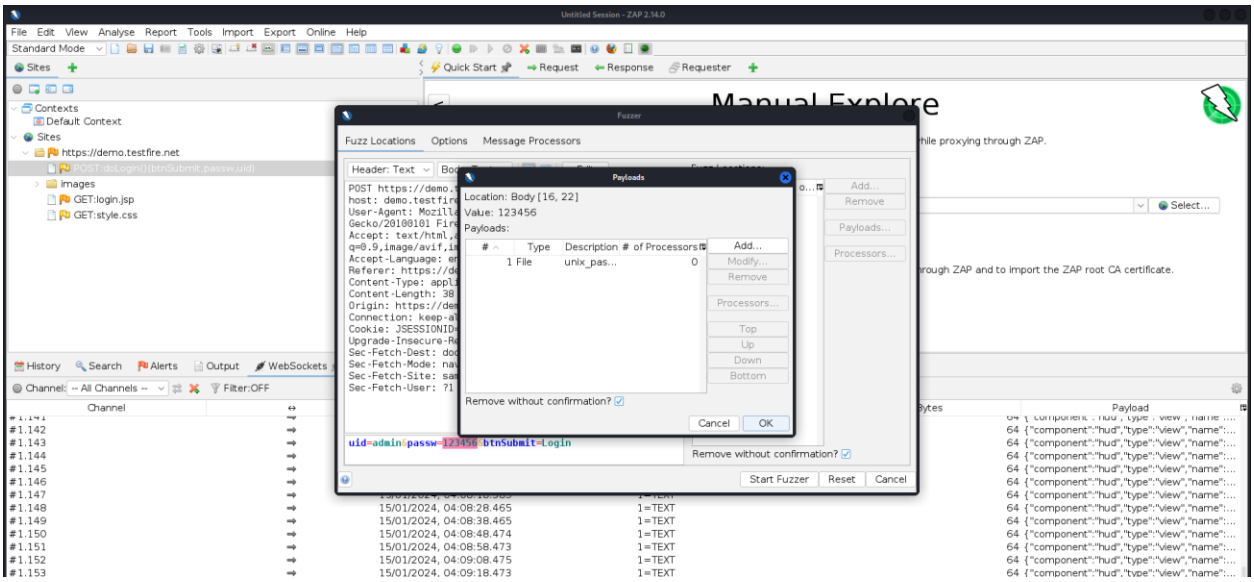


List of passwords in the ‘unix_passwords.txt’ is displayed in the Payloads Preview.

Click on the Add button.

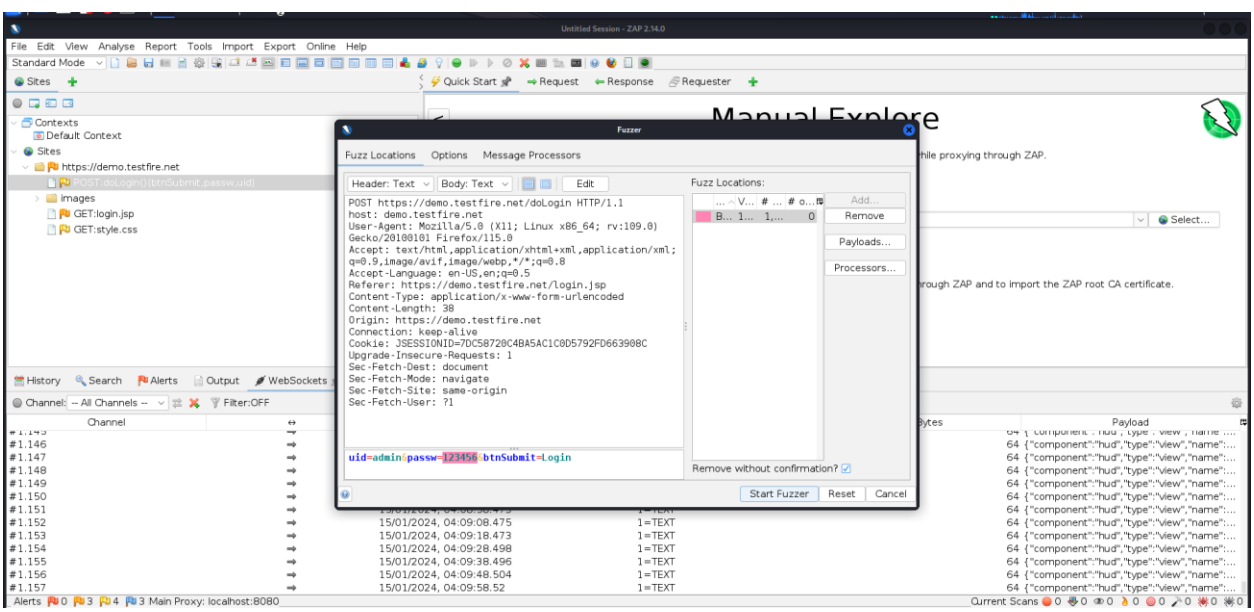


Click on the "OK" button.

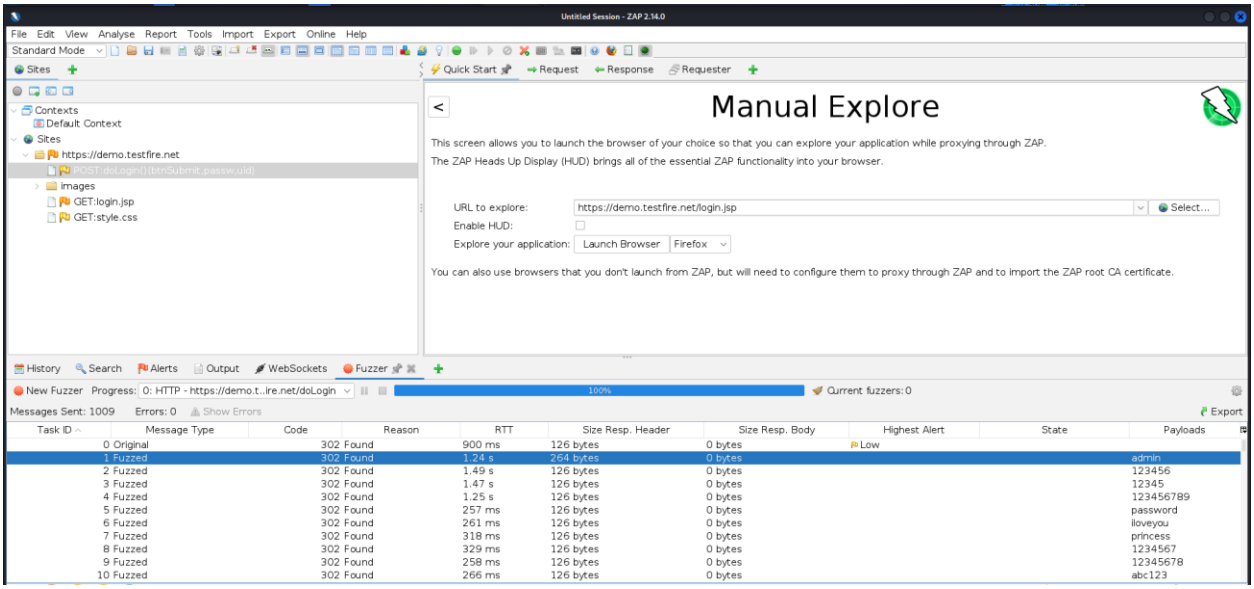


STEP 5: The payload will appear in the Fuzz Locations.

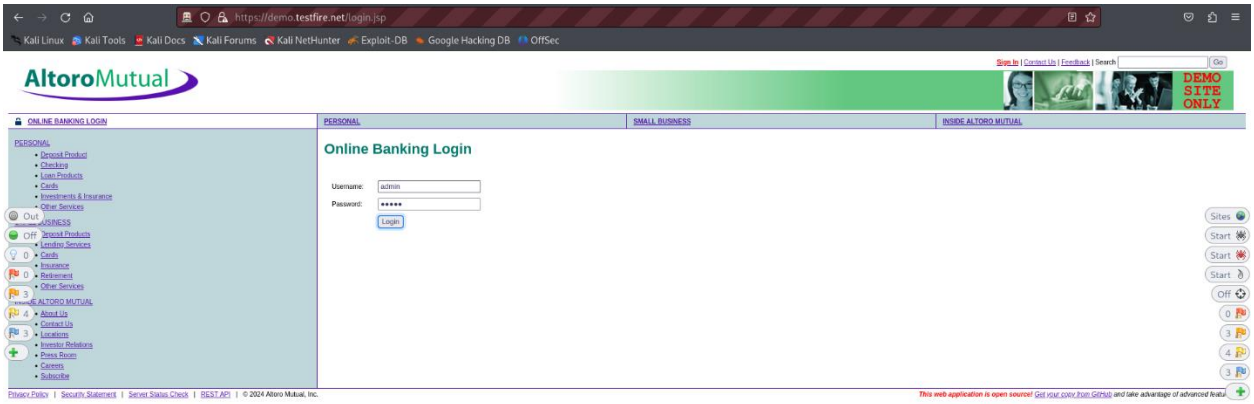
Click on ‘Start Fuzzer’.



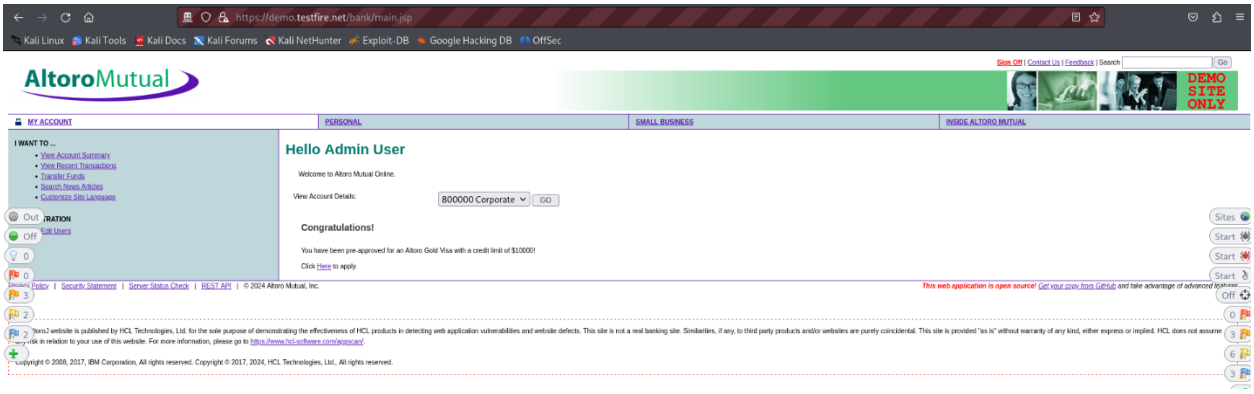
STEP 6: Upon completion of the attack, notice that one message has a different **Size Resp. Header** than the rest of the messages. So try ‘admin’ as password for the username ‘admin’.



STEP 7: Login to the web application with the password discovered in the previous step.
Username: admin
Password: admin



The login would be successful as Admin user.



Solution/Prevention:

To prevent weak password attacks and enhance overall security, consider implementing the following solutions:

- **Enforce Strong Password Policies:** Implement and enforce strong password policies that include requirements for complexity, length, and uniqueness. Encourage or mandate the use of a combination of uppercase letters, lowercase letters, numbers, and special characters.
- **Regular Password Changes:** Encourage users to regularly change their passwords. This practice limits the exposure of credentials and reduces the risk associated with long-term use of the same password.
- **Educate Users:** Provide user education and awareness programs to inform users about the importance of strong passwords and the risks associated with weak ones. Encourage them to avoid easily guessable information and to create unique, complex passwords.
- **Use Multi-Factor Authentication (MFA):** Implement multi-factor authentication to add an extra layer of security beyond passwords. MFA requires users to provide additional verification, such as a code sent to their mobile device, in addition to their password.
- **Password Managers:** Encourage or provide tools for using password managers. Password managers generate and store complex passwords, reducing the reliance on users to create and remember intricate passwords.
- **Regular Security Audits:** Conduct regular security audits and vulnerability assessments to identify weak passwords and enforce compliance with password policies. Automated tools and manual reviews can be employed for this purpose.
- **Monitor and Alert:** Implement monitoring systems to detect and alert on suspicious login activities. Unusual login patterns or multiple failed login attempts from the same account can indicate potential weak password attacks.

Tools Used:

OWASP ZAP (Open Web Application Security Project Zed Attack Proxy)

VULNERABILITY #3:

Vulnerability Name: Reflected Cross-site scripting (XSS)

Vulnerability Description: Cross-site scripting is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data. If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

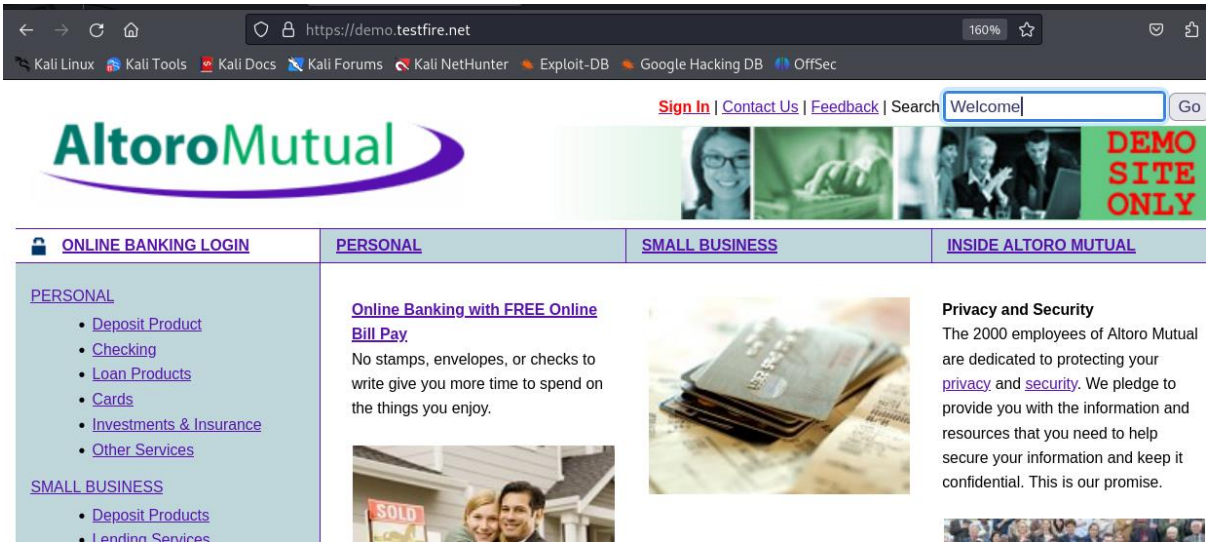
There are three main types of XSS attacks. These are:

- Stored XSS (Persistent XSS): Malicious script injected into a website's database, affecting users accessing compromised content.
- Reflected XSS (Non-Persistent XSS): Malicious script embedded in a URL or input fields, reflected off a web server to victims clicking manipulated links.
- DOM-based XSS: Attack where client-side script manipulates the Document Object Model (DOM), executing payload on the user's browser.

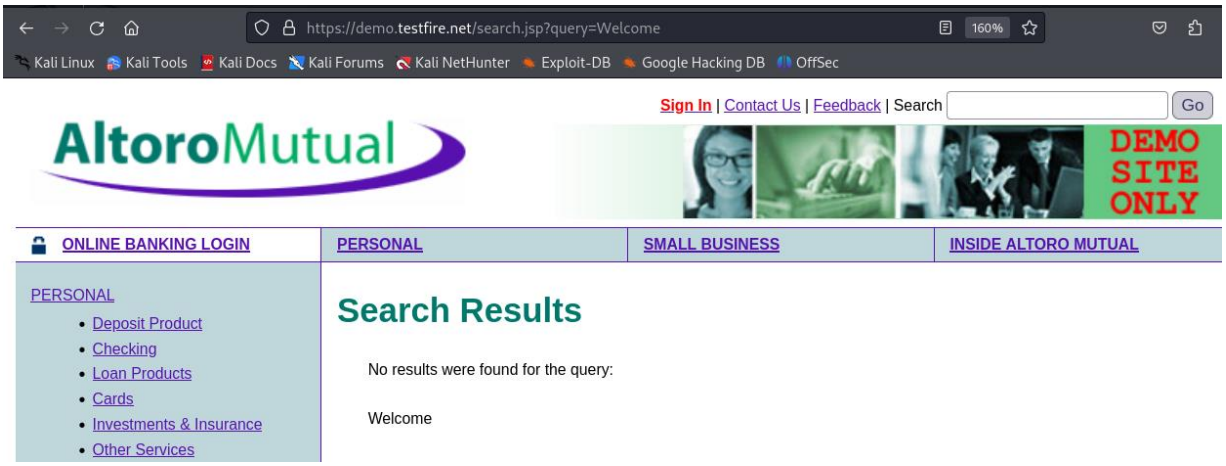
Affected URL: https://demo.testfire.net

Proof of Concept/Steps to reproduce:

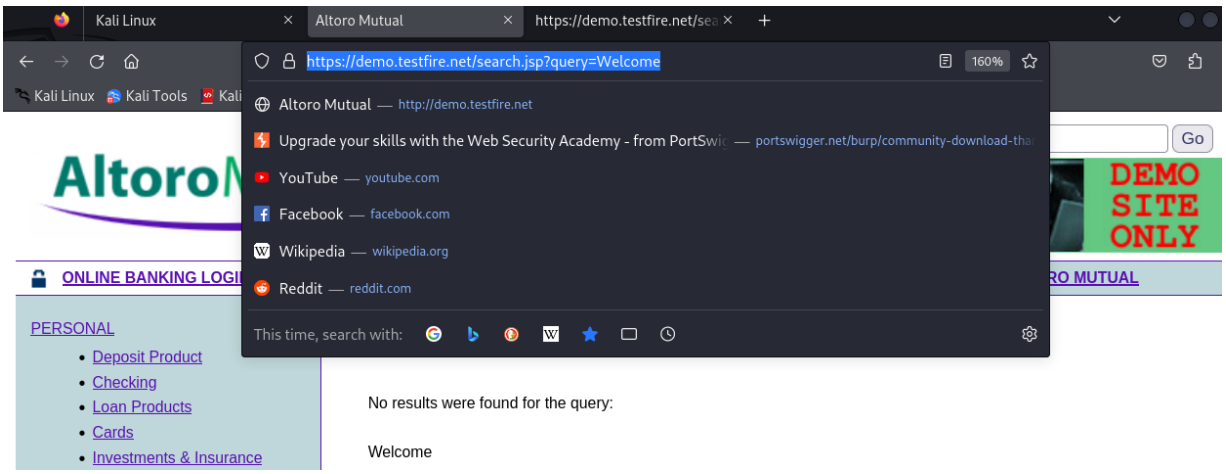
STEP 1: The website has a search function which displays the search string in the URL and webpage.



The “Welcome” string was reflected on the webpage and URL



STEP 2: Copy the URL to send it to XSSER.



STEP 3: Feed the URL to XSSER.

Replace the “Welcome” string with “XSS”, this is done so that XSSER will substitute payload in place of "XSS".

Command: `xsser -u "https://demo.testfire.net/search.jsp?query=XSS"`

The output confirms that the target is vulnerable and the payload is 100% accurate.

```
- Injections: 1
- Failed: 0
- Successful: 1
- Accur: 100.0 %

[*] List of XSS injections:

→ CONGRATULATIONS: You have found: [ 1 ] possible XSS vector! ;-)

[+] Target: https://demo.testfire.net/search.jsp?query=XSS
[+] Vector: [ query ]
[!] Method: URL
[*] Hash: 511aeb3e45fabc6681cffd652666e8be
[*] Payload: https://demo.testfire.net/search.jsp?query=%22%3E511aeb3e45fabc6681cffd652666e8be
[!] Vulnerable: [IE7.0|IE6.0|NS8.1-IE] [NS8.1-G|FF2.0] [O9.02]
[!] Status: XSS FOUND! [WITHOUT --reverse-check VALIDATION!]
```

STEP 4: Scan the target using a basic XSS payload.

Command:

```
xsser -u "https://demo.testfire.net/search.jsp?query=XSS" --Fp="<script>alert(90)</script>"
```

```
kali@kali: ~
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~]
└─$ xsser -u "https://demo.testfire.net/search.jsp?query=XSS" --Fp="<script>alert(90)</script>"
```

The encoded XSS payload is generated.

```
[*] List of XSS injections:

→ CONGRATULATIONS: You have found: [ 1 ] possible XSS vector! ;-)

[+] Target: https://demo.testfire.net/search.jsp?query=XSS
[+] Vector: [ query ]
[!] Method: URL
[*] Hash: 4e79aab539678289491bfdd0cd5e098d
[*] Payload: https://demo.testfire.net/search.jsp?query=%22%3E4e79aab539678289491bfdd0cd5e098d
[!] Vulnerable: [IE7.0|IE6.0|NS8.1-IE] [NS8.1-G|FF2.0] [O9.02]
[*] Final Attack: https://demo.testfire.net/search.jsp?query=%3Cscript%3Ealert%2890%29%3C%2Fscript%3E
[!] Status: XSS FOUND! [WITHOUT --reverse-check VALIDATION!]
```

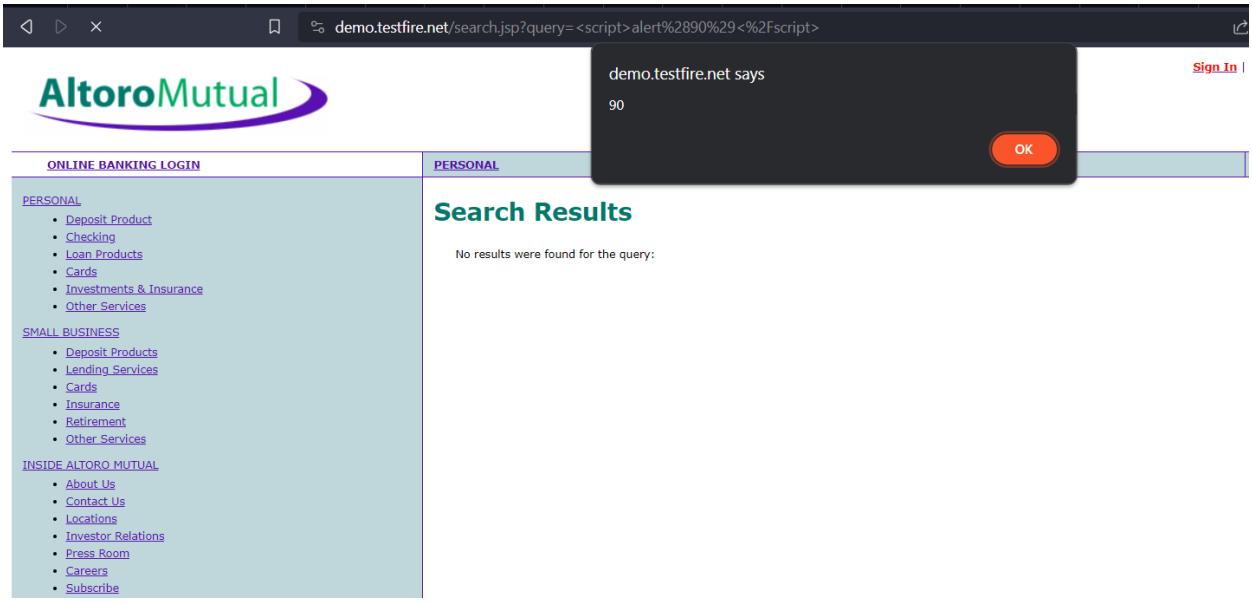
Open the ‘Final Attack’ link in the browser

```
═══════════════════════════════════════════════════════════════════
[*] Final Results:
═══════════════════════════════════════════════════════════════════

- Injections: 1
- Failed: 0
- Successful: 1
- Accur: 100.0 %

═══════════════════════════════════════════════════════════════════
[*] List of XSS injections:
═══════════════════════════════════════════════════════════════════

→ CONGRATULATIONS: You have found: [ 1 ] possible XSS vector! ;-)

____________________

[+] Target: https://demo.testfire.net/search.jsp?query=XSS
[+] Vector: [ query ]
[!] Method: URL
[*] Hash: 4e79aab539678289491bfdd0cd5e098d
[*] Payload: https://demo.testfire.net/search.jsp?query=%22%3E4e79aab539678289491bfdd0cd5e098
[!] Vulnerable: [IE7.0|IE6.0|NS8.1-IE] [NS8.1-G|FF2.0] [O9.02]
[*] Final Attack: https://demo.testfire.net/search.jsp?query=%3Cscript%3Ealert%2890%29%3C%2Fs
[!] Status: XSS FOUND! [WITHOUT --reverse-check VALIDATION!]
```

Open Link
Copy Link Address
Copy Selection Ctrl+Shift+C
Paste Clipboard Ctrl+Shift+V
Paste Selection Shift+Ins
Zoom in Ctrl++
Zoom out Ctrl+-
Zoom reset Ctrl+0
Clear Active Terminal Ctrl+Shift+X
Split Terminal Horizontally Ctrl+Shift+D
Split Terminal Vertically Ctrl+Shift+R
Collapse Subterminal Ctrl+Shift+E
Toggle Menu Ctrl+Shift+M
Hide Window Borders
Preferences...

The XSS payload will be triggered as the website would prompt the alert that is used in the Final Payload parameter in XSSER.

This confirms that the target is prone to Cross-site scripting (XSS) attack.



Solution/Prevention:

To prevent XSS (Cross-Site Scripting) attacks, consider implementing the following measures:

- **Input Validation:** Validate and sanitize user inputs on both the client and server sides. Filter out or escape special characters that can be exploited for injecting malicious scripts.
- **Output Encoding:** Encode user-generated content before rendering it on web pages. Use secure coding practices, such as HTML entity encoding, to prevent browsers from interpreting content as executable code.
- **Content Security Policy (CSP):** Implement Content Security Policy headers to control which resources and scripts are allowed to be executed on a webpage. This helps mitigate the impact of XSS attacks by restricting the execution of unauthorized scripts.
- **HTTP Only and Secure Cookies:** Use the "HttpOnly" flag for cookies to prevent client-side scripts from accessing them. Additionally, set the "Secure" flag to ensure that cookies are only transmitted over secure (HTTPS) connections.
- **Session Management Best Practices:** Implement secure session management practices, including secure cookie attributes, session timeouts, and token-based authentication. This helps prevent session hijacking and unauthorized access.
- **Use Web Application Firewalls (WAF):** Employ WAFs to filter and monitor HTTP traffic between a web application and the Internet. WAFs can detect and block malicious requests, providing an additional layer of protection against XSS attacks.

Tools Used: XSSER (Cross Site “Scripter”)

VULNERABILITY #4:

Vulnerability Name: Broken Access Control - Insecure Direct Object Reference (IDOR)

Vulnerability Description: Insecure direct object references (IDORs) are a subcategory of Broken Access Control. IDORs occur if an application uses user-supplied input to access objects directly and an attacker can modify the input to obtain unauthorized access. It is just one example of many access control implementation mistakes that can lead to access controls being circumvented. IDOR vulnerabilities are most commonly associated with horizontal privilege escalation, but they can also arise in relation to vertical privilege escalation.

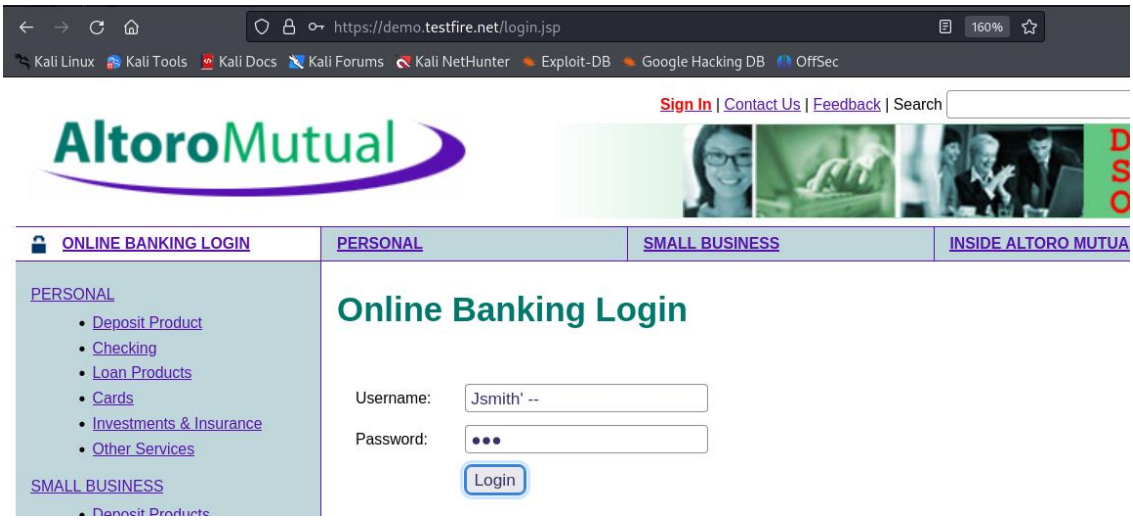
Broken Access Control vulnerabilities lead to Horizontal and Vertical privilege escalation.

- **Vertical privilege escalation:** If a user can gain access to functionality that they are not permitted to access then this is vertical privilege escalation. For example, if a non-administrative user can gain access to an admin page where they can delete user accounts, then this is vertical privilege escalation.
- **Horizontal privilege escalation:** Horizontal privilege escalation occurs if a user is able to gain access to resources belonging to another user, instead of their own resources of that type. For example, if an employee can access the records of other employees as well as their own, then this is horizontal privilege escalation.

Affected URL: https://demo.testfire.net/bank/showAccount?listAccounts=800002

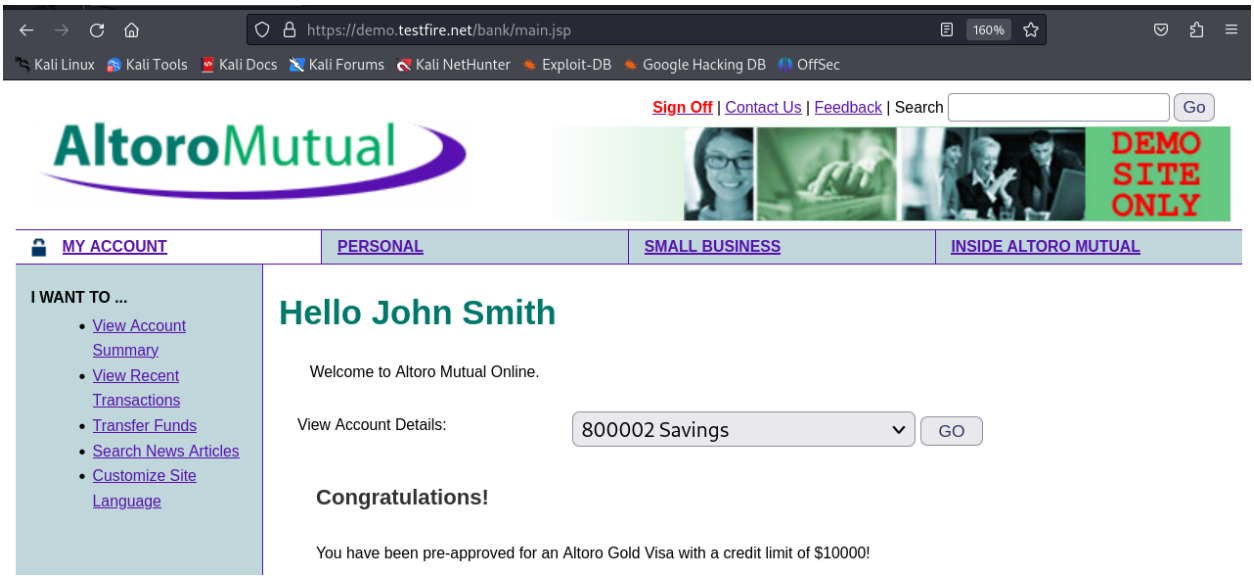
Proof of Concept/Steps to reproduce:

STEP 1: To make sure the website is prone to both the horizontal and vertical privilege escalation, logging into the website as standard user by using the SQL injection vulnerability.

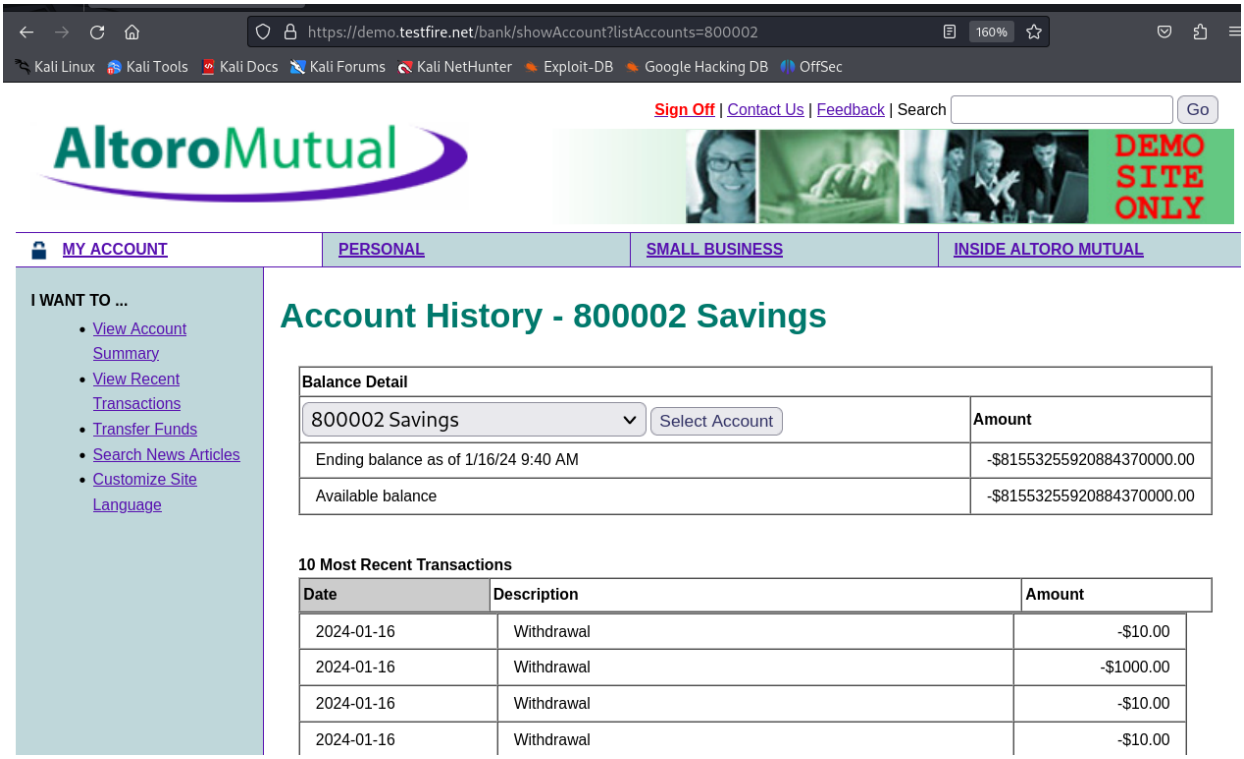


STEP 2: Successfully logged in as ‘John Smith’, who is a standard user.

Click on the ‘Go’ button to view the account details of John Smith.

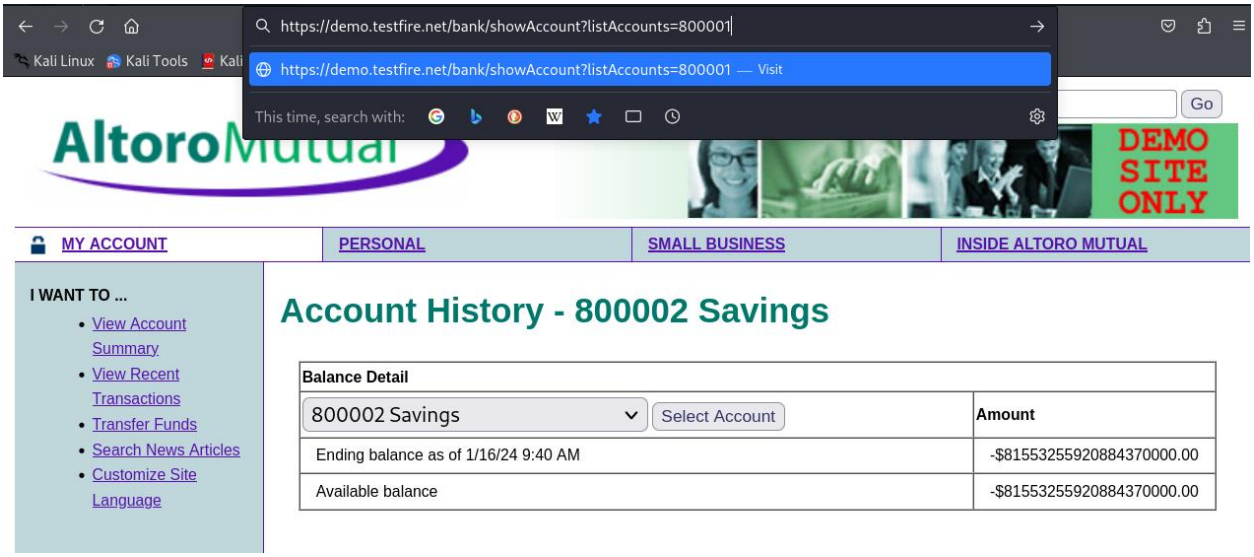


Here the Accounts ID of John Smith is clearly visible in the URL, which is a direct object reference. Changing the Accounts ID would result in other user’s account details.

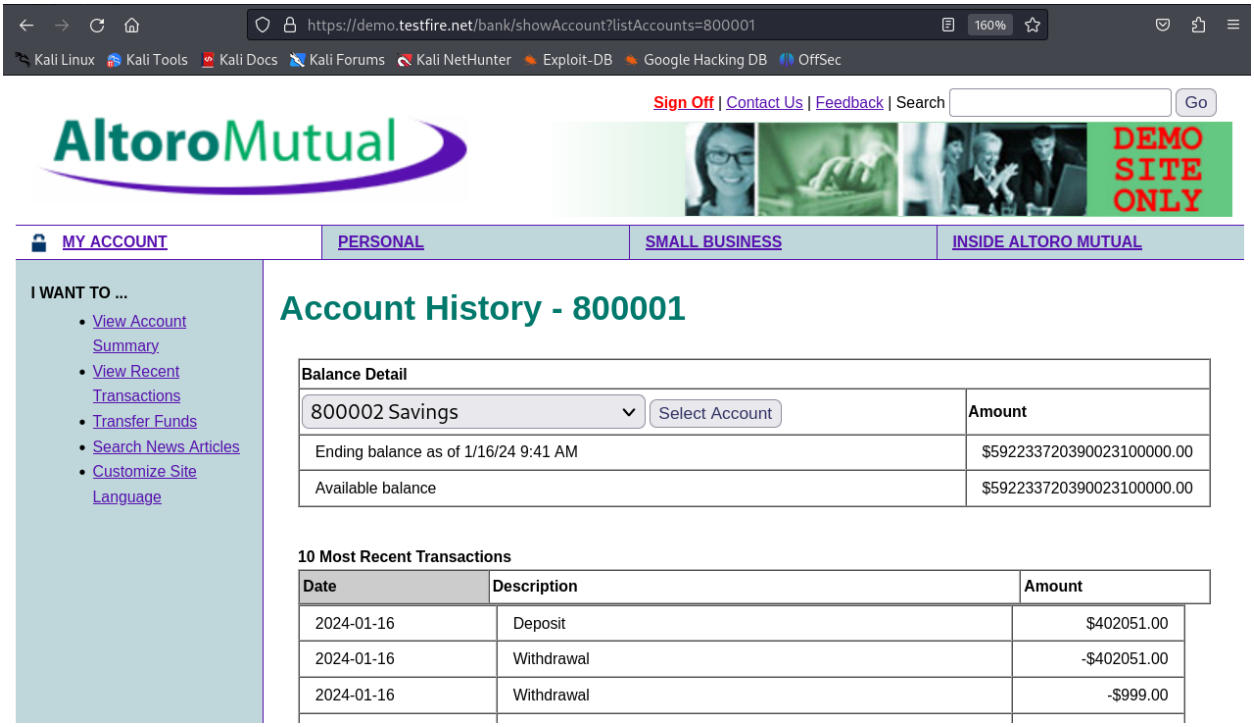


STEP 3: Vertical Privilege Escalation

Change the Accounts ID from 800002 to 800001 to get the account details of privileged user i.e., Admin user.

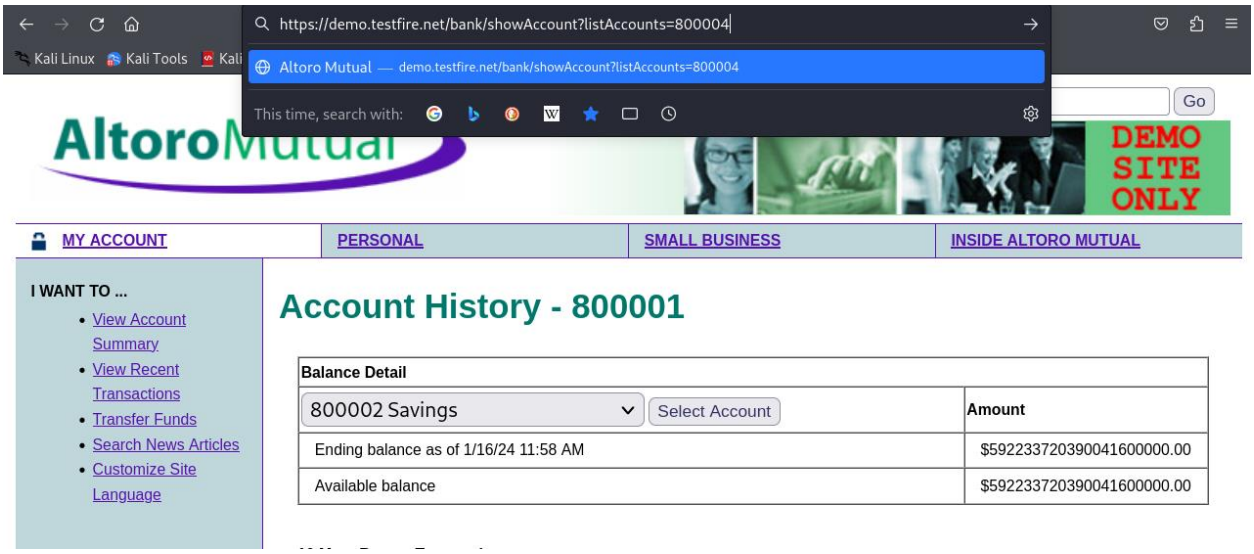


Changing the Accounts ID to 800001, the account details of Admin user is displayed. This is a vertical privilege escalation vulnerability, where the less privileged user John Smith was able to view the account details of higher privileged user.

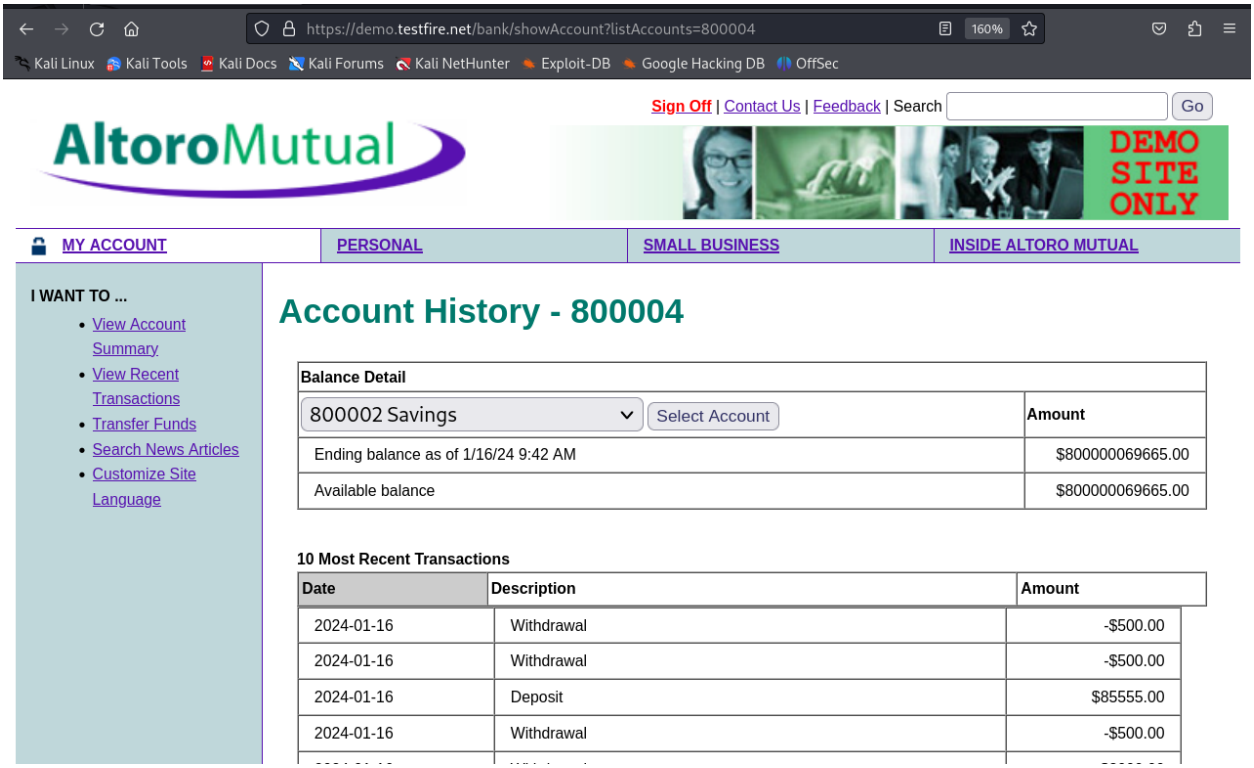


STEP 4: Horizontal Privilege escalation

Change the Accounts ID to 800004 to view the account details of other user.



The account details of the other user Jdoe is being displayed. This is a horizontal privilege escalation vulnerability, where one user knows the account details of other users.



Solution/Prevention:

- **Implement Proper Access Controls:** Enforce strong access controls based on user roles and permissions. Apply the principle of least privilege, ensuring that users only have the minimum level of access required to perform their tasks.
- **Use Object-Level Authorization:** Implement object-level authorization checks to ensure that users are authorized to access specific objects or resources. Verify user permissions before allowing access to or modification of sensitive data.
- **Use UUIDs or Unpredictable Object Identifiers:** Use Universally Unique Identifiers (UUIDs) or other unpredictable identifiers for objects rather than sequential or predictable identifiers. This makes it more challenging for attackers to guess or manipulate object references.
- **Validate User Input:** Perform thorough validation and sanitization of user input, especially when it involves object references. Validate that the user has the proper authorization before processing requests.
- **Apply Contextual Access Controls:** Implement contextual access controls based on the user's context, such as their session, role, or state. Consider the context in which requests are made to determine whether they should be allowed.
- **Use Indirect References:** Use indirect references, such as indexes or pointers, rather than directly exposing sensitive information in URLs or parameters. Ensure that indirect references are properly validated and authorized.
- **Implement Proper Session Management:** Enforce secure session management practices, including secure session token handling, session timeouts, and secure session storage. Regularly review and update session management mechanisms to address security concerns.

Tools Used: NA

VULNERABILITY #5:

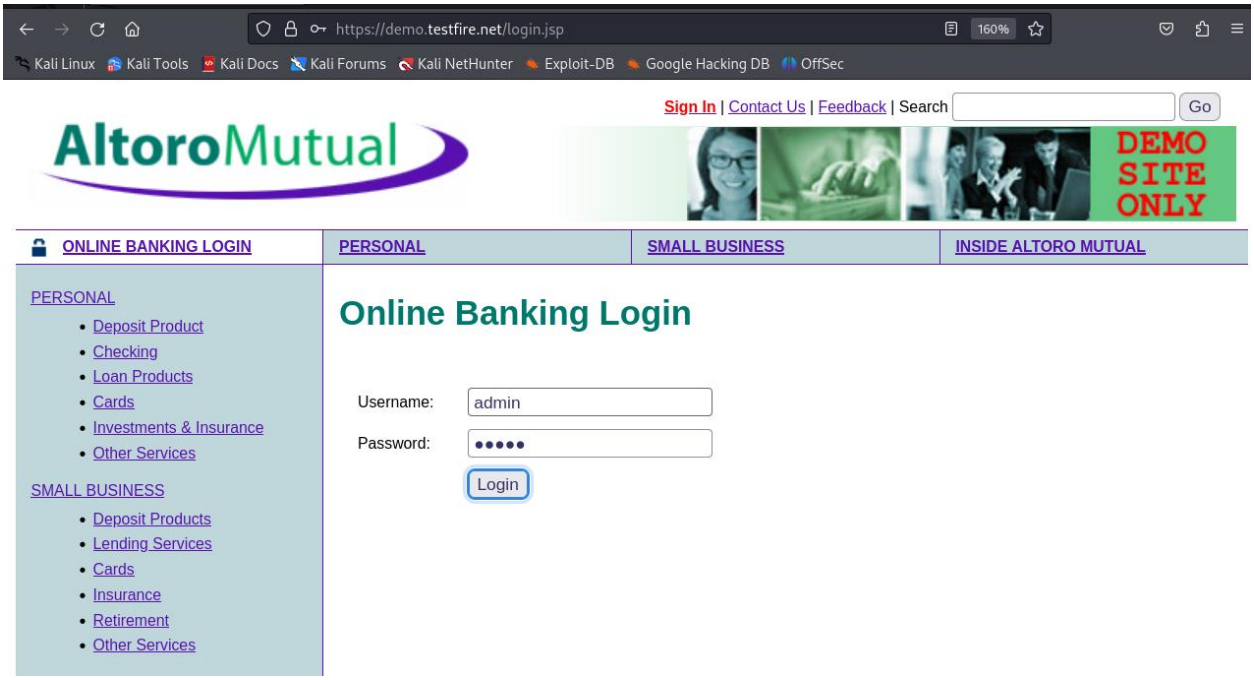
Vulnerability Name: External Redirect vulnerability

Vulnerability Description: External Redirect Vulnerability refers to a security issue in web applications where an attacker can manipulate or redirect users to external, potentially malicious websites. This vulnerability arises when a web application allows user-controlled input, such as URLs or parameters, to influence the destination of a redirect. If the application does not properly validate or sanitize these inputs, an attacker can craft malicious URLs to redirect users to phishing sites, malicious content, or other destinations outside the legitimate scope of the application. URL redirectors represent common functionality employed by web sites to forward an incoming request to an alternate resource. This can be done for a variety of reasons and is often done to allow resources to be moved within the directory structure and to avoid breaking functionality for users that request the resource at its previous location. URL redirectors do not necessarily represent a direct security vulnerability but can be abused by attackers trying to social engineer victims into believing that they are navigating to a site other than the true destination.

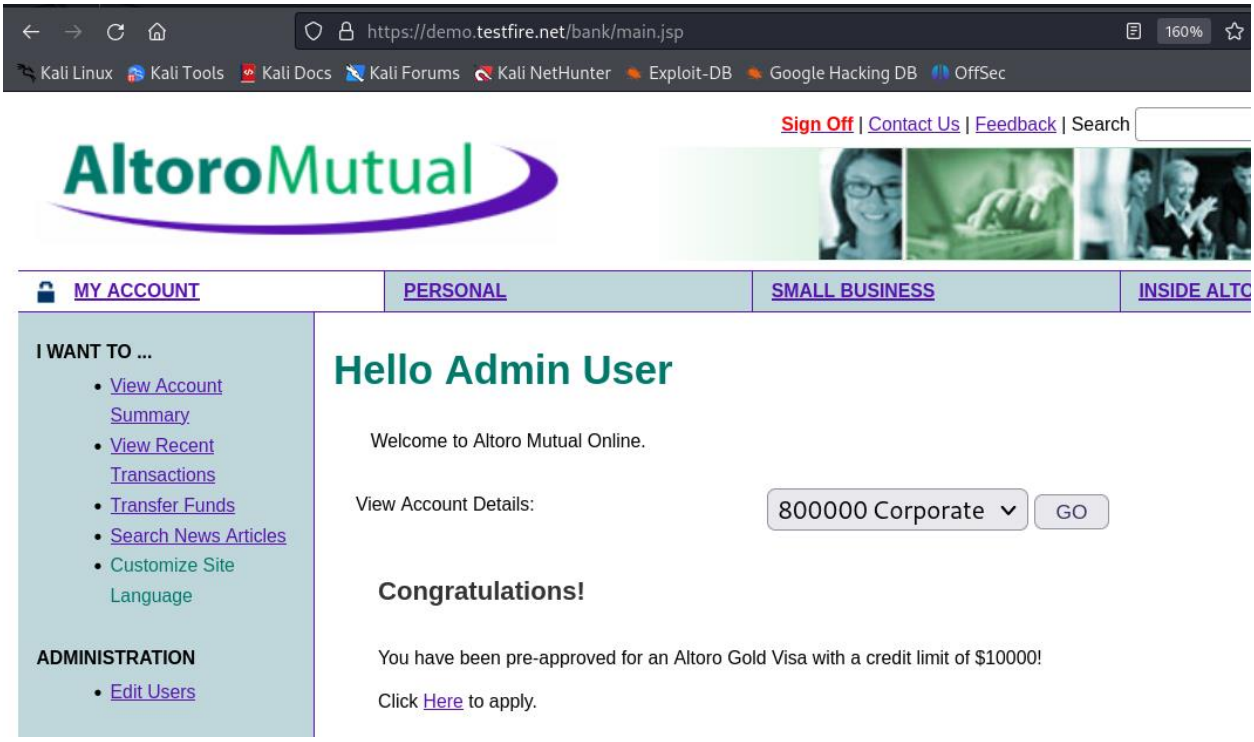
Affected URL: https://demo.testfire.net/bank/customize.jsp

Proof of Concept/Steps to reproduce:

STEP 1: Authenticate into the website using known credentials. Here being logged in as Admin user.

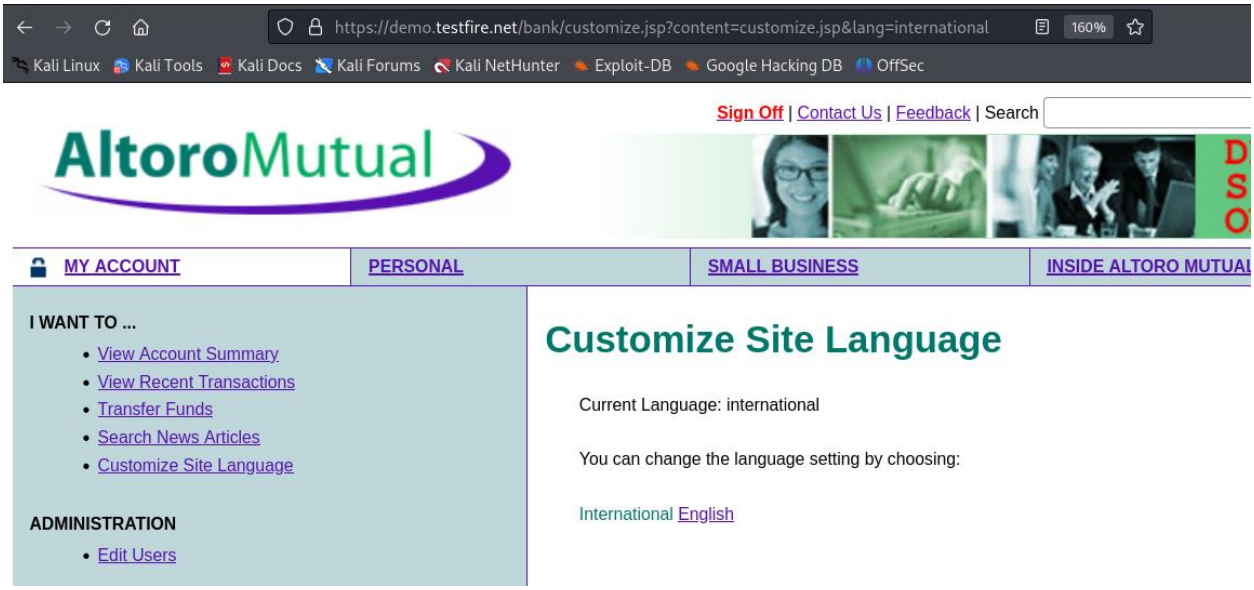


STEP 2: After logged in, click on 'Customize Site Language'

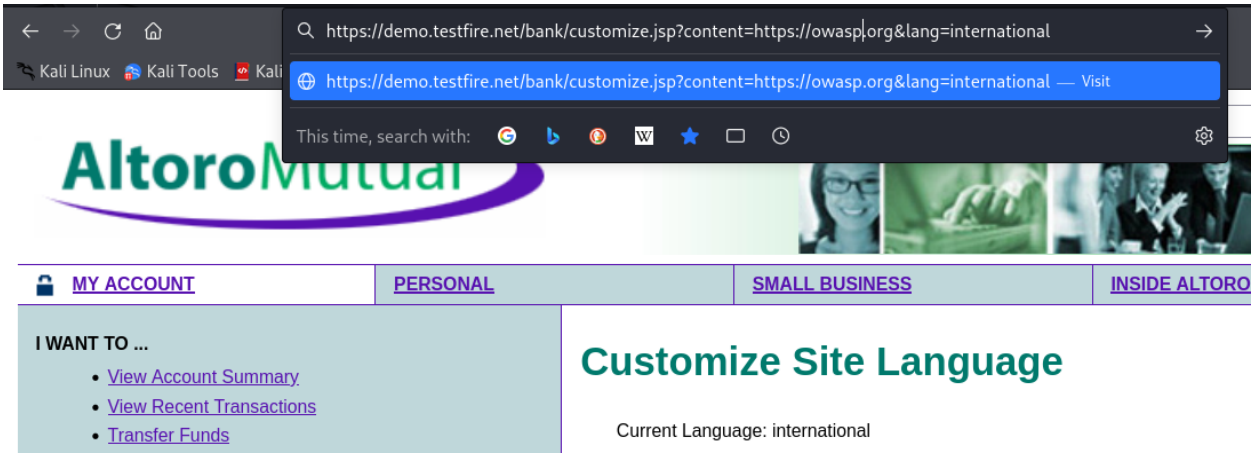


STEP 3: Click on any language to select. Here 'International' is clicked, so the current language changes to International.

In the URL, the 'content' query parameter redirects the browser from English to International and vice versa.



STEP 4: As the ‘content’ query parameter is used to redirect this website to the other website, using the OWASP official website link in the query parameter to redirect from https://demo.tesfire.net to https://owasp.org



Redirection to OWASP website.

Solution/Prevention:

To prevent Redirect vulnerabilities, consider implementing the following measures:

- Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
- Validate and sanitize user inputs, especially those used for redirection purposes. Ensure that only valid and expected values are accepted. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, and consistency across related fields, and conformance to business rules.
- Use an allow list of approved URLs or domains to be used for redirection. Maintain a whitelist of trusted domains to which redirects are allowed. Only permit redirection to URLs on this whitelist.
- Use an intermediate disclaimer page that provides the user with a clear warning that they are leaving your site. Implement a long timeout before the redirect occurs, or force the user to click on the link. Be careful to avoid XSS problems when generating the disclaimer page.
- Minimize instances where users can control or influence redirection. If possible, limit such functionality or apply strict validation.
- Understand all the potential areas where untrusted inputs can enter the software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, databases, and any external systems that provide data to the application.

Tools Used: NA