Chapter 5

Bypassing Client-Side Controls:

How client side is not secure?

Web applications are insecure because users can send any input they want, even if the website tries to control it on the client side. Client-side controls can be bypassed since users fully control their browsers. Applications rely on client-side controls either to hide or restrict data, but these methods are unsafe and can be easily bypassed.

Transmitting Data via the Client:

Web applications often send data to the user's browser and expect it to be returned unchanged, assuming it is safe because it is hidden. However, users fully control their browsers and can modify this data, making such trust unsafe. Although this method simplifies development and improves performance, sending sensitive data through the client can lead to serious security vulnerabilities.

This is commonly done using hidden form fields, cookies, URL parameters, the Referer header, opaque data, and ASP.NET ViewState. Although convenient, all these methods are under the user's control and can be modified, making them unsafe for storing or trusting sensitive data.

Hidden Form Fields:

Hidden form fields are not visible to users but are still part of the page and are sent to the server when a form is submitted. Developers often store sensitive data like prices in hidden fields, assuming users cannot change them. However, users can modify these values using browser tools or intercepting proxies, and if the server trusts this data, it can lead to serious security issues such as price manipulation.

Exploiting Hidden Form Fields on a local website:

Index.html:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Vulnerable Shop</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
<h2>Buy Laptop</h2>
<form id="orderForm">
  Quantity:
  <input type="number" id="quantity" value="1"><br><br>
  <!-- Hidden price field -->
  <input type="hidden" id="price" value="1000">
  <button type="submit">Buy</button>
</form>
<p id="result"></p>
<script src="vulnerable.js"></script>
</body>
</html>
```

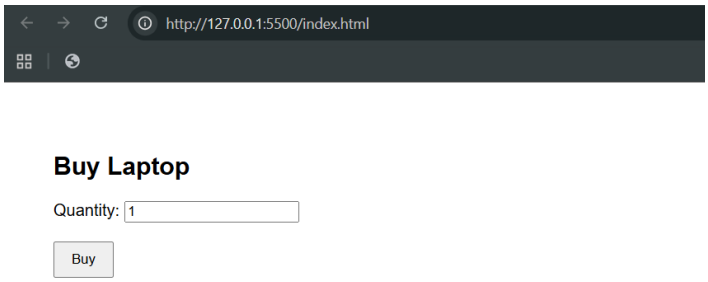
Style.css:

```css
body {
  font-family: Arial, sans-serif;
  padding: 40px;
}

button {
  padding: 8px 16px;
  cursor: pointer;
}
```

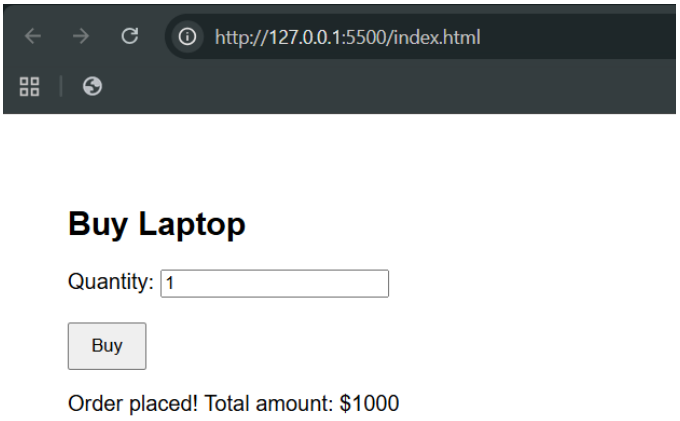
Vulnerable.js:

```javascript
document.getElementById("orderForm").addEventListener("submit", function (e) {
  e.preventDefault();

  const quantity = Number(document.getElementById("quantity").value);
  const price = Number(document.getElementById("price").value); // ❌ trusting client

  const total = quantity * price;

  document.getElementById("result").innerText =
    "Order placed! Total amount: $" + total;
});
```

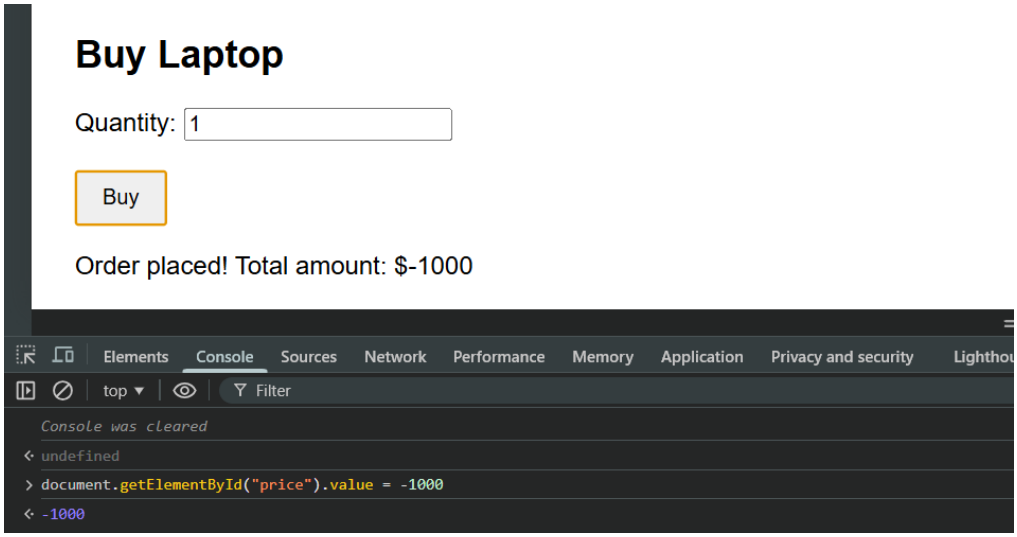
Output:



Output: when we clicked the “buy” button.



Now, what’s wrong here? Let’s try to change something from the console:



Why this is vulnerable?

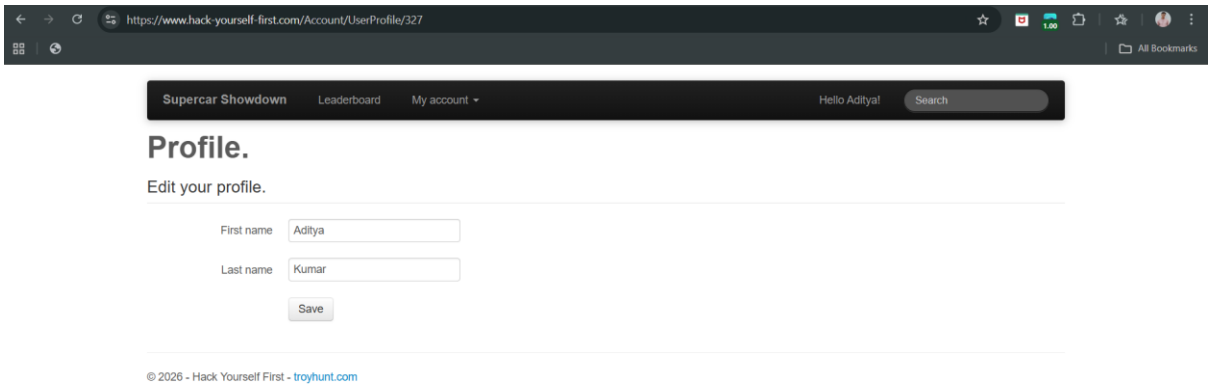
- Price is stored in the page
- Anyone can open DevTools and change:

  document.getElementById("price").value = -1000
- Logic breaks because the app trusts client data

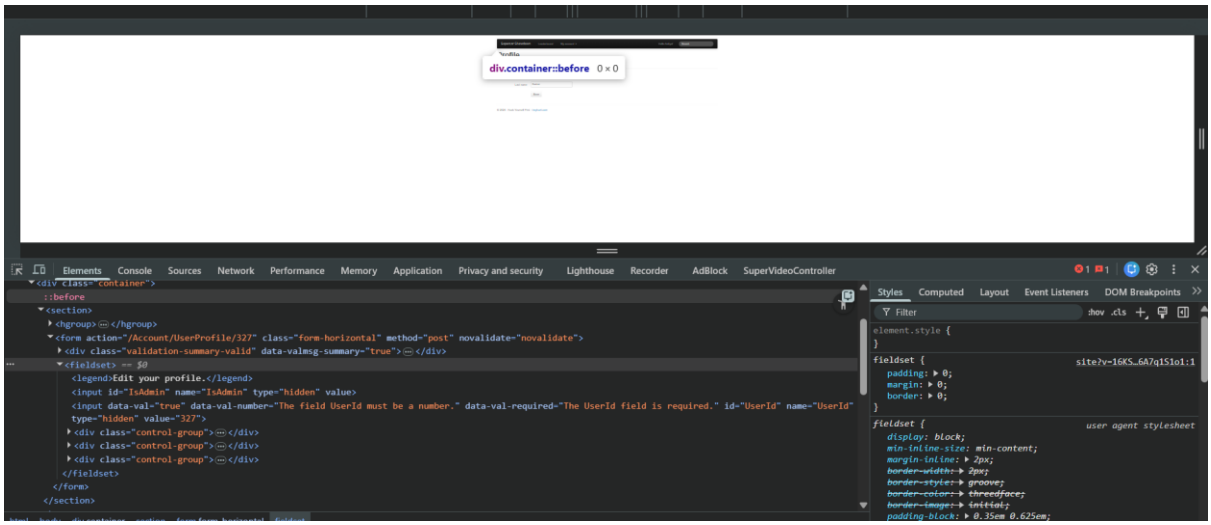
Exploiting Hidden Form Fields on a hosted website:

Tested on https://www.hack-yourself-first.com/

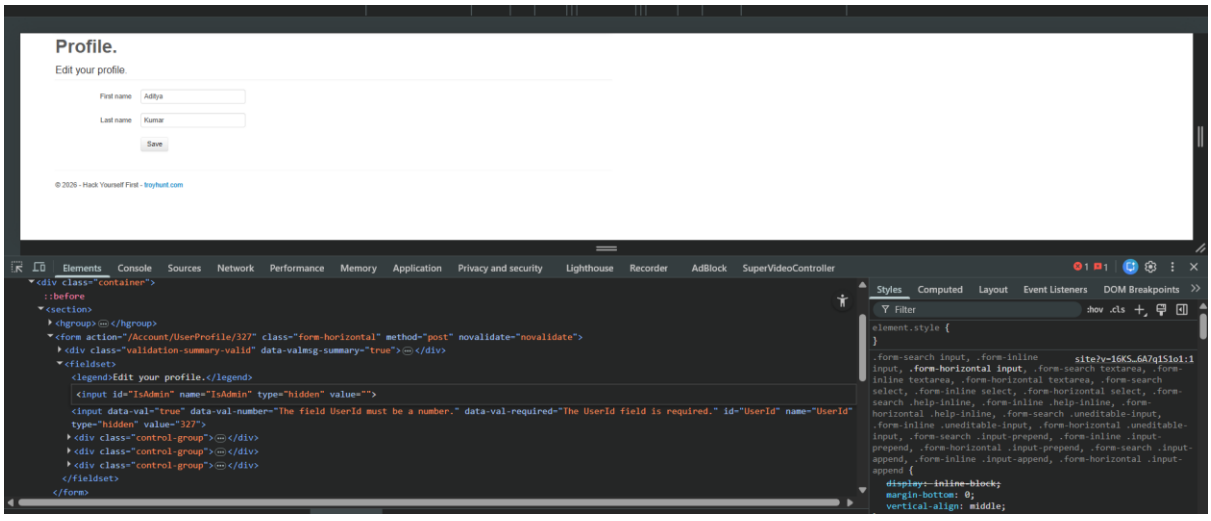
Step1: Go to the login page, and login, then go to “Edit profile”.



Step2: Click on the inspect element. Following screen will appear:



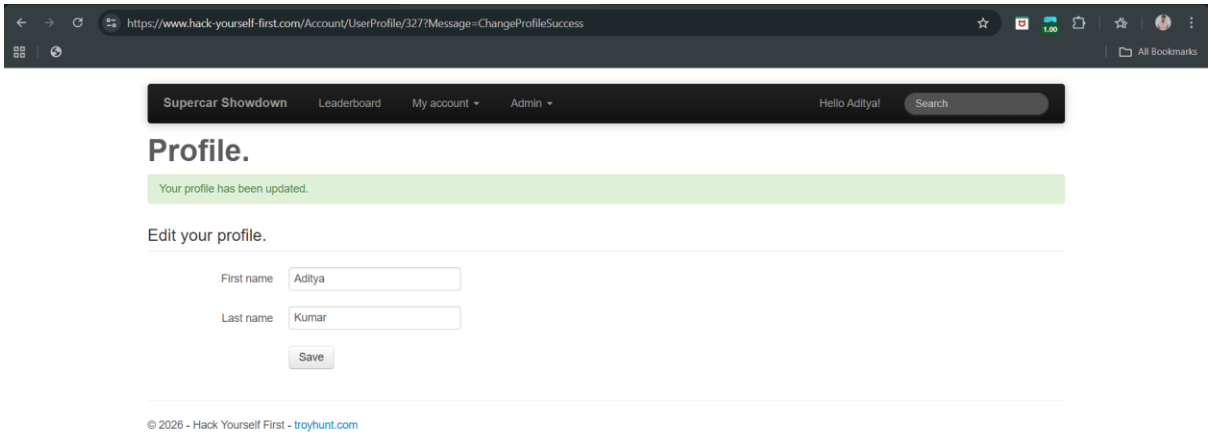
Step3: in the search bar, search for “hidden”, following result seems important. Right click and select as “edit as HTML”.



Step4: Edit that line shown below:

```
<input id="IsAdmin" name="IsAdmin" type="hidden" value="true">
```

Step5: Click on the save button, following screen can be observed.



Clearly, now we have “Admin” panel available. Hence, we managed to do privilege escalation.

Read these links for parameter identification:

1. https://medium.com/geekculture/params-discovering-hidden-treasure-in-webapps-b4a78509290f

HTTP Cookies:

What are HTTP cookies?

Cookies are small bits of information that a website stores in your browser. Your browser sends these cookies back to the website every time you visit again. Because the internet (HTTP) does not remember anything by itself, cookies help websites remember you.

Why websites use cookies?

Websites mainly use cookies for three reasons:

1. Session Management
    - To remember that you are logged in
    - To keep items in your shopping cart
    - To save game progress or similar temporary data
2. Personalization
    - To remember your language choice
    - To keep your theme (dark/light mode)
    - To save preferences
3. Tracking
    - To see how users move around a website
    - To show ads or analyse behaviour

How login works using cookies?

a. You enter your username and password.
b. If they are correct, the server sends back a cookie with a session ID.
c. Your browser stores that cookie.
d. Every time you open another page, your browser sends the cookie back.
e. The server checks the cookie:
    - If it's valid → you stay logged in.
    - If it's invalid or expired → you're logged out.

Creating and sending cookies:

- The server creates cookies using a Set-Cookie message.
- The browser automatically sends cookies back with future requests.

Cookie lifetime (how long cookies live):

1. Permanent cookies

These have:

- An expiration date or
- A time limit

They are deleted when time runs out.

2. Session cookies

These have no expiration date. They are deleted when the browser session ends (but sometimes browsers keep them longer).

Updating cookies:

- The server can change a cookie's value by sending it again with a new value.
- JavaScript in the browser can also change cookies (unless blocked).

Security risks of cookies:

By default:

- Cookies can be seen and changed by users
- Attackers can steal or modify cookies

How cookies are secured?

1. Secure flag

- Cookie is sent only over HTTPS
- Prevents attackers from reading cookies on insecure connections

2. HttpOnly flag

- JavaScript cannot access the cookie
- Protects against hacking attacks like XSS

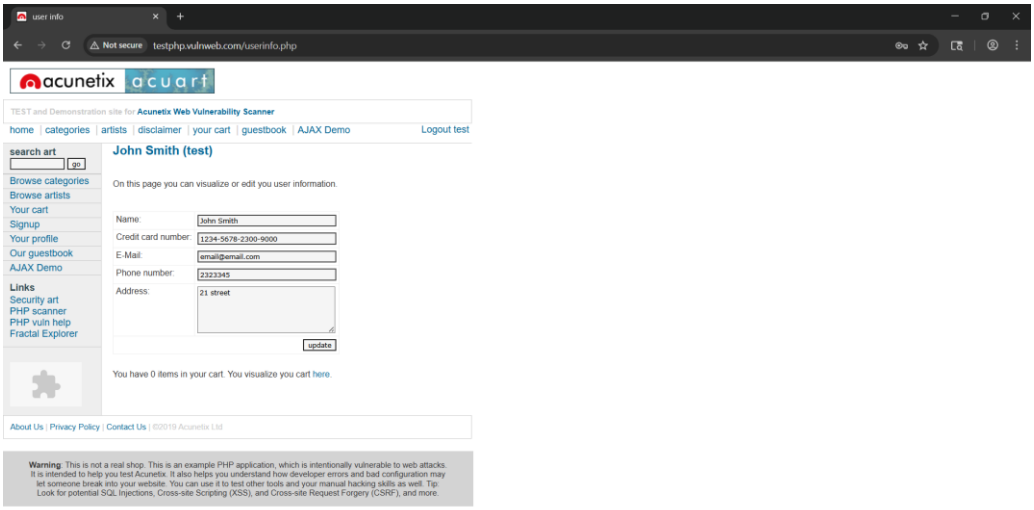
Login/session cookies should always use HttpOnly.

Important idea (very important): Anything stored on the user's computer can be modified by the user. Cookies are stored on the client side (the user's browser), so they cannot be trusted.

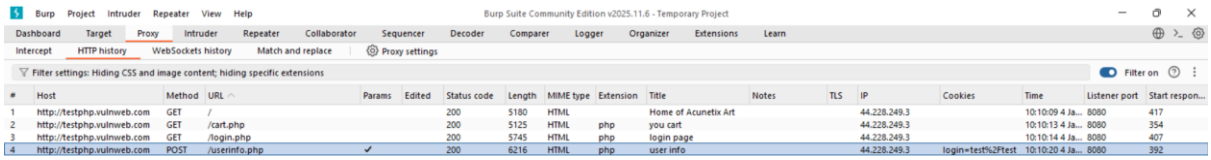
Looking HTTP Cookies using Burp Suite:

Testing on: http://testphp.vulnweb.com/

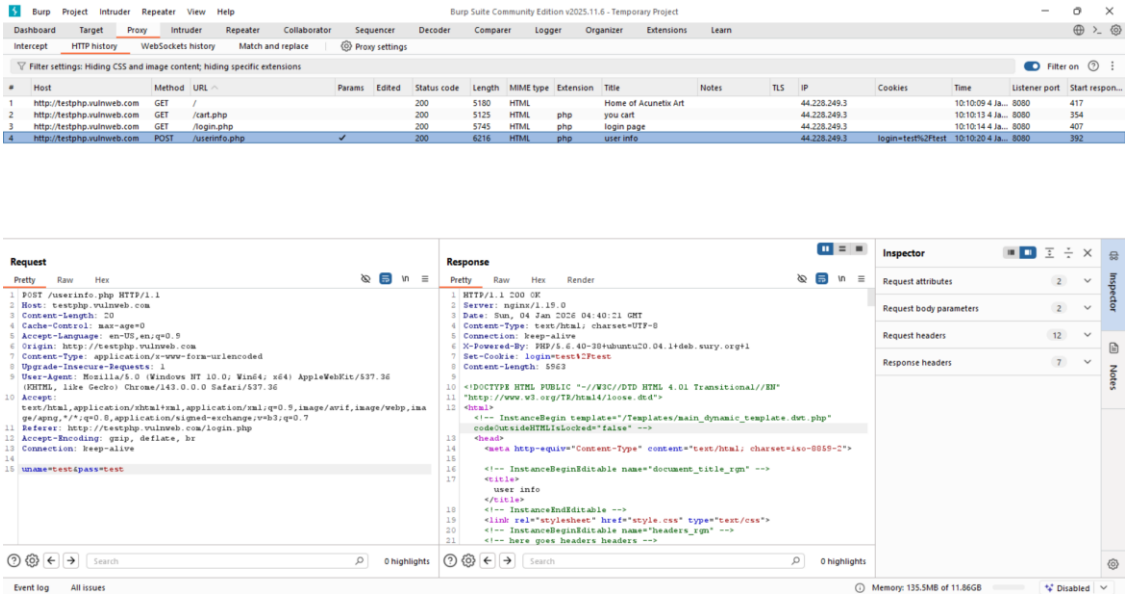
Step1: Open Burp Suite, visit the website given above from the inbuilt browser, login to the page.



Step2: Visit the section Proxy->HTTP History. See the last request '/userinfo.php'.



Step3: Click on it, and see the details.



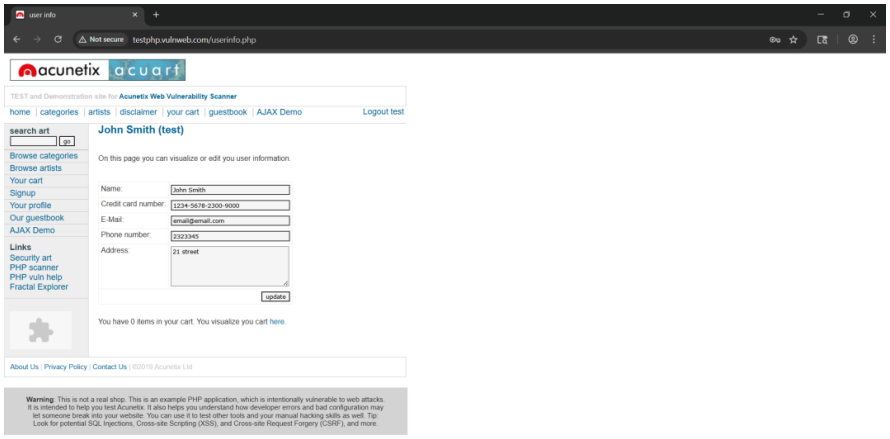
Clearly, we can see the Set-Cookie in the response section having the value login=test%2Ftest, which actually means test/test, and since the 200 OK is given, it means user name is test and password is test as well.

The application stores username and password in a client-side cookie and trusts it for authentication. Since cookies can be modified by users, attackers can manipulate the cookie to impersonate other users or gain unauthorized access.

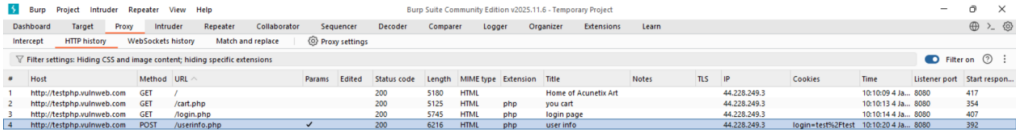
Manipulating HTTP Cookies using Burp Suite:

Testing on: http://testphp.vulnweb.com/

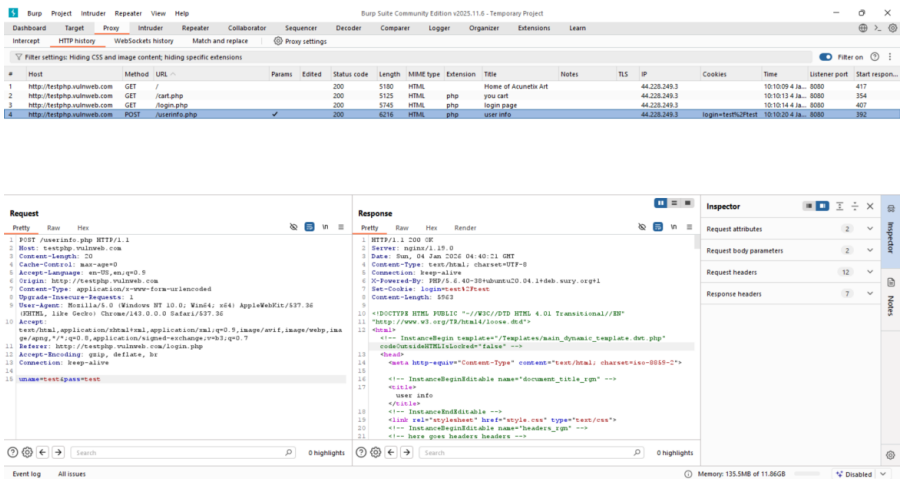
Step1: Open Burp Suite, visit the website given above from the inbuilt browser, login to the page.



Step2: Visit the section Proxy->HTTP History. See the last request '/userinfo.php'.



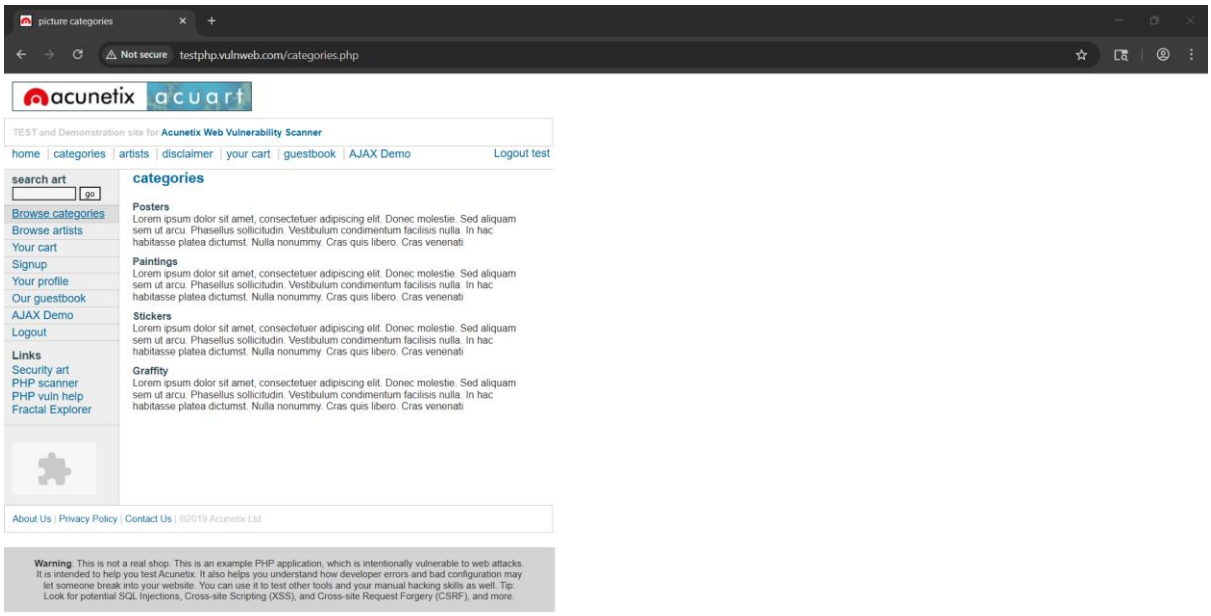
Step3: Click on it, and see the details.



Clearly, we can see the Set-Cookie in the response section having the value login=test%2Ftest, which actually means test/test, and since the 200 OK is given, it means user name is test and password is test as well.

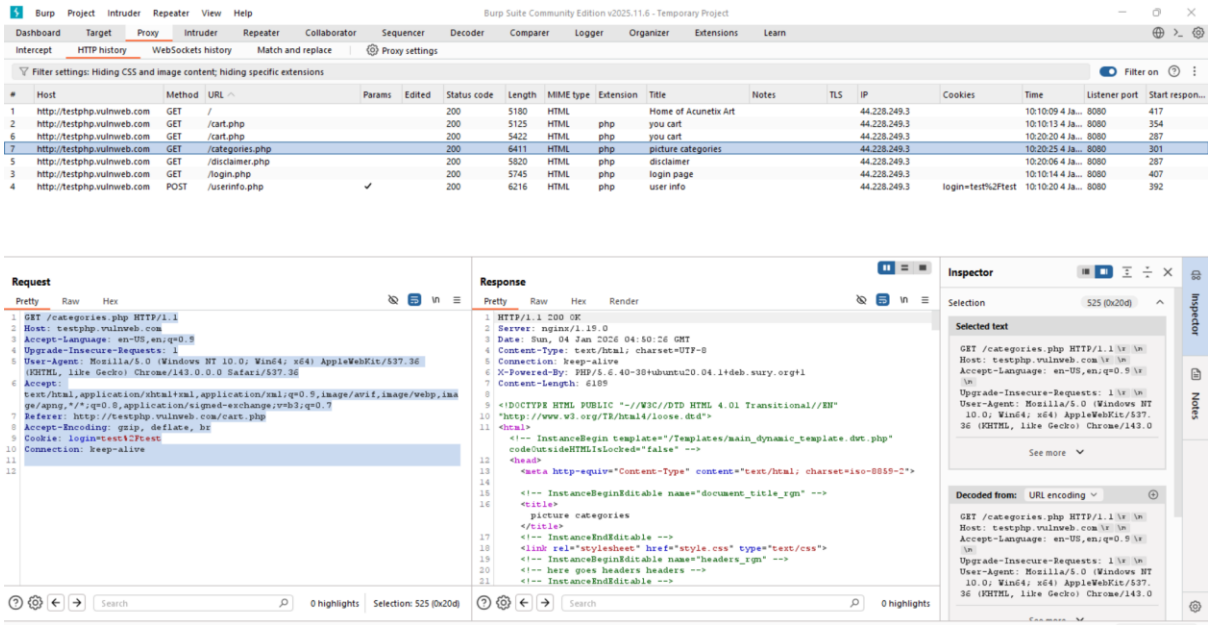
Step4: Now visit any page in the browser when you logged in:

Say:



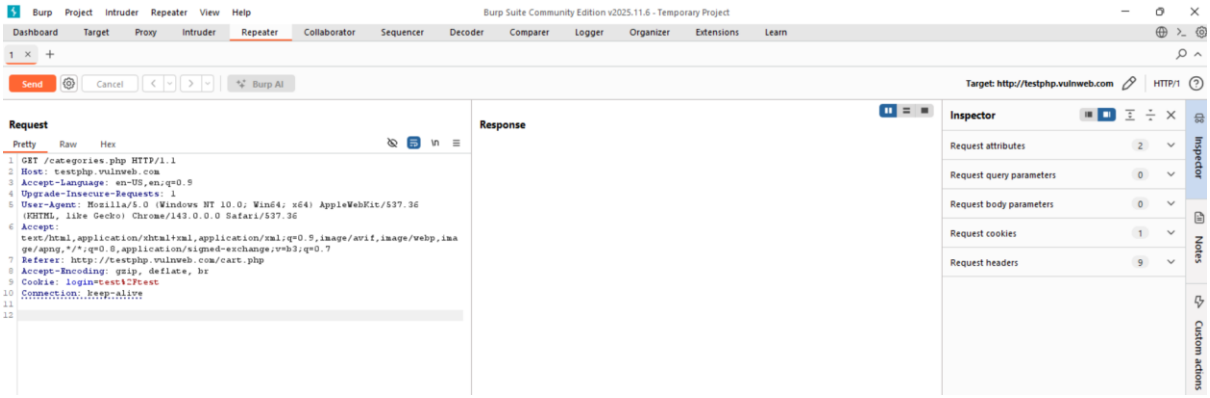
Why to do so? On the next request to the website (any page). The browser automatically sends the cookie back.

In burp:



Clearly, we can see the Cookie in the Request section.

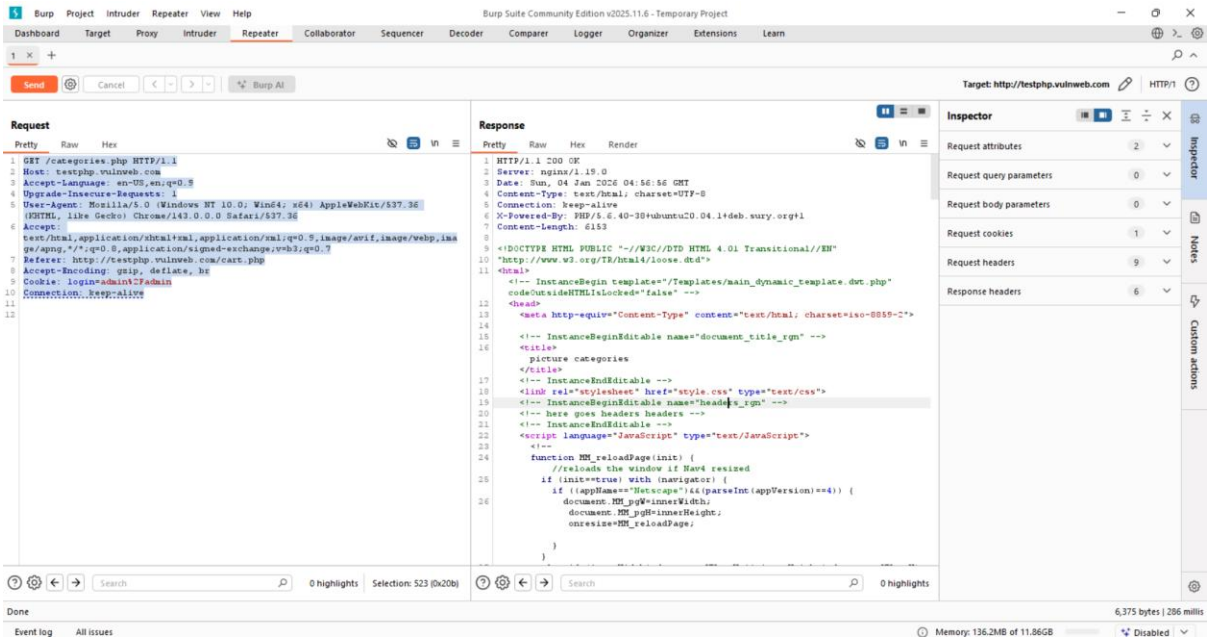
Step5: Send the request to Repeater (Right-click the request -> Click Send to Repeater).



The forwarded request can be seen like this in the “Repeater” section.

Step6: Modify the request as shown below: change the test%2Ftest to admin%2Fadmin.

Step7: Send it…



The server responded with: HTTP/1.1 200 OK

This means: The server accepted your request. It did NOT reject or invalidate the cookie. It trusted the modified cookie value. This is the key security issue.

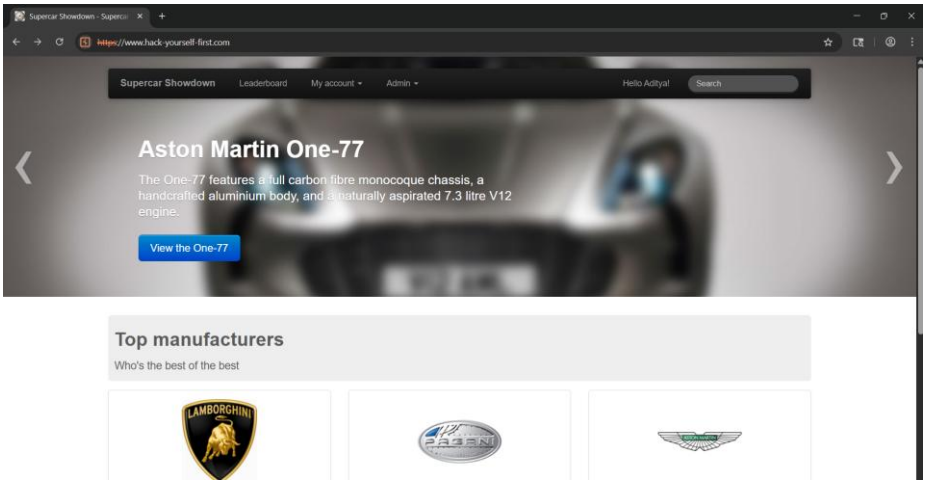
This test proves that:

- Authentication is handled on the client side
- The server does not validate credentials
- Identity is determined purely by a cookie
- Cookie values can be freely manipulated

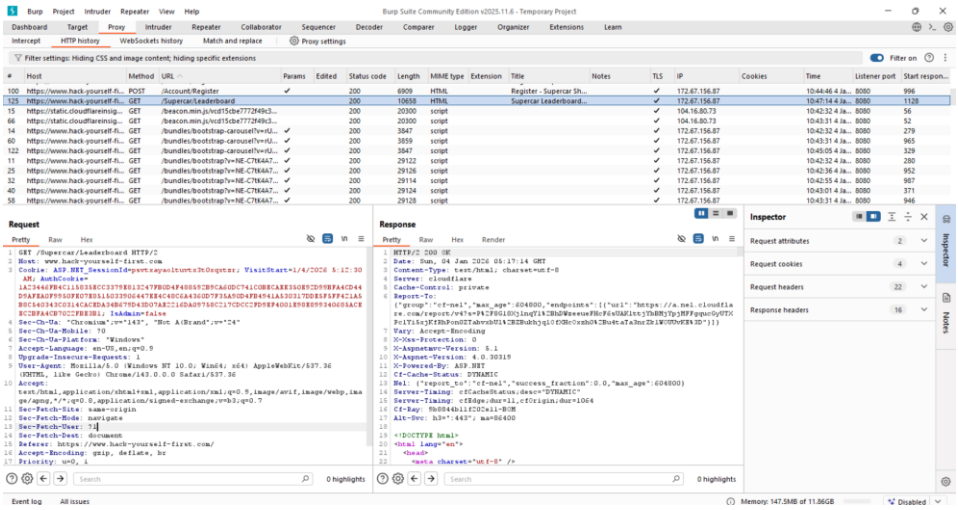
Privilege Escalation via Cookie Manipulation:

Testing on the site: https://www.hack-yourself-first.com/

Step1: Open burp suite, open the browser and visit the page, login as a normal user.

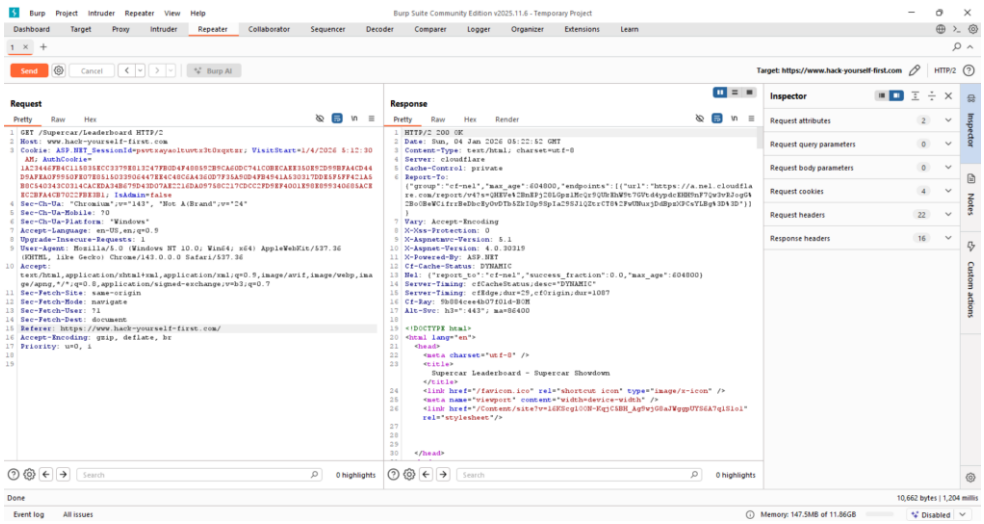


Step2: Visit any site after logging in to see the Cookies set in the “Request” section:



Clearly, two cookies are there, one very long secured, while other is IsAdmin = false.

Step3: Forward the request to the “Repeater”, it can be seen like this:



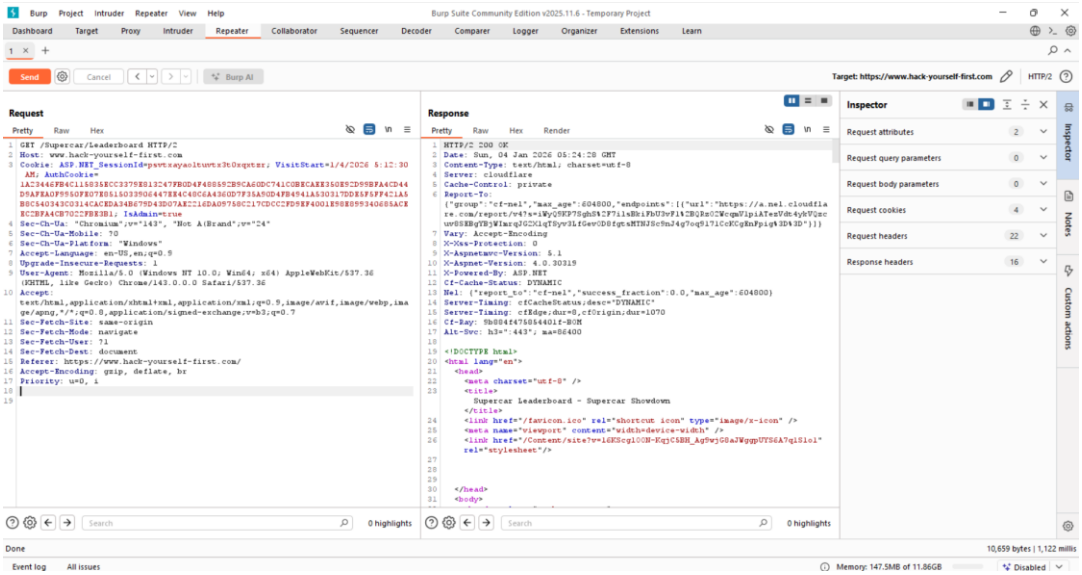
Step4: Change the IsAdmin = False -> IsAdmin = True. Request can be seen like this:

Request

Pretty Raw Hex

```
1  GET /Supercar/Leaderboard HTTP/2
2  Host: www.hack-yourself-first.com
3  Cookie: ASP.NET_SessionId=pswtxayaoltuvtx3t0xqxtzr; VisitStart=1/4/2026 5:12:30
   AM; AuthCookie=
   1AC3446FB4C115835ECC3379E813247FB0D4F488592B9CA60DC741C0BECAEE350E92D99BFA4CD44
   D9AFEA0F9950FE07E8515033906447EE4C48C6A4360D7F35A90D4FB4941A530317DDE5F5FF421A5
   B8C540343C0314CACEDA34B679D43D07AE2216DA09758C217CDCC2FD9EF4001E98E899340685ACE
   EC2BFA4CB7022FBE3B1; IsAdmin=true
4  Sec-Ch-Ua: "Chromium";v="143", "Not A(Brand";v="24"
5  Sec-Ch-Ua-Mobile: ?0
6  Sec-Ch-Ua-Platform: "Windows"
7  Accept-Language: en-US,en;q=0.9
8  Upgrade-Insecure-Requests: 1
9  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
10 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,ima
   ge/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-User: ?1
14 Sec-Fetch-Dest: document
15 Referer: https://www.hack-yourself-first.com/
16 Accept-Encoding: gzip, deflate, br
17 Priority: u=0, i
```

When we clicked the send button:



Observe unauthorized access (HTTP 200). So you told the server you are admin, and the server believed you. Thus, privilege escalation.

How to make sure that HTTP Cookies are secured:

Developers should verify:

- No role/permission stored in plain cookies
- All cookies are signed or random
- Admin routes verify role server-side
- No trust in client-sent flags
- Cookies use Secure + HttpOnly

URL Parameters:

Websites often send information through the URL (the web address). This information is added after a *?* and is called URL parameters.

Example:

https://wahh-app.com/browse.asp?product=VAIOA217S&price=1224.95

In this example:

- product tells the site which item you're viewing
- price tells the site the price of that item

Why this matters?

If you can see parameters in the browser's address bar, you can easily change them just by editing the URL and pressing Enter. No special tools are needed.

Note:

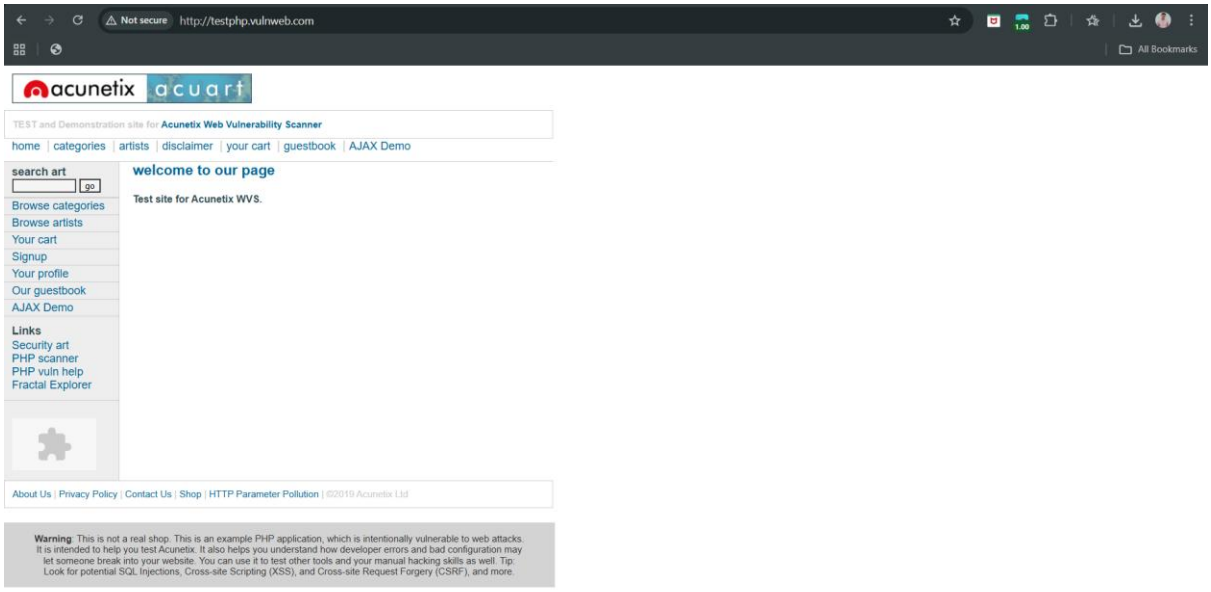
Even if the user can't see the URL or parameters, they can still be:

- Viewed
- Changed

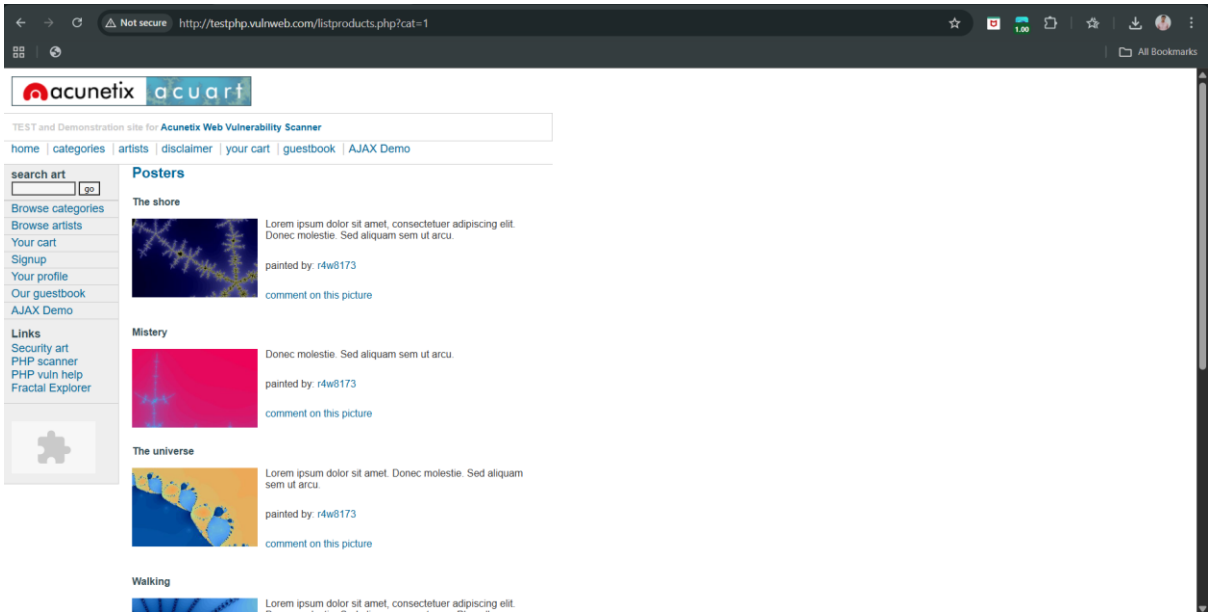
by using tools like an intercepting proxy (a tool that captures and edits web requests).

URL parameters can be viewed and modified by users:

Step1: Visit the website: http://testphp.vulnweb.com/



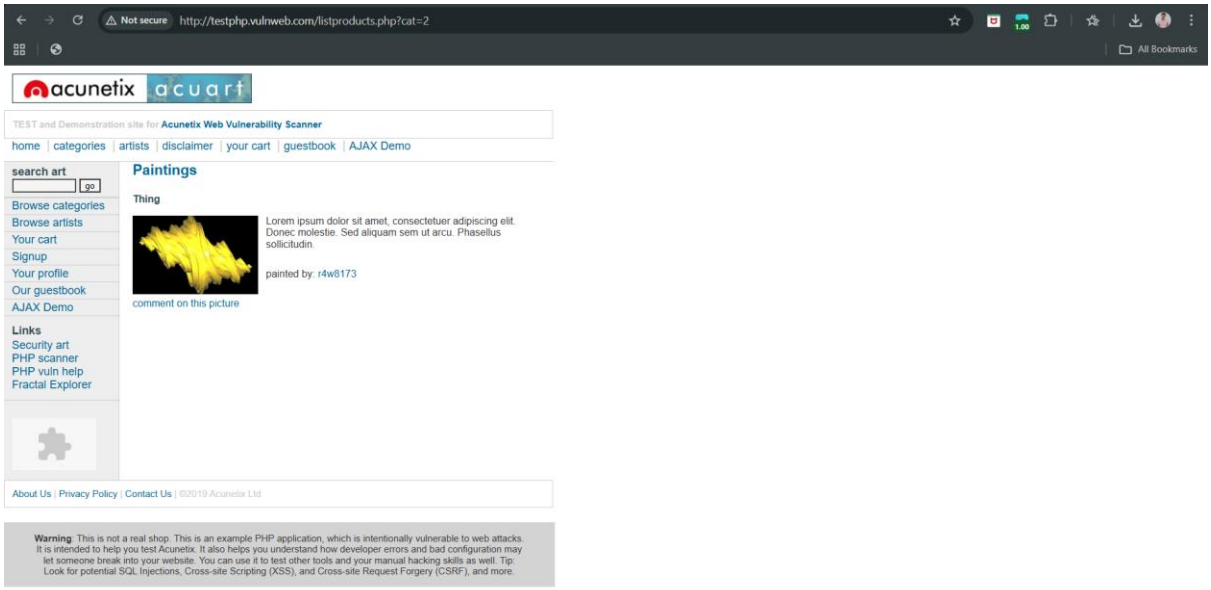
Step2: Visit those pages where the parameters can be seen in the address bar.



Here:

- cat is a URL parameter
- 1 is the category ID

Step3: Modify the URL parameter manually. Change cat = 1-> cat = 2



What happens?

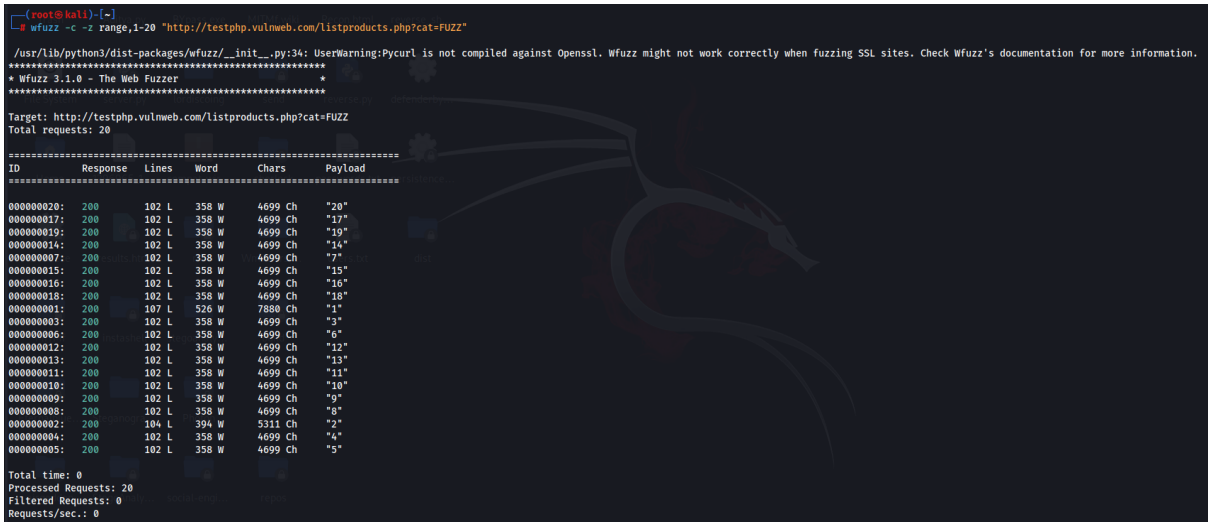
- The page loads a different category
- This proves the site trusts the URL parameter

URL parameters can be modified – testing on Kali Linux:

We will use wfuzz.

Step1: Use the following command to fuzz the ‘cat’ parameter.

Output:



What this means

- Different response sizes = different server behavior
- The application is:
    - Trusting the cat parameter
    - Returning different content based on your input
- This confirms client-side control over server logic

Finding all the subdomains of a website on Kali Linux:

We will use *paramspider* tool

Step1: use the below command to do so.

paramspider -d testphp.vulnweb.com

Output:

Step2: View what it found

Step3: Extract just endpoints + parameter names

```
┌──(root㉿kali)-[~]
└─# cat results/testphp.vulnweb.com.txt \
| sed 's/=.*/=FUZZ/' \
| sort -u

http://testphp.vulnweb.com/?%3Fcmd=FUZZ
http://testphp.vulnweb.com/?%3Fid=FUZZ
http://testphp.vulnweb.com/admin/?C=FUZZ
http://testphp.vulnweb.com/AJAX/infoartist.php?id=FUZZ
http://testphp.vulnweb.com/AJAX/infocateg.php?id=FUZZ
http://testphp.vulnweb.com/artist.php?artist=FUZZ
http://testphp.vulnweb.com/artists.php?+artist=FUZZ
http://testphp.vulnweb.com/artists.php?artist+=FUZZ
http://testphp.vulnweb.com/artists.php?artist=FUZZ
http://testphp.vulnweb.com/artists.php?%CB%93%E2%86%92artist=FUZZ
http://testphp.vulnweb.com/artists.php?file=FUZZ
http://testphp.vulnweb.com/artists.php?oqXf=FUZZ
http://testphp.vulnweb.com/bxss/vuln.php?id=FUZZ
http://testphp.vulnweb.com/categories.php/listproducts.php?cat=FUZZ
http://testphp.vulnweb.com/%CB%93%E2%86%92artists.php?artist=FUZZ
http://testphp.vulnweb.com/%CB%93%E2%86%92listproducts.php?cat=FUZZ
http://testphp.vulnweb.com/?cmd=FUZZ
http://testphp.vulnweb.com/comment.php?aid=FUZZ
```

This is now a perfect fuzzing target list.

Step4: save fuzzable URLs to a file.

```
┌──(root㉿kali)-[~]
└─# cat results/testphp.vulnweb.com.txt \
| sed 's/=.*/=FUZZ/' \
| sort -u > fuzz_targets.txt
```

Step5: Chain ParamSpider → wfuzz

```
┌──(root㉿kali)-[~]
└─# while read url; do
  echo "[*] Fuzzing $url"
  wfuzz -c -z range,1-200 "$url"
done < fuzz_targets.txt

[*] Fuzzing http://testphp.vulnweb.com/?%3Fcmd=FUZZ
 /usr/lib/python3/dist-packages/wfuzz/__init__.py:34: UserWarning:Pycurl is not compiled against Openssl. Wfuzz might not work correctly when fuzzing SSL sites. Check Wfuzz's documentation for more information.
********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://testphp.vulnweb.com/?%3Fcmd=FUZZ
Total requests: 200

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000023:   200        109 L    388 W      4958 Ch     "23"
000000015:   200        109 L    388 W      4958 Ch     "15"
000000024:   200        109 L    388 W      4958 Ch     "24"
000000028:   200        109 L    388 W      4958 Ch     "28"
000000025:   200        109 L    388 W      4958 Ch     "25"
000000026:   200        109 L    388 W      4958 Ch     "26"
000000027:   200        109 L    388 W      4958 Ch     "27"
000000007:   200        109 L    388 W      4958 Ch     "7"
000000003:   200        109 L    388 W      4958 Ch     "3"
000000001:   200        109 L    388 W      4958 Ch     "1"
000000022:   200        109 L    388 W      4958 Ch     "22"
000000021:   200        109 L    388 W      4958 Ch     "21"
000000020:   200        109 L    388 W      4958 Ch     "20"
000000019:   200        109 L    388 W      4958 Ch     "19"
000000018:   200        109 L    388 W      4958 Ch     "18"
000000017:   200        109 L    388 W      4958 Ch     "17"
000000016:   200        109 L    388 W      4958 Ch     "16"
000000014:   200        109 L    388 W      4958 Ch     "14"
000000013:   200        109 L    388 W      4958 Ch     "13"
000000012:   200        109 L    388 W      4958 Ch     "12"
000000011:   200        109 L    388 W      4958 Ch     "11"
000000010:   200        109 L    388 W      4958 Ch     "10"
000000009:   200        109 L    388 W      4958 Ch     "9"
000000006:   200        109 L    388 W      4958 Ch     "6"
```

POST does not protect preset URL parameters.:

- URL parameters override POST
- Attackers don't need forms
- Tools ignore the UI entirely

Even when an application uses POST, preset URL parameters remain fully user-controlled. If the server-side logic relies on them, attackers can manipulate application behaviour by modifying those parameters directly or at scale.

The Referer Header:

How a browser talks to a website

When you use a website, your browser sends HTTP requests to the server. An HTTP request has:

- URL (where you're going)
- Method (GET, POST, etc.)
- Headers (extra information)
- Body (data like usernames, passwords, form inputs)

What is the Referer header?

The Referer header tells the server: "From which page did this request come?"

Why browsers send Referer?

Browsers automatically add the Referer header when:

- You click a link
- You submit a form
- A page loads images, scripts, CSS, etc.

Servers often use it for:

- Analytics
- Logging
- Basic checks (this is where problems start)

Developer assumption:

Developers sometimes think: “The Referer header tells me which page the user came from, so I can trust it.”

This is wrong in security because:

- The user controls the browser
- The user controls the request
- The user controls the headers

Why Referer is especially unreliable?

- Referer is OPTIONAL
- Referer can be faked

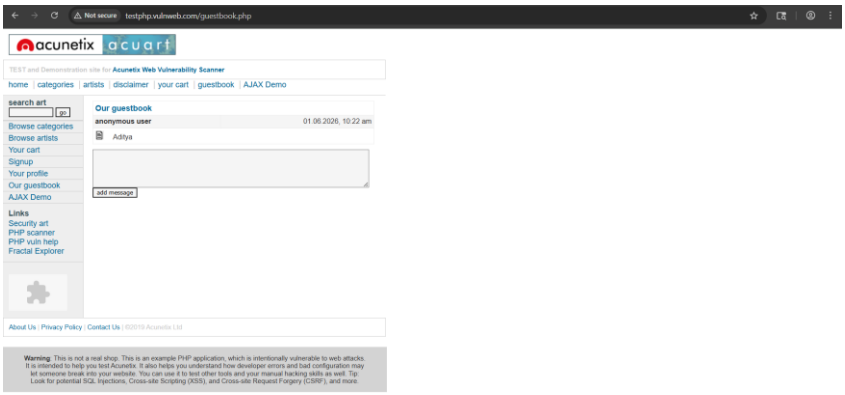
Checking if the Application Trusts the Referer header to control application flow:

Goal: The server relies on the Referer header to allow or deny the POST request

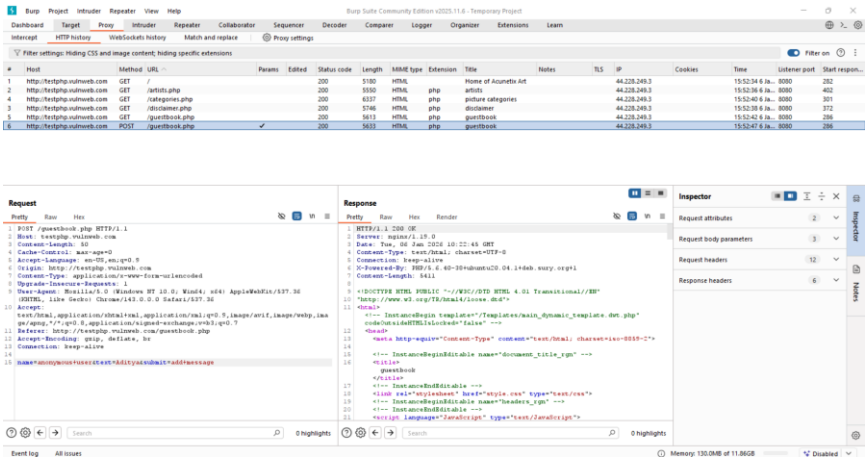
Testing on URL: http://testphp.vulnweb.com/

Step1: Open Burp Suite, visit the page, click other links, make sure to make a POST request. Following request can be seen at Proxy->HTTP History.

Browser:



Burp:



Here, **Important headers (for security testing)**

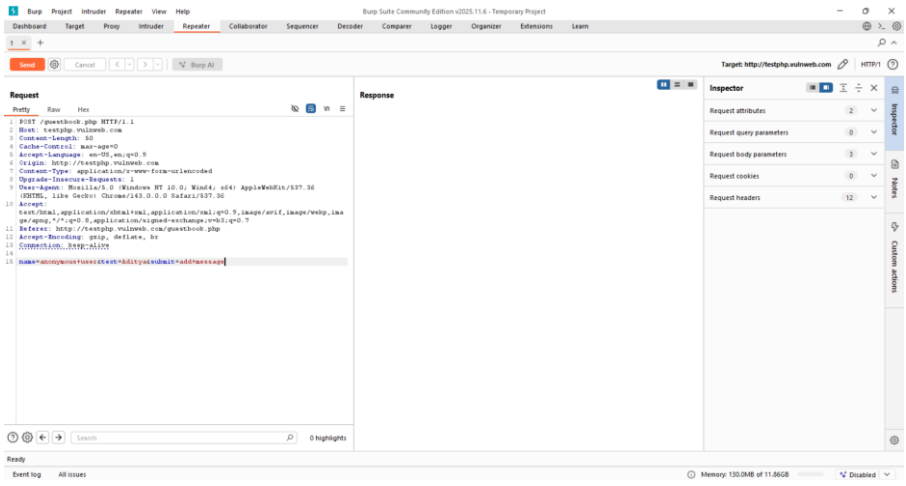
Origin: *http://testphp.vulnweb.com*

Referer: *http://testphp.vulnweb.com/guestbook.php*

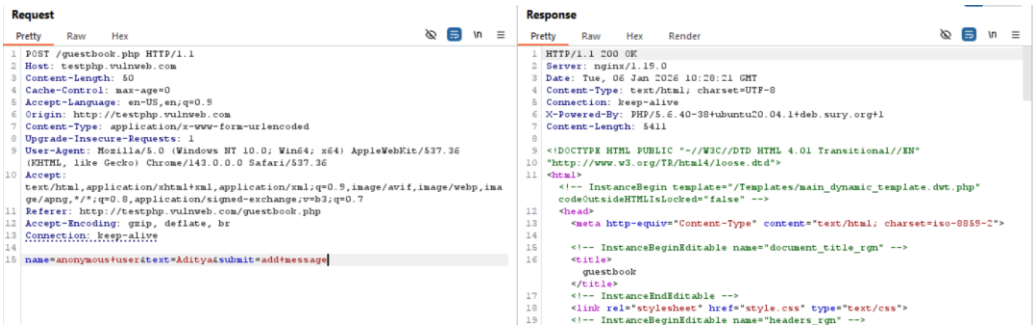
- **Origin** → where the request claims it came from (used mainly for CORS/CSRF)
- **Referer** → exact page that submitted the form

Right now, both look legitimate.

Step2: Send this request to Burp Repeater:



Step3: Baseline request (sending without changing anything).

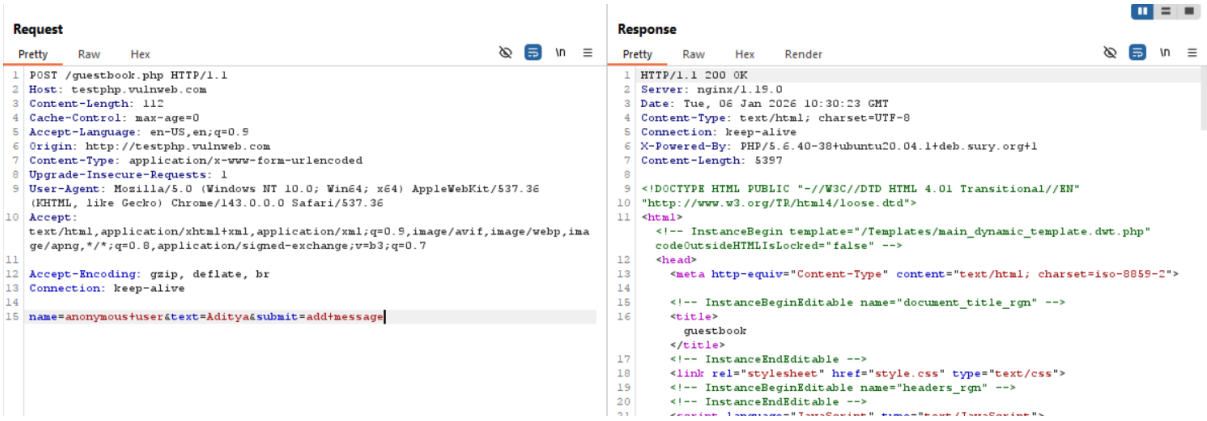


```
Request
Pretty  Raw  Hex

1  POST /guestbook.php HTTP/1.1
2  Host: testphp.vulnweb.com
3  Content-Length: 50
4  Cache-Control: max-age=0
5  Accept-Language: en-US,en;q=0.9
6  Origin: http://testphp.vulnweb.com
7  Content-Type: application/x-www-form-urlencoded
8  Upgrade-Insecure-Requests: 1
9  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
10 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,ima
   ge/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11 Referer: http://testphp.vulnweb.com/guestbook.php
12 Accept-Encoding: gzip, deflate, br
13 Connection: keep-alive
14
15 name=anonymous+user&text=Aditya&submit=add+message
```

```
Response
Pretty  Raw  Hex  Render

1  HTTP/1.1 200 OK
2  Server: nginx/1.19.0
3  Date: Tue, 06 Jan 2026 10:28:21 GMT
4  Content-Type: text/html; charset=UTF-8
5  Connection: keep-alive
6  X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1
7  Content-Length: 5411
8
9  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
10 "http://www.w3.org/TR/html4/loose.dtd">
11 <html>
     <!-- InstanceBegin template="/Templates/main_dynamic_template.dwt.php"
     codeOutsideHTMLIsLocked="false" -->
     <head>
       <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
12
13
14
15   <!-- InstanceBeginEditable name="document_title_rgn" -->
16   <title>
       guestbook
     </title>
17   <!-- InstanceEndEditable -->
18   <link rel="stylesheet" href="style.css" type="text/css">
19   <!-- InstanceBeginEditable name="headers_rgn" -->
```

HTTP 200 OK. This confirms The requests works normally.

Now, we will modify the request by modifying the Referer header: (deleting it entirely)



```
Request
Pretty  Raw  Hex

1  POST /guestbook.php HTTP/1.1
2  Host: testphp.vulnweb.com
3  Content-Length: 112
4  Cache-Control: max-age=0
5  Accept-Language: en-US,en;q=0.9
6  Origin: http://testphp.vulnweb.com
7  Content-Type: application/x-www-form-urlencoded
8  Upgrade-Insecure-Requests: 1
9  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
10 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,ima
   ge/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11
12 Accept-Encoding: gzip, deflate, br
13 Connection: keep-alive
14
15 name=anonymous+user&text=Aditya&submit=add+message
```

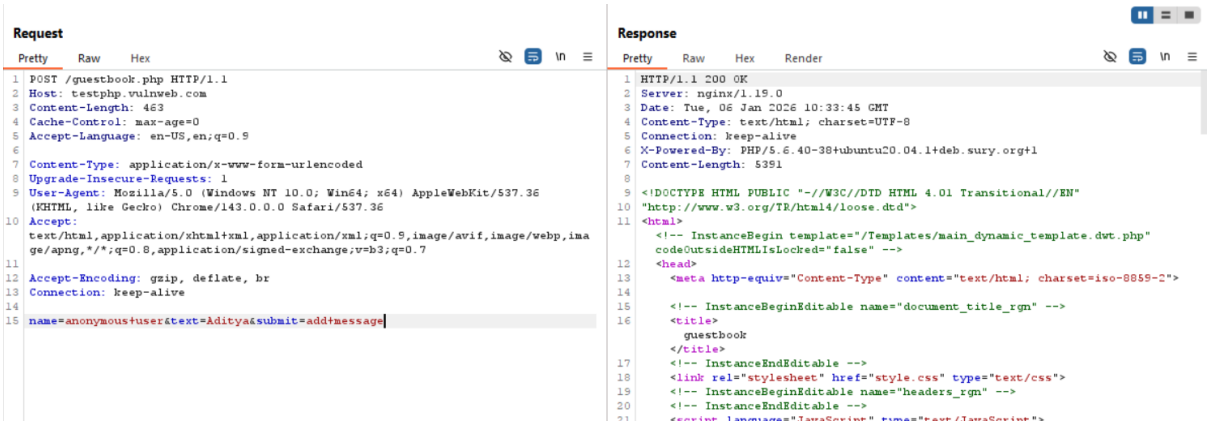
```
Response
Pretty  Raw  Hex  Render

1  HTTP/1.1 200 OK
2  Server: nginx/1.19.0
3  Date: Tue, 06 Jan 2026 10:30:23 GMT
4  Content-Type: text/html; charset=UTF-8
5  Connection: keep-alive
6  X-Powered-By: PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1
7  Content-Length: 5397
8
9  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
10 "http://www.w3.org/TR/html4/loose.dtd">
11 <html>
     <!-- InstanceBegin template="/Templates/main_dynamic_template.dwt.php"
     codeOutsideHTMLIsLocked="false" -->
     <head>
       <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
12
13
14
15   <!-- InstanceBeginEditable name="document_title_rgn" -->
16   <title>
       guestbook
     </title>
17   <!-- InstanceEndEditable -->
18   <link rel="stylesheet" href="style.css" type="text/css">
19   <!-- InstanceBeginEditable name="headers_rgn" -->
20   <!-- InstanceEndEditable -->
```

There is NO Referer header at all. Yet the server responded: HTTP/1.1 200 OK. And the guestbook entry was accepted. It means: **“**The server does NOT care where this request came from**”**.

This demonstrates:

- Missing CSRF protection
- Client-side trust
- No workflow enforcement

Even if we remove the “origin”, 200 OK is there in response.



Opaque Data:

What is visible data?

Some data is easy to read: quantity=2. Anyone looking at the request can understand what it means.

What is hidden data?

Some data is sent but not shown to the user. Example: <input type="hidden" name="price" value="500">. This data is still readable if you inspect the page.

What is opaque data?

Opaque data is data that:

- Is sent by the browser
- Cannot be understood by humans
- Looks like random letters and numbers

Example: 262a4844206559224f45...

This happens because the data is:

- Encrypted (locked with a key), or
- Obfuscated (scrambled to confuse people)

Why do websites use opaque data?

Websites use it to:

- Hide important information (like price, product ID)

- Prevent users from changing values
- Add a layer of security

Instead of sending: price=500

They send: enc=262a4844206559224f45...

What happens on the server?

When the server receives opaque data:

1) It decrypts or decodes the data
2) Turns it back into real information (like price = 500)
3) Uses it to process the order

Why is this important for security?

Even though the data looks safe:

- The server still processes the real decoded value
- That processing might have bugs or weaknesses

But:

- You can't send plain text
- You must send data in the same encrypted/scrambled format

Opaque Data Manipulation:

App.py: we created locally.

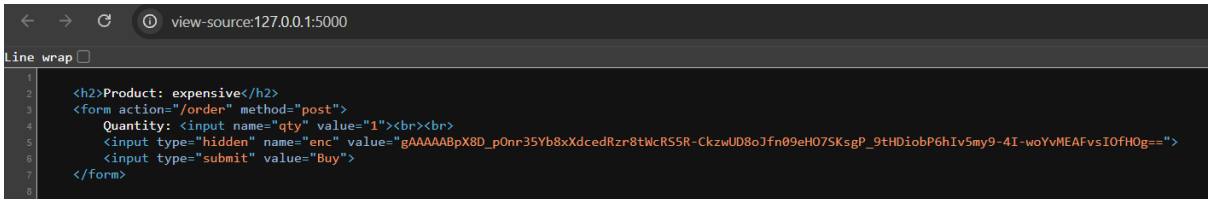
app.py > ...

```python
from flask import Flask, request, render_template_string
from cryptography.fernet import Fernet
app = Flask(__name__)
# Hardcoded key (bad practice, intentional for learning)
key = Fernet.generate_key()
cipher = Fernet(key)
# Fake product database
PRODUCTS = {
    "cheap": 100,
    "expensive": 1000
}
@app.route("/")
def index():
    product = request.args.get("product", "expensive")
    price = PRODUCTS[product]
    # Encrypt the price (opaque data)
    enc_price = cipher.encrypt(str(price).encode()).decode()
    return render_template_string("""
    <h2>Product: {{product}}</h2>
    <form action="/order" method="post">
        Quantity: <input name="qty" value="1"><br><br>
        <input type="hidden" name="enc" value="{{enc}}">
        <input type="submit" value="Buy">
    </form>
    """, enc=enc_price, product=product)
@app.route("/order", methods=["POST"])
def order():
    qty = int(request.form["qty"])
    enc = request.form["enc"]
    # Decrypt opaque value
    price = int(cipher.decrypt(enc.encode()).decode())
    total = price * qty
    return f"<h3>Total charged: ${total}</h3>"
if __name__ == "__main__":
    app.run(debug=True)
```

Step1: Run the application using Python app.py

Step2: Open the browser and visit http://127.0.0.1:5000/

Step3: See the page source.



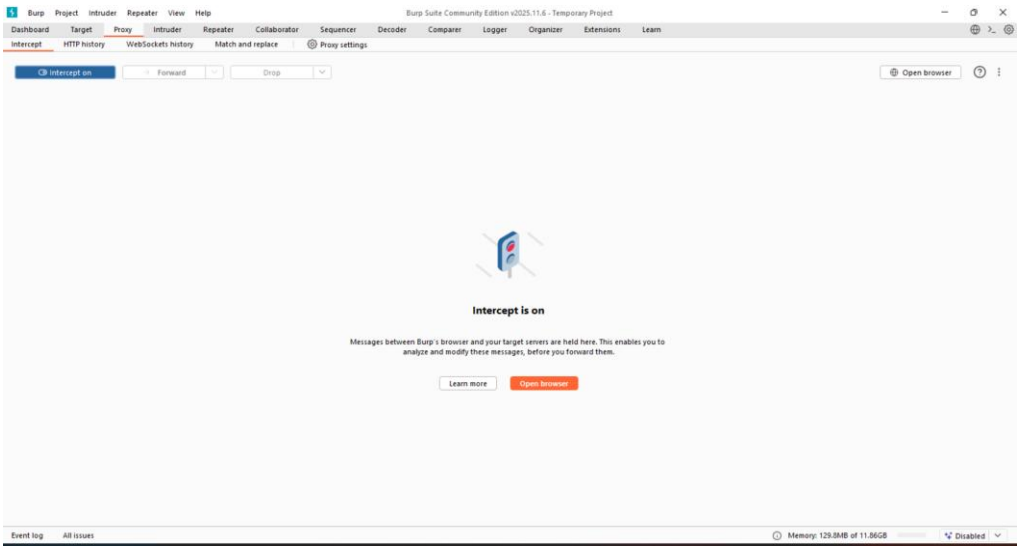
This is opaque

- You can't read the price
- You *know* it contains the price
- The server will decrypt it

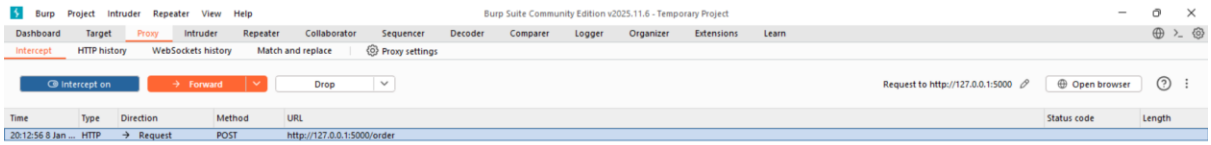
Step4: Use Burp Suite or OWASP ZAP.

- Enable intercept
- Click Buy
- Capture the POST request:

Burp:

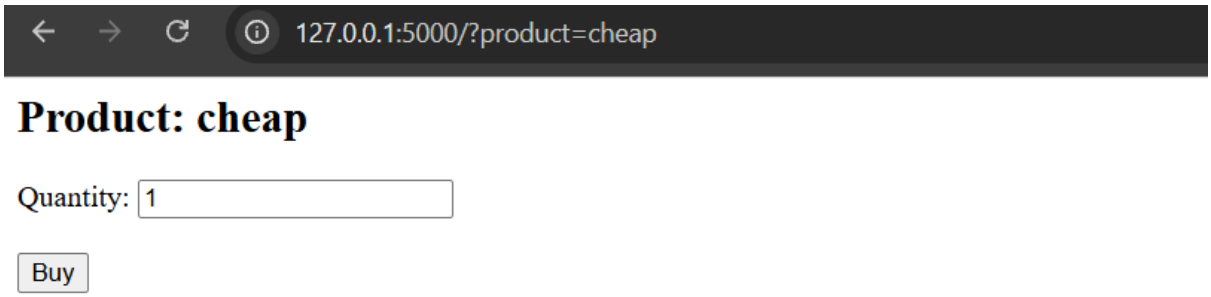


Burp when “Buy” button clicked:

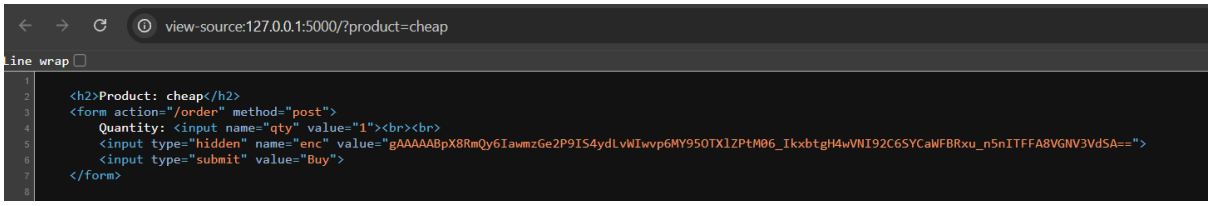


Step5: Replay Attack. We want to: Use encrypted price from a cheap product on an expensive one. Open cheap product: http://127.0.0.1:5000/?product=cheap

Browser:



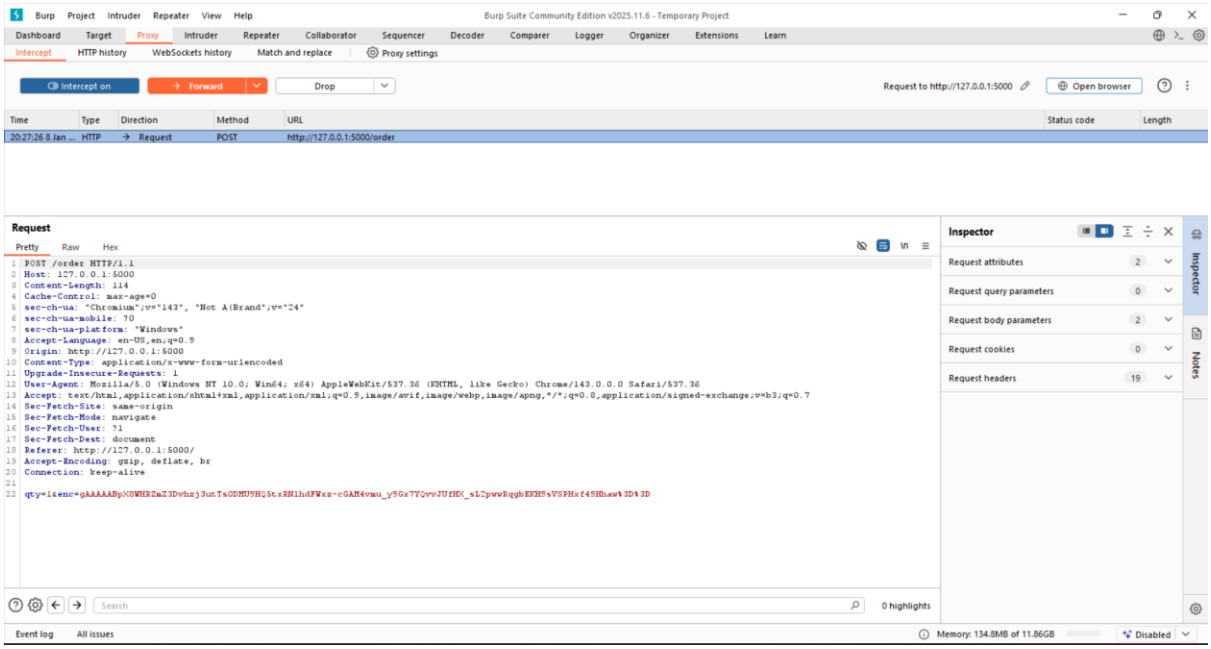
Source code: see the difference in the value.



Copy it.

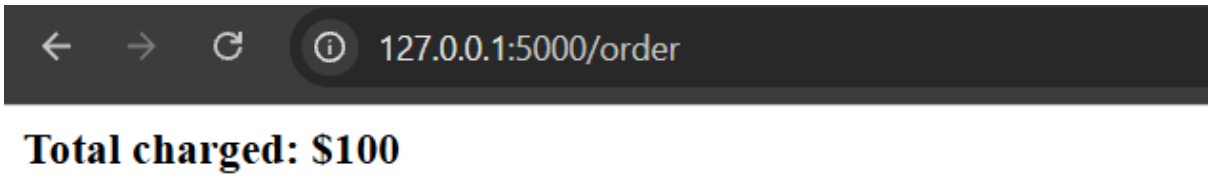
Step6: Go back to the original http://127.0.0.1:5000/ where there is high price. This can be seen in browser as:

Step7: Click on “Buy” again. This can be seen in burp as:



Change the enc.

Step8: Then forward it.



Thus, Copy the encrypted price from a cheaper product and submit it in its place.

The ASP.NET ViewState:

What is ViewState?

- ViewState is a hidden field automatically added by ASP.NET pages.
- It stores page state (UI data) between requests.
- The server sends it to the browser, and the browser sends it back on the next request.

Why ViewState exists?

- Browsers normally send back only user input, not full page data.
- ViewState allows ASP.NET to:
    1. Remember dropdown values, controls, UI state
    2. Rebuild the page without storing everything server-side
- Improves server performance.

How ViewState works?

1) Server serializes page data into ViewState
2) Sends it to the browser as a hidden field
3) Browser returns it on form submission
4) Server deserializes it and restores page state

Important: ViewState is NOT encrypted

- ViewState is usually Base64-encoded, not encrypted.
- Anyone can decode it.
- Decoding may reveal sensitive values like: price = 1224.95

ViewState stores page data on the client to preserve UI state, but if sensitive data is stored there, it can be read or tampered with unless strong protection is enabled.

Capturing User Data: HTML Forms:

HTML forms collect user input and send it to the server as name/value pairs. They often use client-side rules to restrict or validate data, but these controls run on the user's device. Because attackers can easily bypass them, client-side validation alone is not secure.

Some ways are:

1. Length limits
2. Script based validation
3. Disabled elements

Length Limits:

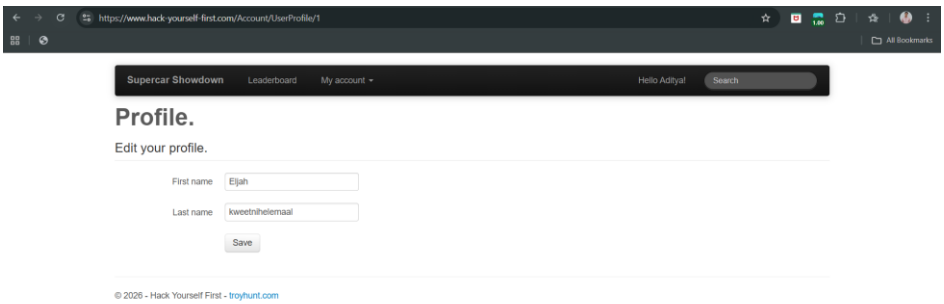
HTML length limits only restrict what users can type in the browser and can be easily bypassed. Browsers use caching and 304 responses to avoid re-downloading unchanged files. By removing cache headers, attackers can force the server to resend and modify client-side validation code.

Testing about Length Limits in forms:

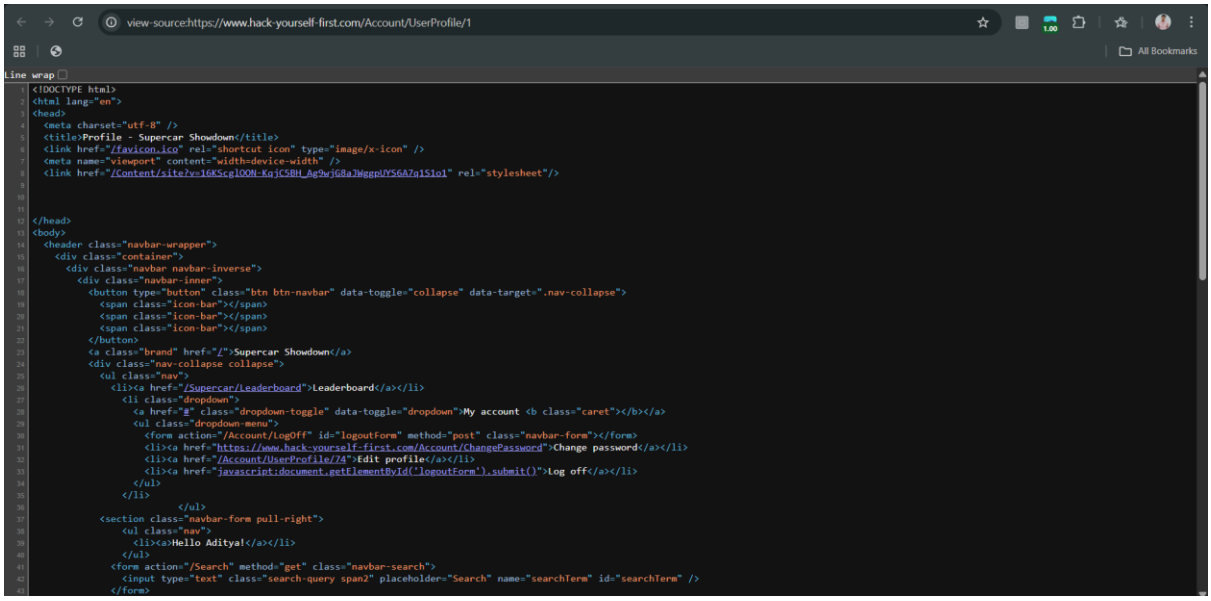
URL: https://www.hack-yourself-first.com/Account/UserProfile/1

Step1: Open the URL, following page will open:

Browser:



Now, open the page source: following screen will appear.



There is NO maxlength attribute at all.

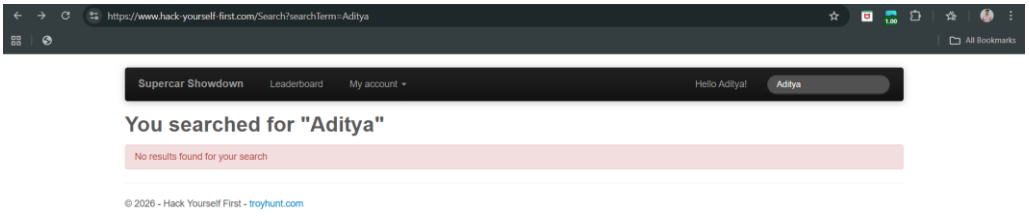
This means:

- The browser does not restrict input length
- Users can type very long messages
- All responsibility is on the server to handle size safely

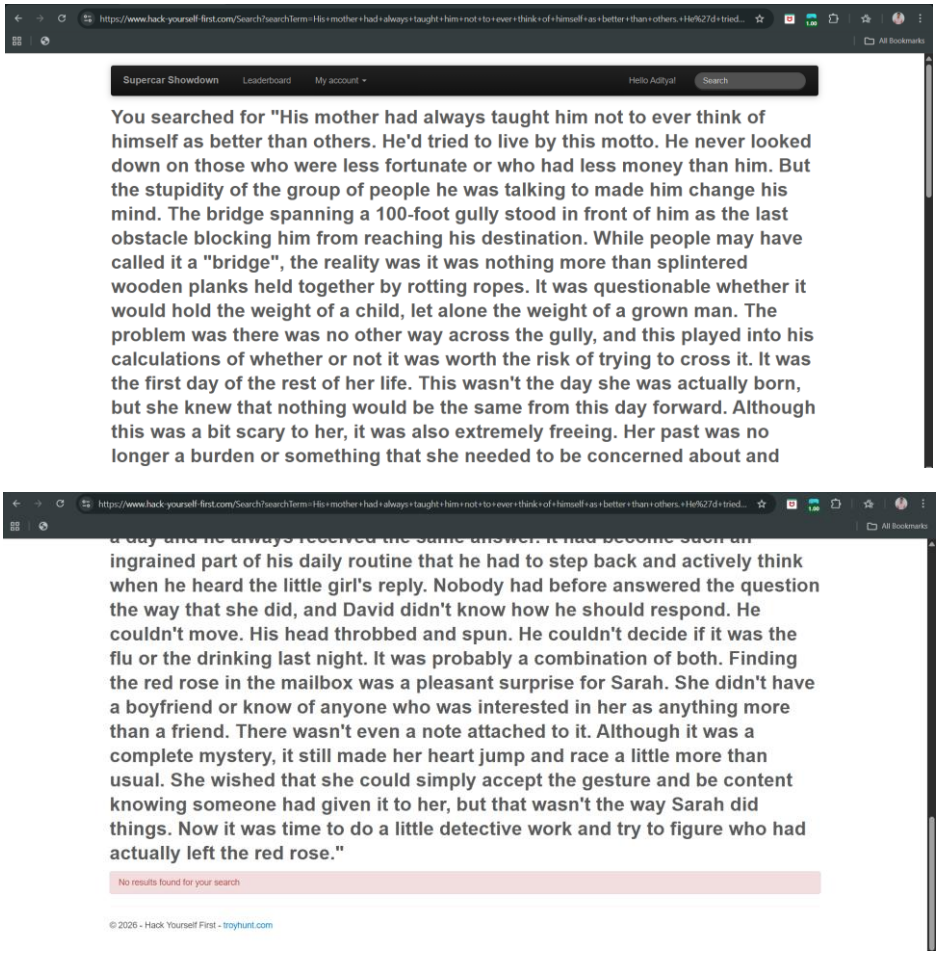
So this page is actually worse than a maxlength issue — it has no client-side length control.

Step2: So now, test direction changed to “How does the server behave when I submit a very long but valid message?”

For short input in the “Search” bar:



For large input:



Result:

- Page still loads normally
- Entire long text is reflected back in:

You searched for "VERY LONG TEXT…"

- No error
- No truncation
- No crash

Meaning: The server accepts and processes very large input.

This can lead to:

- Performance issues
- Extremely long URLs
- Memory/database stress
- Easier exploitation when combined with XSS or injection flaws

Script-based validation:

What is Script-Based Validation?

Script-based validation means checking user input using JavaScript in the browser before the form is sent to the server. Since, HTML can't do everything by itself, hence developers often use JavaScript to do more detailed checking.

This Validation Happens Only in the Browser. This is client-side validation, not server-side.

Why Is This Weak / Easy to Bypass?

1. JS can be bypassed if JS is turned off
2. Request can be modified
3. JS can be modified by itself

Script-based validation testing on a local vulnerable website:

Server.js:

```
JS server.js > ...
1   const express = require("express");
2   const bodyParser = require("body-parser");
3
4   const app = express();
5   app.use(bodyParser.urlencoded({ extended: false }));
6
7   // Serve static HTML
8   app.use(express.static("public"));
9
10  // ❌ Vulnerable order endpoint
11  app.post("/order", (req, res) => {
12    const quantity = req.body.quantity;
13    const price = req.body.price;
14
15    // ❌ NO server-side validation
16    const total = quantity * price;
17
18    res.send(`
19      <h2>Order Confirmation</h2>
20      Quantity ordered: ${quantity} <br>
21      Price per item: ${price} <br>
22      <strong>Total cost: ${total}</strong>
23    `);
24  });
25
26  app.listen(3000, () => {
27    console.log("Vulnerable app running on http://localhost:3000");
28  });
```

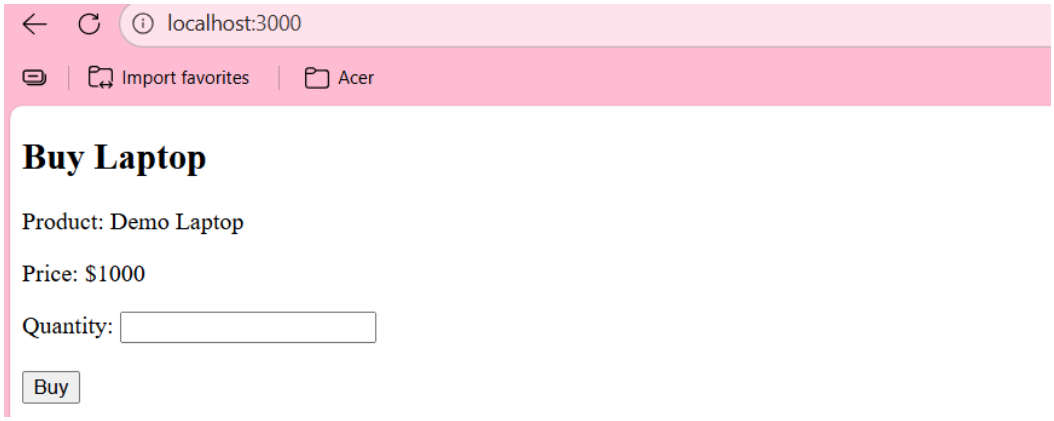
Public/index.html:

```
public > index.html > html
1   <!DOCTYPE html>
2   <html>
3   <head>
4     <title>Vulnerable Shop</title>
5     <script>
6       function validateForm() {
7         let qty = document.getElementById("quantity").value;
8         // ❌ Client-side validation ONLY
9         if (!/^\d+$/.test(qty)) {
10          alert("Quantity must be a positive number!");
11          return false;
12        }
13        return true;
14      }
15    </script>
16  </head>
17  <body>
18    <h2>Buy Laptop</h2>
19    <p>Product: Demo Laptop</p>
20    <p>Price: $1000</p>
21    <form action="/order" method="POST" onsubmit="return validateForm()">
22      Quantity:
23      <input type="text" id="quantity" name="quantity"><br><br>
24      <!-- ❌ Hidden field controlled by client -->
25      <input type="hidden" name="price" value="1000">
26      <input type="submit" value="Buy">
27    </form>
28  </body>
29  </html>
```

Terminal:

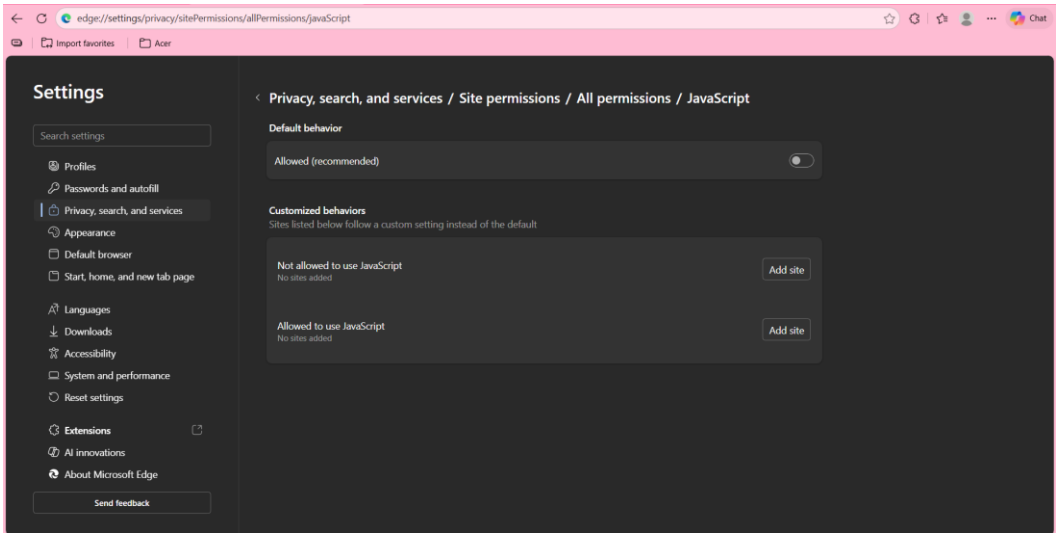
```
PS E:\Tounderstnad\vulnerable-shop> node server.js
Vulnerable app running on http://localhost:3000
```

Browser:

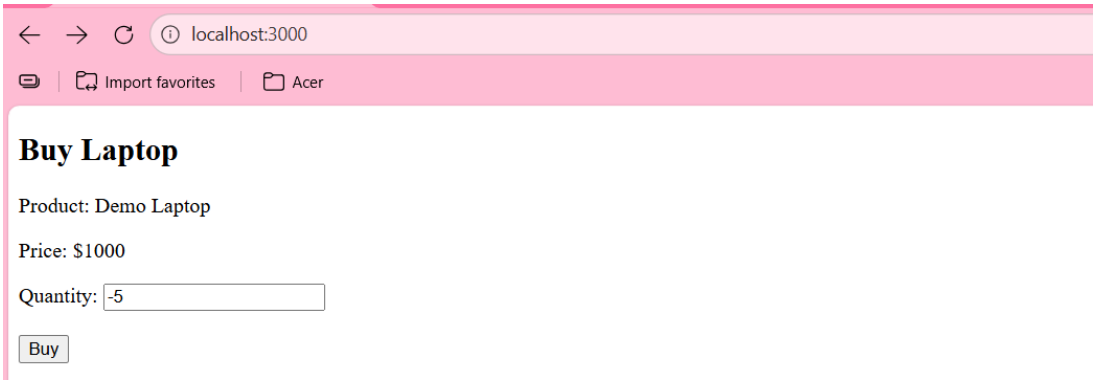


Check1: Close the JS from the browser.

Settings:



Enter any negative value:

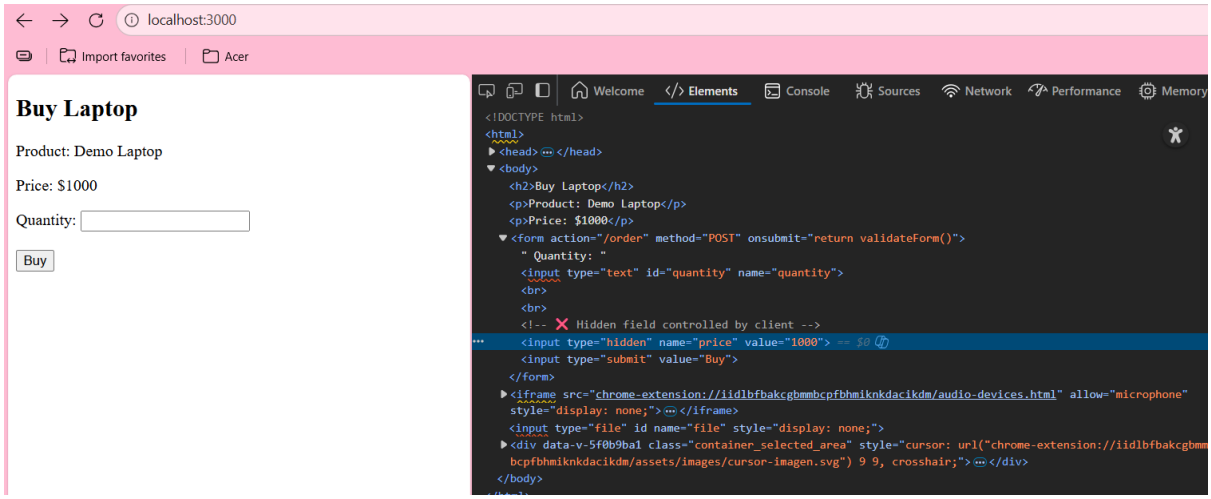


Click on buy:

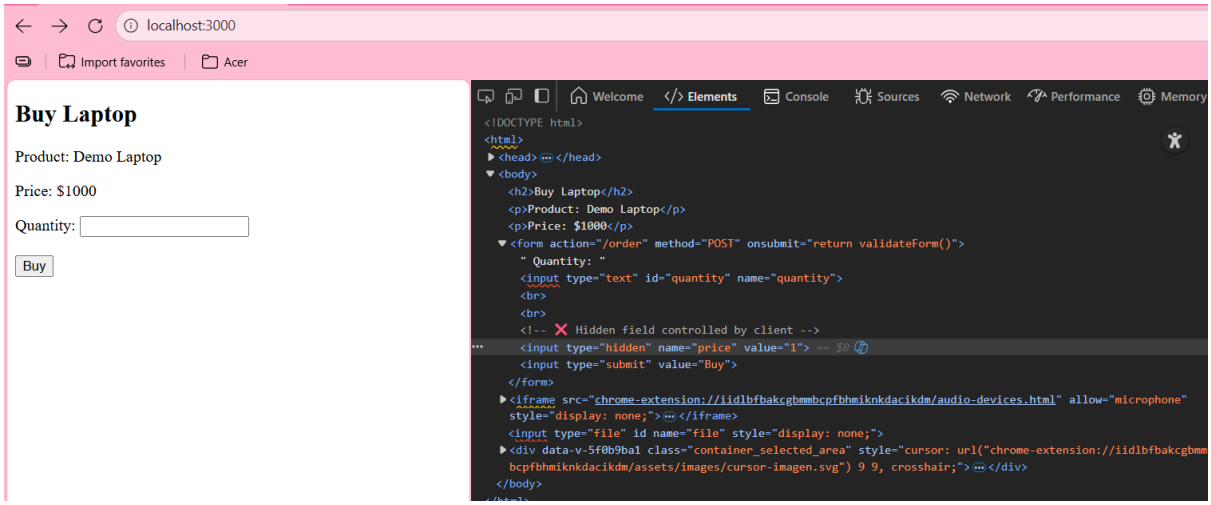
Server accepted it.

Check2: Manipulating from the inspect section.

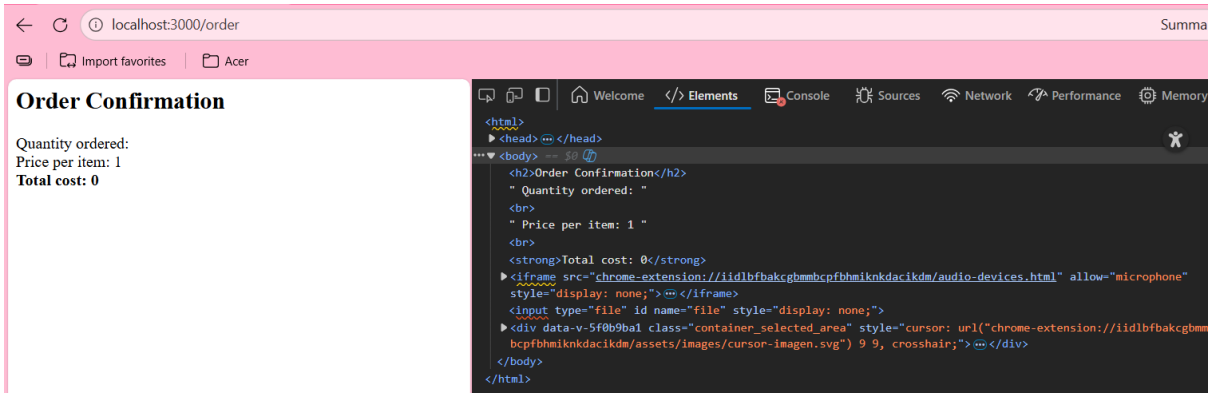
Originally: See 1000.



Manipulated:



Click on the “Buy” button: Clearly, it accepted it.



Disabled Elements:

Disabled elements are a client-side restriction only. If the server processes them when manually submitted, the application is vulnerable.

Disabled Elements vulnerability on a local website:

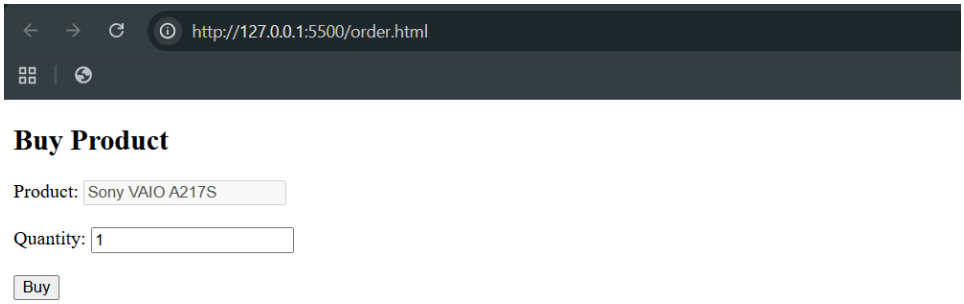
Order.html:

```
order.html > html > body > form > br
<!DOCTYPE html>
<html>
<head>
    <title>Buy Product</title>
</head>
<body>

<h2>Buy Product</h2>

<form action="/order" method="POST">

    <!-- Disabled field -->
    Product:
    <input type="text" name="product" value="Sony VAIO A217S" disabled>
    <br><br>

    Quantity:
    <input type="number" name="quantity" value="1">
    <br><br>

    <!-- Hidden field -->
    <input type="hidden" name="price" value="1200">

    <button type="submit">Buy</button>

</form>

</body>
</html>
```

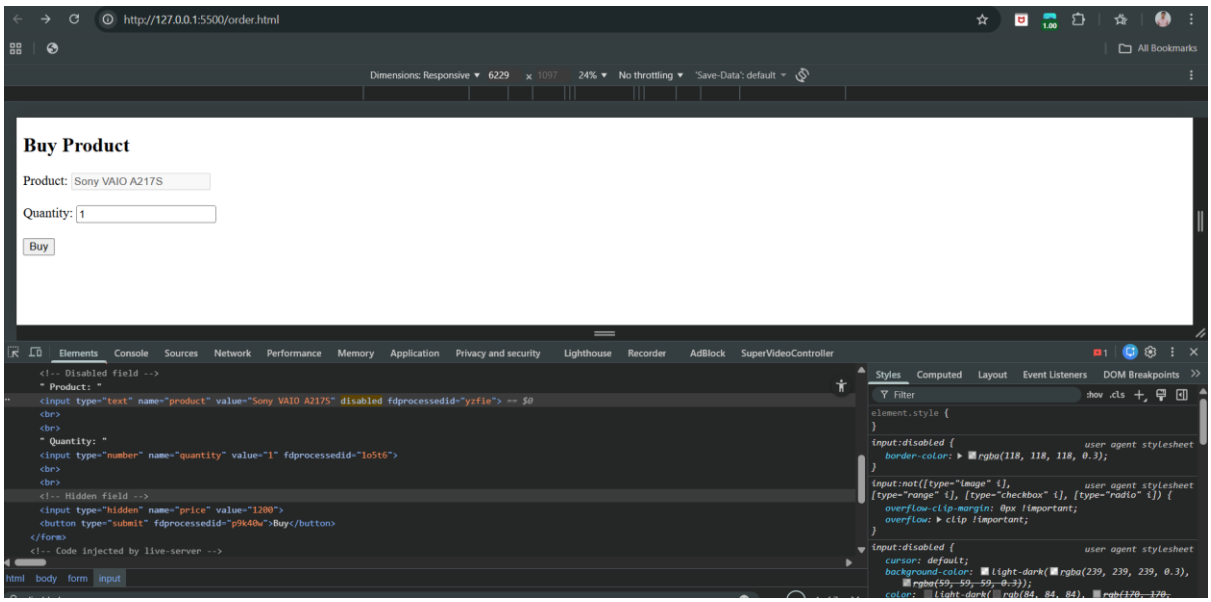
Server.js:

```
JS server.js > ...
1   // server.js (INTENTIONALLY VULNERABLE)
2   const express = require('express');
3   const bodyParser = require('body-parser');
4   const app = express();
5   app.use(bodyParser.urlencoded({ extended: false }));
6   // Serve the HTML page
7   app.get('/', (req, res) => {
8       res.sendFile(__dirname + '/order.html');
9   });
10  // Handle form submission
11  app.post('/order', (req, res) => {
12      // ❌ TRUSTING CLIENT INPUT
13      const product  = req.body.product;
14      const quantity = req.body.quantity;
15      const price    = req.body.price;
16      const total = quantity * price;
17      res.send(`
18          <h2>Order Details</h2>
19          Product: ${product}<br>
20          Quantity: ${quantity}<br>
21          Price: $${price}<br>
22          Total: $${total}
23      `);
24  });
25  app.listen(3000, () => {
26      console.log('Server running on http://localhost:3000');
27  });
```

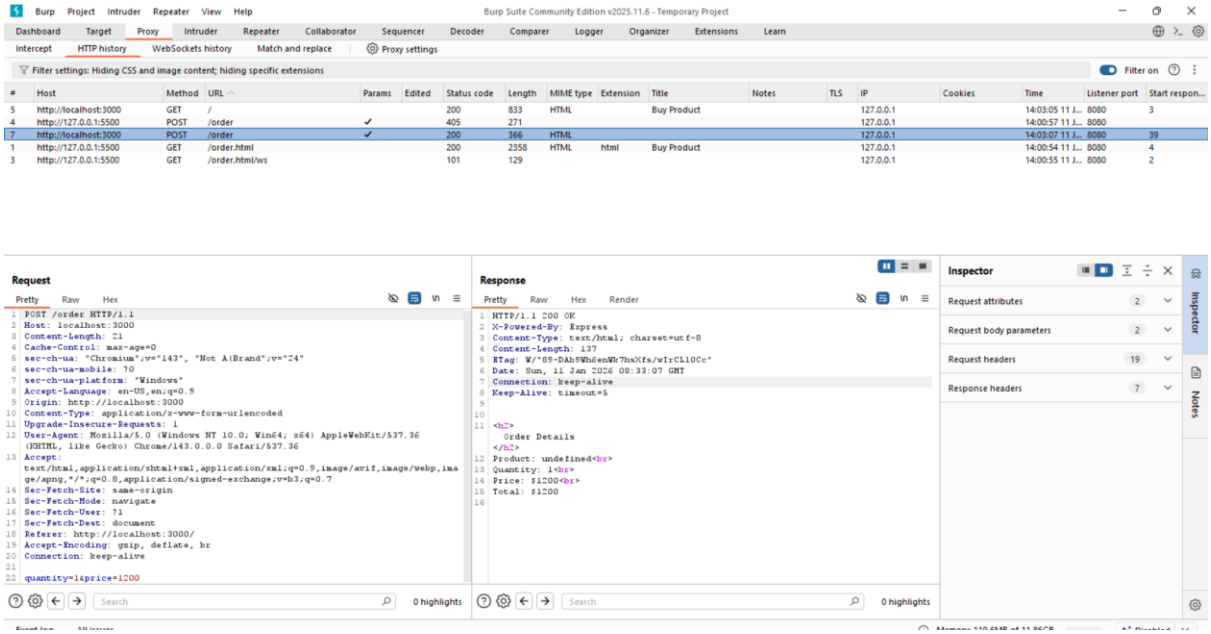
Output:



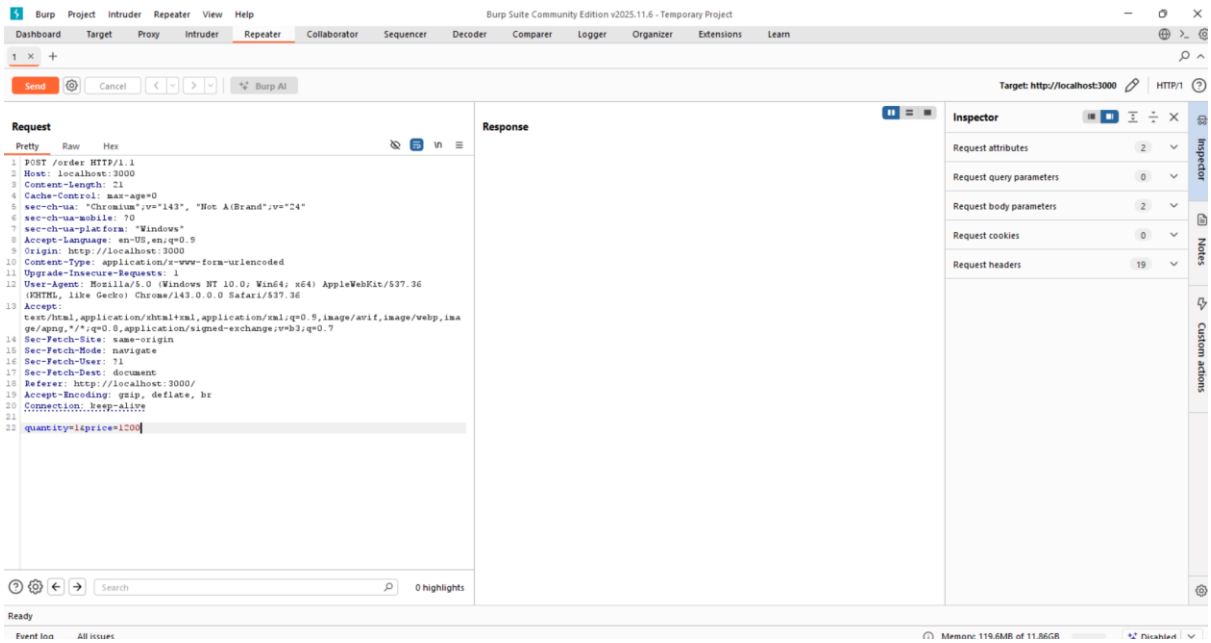
Step1: Open the inspect section of the page to test, and look for word “disabled”.



Step2: open the burp and visit the page, and click on the “Buy” button. See the proxy-> HTTP History.



Step3: Send it to the “repeater”. It can be seen in repeater like this:



Step4: Manually modify the request body.

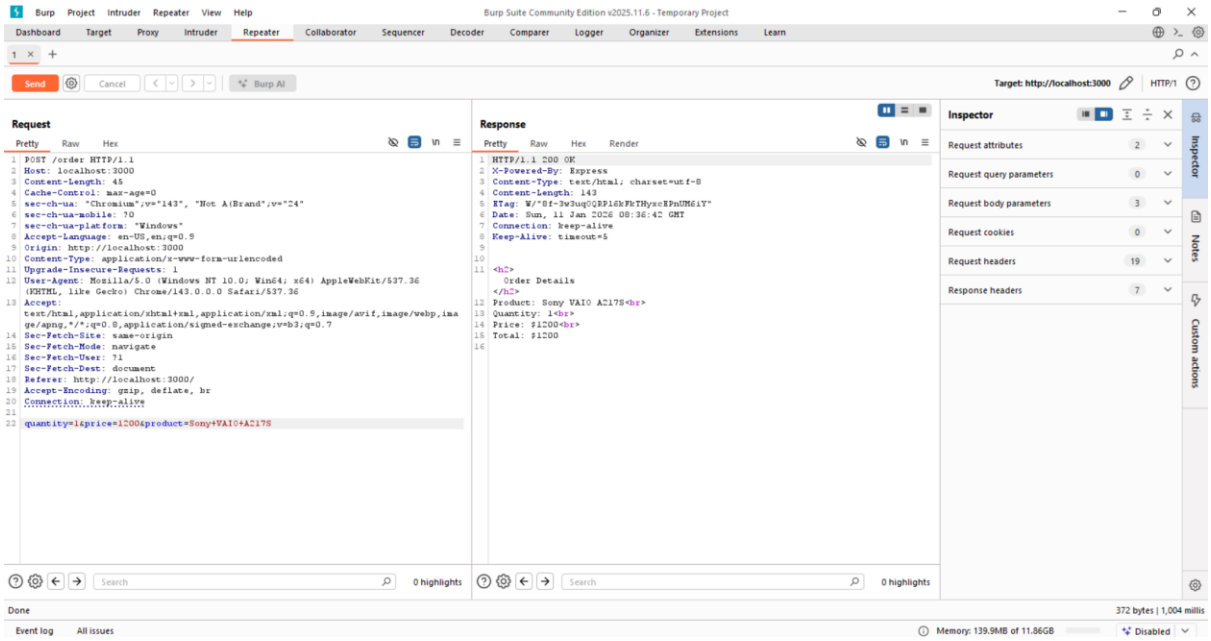
New request:

Request

Pretty Raw Hex

```
1  POST /order HTTP/1.1
2  Host: localhost:3000
3  Content-Length: 21
4  Cache-Control: max-age=0
5  sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
6  sec-ch-ua-mobile: ?0
7  sec-ch-ua-platform: "Windows"
8  Accept-Language: en-US,en;q=0.9
9  Origin: http://localhost:3000
10 Content-Type: application/x-www-form-urlencoded
11 Upgrade-Insecure-Requests: 1
12 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
13 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,ima
   ge/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
14 Sec-Fetch-Site: same-origin
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-User: ?1
17 Sec-Fetch-Dest: document
18 Referer: http://localhost:3000/
19 Accept-Encoding: gzip, deflate, br
20 Connection: keep-alive
21
22 quantity=1&price=1200&product=Sony+VAIO+A217S
```

Send it: following response can be seen.



Thus, Trusted client-side control, hence vulnerable.

This proves Disabled Elements Bypass:

- Disabled field was successfully submitted
- Server processed it
- Client-side control completely bypassed

Capturing User Data: Thick-Client Components:

Thick-client components are advanced tools used in web applications to collect user data, such as Java applets, ActiveX controls, and Flash object. They can validate and process data on the user's side before sending it to the server, which often leads developers to trust them too much. However, if the data is sent in a clear and readable form, it can still be intercepted and modified, making client-side validation easy to bypass.

Security improves when the data is obfuscated before transmission, but changing such data usually causes the server to reject it. Bypassing these protections then requires reverse-engineering the component, which shows that thick-client components can still introduce serious security vulnerabilities.

Java Applets:

What are Java Applets?

Java applets are small programs that run inside a web browser. They are often used to build thick-client components because:

- They work on many operating systems (cross-platform)
- They run in a sandbox, which means they are restricted and cannot easily harm the user's computer

Because of this sandbox, Java applets usually cannot access files or system settings. Their main job is to handle things inside the browser, like collecting user input or running small programs such as games. Moreover, keep in mind Java Applets are deprecated.

Note: Java Applets, Flash Objects and ActiveX controls are deprecated.

Handling Client-Side Data Securely:

Transmitting Data via the Client:

Web applications should never trust the client with critical data such as prices or discounts. All important values should be stored and calculated on the server. If critical data must be sent to the client, it should be properly encrypted or signed, while avoiding replay attacks and cryptographic weaknesses. Special care must be taken in frameworks like ASP.NET, where features such as ViewState can expose sensitive data if used incorrectly.

Validating Client-Generated Data:

Client-generated data can never be securely validated on the client, because all client-side controls can eventually be bypassed. Simple methods like HTML and JavaScript offer no security, and even complex or obfuscated client-side logic only slows attackers down. The only secure approach is to validate all data on the server and treat client input as untrusted. However, client-side controls still have legitimate uses, such as improving usability, preventing certain client-side attacks, and securely transmitting encrypted data when implemented correctly.

Logging and Alerting:

Client-side validation should be combined with server-side logging and alerting to help detect suspicious behaviour. If the server receives input that should have been blocked on the client side, it may indicate an attempt to bypass controls and should be logged and possibly trigger alerts. Applications can respond by monitoring activity, ending sessions, or suspending accounts. However, systems must be careful to avoid false positives, especially when users have JavaScript disabled, since client-side validation may not run in those cases.