AI RED TEAMER

RESEARCHED BY:
RASHID IQBAL

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

This is Part-2 of Documented Operations on Kali Linux

--(DOCUMENTED OPERATIONS ON KALI)--

WHAT I NEED TO KNOW ABOUT WEB APPS & WHERE THEY CAN BE ATTACKED:

Here, I'm focused on the **fundamentals** of how a website actually works and the attack surface – that is, all the places an attacker might try to break in.

HOW WEBSITES TALK:

The internet works on a simple **"ask & answer"** system, which we call **client-server architecture**.

- **I am the Client (My Web Browser):** I'm the one who asks for things
- **The Website is the Server:** It's the one that gives me the answers (like a shop)

This asking & answering is done using something called **HTTP (Hypertext Transfer Protocol)**. When I browse a website, I'm sending a request, and the server sends back a response.

Practical Example:

1. I click on a link (My Request)
2. My browser sends a GET method requests to the server: "Please GET me the homepage"
3. The server finds the homepage data and sends it back (The Response)
4. If I log in, my browser uses a POST method request: "Please POST this username and password to log me in."

THE PIECES OF A REQUEST:

Every time I send a request to a server, it's made up of several important parts that the server needs to understand.

- **Parameters:** These are the specific pieces of data I send. If I search for "shoes", the word "shoes" is a parameter.
  Practical: I see parameters in the URL like **example.com/search?q=shoes**. The attacker can change this!
- **Cookies:** These are small notes that the server gives my browser to remember me later. They help me stay logged in.
  Practical: If I log in, the server gives me a unique cookie. If an attacker steals my cookie, they might be able to pretend to be me.
- **Headers:** These are like the envelopes of the request. They contain **technical details**, like what browser I'm using or what language I speak.
- **Forms:** This is the **visible stuff** I fill out, like a login box or a comment section.

😈 COMMON WEB VULNERABILITIES:

These are the **most common flaws** that attackers look for. If I can understand these, I can look for them too!

- **XSS (Cross-Site Scripting):**

  This happens when the website lets an attacker put malicious code (usually JavaScript) into a webpage that other users see.

  Practical:

  An attacker puts code into a comment box. When I read the comment, the malicious code runs in my browser and might steal my cookie.

- **SQL Injection (SQLi):**

  SQL is the language used to talk to the website's database (where all the important info is stored). This happens when an attacker puts malicious commands into a form field that tricks the server into running them.

  Practical:

  An attacker types a specific code snippet into the login username field. Instead of checking a password, the database is tricked into showing all user passwords.

- **Command Injection:**

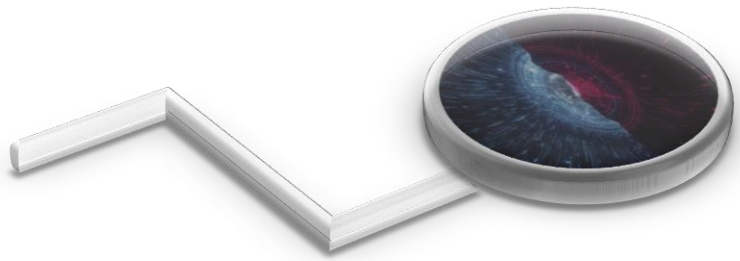
  This is similar to SQLi, but instead of tricking the database, the attacker tricks the server into running system commands (like running a program or deleting a file) on the server itself.

- **File Upload Flaws:**

  If a website lets me upload a picture, and it doesn't check the file type properly, an attacker could upload a malicious program instead of an image.

- **Authentication Flaws:**

  These are problems with how the server handles logging in. Maybe it accepts weak passwords, or it doesn't log me out properly, or it allows me to guess passwords too quickly.



🛠️ BURP SUITE: HOW I DEFINE IT TECHNICALLY:

I think of **Burp Suite** as my main toolkit for checking the security of websites. It's a large, graphical platform made up of lots of smaller tools.

My most crucial tool within Burp is the **Intercepting Proxy**. I set this up to sit directly between my web browser (me) and the website's server. This lets me stop, look at, change, and resend all the traffic (HTTP/S) instantly. I have full control over what goes to the server and what comes back to me.

🎯 WHY I THINK BURP SUITE IS THE BEST:

I consider Burp Suite the best tool in the industry for **web testing** because it combines many features I need into one place, making my job much smoother:

- **Easy to Use:** I can move a web request I find straight into other Burp tools, like the one for repeating tests or the one for big automated attacks. This means I never waste time copying and pasting.
- **I'm in Charge:** I need to use my brain to find tricky problems. Burp lets me manually change every tiny detail of the request – like forms and cookies – so I can fully control how I test the website.
- **Paid Help:** If I pay for the Pro version, I get a smart scanner. It automatically looks at the whole website for me and finds hundreds of common flaws (like XSS), saving me lots of effort at the start.

💰 DO I GET IT FOR FREE ON KALI LINUX?

Yes, I get Burp Suite for free on Kali Linux, but only the core tools.

- The **Burp Suite Community Edition** is usually already installed on my Kali system.

  This free version gives me the essential manual tools:

  - **Proxy** : I use it to stop, looks at, and edit web traffic
  - **Repeater** : I use it to manually re-send a single request many times
  - **Decoder** : I use it to read and change data formats
  - **Comparer** : I use it to see the difference between two things

  This is everything I need to learn manual testing.

- The **Burp Suite Professional Edition** is the paid version.

  If I want to Automate Scanner, the full power of Intruder, and other advanced features, I have to buy this one.

💻 HOW I INSTALL IT ON KALI LINUX:

Since, the **Community Edition** is so fundamental, it's usually already there when I start up Kali. I can just search for it in my application menu.

If I ever need to install or update it using the command line, I just use these standard commands:

- **sudo apt update**
- **sudo apt install burpsuite**

VS HOW BURP SUITE AND NMAP ARE DIFFERENT:

I use both **Burp Suite** & **Nmap**, but for completely different purposes. They check different layers of the web:

- I use **Burp Suite** to look at the **Application Layer (Layer 7)**.
  My focus is on the website's actual code and logic – the HTTP requests, forms, cookies, and parameters. I am looking for flaws like SQL Injection or XSS inside the application. I think of this as being a locksmith checking the inner mechanism of a safe.
- I use **Nmap** to look at the **Network Layer (Layers 3 & 4)**.
  My focus is one the server's infrastructure – what ports are open (like port 80 or 443), what operating system the server is running, and what services are active. I think of this as being a cartographer mapping the walls and doorways of a building.

PRACTICALS ON BURP SUITE COMMUNITY EDITION:

Source of Study: Burp Suite Online Documentation

THE PROXY TOOL: MY TECHNICAL MIDDLEMAN:

The Proxy is the **heart** of Burp Suite and is technically on Intercepting HTTP/S Proxy Server.

In easy words, I think of the Proxy as my technical middleman or a **traffic cop**.

- It's a server that runs on my own computer.
- I force my web browser to send all its traffic (HTTP/S) through this server first.
- Because it's intercepting, it lets me stop and hold every single message (request) my browser sends and every answer (response) the website sends back.
- I use this power to study, edit, and forward the data before it ever reaches its destination. It gives me total control over the communication between and the target server.

How I Stop & Look at Web Traffic:

Step 1: I Get Ready

- I go to the **Proxy > Intercept** tab in Burp.
- I turn the intercept switch to **Intercept On**. This is like putting up a stop sign.
- I click Open Browser. This launches a special web browser that is already set up to work with Burp.

Step 2: I Catch a Request

- In the special browser, I try to go to a website like **https://portswigger.net**
- I notice the **page doesn't load**. That's because Burp has caught the request!
- I look back at the **Proxy > Intercept** tab in Burp and see the request frozen there. This is where I can look closely at it or change it before it leaves my computer.

Step 3: I Send it on

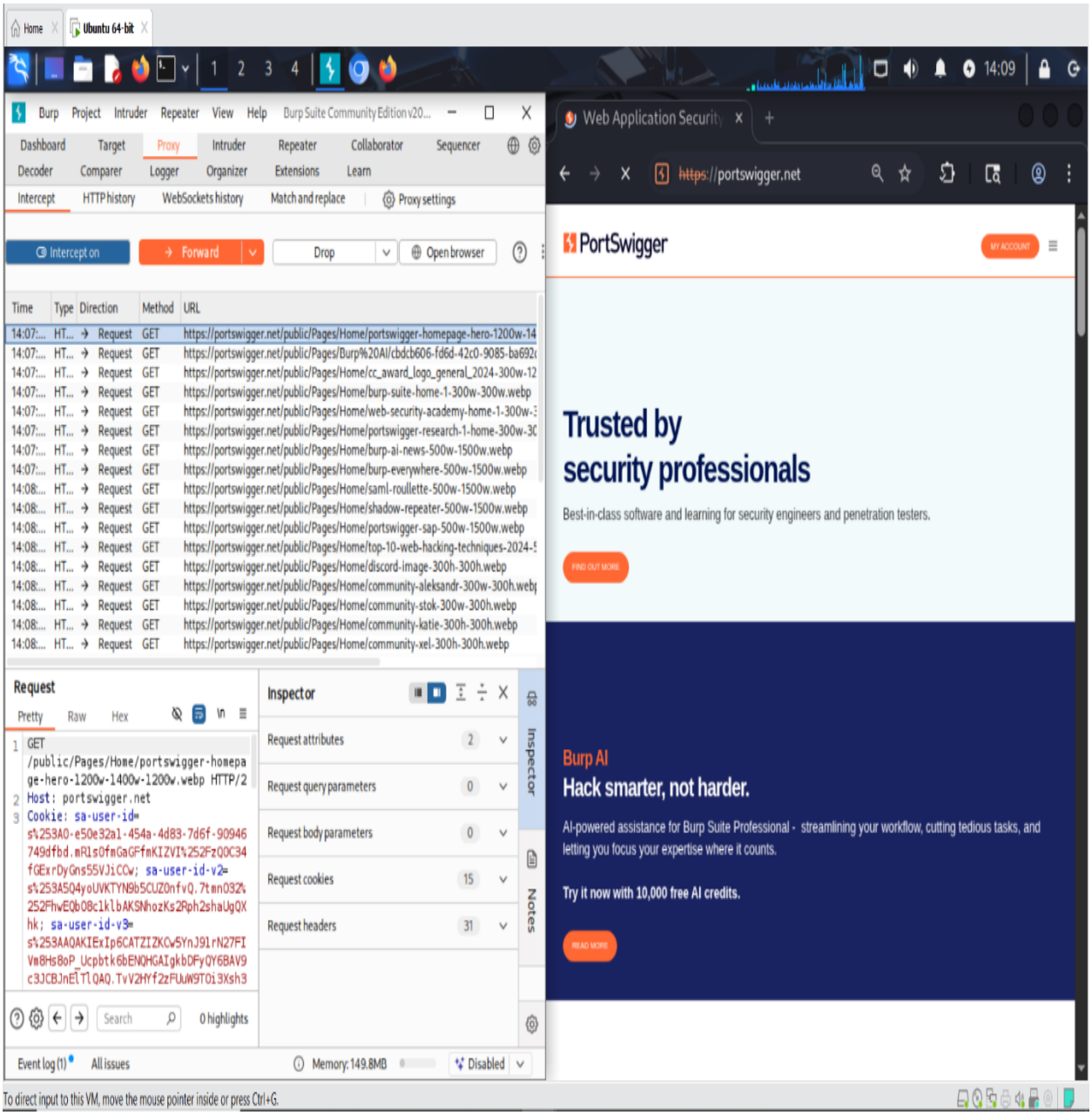
- I click the **Forward** button. This lets the frozen request finally go to the website's server.
- I click **Forward** again and again if necessary until the whole website page loads in my browser.

Step 4: I Turn Off the Stop Sign

- Since browsers send so many **small requests**, I usually don't want to stop every single one.
- I go back to Burp and set the intercept switch to **Intercept off**.
- Now, I can browse the site normally, and Burp doesn't stop anything.

Step 5: I Review My History

- Even when the intercept is off, Burp is **still secretly watching**.
- I go to the **Proxy > HTTP history** tab.
- Here, I can see a record of every single piece of traffic that passed through Burp. I can click on any entry to see the exact message I sent and the exact answer the server gave. This is often the easiest way for me to review my actions after I've finished browsing.



I Modify HTTP Requests with Burp Proxy:

I'm going to show you how I use **Burp Proxy** to change a request before it reaches the server. This is a key skill for identifying how a website handles unexpected data.

Note: To follow this exact process, I would need an account on **PortSwigger's** Web Security Academy to access their training lab.

Step 1: I Prepare the Browser

- In Burp, I go to the **Proxy > Intercept** tab and make sure the **interception is off**.
- I launch Burp's browser and go to the specific training lab URL.
  https://0a94005b047d8dfd8112214f006d006a.web-security-academy.net/
  Lab: Excessive trust in client-side controls
- When the lab loads, I click Access the Lab.

Step 2: I Log In

- On the fake shopping site, I click **My Account**.
- I log in with the provided credentials…
  Username: wiener
  Password: peter
- I notice that my account has only $100 of store credit.

Step 3: I Find the Item

- I click **Home** to return to the main page.
- I select the product page for the **Lightweight "l33t" leather jacket**.

Step 4: I Intercept the "Add the Cart" Request

- I switch back to **Burp**, go to **Proxy > Intercept**, and turn **interception on**.
- In the browser, I click the button to add the leather jacket to my cart.
- Burp immediately stops the resulting request. This is usually the **POST /cart request**.
- I look closely at the intercepted request. I see a parameter in the body of the request called **price**, and its value matches the item's cost in cents.

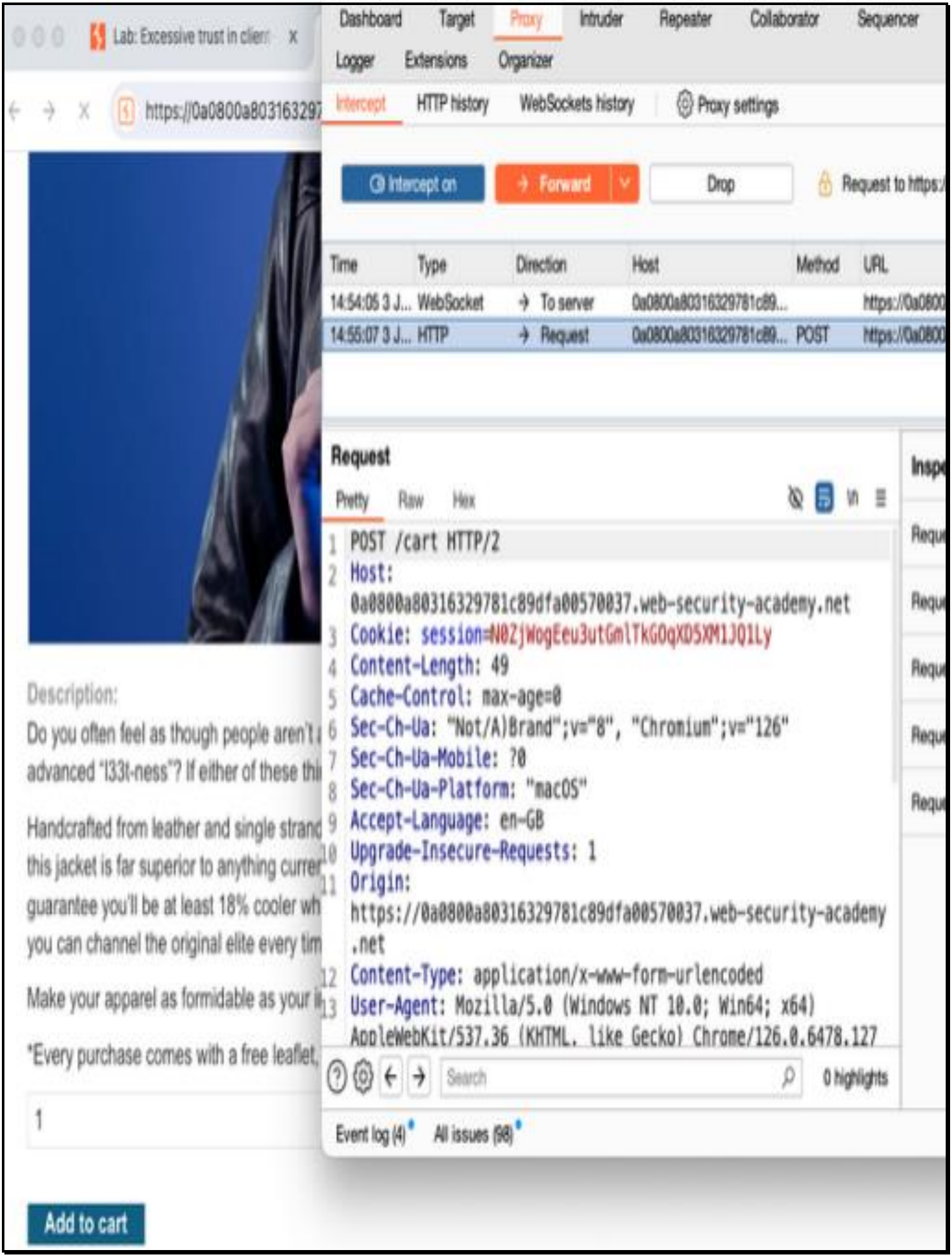
Step 5: I Modify the Request

- I change the value of the price parameter. Instead of the high cost, I change it to **1** (for one cent).
- I click **Forward** (or Forward all) to send my modified request to the server.
- After the request is sent, I switch the **interception off** again so my browsing isn't stopped more.

Step 6: I Exploit the Vulnerability

- In Burp's browser, I click the **basket icon** to view my cart.
- I see that the jacket has been added for only one cent! (This change was only possible because I **modified the request** using Burp Proxy)
- I click **Place order** to buy the jacket for a ridiculously low price.

I've successfully **intercepted**, **reviewed**, and **manipulated an HTTP request!** This shows how important it is for websites to **never trust data sent by the client**.


Lab: Excessive trust in clien
https://0a0800a803163297
Dashboard
Target
Proxy
Intruder
Repeater
Collaborator
Sequencer
Logger
Extensions
Organizer
Intercept
HTTP history
WebSockets history
Proxy settings
Intercept on
Forward
Drop
Request to https:/
Time
Type
Direction
Host
Method
URL
14:54:05 3 J...
WebSocket
To server
0a0800a80316329781c89...
https://0a0800
14:55:07 3 J...
HTTP
Request
0a0800a80316329781c89...
POST
https://0a0800
Request
Pretty
Raw
Hex
1 POST /cart HTTP/2
2 Host:
0a0800a80316329781c89dfa00570037.web-security-academy.net
3 Cookie: session=N0ZjWogEeu3utGmlTkGOqXD5XM1JQ1Ly
4 Content-Length: 49
5 Cache-Control: max-age=0
6 Sec-Ch-Ua: "Not/A)Brand";v="8", "Chromium";v="126"
7 Sec-Ch-Ua-Mobile: ?0
8 Sec-Ch-Ua-Platform: "macOS"
9 Accept-Language: en-GB
10 Upgrade-Insecure-Requests: 1
11 Origin:
https://0a0800a80316329781c89dfa00570037.web-security-academy
.net
12 Content-Type: application/x-www-form-urlencoded
13 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.6478.127
Search
0 highlights
Event log (4)
All issues (98)
Description:
Do you often feel as though people aren't
advanced "l33t-ness"? If either of these thi
Handcrafted from leather and single strand
this jacket is far superior to anything currer
guarantee you'll be at least 18% cooler wh
you can channel the original elite every tim
Make your apparel as formidable as your
*Every purchase comes with a free leaflet,
1
Add to cart

I Set the Target Scope in Burp Suite:

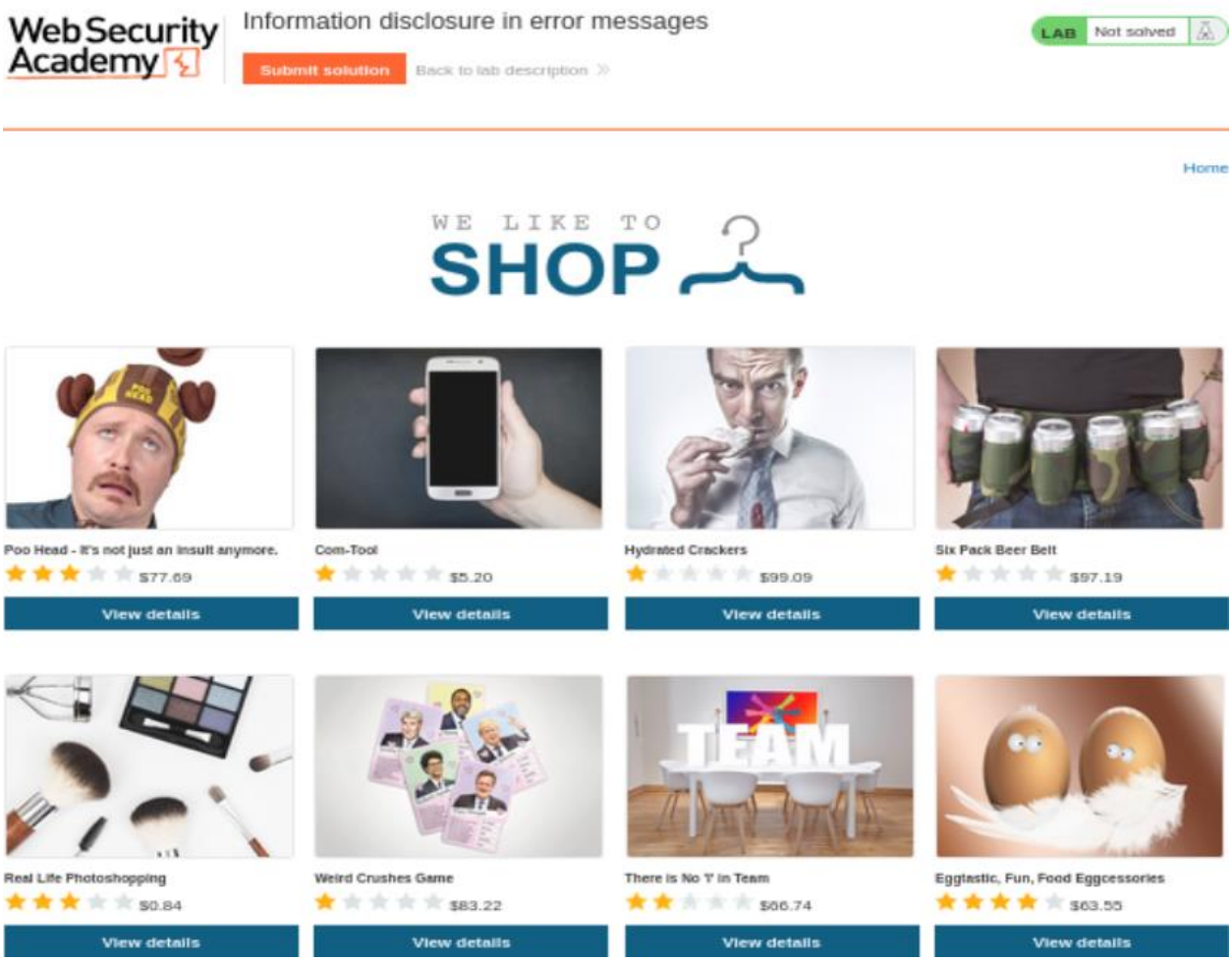
I use the **Target Scope** feature in Burp Suite to tell it exactly which websites I want to focus on. This is important because it filters out the traffic noise from things like Google Analytics or other sites, letting me concentrate only on my target.

Step 1: I Launch the Browser and Access the Site

- I launch Burp's browser and visit the specific URL for the lab.
  https://portswigger.net/web-security/information-disclosure/exploiting/lab-infoleak-in-error-messages
  Lab: Information disclosure in error messages
- After the page loads, I click **Access the Lab** and log in if prompted.
- I see my own instance of the fake shopping website.

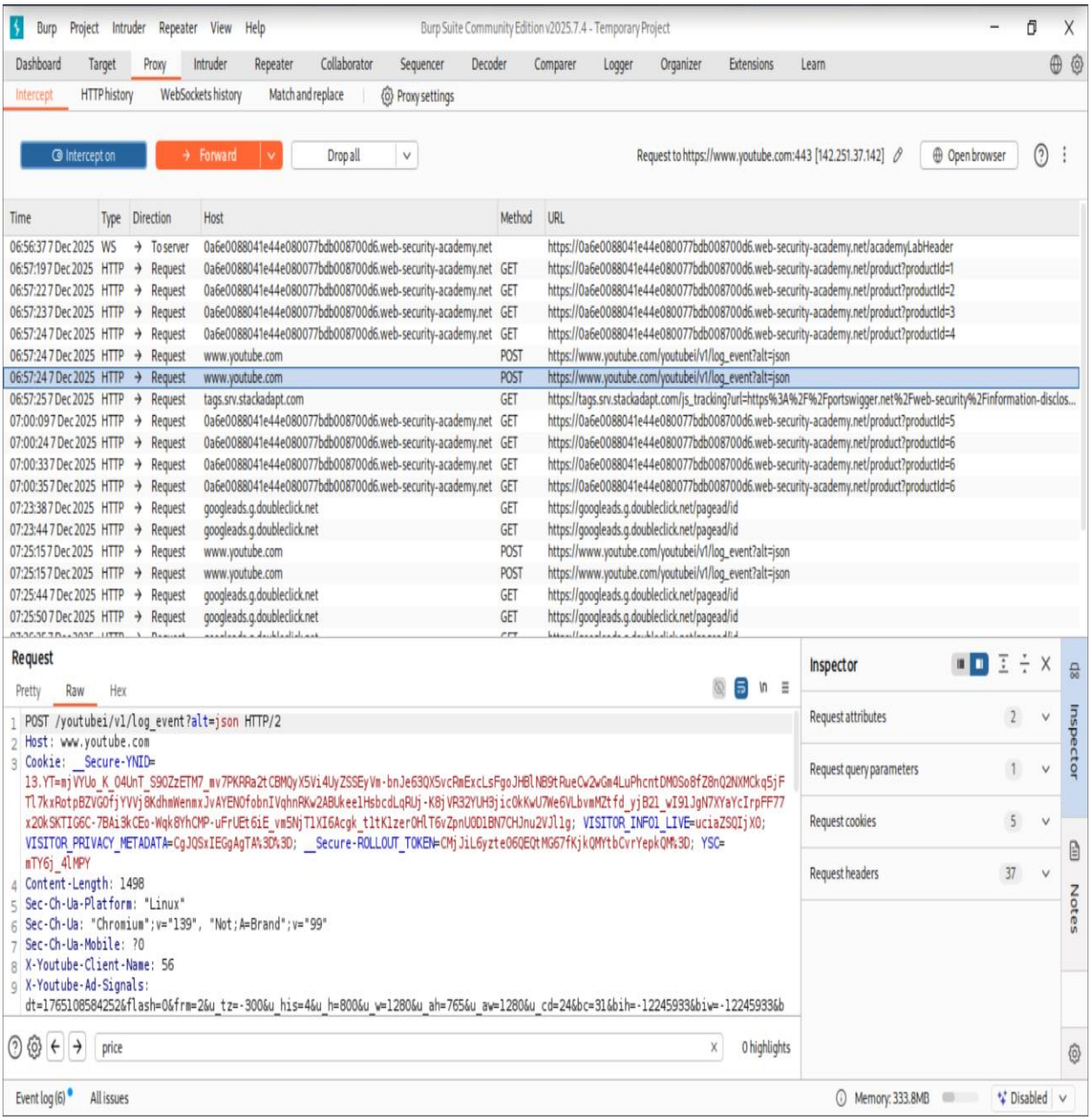
Step 2: I Browse the Target Site

- In the browser, I click on a few pages, like product pages, to generate some traffic.



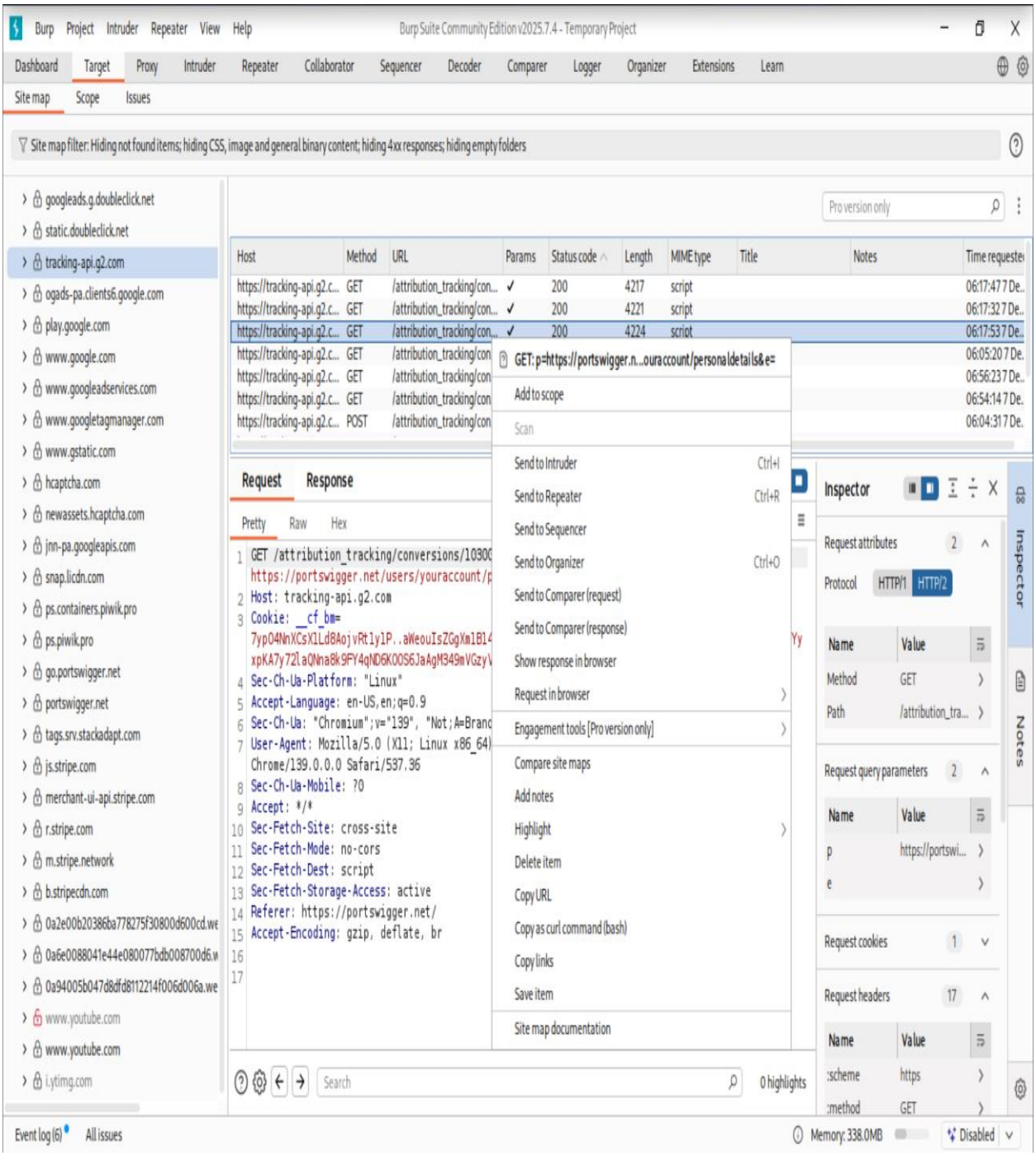
Step 3: I Study the HTTP History

- I go to the **Proxy > HTTP history** tab in Burp.
- To make the history easier to read, I click the header of the leftmost column (#) to sort the requests with the most recent ones at the top.
- I notice that the history contains traffic not only from my target site but also from **third-party sites** like YouTube or Google Analytics, which I don't care about.



Step 4: I Set My Target Scope

- I go to the **Target > Site map** tab.
- In the panel on the left, I see a list of all the hosts my browser has communicated with.
- I right-click on the specific host (the one for my target site) and select **Add to Scope**.

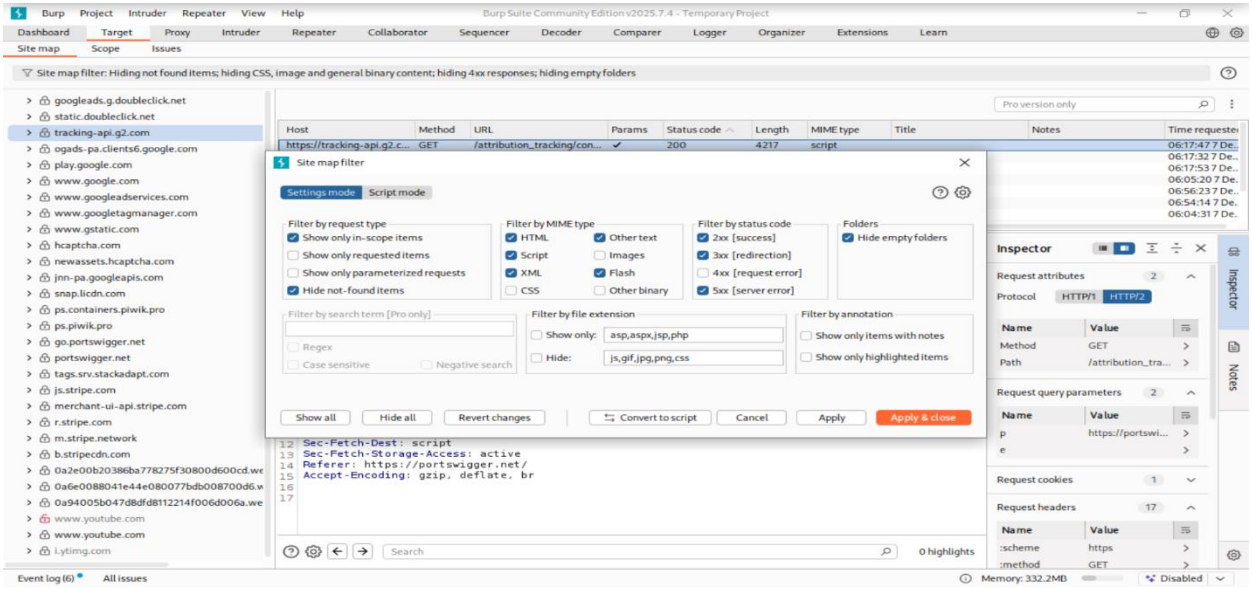


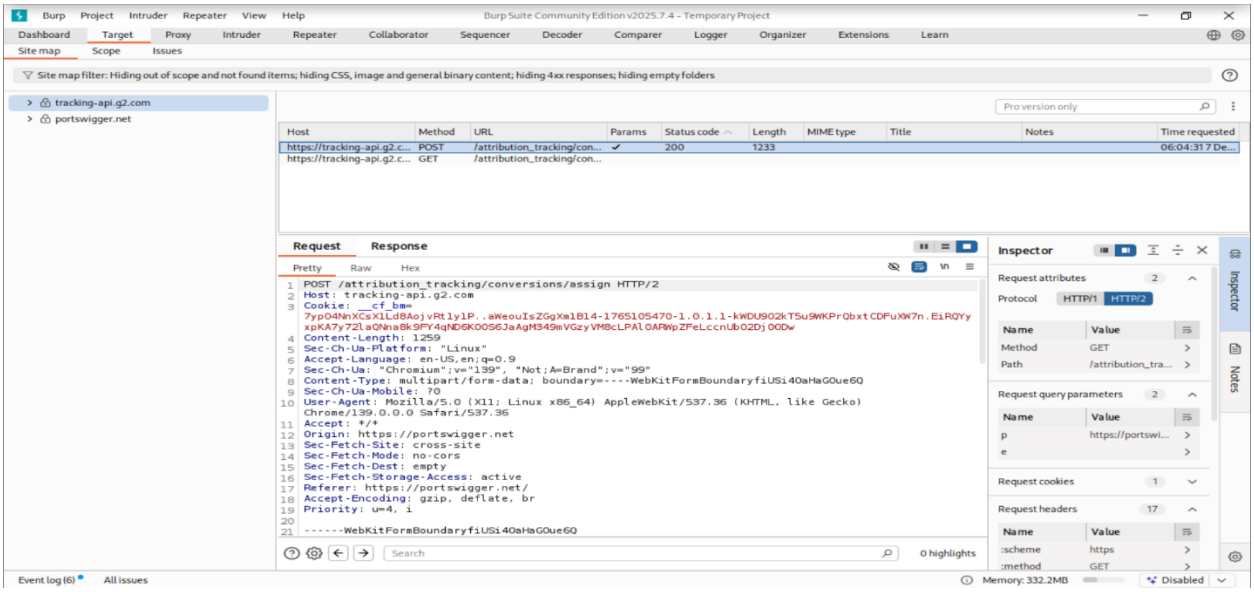
Step 5: I Filter the HTTP History

- Now, I go back to the **Proxy > HTTP history** tab.
- I click on the **Display filter** located above the history log.
- I select the option **Show only in-scope items**.
- I scroll through my history again, and now it only shows requests and responses from my target website! **All the unnecessary traffic is hidden**.

By setting the scope, I've simplified my work and can focus entirely on the traffic I'm interested in.

Now, any new traffic I generate on the target site will also only show up in the history, keeping things tidy.





SETTING UP DVWA / OWASP JUICE SHOP, CAPTURING A LOGIN REQUEST USING BURP SUITE, AND ANALYZING EVERY COMPONENT – METHOD, URL, PARAMETERS, HEADERS, AND COOKIES – TO UNDERSTAND THEIR FUNCTION AND CRITICAL ROLE IN SECURITY TESTING FOR POTENTIAL VULNERABILITIES

💻 SETTING UP MY LAB & TOOL:

First, I set up and **opened DVWA (Damn Vulnerable Web App)** in my local lab environment. After confirming it was running, I configured my browser to use **Burp Suite** as a **Proxy**. I ensured Burp Suite's **'Intercept is on'** feature was active, which allowed me to **capture network traffic** between my browser and DVWA.

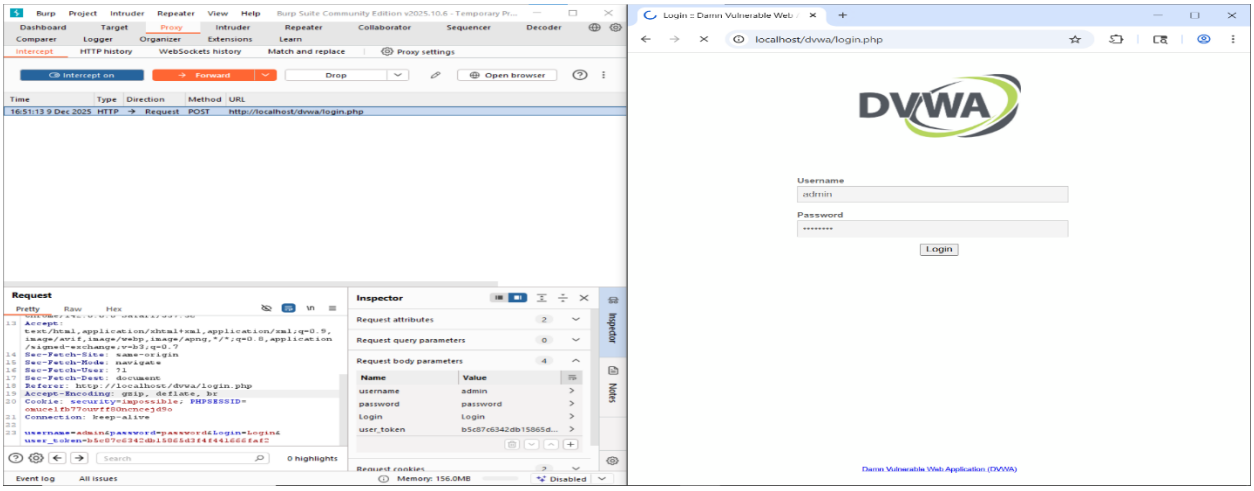
🔒 CAPTURING & ANALYZING THE LOGIN REQUEST:

Next, I navigated to the **DVWA login page**. To capture a login request, I entered arbitrary credentials (like 'admin' and 'password') and clicked the 'Login' button. Burp Suite intercepted the outgoing HTTP request, which I analyzed.

Here is my analysis of the key parts of the intercepted HTTP request:

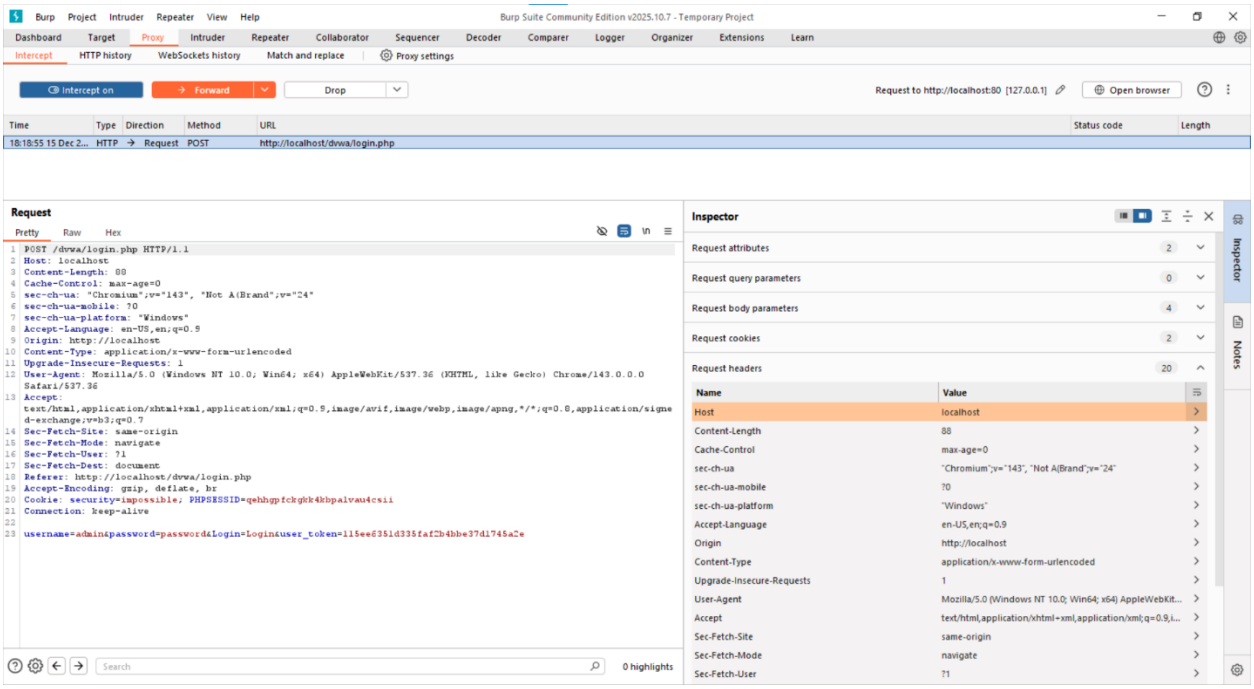
REQUEST LINE (METHOD & URL):

PART	DESCRIPTION	IMPORTANCE IN TESTING
METHOD	The HTTP method used, which was typically **POST** for a login form	This tells the server I want to send data to be processed. I'll test to see if other methods (like **GET**) are incorrectly allowed, which can expose data
URL	The Uniform Resource Locator, which specifies the resource on the server being requested, like **/vulnerabilities/login.php**	This defines the target script or page. I use this to look for **insecure direct object references** or test if sensitive pages can be accessed without authentication



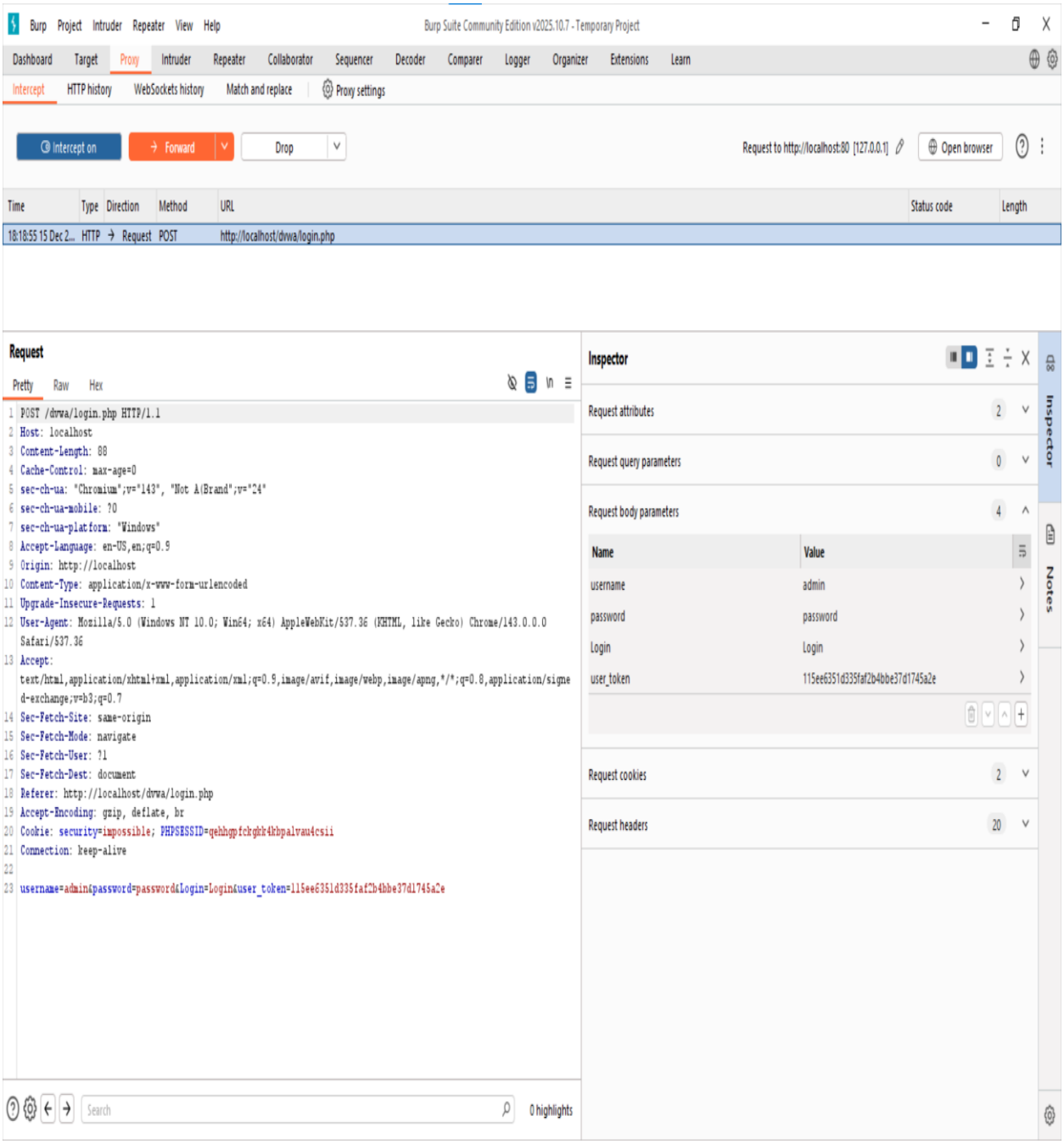
REQUEST HEADERS:

PART	DESCRIPTION	IMPORTANCE IN TESTING
HOST	Specifies the **domain name** of the server	It helps the server know which website the request is for, which is crucial in environments hosting multiple sites (virtual hosts)
USER AGENT	Identifies the **client software** (my browser & version) making the request	I can manipulate this to test for poor access controls based on the agent or to bypass simple bot detection
CONTENT TYPE	Indicated the **format** of the data in the body (e.g., **application/x-www-form-urlencoded**)	I check this to ensure the server correctly processes different data formats, which is key for testing vulnerabilities like **XML External Entity (XXE) injections** if other types are accepted
CONTENT LENGTH	Indicates the **size** (in bytes) of the request body	A basic integrity check. In some advanced attacks, I might modify this to confuse and bypass security filters
REFERER	Indicates the **previous page** I was on before clicking 'Login'	I check this to see if the server relies on it for security (a bad practice) and test for **Cross-Site Request Forgery (CSRF) vulnerabilities** that use a broken **Referer** check



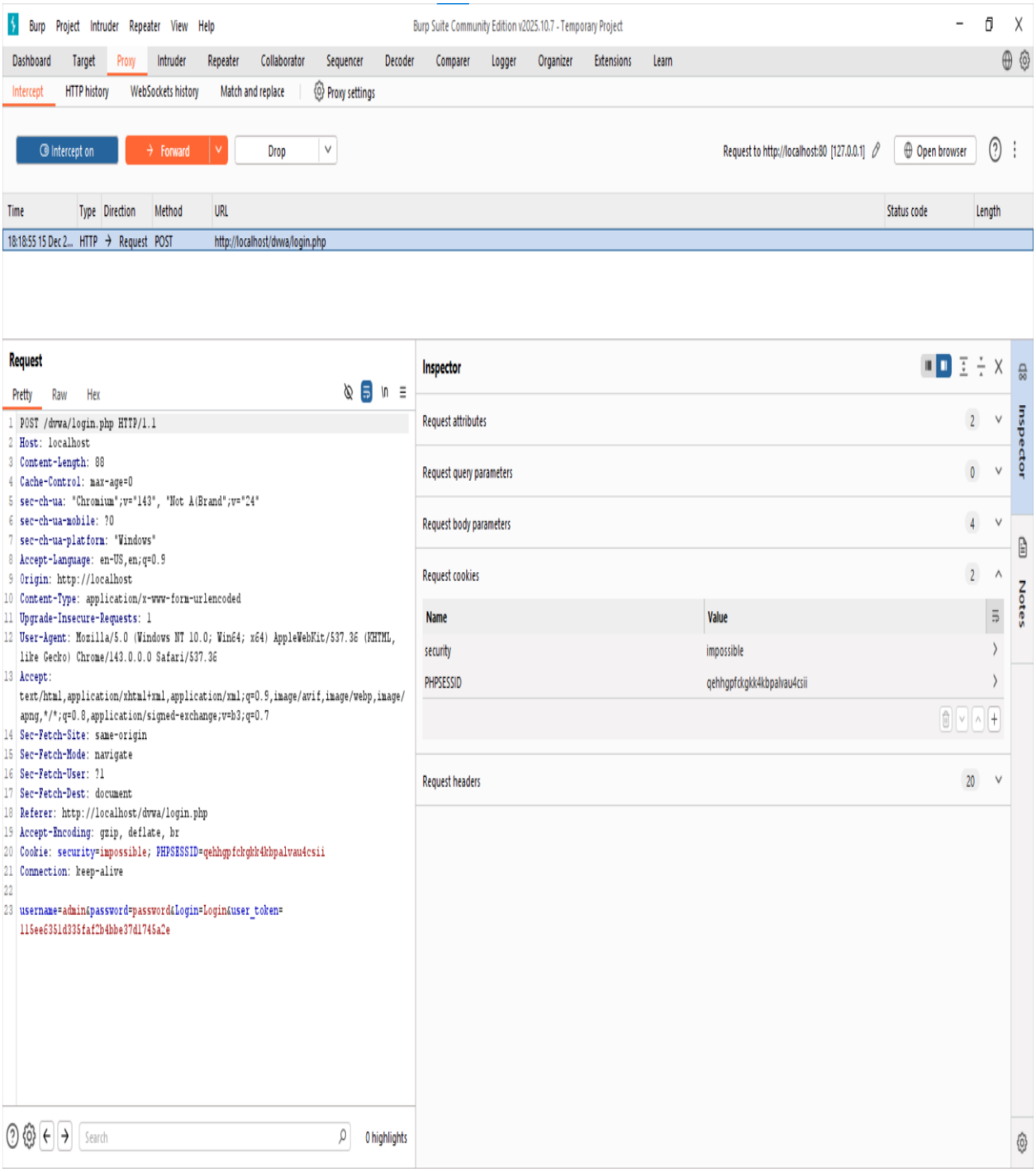
REQUEST BODY (PARAMETERS):

PART	DESCRIPTION	IMPORTANCE IN TESTING
PARAMETERS	The data being sent to the server, typically username & password	**This is the most critical part for web application testing.** I change the values here to test for almost all injection flaws, including **SQL Injection** and **Cross-Site Scripting (XSS)**



COOKIES:

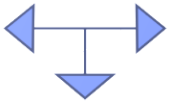
PART	DESCRIPTION	IMPORTANCE IN TESTING
COOKIES	Small pieces of data sent by the server and stored by the browser, often including a Session ID (e.g., PHPSESSID in DVWA)	The Session ID is how the server tracks my login state. I check if the ID is predictable or if another user's cookie can be used to hijack their session



EXPLAINING CROSS-SITE SCRIPTING (XSS) FROM MY PERSPECTIVE. I DESCRIBED HOW I DIFFERENTIATE BETWEEN REFLECTED AND STORED XSS, HOW I USE INPUT VALIDATION AND OUTPUT ENCODING TO PREVENT XSS, AND HOW I WOULD MANUALLY TEST FOR XSS AND VERIFY IT USING BURP SUITE

😈 CROSS-SITE SCRIPTING (XSS):

Cross-Site Scripting (XSS) is a web vulnerability where I am able to inject malicious JavaScript into a website, and that code executes in the browser of another user instead of on the server. This allows me to steal cookies, session tokens, or perform actions on behalf of the victim without their knowledge.



◇ DIFFERENCE BETWEEN REFLECTED XSS & STORED XSS:

➤ REFLECTED XSS:

Reflected XSS occurs when my injected script is not stored on the server but is immediately reflected back in the website's response.

Explanation:

I inject malicious JavaScript through a URL parameter, search box, or form input. When the victim clicks the crafted link or submits the request, the server reflects my input in the response page. Since the website does not properly sanitize the input, the browser executes my script instantly.

Key Point:

The attack works only when the victim interacts with the malicious link or request.

➤ STORED XSS:

Stored XSS occurs when my malicious script is saved permanently in the website's database.

Explanation:

I inject JavaScript into a comment section, review field, or user profile. The server stores my input in the database without proper filtering. Every time any user opens that page, my script is loaded and executed in their browser automatically.

Key Point:

Stored XSS is more dangerous because it affects every user who visits the infected page.

◇ HOW INPUT VALIDATION AND OUTPUT ENCODING PREVENT XSS:

➤ INPUT VALIDATION:

Input validation is the process where the server checks and restricts what kind of data I am allowed to submit.

Explanation:

If the website validates input properly, it blocks dangerous characters, tags, or scripts such as **<script>**, **onerror**, or **JavaScript keywords**. This prevents me from injecting malicious payloads at the entry point.

Limit:

Input validation alone is not always enough, because attackers can sometimes bypass filters.

➤ OUTPUT ENCODING:

Output encoding ensures that user input is displayed as plain text rather than executable code.

Explanation:

Even if I manage to submit JavaScript, the website converts special characters into safe formats. When the browser receives the response, it treats my input as text instead of code, so my script never executes.

Key Advantage:

Output encoding protects the user even if malicious input reaches the output.

◇ MANUAL TESTING FOR XSS AND VERIFICATION USING BURP SUITE:

➤ MANUAL TESTING FOR XSS:

I manually test all input fields such as **search boxes**, **login forms**, **comment sections**, and **URL parameters**.

Explanation:

I inject basic XSS payloads like **<script>alert(1)</script>** or simple event-based payloads. If the browser shows an alert or executes JavaScript, I confirm the presence of XSS.

➤ VERIFYING XSS IN BURP SUITE:

Burp Suite helps me analyze and manipulate HTTP requests and responses.

Explanation:

I intercept requests using Burp Proxy and inject my payload directly into parameters. Then I analyze the server's response. If my payload appears unencoded or is stored in the response body, I confirm the vulnerability. I can also replay requests to check whether the XSS is reflected or stored.

IDENTIFYING & ANALYZING XSS VULNERABILITIES IN DVWA / OWASP JUICE SHOP FROM MY PERSPECTIVE. I IDENTIFIED AT LEAST ONE REFLECTED AND ONE STORED XSS VULNERABILITY, DESCRIBED WHERE EACH WAS FOUND, USED APPROPRIATE PAYLOADS, VERIFIED SUCCESSFUL EXECUTION THROUGH ALERT POP-UPS, AND EXPLAINED THE SECURITY IMPACT ALONG WITH EFFECTIVE PREVENTION TECHNIQUES

😈 IDENTIFICATION OF VULNERABILITY: REFLECTED XSS (LOW LEVEL)

I identified the vulnerability as **Reflected Cross-Site Scripting (XSS)**.

I entered special characters such as **<>** into the **input field**, and I observed that my input was immediately reflected back on the webpage without proper sanitization or encoding. Since the payload was not stored in the database and was executed only within the same request-response cycle, this confirmed that the vulnerability was **Reflected XSS** rather than **Stored XSS**.

This immediate reflection of my input in the server's response indicated that the application was processing user input insecurely and returning it directly to the browser.

➤ Common Places Where I Test for Reflected XSS:

I usually test for Reflected XSS in areas where user input is displayed instantly on the same page, such as:

- The **XSS (Reflected) module in DVWA**, specifically the *Name* input field
- Search fields where the entered query is shown immediately in the results
- URL parameters, where input is passed directly through the browser's address bar

In DVWA, I specifically tested the vulnerability in the XSS (Reflected) section by entering payloads such as **<>** and simple JavaScript scripts. When I saw my input reflected instantly in the response without encoding. I confirmed the presence of Reflected XSS.

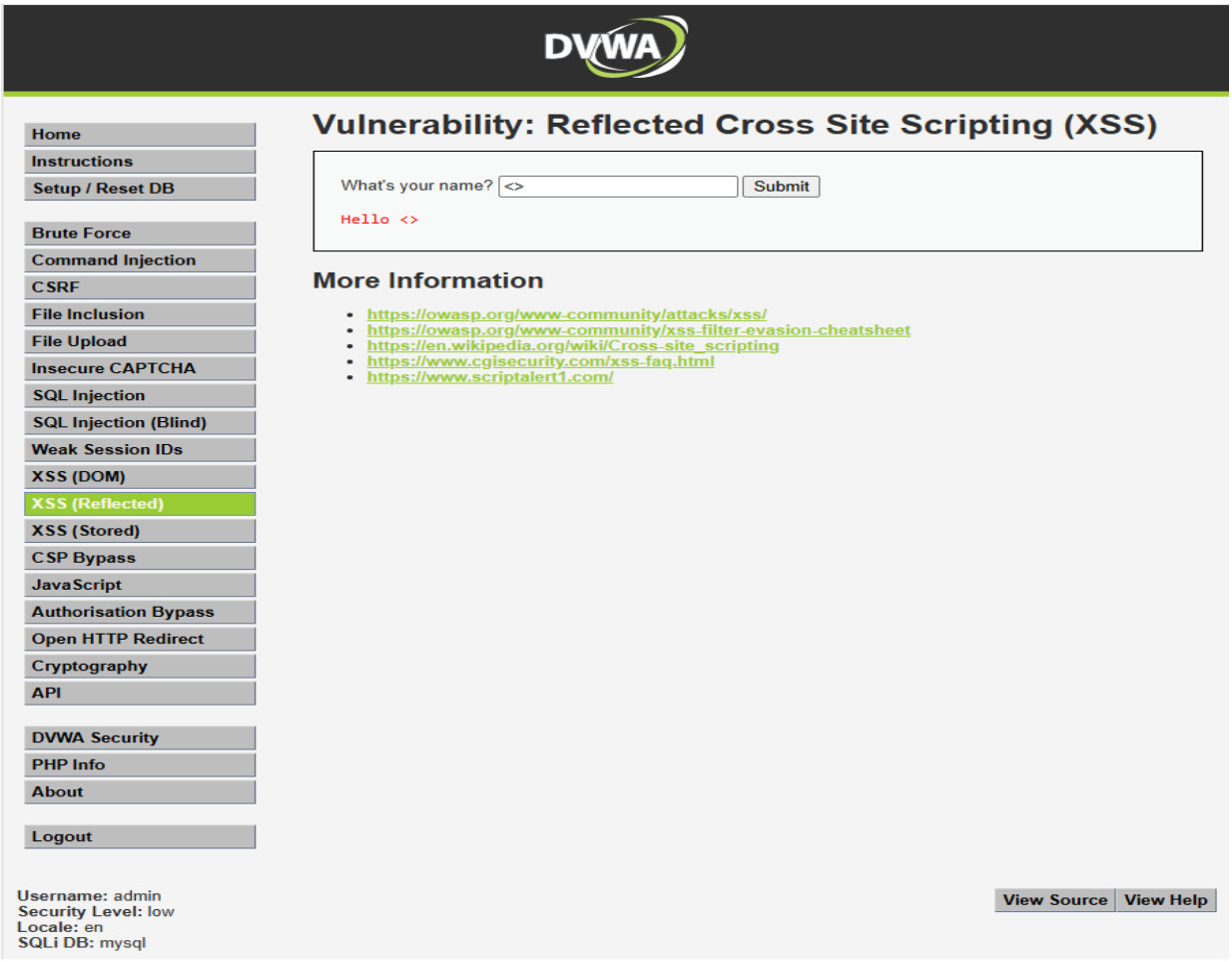
➤ Possible Scenario in Story form:

I was actively hunting for small mistakes in a web application – the kind of mistakes attackers silently wait for. When I injected the payload … **<script>alert(document.cookie)</script>**

And saw it execute instantly, I knew I had found an open door. At first, it was just an alert popup, but to me, that alert was proof of execution and control.

I imagined a real user logging in from a public café, trusting the website. When that user clicked a crafted link I had shared, my injected script would execute in their browser and silently steal their session cookie, I could access their browser without needing a username or password, viewing their private data exactly as they did.

From there, I could modify account settings, post content on their behalf, or spread the same malicious link to other users, allowing the attack to propagate without ever touching the server again. To the victims, everything would appear normal – no errors, no warnings – while I remained invisible in the background.




```
<li class=""><a href="../../vulnerabilities/exec/">Command Injection</a></li>
<li class=""><a href="../../vulnerabilities/csrf/">CSRF</a></li>
<li class=""><a href="../../vulnerabilities/fi/.?page=include.php">File Inclusion</a></li>
<li class=""><a href="../../vulnerabilities/upload/">File Upload</a></li>
<li class=""><a href="../../vulnerabilities/captcha/">Insecure CAPTCHA</a></li>
<li class=""><a href="../../vulnerabilities/sqli/">SQL Injection</a></li>
<li class=""><a href="../../vulnerabilities/sqli_blind/">SQL Injection (Blind)</a></li>
<li class=""><a href="../../vulnerabilities/weak_id/">Weak Session IDs</a></li>
<li class=""><a href="../../vulnerabilities/xss_d/">XSS (DOM)</a></li>
<li class="selected"><a href="../../vulnerabilities/xss_r/">XSS (Reflected)</a></li>
<li class=""><a href="../../vulnerabilities/xss_s/">XSS (Stored)</a></li>
<li class=""><a href="../../vulnerabilities/csp/">CSP Bypass</a></li>
<li class=""><a href="../../vulnerabilities/javascript/">JavaScript</a></li>
<li class=""><a href="../../vulnerabilities/authbypass/">Authorisation Bypass</a></li>
<li class=""><a href="../../vulnerabilities/open_redirect/">Open HTTP Redirect</a></li>
<li class=""><a href="../../vulnerabilities/cryptography/">Cryptography</a></li>
<li class=""><a href="../../vulnerabilities/api/">API</a></li>
</ul><ul class="menuBlocks"><li class=""><a href="../../security.php">DVWA Security</a></li>
<li class=""><a href="../../phpinfo.php">PHP Info</a></li>
<li class=""><a href="../../about.php">About</a></li>
</ul><ul class="menuBlocks"><li class=""><a href="../../logout.php">Logout</a></li>
</ul>
                </div>

            </div>

            <div id="main_body">


<div class="body_padded">
    <h1>Vulnerability: Reflected Cross Site Scripting (XSS)</h1>

    <div class="vulnerable_code_area">
        <form name="XSS" action="#" method="GET">
            <p>
                What's your name?
                <input type="text" name="name">
                <input type="submit" value="Submit">
            </p>

        </form>
        <pre>Hello <></pre>
    </div>

    <h2>More Information</h2>
    <ul>
        <li><a href="https://owasp.org/www-community/attacks/xss/" target="_blank">https://owasp.org/www-community/attacks/xss/</a></li>
        <li><a href="https://owasp.org/www-community/xss-filter-evasion-cheatsheet" target="_blank">https://owasp.org/www-community/xss-filter-evasion-cheatsheet</a></li>
        <li><a href="https://en.wikipedia.org/wiki/Cross-site_scripting" target="_blank">https://en.wikipedia.org/wiki/Cross-site_scripting</a></li>
        <li><a href="https://www.cgisecurity.com/xss-faq.html" target="_blank">https://www.cgisecurity.com/xss-faq.html</a></li>
        <li><a href="https://www.scriptalert1.com/" target="_blank">https://www.scriptalert1.com/</a></li>
    </ul>
</div>

                <br /><br />


            </div>

            <div class="clear">
            </div>
```

😈 IDENTIFICATION OF VULNERABILITY: STORED XSS (LOW LEVEL)

I identified the vulnerability as **Stored Cross-Site Scripting (XSS)**.

While testing the application at the **Low security level**, I first verified whether special JavaScript characters like **> < ‘ “** were allowed in the input fields. I observed that the application did **not sanitize or encode** user input before storing it, which confirmed insecure input handling.

After validating this behavior, I injected the payload:

<script>alert(‘Hello world’)</script>

When I saved the input and revisited the page, the script executed automatically every time the page loaded. This confirmed that the payload was **stored in the backend** and executed whenever the content was accessed. Since the script persisted beyond a single request and affected all users viewing the page, the vulnerability was clearly identified as **Stored XSS**, not **Reflected XSS**.

This demonstrated that the application was trusting user input and rendering it directly in the browser without proper validation or output encoding.

➤ Common Places Where I Test for Stored XSS:

I usually test for Stored XSS in areas where user input is saved and later displayed, such as:

- Comment sections
- Feedback or message forms
- User profile fields (name, bio, description)
- DVWA’s **XSS (Stored)** module, especially the message input fields

In DVWA, I confirmed the vulnerability in the **Stored XSS (Low)** section when my injected script executed automatically on every page load.

➤ What <script>alert(‘Hello world’)</script> Does:

This payload injects a JavaScript <script> tag into the application. When the page loads, the browser executes the script.

Although **alert(‘Hello world’)** only displays a popup, it serves as **proof of execution**, confirming that arbitrary JavaScript can be run inside the victim’s browser.

➤ Possible Scenario in Story form:

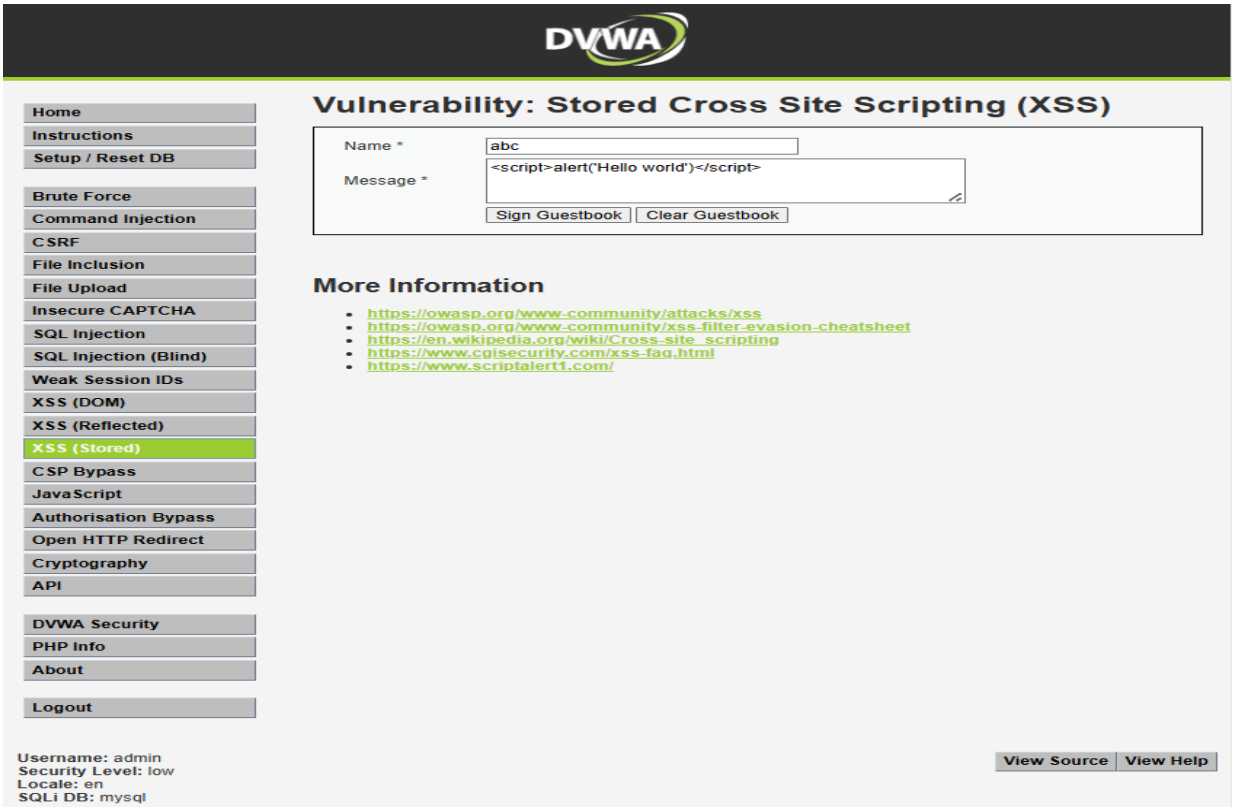
I was carefully analyzing the application, searching for input that didn’t just reflect – but stayed. When my payload returned again after a refresh, I knew the application, I knew the application had stored my code.

At first, it was just a popup saying “Hello world”. But for me, that message meant persistence. Any user who visited that page would unknowingly execute my script.

Instead of an alert, I could extract session cookies, hijack active logins, modify page content, or inject scripts to other stored fields. The attack wouldn’t require repeated interaction – once stored, it would execute automatically.

Everything would appear normal to the user. Meanwhile, I would remain invisible, operating entirely from the client side.

Note: Testing commands are available on GitHub repositories




```
<li class=""><a href="../../vulnerabilities/exec/">Command Injection</a></li>
<li class=""><a href="../../vulnerabilities/csrf/">CSRF</a></li>
<li class=""><a href="../../vulnerabilities/fi/.?page=include.php">File Inclusion</a></li>
<li class=""><a href="../../vulnerabilities/upload/">File Upload</a></li>
<li class=""><a href="../../vulnerabilities/captcha/">Insecure CAPTCHA</a></li>
<li class=""><a href="../../vulnerabilities/sqli/">SQL Injection</a></li>
<li class=""><a href="../../vulnerabilities/sqli_blind/">SQL Injection (Blind)</a></li>
<li class=""><a href="../../vulnerabilities/weak_id/">Weak Session IDs</a></li>
<li class=""><a href="../../vulnerabilities/xss_d/">XSS (DOM)</a></li>
<li class=""><a href="../../vulnerabilities/xss_r/">XSS (Reflected)</a></li>
<li class="selected"><a href="../../vulnerabilities/xss_s/">XSS (Stored)</a></li>
<li class=""><a href="../../vulnerabilities/csp/">CSP Bypass</a></li>
<li class=""><a href="../../vulnerabilities/javascript/">JavaScript</a></li>
<li class=""><a href="../../vulnerabilities/authbypass/">Authorisation Bypass</a></li>
<li class=""><a href="../../vulnerabilities/open_redirect/">Open HTTP Redirect</a></li>
<li class=""><a href="../../vulnerabilities/cryptography/">Cryptography</a></li>
<li class=""><a href="../../vulnerabilities/api/">API</a></li>
</ul><ul class="menuBlocks"><li class=""><a href="../../security.php">DVWA Security</a></li>
<li class=""><a href="../../phpinfo.php">PHP Info</a></li>
<li class=""><a href="../../about.php">About</a></li>
</ul><ul class="menuBlocks"><li class=""><a href="../../logout.php">Logout</a></li>
</ul>
                </div>

            </div>

            <div id="main_body">


<div class="body_padded">
    <h1>Vulnerability: Stored Cross Site Scripting (XSS)</h1>

    <div class="vulnerable_code_area">
        <form method="post" name="guestform" ">
            <table width="550" border="0" cellpadding="2" cellspacing="1">
                <tr>
                    <td width="100">Name *</td>
                    <td><input name="txtName" type="text" size="30" maxlength="10"></td>
                </tr>
                <tr>
                    <td width="100">Message *</td>
                    <td><textarea name="mtxMessage" cols="50" rows="3" maxlength="50"></textarea></td>
                </tr>
                <tr>
                    <td width="100"> </td>
                    <td>
                        <input name="btnSign" type="submit" value="Sign Guestbook" onclick="return validateGuestbookForm(this.form);" />
                        <input name="btnClear" type="submit" value="Clear Guestbook" onClick="return confirmClearGuestbook();" />
                    </td>
                </tr>
            </table>
            <input type='hidden' name='user_token' value='898cd12c4884e4a860f4c0b98a8f88d7' />
        </form>

    </div>
    <br />

    <div id="guestbook_comments">Name: abc<br />Message: &amp;lt;script&amp;gt;alert(\&amp;#039;Hello world\&amp;#039;)&amp;lt;/script&amp;gt;<br /></div>

    <br />
```

IDENTIFYING & ANALYSIS SQL INJECTION (SQLi) VULNERABILITIES IN DVWA / OWASP JUICE SHOP FROM MY PERSPECTIVE

😈 SQL INJECTION (SQLi):

SQL Injection is a **web security vulnerability** where I can **interfere with the queries** that an application makes its database. This allows me to view, modify, or even delete data I shouldn't be able to access. In serious cases, I can completely take over the database server.

🔷 HOW SQL QUERIES WORK & HOW USER INPUT MODIFIES THEM:

➤ How A Normal Query Works:

When a user logs in, the application might build an SQL query like this:

SELECT * FROM users WHERE username = 'USER_INPUT' AND password = 'PASSWORD_INPUT';

The database checks if the **username** and **password** match a record. If they do, I'm logged in.

➤ How I Modify the Query (The Injection):

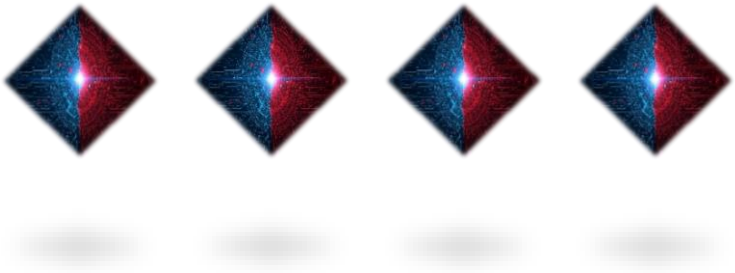
I exploit this by entering special characters, like a single quote ('), into the input field.

Example: If I put the username as **' OR 1=1 - -** (or something similar), the final query becomes:

SELECT * FROM users WHERE username = '' OR 1=1 – ' AND password = 'PASSWORD _INPUT';

- The **'** ends the username string
- The **OR 1=1** is always true, so the **WHERE** condition is always met
- The **- -** (or **#** in some databases) tells the database to treat the rest of the original query (including the password check) as a **comment**, effectively ignoring it.

Result: The database executes the modified query, sees **WHERE TRUE**, and logs me in as the first user (often an administrator) **without needing a password**.



◇ IDENTIFYING VULNERABLE PARAMETERS:

I look for any place where I can input data that will be used in an SQL query.

- **Login forms:** Username & password fields
- **Search fields:** Inputs that filter results from a database
- **URL parameters:** Things like **id=** in a URL such as **product.php?id=123**
- **Hidden fields:** Data sent in **POST requests** that I can intercept and change

My goal is to find parameters that the application uses directly in a database query **without cleaning or sanitizing** the input first.

◇ TESTING MANUALLY USING PAYLOADS:

I use specific strings, called payloads, to test if an input is vulnerable.

➤ Payload 1: The Single Quote (')

- **Action:** I add a single quote to the end of a parameter (e.g. product.php?id=1)
- **Expected vulnerable result:** If the application is vulnerable, the single quote will break the intended SQL syntax, causing a database error to be displayed on the page. This is a clear indicator of a potential SQLi flaw

➤ Payload 2: The Truthy / Comment (' OR 1=1 - -)

- **Action:** I try to change the logic, as shown in the example above, in a login field
- **Expected vulnerable result:** The application should grant me unauthorized access (e.g., login me in without a valid password)

➤ Payload 3: The Order By / Group by Clause

- **Action:** I use a query structure to test how many columns the database result set has (e.g., **product.php?id=1 ORDER BY 5- -**)
- **Expected vulnerable result:** If the application has 4 columns, **ORDER BY 5** will cause an error (because there is no 5th column). **ORDER BY 4** will succeed. This helps me map out the database structure for more complex attacks later.

PRACTICAL IMPLEMENTATION OF SQL INJECTION ON DVWA ON A LOW-LEVEL SECURITY

😈 SQL INJECTION (SQLi) – DVWA (LOW SECURITY):

◇ INITIAL TESTING & ERROR IDENTIFICATION:

While testing SQL Injection in **DVWA at low security**, I first entered a **single quote** (**'**) in the **User ID** field.

As a result, I received a **Fatal Error**.

➤ What is a Fatal Error:

A **fatal error** is a database error that occurs when the SQL query breaks due to invalid input. It usually means my input is being directly used in the SQL query without proper sanitization, which confirms the application is vulnerable to SQL Injection.

◇ BYPASSING THE QUERY LOGIC:

Next, I entered the payload:

1' OR '1' = '1

Because **'1' = '1' is always true**, the WHERE condition became true for all records.

As a result, the application returned **multiple user records**, such as:

- **admin**
- **Gordon Brown**
- **Hack Me**
- **Pable Picasso**
- **Bob Smith**

This confirmed that I successfully **bypassed the query restriction** and forced the database- to display all rows.

◇ USING UNION-BASED SQL INJECTION:

After confirming the vulnerability, I tested **UNION-based SQL Injection** by entering:

1' OR '1' = '1 UNION SELECT * FROM password

This showed me that **UNION SELECT** can be used to combine my malicious query with the original query and extract additional data from other tables.

From my research, I learned that **UNION SELECT** allows me to retrieve data from another table **only if the number of columns matches**.

🔷 EXTRACTING USERNAMES & PASSWORD HASHES:

Then I used the payload:

' UNION SELECT user, password FROM users#

With this, the database returned **usernames and password hashes**, for example:

- **Admin** → 5f4dcc3b5aa765d61d8327deb882cf99
- **Gordonb** → e99a18c428cb38d5f260853678922e03
- **Pablo** → 0d107d09f5bbe40cade3de5c71e9e9b7

Here, the **First name field displayed usernames**, and the **Surname field displayed hashed passwords**.

🔷 HASH IDENTIFICATION & CRACKING:

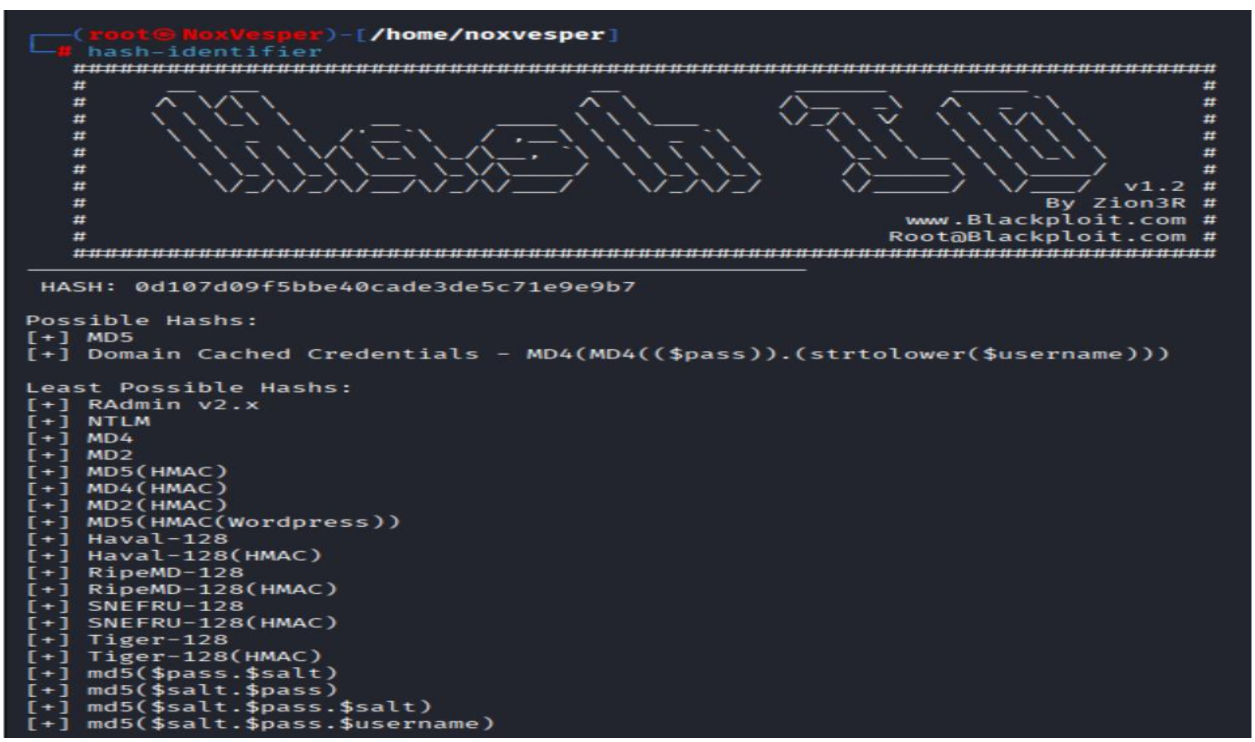
I identified that the surnames were **hashed passwords**.

To analyzed them, I used **hash-identifier** in Kali Linux by running it -pin the terminal and pasting the hash value.

The tool showed possible hash types, which allowed me to attempt password cracking **ethically** using:

- **John the Ripper**
- **Hashcat**

By testing the cracked passwords carefully, I can **verify credentials in a controlled lab environment** like DVWA without harming real systems.



I IDENTIFIED INFORMATION DISCLOSURE AND BROKEN AUTHENTICATION AS SERIOUS SECURITY ISSUES BY OBSERVING EXPOSED COMMENTS, DEBUG INFORMATION, AND ACCESSIBLE BACKUP FILES. I TESTED THE SYSTEM FOR WEAK AUTHENTICATION BY ATTEMPTING DEFAULT CREDENTAILS AND WEAK PASSWORDS. I UNDERTSOOD THAT THESE ISSUES ARE DANGEROUS BECAUSE THEY REVEAL SENSITIVE INFORMATION AND ALLOWED UNAUTHORIZED ACCESS, WHICH CAN LEAD TO COMPLETE SYSTEM COMPROMISE.

😈 INFORMATION DISCLOSURE & BROKEN AUTHENTICATION:

Information Disclosure is a security weakness where I am able to access sensitive information that should not be visible to me. This information can include usernames, internal paths, configuration details, or system logic.

Broken Authentication occurs when I am able to bypass or weaken the login mechanism of an application, allowing me to impersonate other users or gain unauthorized access.

◇ COMMON INFORMATION DISCLOSURE ISSUES:

➤ Comments & Debug Information:

I inspect the page source, JavaScript files, and server responses.

Explanation:

I often find developer comments, debug messages, or stack traces that reveal database names, API keys, file paths, or application logic. This information helps me understand how the system works internally.

➤ Backup & Sensitive Files:

I try accessing files like **.bak**, **.old**, **.zip**, **.env**, or **.log**

Explanation:

If the server allows access to these files, I can download backups containing source code, credentials, or configuration settings, giving me a deeper insight into the application.

◇ TESTING FOR WEAK AUTHENTICATION:

➤ Default Credentials:

I test common username and password combinations such as **admin / admin** or **admin / password**.

Explanation:

If the application still uses default credentials, I can log in without any real effort and gain full control of the account.

➤ Weak Passwords:

I attempt simple and predictable passwords.

Explanation:

When password policies are weak or not enforced, I can easily guess or brute-force credentials and gain unauthorized access.

◇ **WHY THESE ISSUES ARE DANGEROUS:**

➤ **Danger of Information Disclosure:**

Exposed information gives me a roadmap of the system.

Explanation:

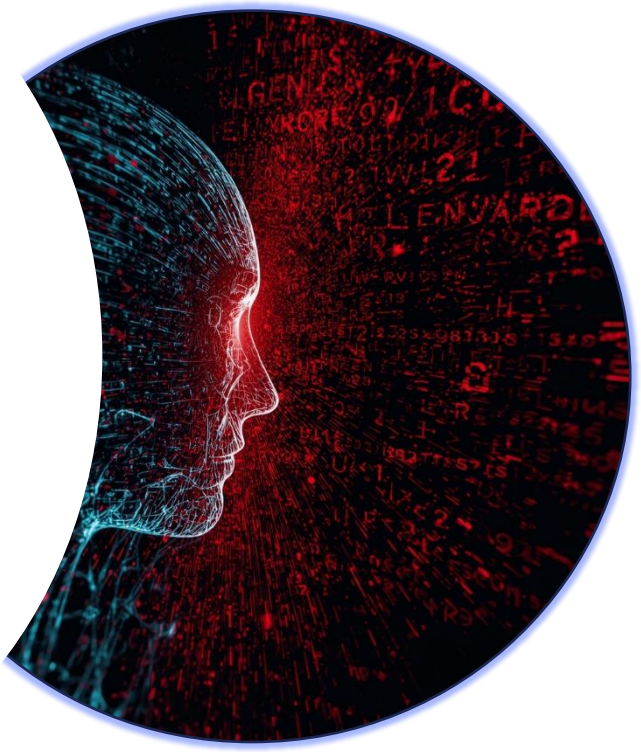
With internal details, I can plan targeted attacks, find hidden endpoints, and exploit other vulnerabilities more efficiently.

➤ **Danger of Broken Authentication:**

Broken authentication allows me to become someone I am not.

Explanation:

Once I bypass authentication, I can access private data, perform privileged actions, or fully compromise the application and its users.



The second phase of this research is completed.