AI RED TEAMER

RESEARCHED BY:
RASHID IQBAL

"I don't want fame, I want mastery"

This is Part-3 of Documented Operations on Kali Linux

--(DOCUMENTED OPERATIONS ON KALI)--

DELVING DEEP INTO CONCEPT OF BROKEN ACCESS CONTROL, IDOR (INSECURE DIRECT OBJECT REFERENCE) AND TESTING URLs, PARAMETERSE, AND API ENDPOINTS FOR ACCESS ISSUES.

🛠️ BROKEN ACCESS CONTROL:

HOW I DEFINE IT TECHNICALLY:

I think of **Access Control** as the security mechanism that dictates what a user is **allowed** to do, see, or touch within a web application. It is the gatekeeper that sits between a user and a resource.

Access control typically relies on **three concepts**:

I.	**Authentication**	:	Who you are (logging in)
II.	**Authorization**	:	What you are allowed to do (permissions)
III.	**Session Management**	:	Tracking your state as you move through the application

Broken Access Control is a complete failure in the authorization step. It happens when an application's code allows a user to perform an action or view a resource that their current permission level **should not** permit. It is the most critical and frequently found vulnerability (often **#1 on the OWASP Top 10**)

🎯 WHY BROKEN ACCESS CONTROL FAILS:

I consider Broken Access Control a common failure because it is often overlooked or implemented inconsistently throughout a large application. A major failure type is **Missing Authorization Checks**.

MISSING AUTHORIZATION CHECKS:

The root cause of failure is usually the developer forgetting to ask one simple question at the server level:
"**Does the current user have permission to access this resource?**"

- **Trusting the User:**
  The developer trusts the front-end (browser) to enforce permissions (e.g., hiding an "Admin" button). An attacker bypasses this by directly sending a request to the server, which then fails to check the user's role.
- **Horizontal vs Vertical Privilege:**
  - **Horizontal Privilege Escalation (IDOR):**
    A user accesses resources belonging to another user with the same privilege level (e.g., User A accessing the User B's private account details).
  - **Vertical Privilege Escalation:**
    A low-privilege user accesses a high-privilege function (e.g., a "Customer" accessing an "Admin" page)

💰 **KEY VULNERABILITIES & TESTING METHODS:**

I have to use my brain and manual testing techniques to find these issues, as automated scanners often struggle with the subtle logic errors that cause access control failures.

IDOR (INSECURE DIRECT OBJECT REFERENCE):

This is the most common manifestation of Broken Access Control. It happens when the application exposes a reference to an internal object (like a file path or database key) and doesn't check the requesting user's permission to access that specific object.

Example:

A banking application uses a URL like:
*https://bank.com/account?id=**1234***

The attacker changes the ID parameter to another user's account number:
*https://bank.com/account?id=**5678***

If the application does not check that the logged-in user is the owner of account 5678, the attacker has successfully performed an IDOR attack.

TESTING URLs, PARAMETERS, AND API ENDPOINTS:

I use my Burp Suite Proxy & Repeater tools extensively for this manual testing…

TESTING TARGET	BURP TOOL USED	TESTING TECHNIQUE
URL PATHS	Proxy & Repeater	I try to access restricted paths by manually changing the URL. Example: Accessing **/admin/** or **/user_management/** when logged in as a standard user.
PARAMETERS	Proxy & Repeater	I focus on parameters in the URL query string or POST body **(like id= , username= , account_id=).** I change my user ID to another user's ID to check for IDOR.
API ENDPOINTS	Proxy & Repeater	Modern apps use APIs. I capture the API call **(e.g., POST /api/v1/user/1234/details)** and change the resource ID **(1234 to 5678)** to see if the server validates ownership.

FORCED BROWSING AND MISSING AUTHORIZATION CHECKS:

Forced browsing is the act of manually forcing the browser (or Burp Repeater) to request pages that are not linked or visible in the application's user interface.

- **Unauthenticated Access:**
  I first **log out** completely and then try to browse the pages that should require a login (e.g., **/dashboard**). If I can access them, that is a critical failure.
- **Role-Based Access:**
  I **log in** as a low-privilege user and then force-browse to pages that only high-privilege users (like **Administrators**) should see. If the page loads and allows me to perform an action, the authorization check is missing or broken.

DURING MY TESTING ON A LOCAL LAB (DVWA / JUICE SHOP), I IDENTIFIED AN **IDOR VULNERABILITY** BY CHANGING THE **USER_ID** PARAMETER IN THE REQUEST; IN THE ORIGINAL REQUEST I ACCESSED MY OWN DATA, BUT AFTER MODIFYING THE **USER_ID** VALUE, I WAS ABLE TO VIEW ANOTHER USER'S INFORMATION, PROVING MISSING AUTHENTICATION CHECKS. I ALSO FOUND AN **AUTHENTICATION / AUTHORIZATION ISSUE** WHERE A SENSITIVE PAGE WAS ACCESSIBLE DIRECTLY VIA URL WITHOUT LOGGING IN; THE ORIGINAL REQUEST WAS BLOCKED IN THE UI, BUT THE MODIFIED DIRECT REQUEST GRANTED ACCESS. THE MAIN CHANGE IN BOTH CASES WAS SIMPLE PARAMETER / URL MANIPULATION WITHOUT VALID PERMISSION CHECKS. I ATTACHED SCREENSHOTS OF THE ORIGINAL AND MODIFIED REQUESTS, AND THE IMPACT IS SERIOUS BECAUSE AN ATTACKER CAN ACCESS OTHER USERS' DATA OR RESTRICTED FUNCTIONS, LEADING TO DATA LEAKAGE AND PRIVILEGE ABUSE.

IDOR VIA USER ID MANIPULATION:

WHAT I DID:

I opened the **SQL Injection page** and first entered my own **user ID** to see my data.

WHAT I CHANGED:

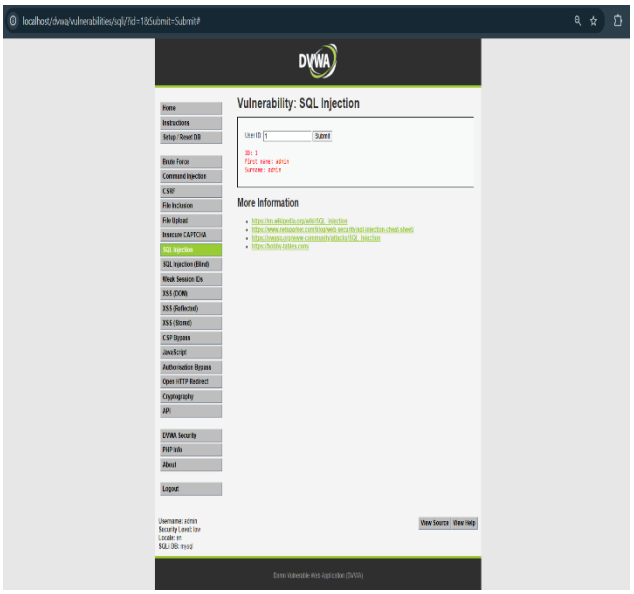
After the page loaded, I manually changed the **user ID** value in the URL to another number **(i.e. 2, 3, 4, etc)**.

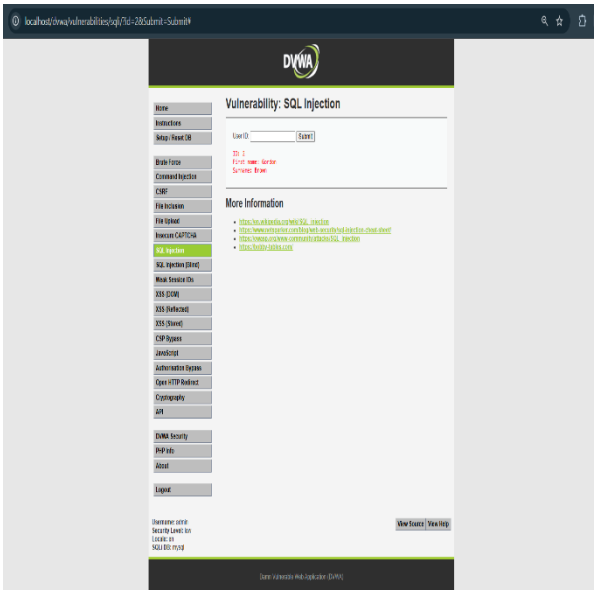
WHAT HAPPENED:

The application showed another user's data without asking for permission or login again.

CONCLUSION:

This proved by simply **changing a user ID**, I could access other users' information, which demonstrates an **IDOR vulnerability**. This is a type of **Broken Access Control vulnerability** where the application trusts user-supplied input (like a user ID) and allows access to another user's data without proper authorization checks.





UNAUTHORIZED ACCESS TEST (DVWA) – BROKEN ACCESS CONTROL:

ORIGINAL REQUEST:

I accessed a **DVWA vulnerability page** normally while logged in, using its direct URL in the browser.

MODIFIED REQUEST:

After **logging out**, I entered the same URL directly into the browser without providing any login credentials.

WHAT CHANGED:

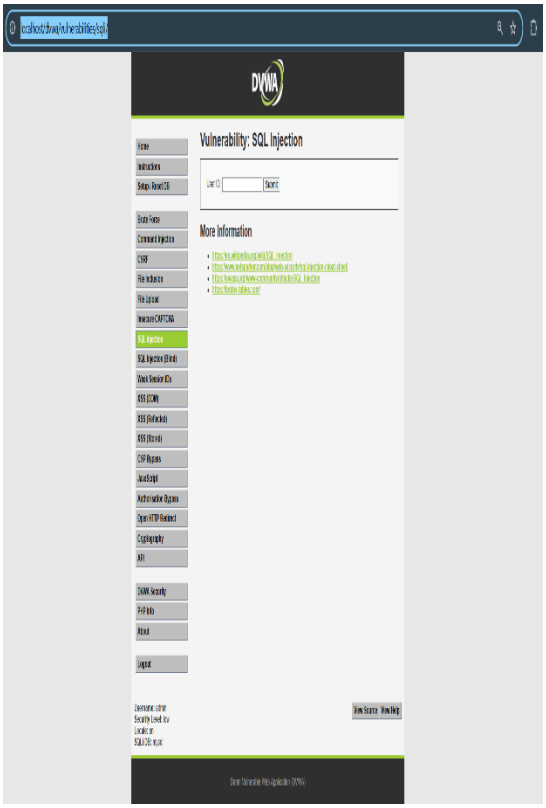
The only change was that I was no longer authenticated, but **DVWA** redirected me back to the **login page** instead of allowing access.

EXPLANATION:

This behavior showed that **DVWA** correctly checks authentication before allowing access to its pages.

CONCLUSION:

I was not able to access any page or function without proper authentication. Therefore, the vulnerability known as **Broken Access Control** was not present in my practical testing on **DVWA**.



TESTING SESSION MANAGEMENT IN THE APPLICATION BY FIRST OBSERVING HOW SESSION COOKIES WERE CREATED AND USED TO MAINTAIN MY LOGIN STATE. I ANALYZED THE SESSION TOKENS TO CHECK WHETHER THEY WERE WEAK OR PREDICTABLE, AS SUCH TOKENS CAN ALLOW AN ATTACKER TO HIJACK A VALID SESSION. I ALSO TESTED FOR SESSION FIXATION BY ATTEMPTING TO REUSE A SESSION ID BEFORE AND AFTER LOGIN TO SEE IF IT REMAINED THE SAME. FINALLY, I TESTED THE LOOUT FUNCATIONALITY TO CONFIRM THAT THE SESSION WAS PROPERLY INVALIDATION AND COULD NOT BE REUSED AFTER LOGGING OUT.

🛠️ SESSION MANAGEMENT TESTING:

HOW I DEFINE IT TECHNICALLY:

I understand **Session Management** as the way a web application remembers me after I log in. When I enter my username and password, the application does not ask me to log in again on every page. Instead, it creates a session and assigns me a unique session ID, usually stored in a session cookie. This session ID tells the server, "This request is coming from the same logged-in user."

Session Management is built on three main ideas:

A.	**Session Creation**	:	The server creates a unique session ID after successful login
B.	**Session Usage**	:	The session ID is sent with every request to prove I am authenticated
C.	**Session Termination**	:	The session must be destroyed when I log out or when it expires

If any of these steps are weak, an attacker can take over a user's session without knowing the password.

🎯 WHY SESSION MANAGEMENT FAILS:

I consider **session management** failures dangerous because they completely break authentication. Even if the login system is strong, poor session handling allows attackers to skip login and directly act as a valid user. These issues usually happen because developers do not properly protect, regenerate, or destroy session IDs.

🔍 HOW SESSION COOKIES WORK (WHAT I TESTED):

I observed how the application generated **session cookies** after I logged in. I checked how the cookie was used to maintain my **login state** while I moved between different pages. I also verified whether the session cookies changed after login and whether it was required to access protected areas of the application.

⚠️ WEAK SESSION TOKENS & PREDICTABLE COOKIES:

I analyzed the **session token** to see if it was simple, short, or predictable. Weak session tokens are risky because attackers can **guess or brute-force** them. If an attacker obtains a valid **session ID**, they can impersonate a user without needing **login credentials**.

🔗 SESSION FIXATION:

I tested for **session fixation** by checking whether the same **session ID** was used before and after login. If the session ID doesn't change after authentication, an attacker can force a victim to use a known **session ID** and later hijack the session once the victim logs in.

🚪 LOGOUT & SESSION INVALIDATION TESTING:

I tested the logout feature to ensure the session was properly destroyed on the server side. I checked whether I could reuse the old **session ID** or access restricted pages after logging out. Proper **session invalidation** ensures that once a user logs out, their session cannot be reused by anyone.

SESSION MANAGEMENT TESTING – PRACTICAL LAB:

LAB GOAL:

Understand how sessions work, identify weak session handling, and test how sessions can be **hijacked** or **misused**.

- **Target** : Any test app (DVWA / Juice Shop / PortSwigger lab)
- **Tool** : Browser + DevTools / Burp Suite
- **Account** : Test user credentials

SESSION CREATION TEST:

Objective: Check how the **session ID** is created

Steps:

1. Open the **login page**
2. **Log in** with valid credentials
3. Open **DevTools → Application → Cookies**
4. Identify the **session cookie** (e.g., PHPSESSID, JSESSIONID)

What I Check:

- Is the **session ID** long and random?
- Is it **created only after login**?

Weakness Indicator:

- Short or readable **session IDs**
- Session ID **exists before login**

Conclusion:

In **DVWA**, a **session** created both before and after login, so what does this means security-wise?

In **DVWA**, the application creates a session even before login to track a **user as a guest** and **manage basic interactions**. After **login**, this same session should ideally be regenerated to represent an **authenticated user**. If the **session ID** remains the same before and after login, it **indicates a session fixation vulnerability**, because an attacker could pre-set a known session ID and **hijack the session once the victim logs in**. In secure implementations, the **session ID** must change after authentication.

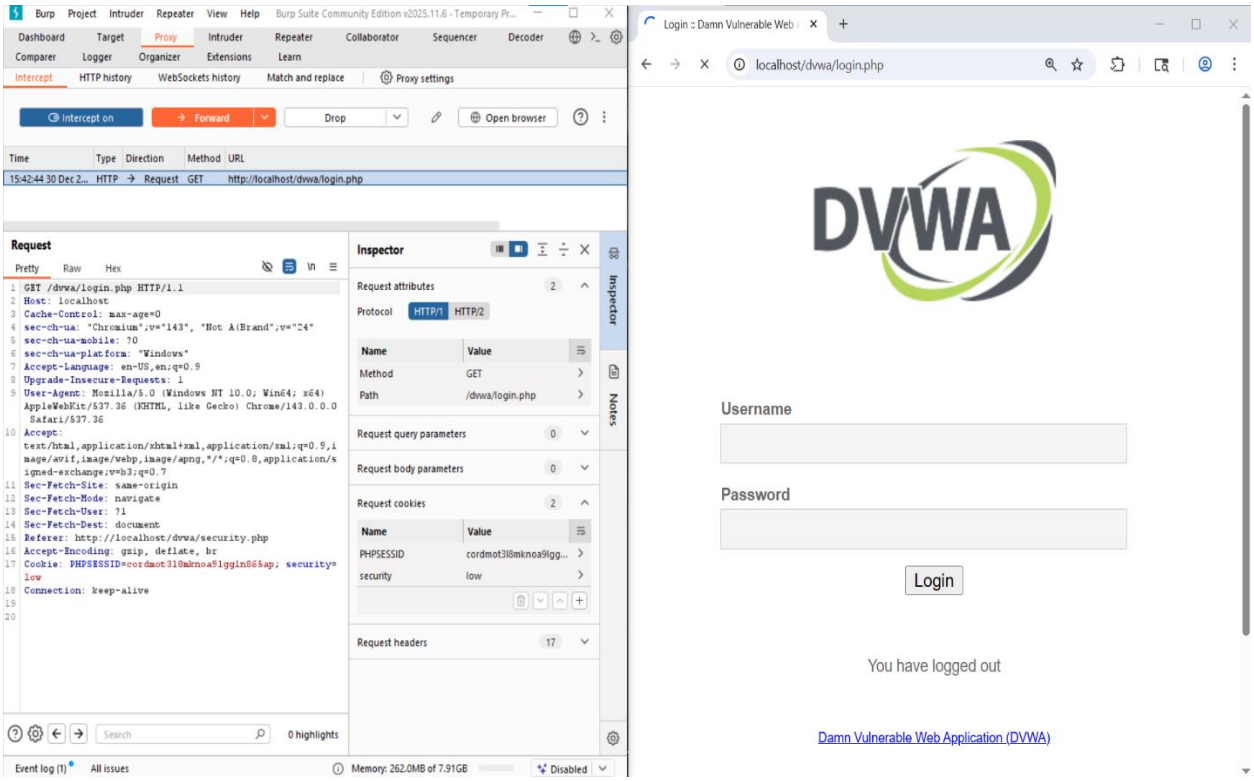


Figure 1: Before Login (low security) PHPSESSID – same to After Login (low security)

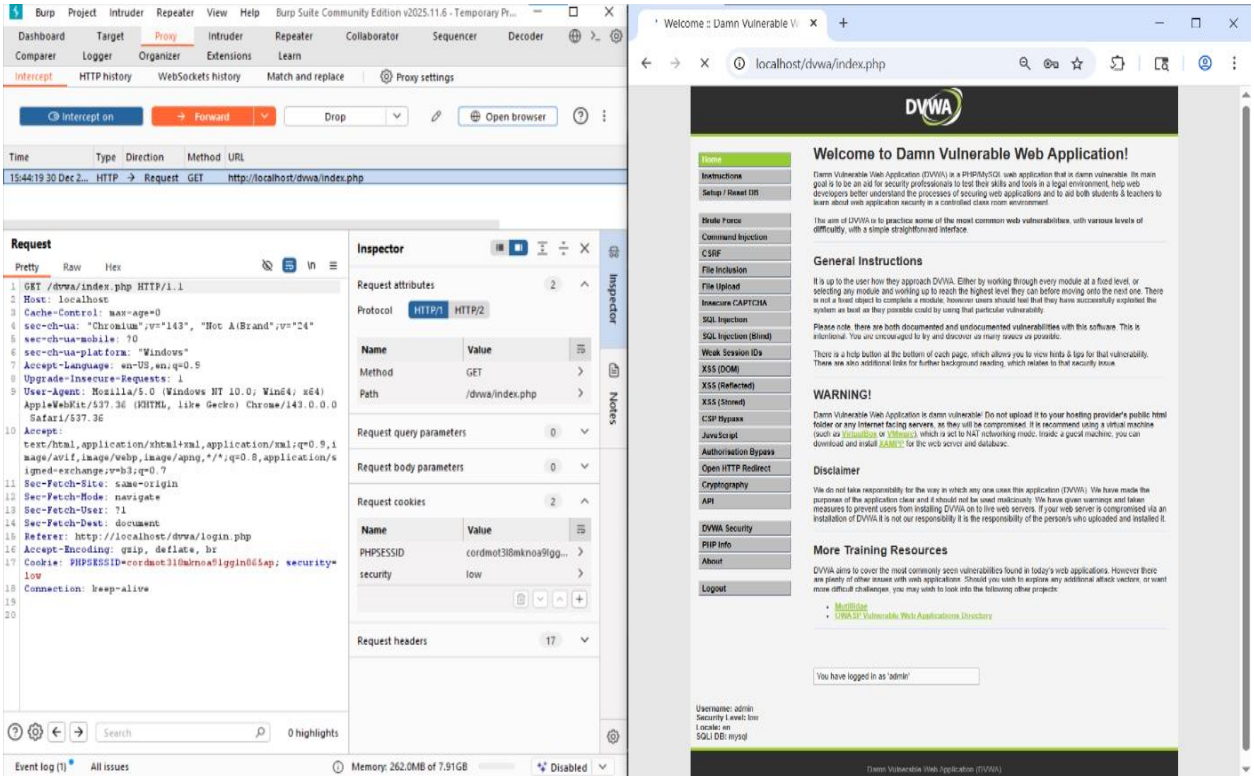


Figure 2: After Login (low security) PHPSESSID - same to Before Login (low security)

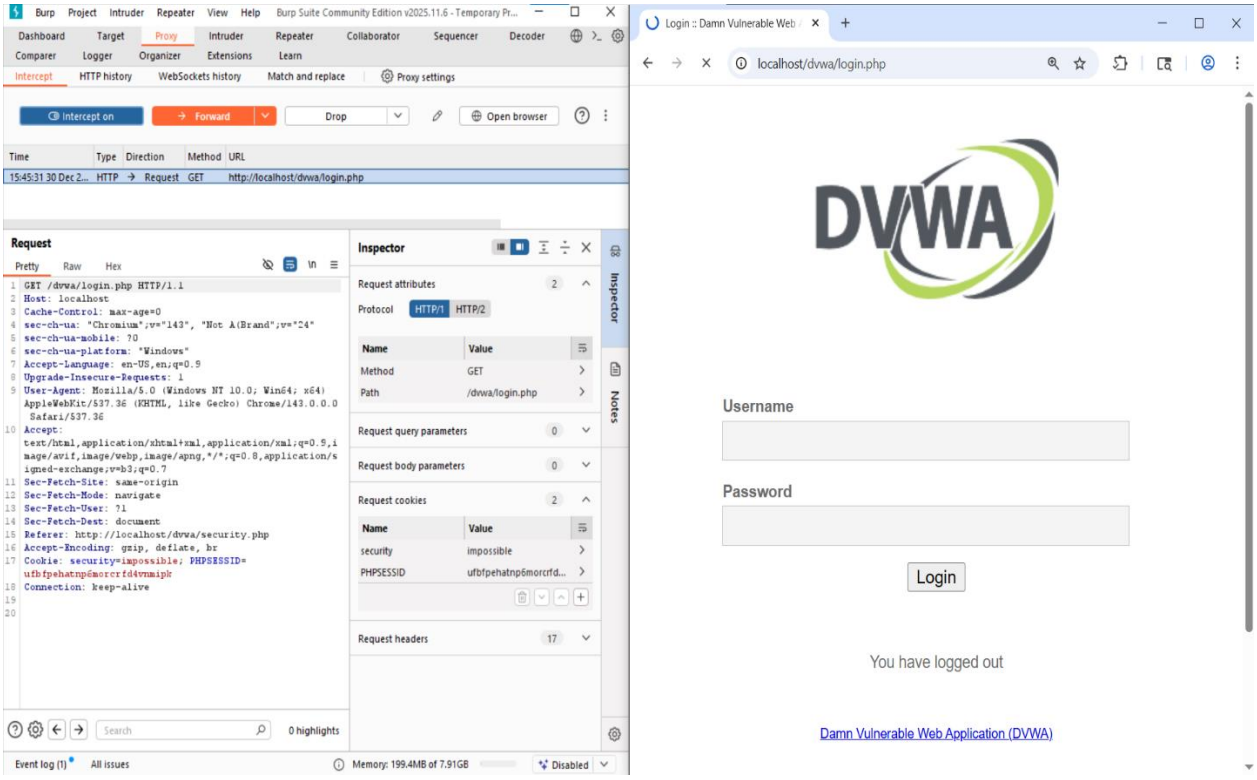


Figure 3: Before Login (impossible security) PHPSESSID - differ from After Login (impossible security)

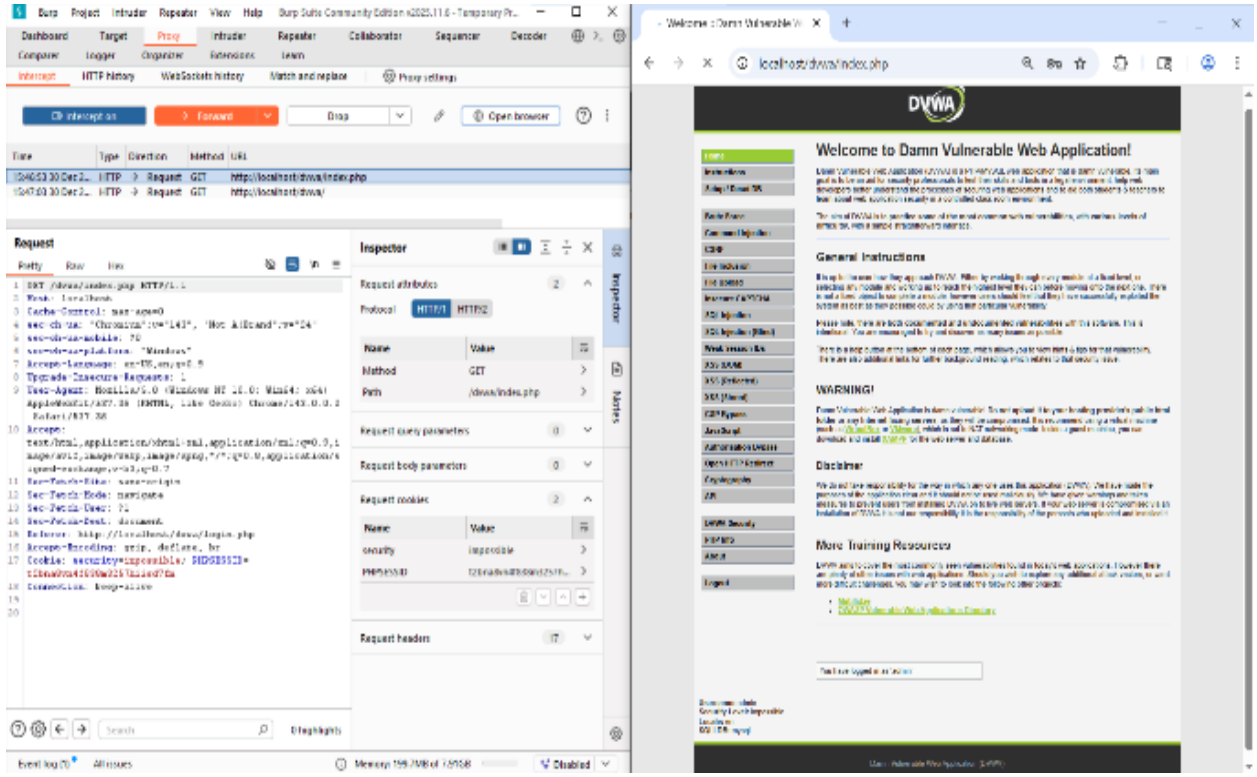


Figure 4: After Login (impossible security) PHPSESSID - differ from Before Login (impossible security)

SESSION USAGE TEST:

Objective: Verify how the **session ID** is used

Steps:

1. Stay **logged in**
2. Navigate across **protected pages**
3. Observe requests in **Burp** or **DevTools**

What I Check:

- Is the same **session ID** sent with every request?
- Does removing the cookie log me out?

Weakness Indicator:

- Access allowed without **session cookie**
- Multiple users sharing same session ID

SESSION TERMINATION TEST:

Objective: Confirm proper **session destruction**

Steps:

1. **Log out** normally
2. Copy the old **session ID**
3. Paste it back into the browser or **Burp**
4. Try accessing a **protected page**

What I Check:

- Is the **session invalidated server-side**?

Weakness Indicator:

- If the old **session ID** still allows access after logout, it proves that the session was not invalidated server-side, making the application **vulnerable** to **session hijacking**.

What would you do if modifying a cookie causes an application to behave abnormally, such as skipping the login pages?

If I notice abnormal session behavior, such as the application automatically redirecting me to protected pages instead of showing the login page, my first step is to isolate the problem. I open the same application in an Incognito or Private window (Pressing Ctrl + Shift + N) because Incognito does not store cookies or sessions. If the login page appears normally there, it confirms that the issue is caused by stored cookies or an old session in my normal browser. After that, I go back to my normal browser and clear the site data and cookies for that domain, which removes the existing session information. This forces the application to create a fresh unauthenticated session, and the application starts behaving normally again.

TESTING PARAMETER MANIPULATION IN THE APPLICATION BY OBSERVING HOW USER-CONTROLLED PARAMETERS WERE PASSED THROUGH "HIDDEN FIELDS", "GET", AND "POST" REQUESTS. I MODIFIED SENSITIVE VALUES SUCH AS PRICE, ROLE, DISCOUNT, QUANTITY, AND ACCESS LEVEL TO CHECK WHETHER THE SERVER PROPERLY VALIDATED THEM. I ANALYZED WHETHER CLIENT-SIDE CONTROLS WERE ENFORCED ONLY IN THE FRONTEND OR ALSO VERIFIED ON THE SERVER. FINALLY, I TESTED IF THE APPLICATION TRUSTED CLIENT-SIDE VALUES, WHICH COULD ALLOW AN ATTACKER TO BYPASS BUSINESS LOGIC, GAIN UNAUTHORIZED ACCESS, OR MANIPULATE TRANSACTIONS.

🛠️ PARAMETER MANIPULATION TESTING

HOW I DEFINE IT TECHNICALLY:

I understand **Parameter Manipulation** as the weakness that occurs when a web application trusts values sent from the client (browser) without proper server-side validation. These values are usually passed through URL parameters (GET), form data (POST), cookies, or hidden fields. Since the client is fully under user control, an attacker can **modify** these parameters to change the application's behavior in unintended ways.

Parameter Manipulation is built on four main ideas:

- **Parameter Exposure:** Application sends important values (price, role, ID) to the client.
- **Client-Side Control:** Values are controlled or restricted only by frontend logic.
- **Parameter Modification:** Attacker alters parameters using browser tools or intercepting proxies.
- **Server Trust:** Server accepts modified values without verification.

If the server trusts these manipulated values, attackers can gain unauthorized access or benefits.

🎯 WHY PARAMETER MANIPULATION IS DANGEROUS:

I consider **parameter manipulation** critical because it allows attackers to bypass business logic without exploiting complex vulnerabilities. Even when authentication is strong, changing simple parameters can lead to privilege escalation, financial loss, or unauthorized actions. The root cause is trusting user-supplied input instead of enforcing rules on the server side.

🔍 HIDDEN FIELDS & CLIENT-SIDE CONTROLS (WHAT I TESTED):

I inspected hidden form fields and client-side controls that stored sensitive values such as role, price, or user ID. I tested whether modifying these hidden values affected server-side behavior. Hidden fields are not secure because they can be easily viewed and changed by attackers.

⚠️ CHANGING CRITICAL VALUES (PRICE, ROLE, ACCESS LEVEL):

I **tested by changing parameters** like price, discount, quantity, role, and access level to see if the application accepted unauthorized values. If a normal user can change their role to admin or reduce a product price, it clearly shows broken server-side validation.

💉 GET & POST PARAMETER MANIPULATION:

I analyzed both **GET** and **POST** requests using interception tools. I modified parameter values in URLs and request bodies to observe how the server handled unexpected or unauthorized input. Secure applications must validate all parameters regardless of the request method.

🚨 **WHY TRUSTING CLIENT-SIDE VALUES IS A VULNERABILITY:**

I concluded that **trusting client-side values is dangerous** because the client cannot be trusted. Attackers can fully control requests sent to the server. Any security decision based on client-side input alone can be bypassed. Proper protection requires **strict server-side validation**, **authorization checks**, and **business logic enforcement**.

This pattern shows how **Parameter Manipulation** directly leads to **broken access control**, **business logic abuse**, and **privilege escalation** if not handled securely.

Three Phases …

One Completion

Project Successfully Closed