

Chapter 5



WEB APPLICATION SECURITY

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:

```
index.html > html > body > p#result
1  <!DOCTYPE html>
2  <html>
3  <head>
4  | <title>Vulnerable Shop</title>
5  | <link rel="stylesheet" href="style.css">
6  </head>
7  <body>
8  <h2>Buy Laptop</h2>
9  <form id="orderForm">
10 | Quantity:
11 | <input type="number" id="quantity" value="1"><br><br>
12 | <!-- Hidden price field -->
13 | <input type="hidden" id="price" value="1000">
14 | <button type="submit">Buy</button>
15 </form>
16 <p id="result"></p>
17 <script src="vulnerable.js"></script>
18 </body>
19 </html>
```

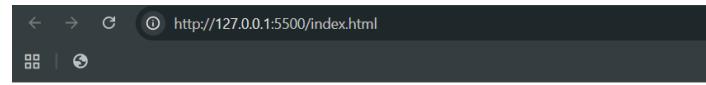
Style.css:

```
style.css > ...
1  body {
2  |   font-family: Arial, sans-serif;
3  |   padding: 40px;
4  }
5
6  button {
7  |   padding: 8px 16px;
8  |   cursor: pointer;
9  }
```

Vulnerable.js:

```
vulnerable.js > ...
1  document.getElementById("orderForm").addEventListener("submit", function (e) {
2  |   e.preventDefault();
3
4  const quantity = Number(document.getElementById("quantity").value);
5  const price = Number(document.getElementById("price").value); // ✖ trusting client
6
7  const total = quantity * price;
8
9  document.getElementById("result").innerText =
10 |   "Order placed! Total amount: $" + total;
11});
```

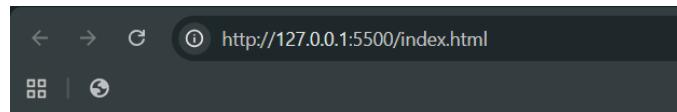
Output:



Buy Laptop

Quantity:

Output: when we clicked the “buy” button.



Buy Laptop

Quantity:

Order placed! Total amount: \$1000

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

```
Console was cleared
< undefined
> document.getElementById("price").value = -1000
< -1000
```

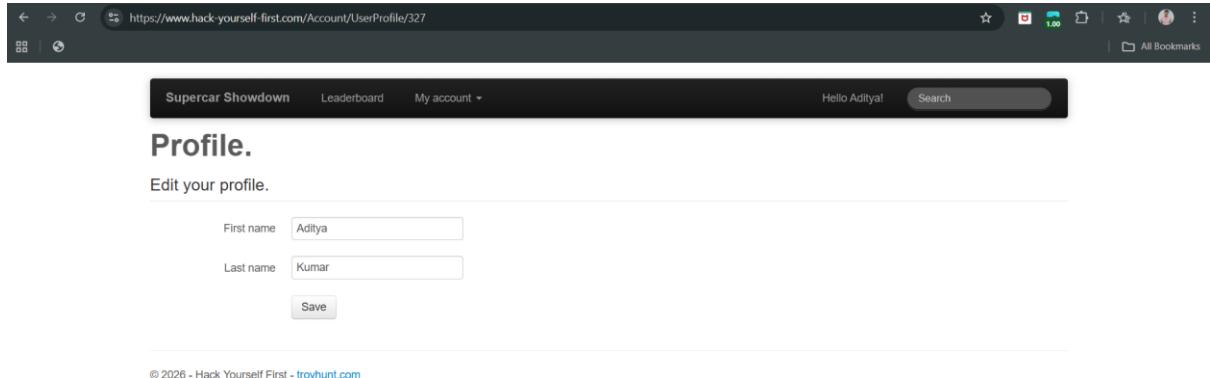
Why this is vulnerable?

- 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:

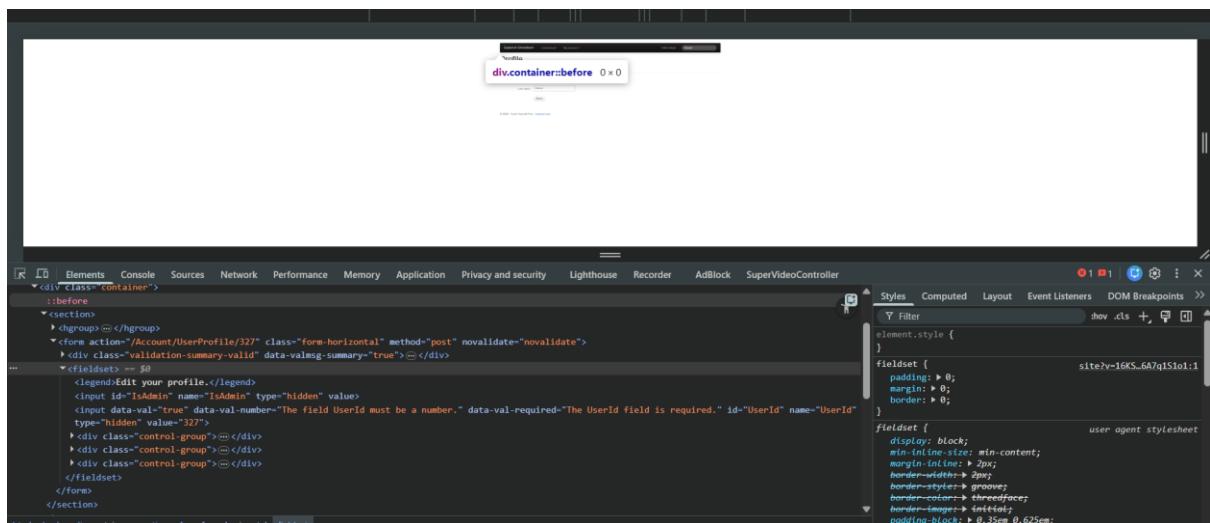
Tested on <https://www.hack-yourself-first.com/>

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



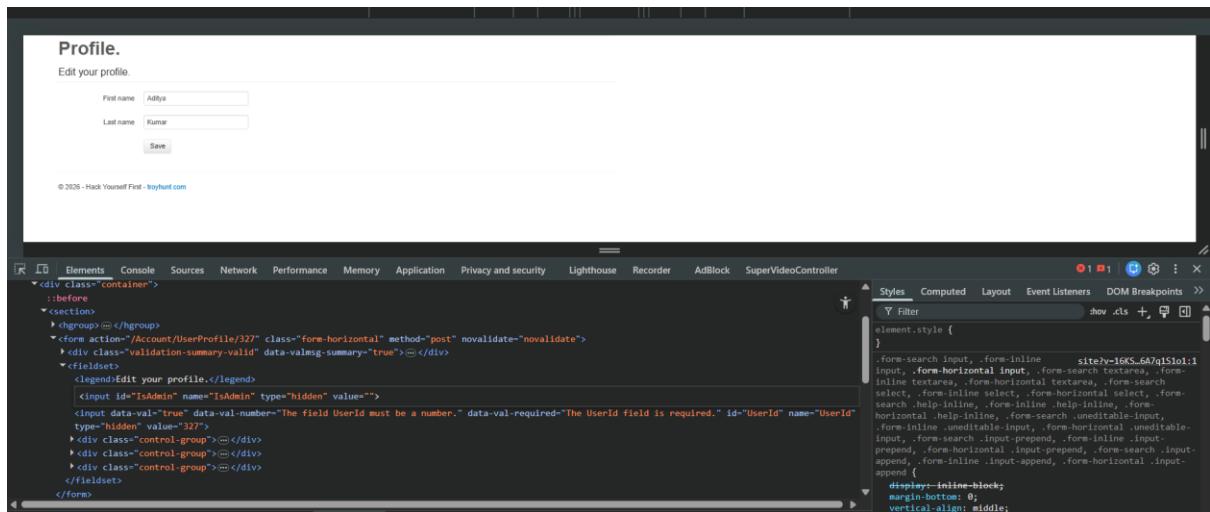
The screenshot shows a web browser window with the URL <https://www.hack-yourself-first.com/Account/UserProfile/327>. The page title is "Profile". The content area contains a form with two input fields: "First name" (Aditya) and "Last name" (Kumar). Below the form is a "Save" button. At the bottom of the page, there is a copyright notice: "© 2026 - Hack Yourself First - troyhunt.com".

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



The screenshot shows a browser developer tools window with the "Elements" tab selected. The DOM tree shows a `<div class="container">` element with a `::before` pseudo-element. The styles panel shows the CSS for the `element.style` and `fieldset` rules. The `element.style` rule includes properties like `padding: 0`, `margin: 0`, and `border: 0`. The `fieldset` rule includes properties like `display: block`, `min-inline-size: min-content`, `margin-top: 2px`, `border-width: 1px`, `border-style: groove`, `border-color: #3399FF`, and `border-image: none`.

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

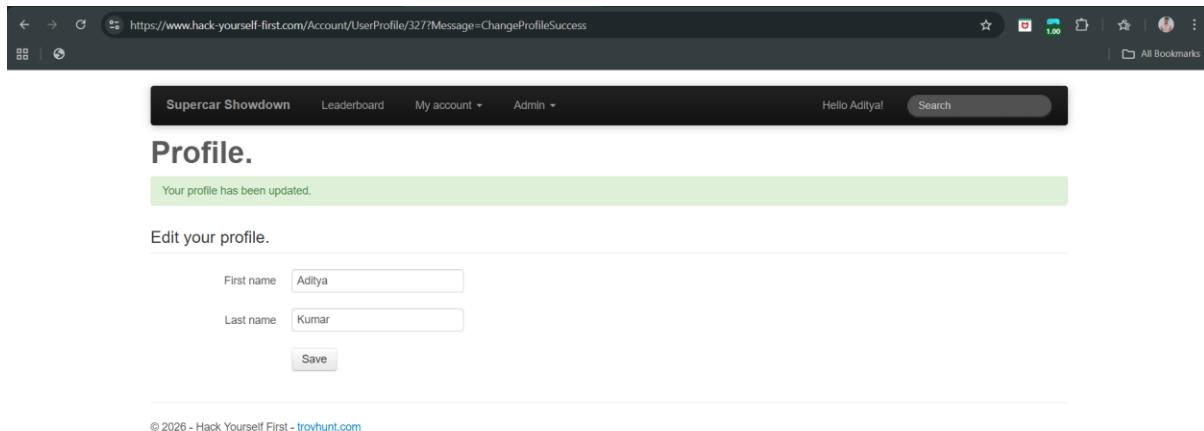


The screenshot shows a browser developer tools window with the "Elements" tab selected. The DOM tree highlights a `<input id="IsAdmin" name="IsAdmin" type="hidden" value="">` element. The styles panel shows the CSS for the `input` element, which includes properties like `display: inline-block`, `margin-bottom: 0`, and `vertical-align: middle`.

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'.

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS	IP	Cookies	Time	Listener port	Start response...
1	http://testphp.vulnweb.com	GET	/			200	5180	HTML		Home of Acunetix Art			44.228.249.3		10:10:09 4.Jan. 8080	417	
2	http://testphp.vulnweb.com	GET	/cart.php			200	5125	HTML	php	you cart			44.228.249.3		10:10:13 4.Jan. 8080	354	
3	http://testphp.vulnweb.com	GET	/login.php			200	5745	HTML	php	login page			44.228.249.3		10:10:14 4.Jan. 8080	407	
4	http://testphp.vulnweb.com	POST	/userinfo.php		✓	200	6216	HTML	php	user info			44.228.249.3	login=test%2Ftest	10:10:20 4.Jan. 8080	392	

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 screenshot shows the Burp Suite interface with the "Repeater" tab selected. In the "Request" pane, there is a single line of text representing a forwarded HTTP request. In the "Response" pane, there is no visible response. To the right, the "Inspector" pane displays various request details: attributes (2), query parameters (0), body parameters (0), cookies (1), and headers (9). The "Notes" and "Custom actions" sections are empty.

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.

The screenshot shows the Burp Suite interface with the "Repeater" tab selected. The "Request" pane contains a modified version of the previous request, where the "Cookie" field has been changed from "login=test%2Ftest" to "login=admin%2Fadmin". The "Response" pane is still empty. The "Inspector" pane shows the updated request details.

Step7: Send it...

The screenshot shows the Burp Suite interface with the "Repeater" tab selected. The "Request" pane shows the modified request. The "Response" pane now displays the server's response, which includes HTML content and some JavaScript. The "Inspector" pane shows the updated request details. At the bottom, the status bar indicates "6,375 bytes | 286 millis".

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:

The screenshot shows the Burp Suite interface with the Repeater tab selected. The Request pane displays a complex GET request to 'www.hack-yourself-first.com'. The Response pane shows the server's response, which includes a large JSON payload. The Inspector tab is open, showing detailed information about the request attributes, query parameters, body parameters, and headers. The status bar at the bottom indicates 10,662 bytes transferred in 1,204 milliseconds.

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

The screenshot shows the Burp Suite interface with the Request tab selected. The request has been modified to include 'IsAdmin=true' in the URL. The rest of the request is identical to the one in Step 3. The status bar at the bottom indicates 10,662 bytes transferred in 1,204 milliseconds.

When we clicked the send button:

The screenshot shows the Burp Suite interface with the Response tab selected. The response is identical to the one in Step 3, indicating that the modification did not change the response. The status bar at the bottom indicates 10,659 bytes transferred in 1,122 milliseconds.

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/>

The screenshot shows a web browser window with the following details:

- Address Bar:** Not secure http://testphp.vulnweb.com
- Title Bar:** TEST and Demonstration site for Acunetix Web Vulnerability Scanner
- Content Area:**
 - Search bar: search art go
 - Sidebar menu:
 - Browse categories
 - Browse artists
 - Your cart
 - Signup
 - Your profile
 - Our guestbook
 - AJAX Demo
 - Image: A puzzle piece icon.
 - Warning message: "Warning: This is not a real shop. This is an example PHP application, which is intentionally vulnerable to web attacks. It is intended to help you test Acunetix. It also helps you understand how developer errors and bad configuration may let someone break into your website. You can use it to test other tools and your manual hacking skills as well. Tip: Look for potential SQL Injections, Cross-site Scripting (XSS), and Cross-site Request Forgery (CSRF), and more."
- Footer:** About Us | Privacy Policy | Contact Us | Shop | HTTP Parameter Pollution | ©2010 Acunetix Ltd

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

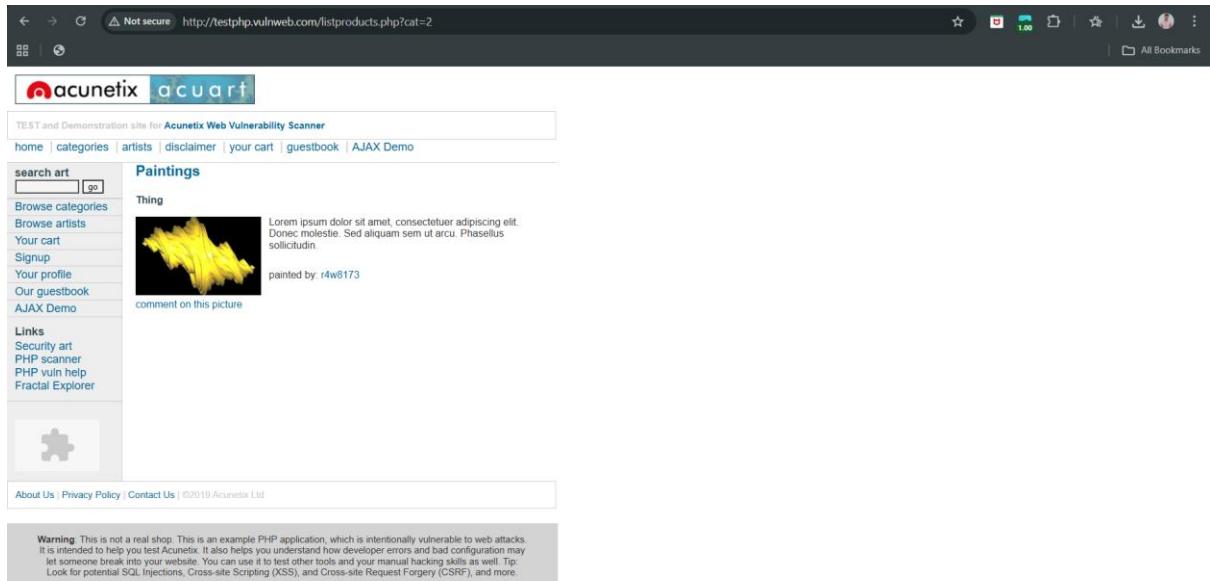
The screenshot shows a web browser window with the following details:

- Address Bar:** Not secure http://testphp.vulnweb.com/listproducts.php?cat=1
- Title Bar:** TEST and Demonstration site for Acunetix Web Vulnerability Scanner
- Content Area:**
 - Search bar: search art go
 - Sidebar menu:
 - Browse categories
 - Browse artists
 - Your cart
 - Signup
 - Your profile
 - Our guestbook
 - AJAX Demo
 - Image: A puzzle piece icon.
 - Section header: Posters
 - Item 1: The shore
 - Thumbnail: A yellow and black abstract painting.
 - Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec molestie. Sed aliquam sem ut arcu.
 - Author: painted by r4w8173
 - Action: comment on this picture
 - Item 2: Mystery
 - Thumbnail: A red abstract painting.
 - Description: Donec molestie. Sed aliquam sem ut arcu.
 - Author: painted by r4w8173
 - Action: comment on this picture
 - Item 3: The universe
 - Thumbnail: A blue abstract painting.
 - Description: Lorem ipsum dolor sit amet. Donec molestie. Sed aliquam sem ut arcu.
 - Author: painted by r4w8173
 - Action: comment on this picture
 - Item 4: Walking
 - Thumbnail: A blue abstract painting.
 - Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit.
 - Author:
 - Action:

Here:

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

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



The screenshot shows a web browser window with the URL `http://testphp.vulnweb.com/listproducts.php?cat=2`. The page content is identical to the original 'Paintings' page, displaying a yellow fractal image titled 'Thing' with a caption about ipsum dolor sit amet. A sidebar on the left contains links like 'search art', 'Browse categories', and 'Links'. A warning banner at the bottom states: 'Warning: This is not a real shop. This is an example PHP application, which is intentionally vulnerable to web attacks. It is intended to help you test Acunetix. It also helps you understand how developer errors and bad configuration may let someone break into your website. You can use it to test other tools and your manual hacking skills as well. Tip: Look for potential SQL Injections, Cross-site Scripting (XSS), and Cross-site Request Forgery (CSRF), and more.'

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.

```
[root@kali]# wfuzz -c -z range,1-20 "http://testphp.vulnweb.com/listproducts.php?cat=FUZZ"
```

Output:

```
[root@kali]# wfuzz -c -z range,1-20 "http://testphp.vulnweb.com/listproducts.php?cat=FUZZ"
[...]
[*] wFuzz 3.1.0 - The Web Fuzzer
[*] http://testphp.vulnweb.com/listproducts.php?cat=FUZZ
Total requests: 20

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

000000020: 200      102 L   358 W   4699 Ch  "28"
000000017: 200      102 L   358 W   4699 Ch  "17"
000000019: 200      102 L   358 W   4699 Ch  "19"
000000014: 200      102 L   358 W   4699 Ch  "14"
000000008: 200      102 L   358 W   4699 Ch  "7"
000000015: 200      102 L   358 W   4699 Ch  "15"
000000016: 200      102 L   358 W   4699 Ch  "16"
000000018: 200      102 L   358 W   4699 Ch  "18"
000000001: 200      107 L   526 W   7880 Ch  "1"
000000003: 200      102 L   358 W   4699 Ch  "3"
000000006: 200      102 L   358 W   4699 Ch  "6"
000000012: 200      102 L   358 W   4699 Ch  "12"
000000013: 200      102 L   358 W   4699 Ch  "13"
000000011: 200      102 L   358 W   4699 Ch  "11"
000000010: 200      102 L   358 W   4699 Ch  "10"
00000009: 200      102 L   358 W   4699 Ch  "9"
00000008: 200      102 L   358 W   4699 Ch  "8"
00000007: 200      104 L   304 W   5311 Ch  "7"
00000004: 200      102 L   358 W   4699 Ch  "4"
00000005: 200      102 L   358 W   4699 Ch  "5"

Total time: 0
Processed Requests: 20
Filtered Requests: 0
Requests/sec.: 0
```

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
```



```
(root㉿kali)-[~]
# paramspider -d testphp.vulnweb.com
```

Output:



```
(root㉿kali)-[~]
# paramspider -d testphp.vulnweb.com

with <3 by @0xasm0d3us

[INFO] Fetching URLs for testphp.vulnweb.com
[INFO] Found 11732 URLs for testphp.vulnweb.com
[INFO] Cleaning URLs for testphp.vulnweb.com
[INFO] Found 890 URLs after cleaning
[INFO] Extracting URLs with parameters
[INFO] Saved cleaned URLs to results/testphp.vulnweb.com.txt
```

Step2: View what it found



```
(root㉿kali)-[~]
# cat results/testphp.vulnweb.com.txt

http://testphp.vulnweb.com/bxs/vuin.php?id=FUZZ
http://testphp.vulnweb.com/ipp/params.php?tp=FUZZ
http://testphp.vulnweb.com/ipp/params.php?tp=FUZZ
http://testphp.vulnweb.com/redirect.php?r=FUZZ&user=FUZZ
http://testphp.vulnweb.com/redirect.php?r=FUZZ&user=FUZZ
http://testphp.vulnweb.com/comment.php?pid=FUZZ
http://testphp.vulnweb.com/listproducts.php?artist=FUZZ&cat=FUZZ
http://testphp.vulnweb.com/search.php?cookie=FUZZ
http://testphp.vulnweb.com/?z3Fcnd=FUZZ
http://testphp.vulnweb.com/listproducts.php?cat=FUZZ&artist=FUZZ
http://testphp.vulnweb.com/search.php?test=FUZZ&cat=FUZZ&pid=FUZZ
http://testphp.vulnweb.com/index.php?id=FUZZ&user=FUZZ&gol=FUZZ
http://testphp.vulnweb.com/Mod_Rewrite_Shop/details.php?id=FUZZ
http://testphp.vulnweb.com/redirect.php?r=FUZZ
http://testphp.vulnweb.com/AJAX/infoartist.php?id=FUZZ&VneN=FUZZ
http://testphp.vulnweb.com/ipp/index.php?pp=FUZZ
http://testphp.vulnweb.com/listproducts.php?cat=FUZZ
http://testphp.vulnweb.com/search?q=FUZZ
http://testphp.vulnweb.com/Login.php?id=FUZZ
http://testphp.vulnweb.com/XCB93E286592listproducts.php?cat=FUZZ
http://testphp.vulnweb.com/ipp/params.php?aaaaa2f=FUZZ&pp=FUZZ
http://testphp.vulnweb.com/Mod_Rewrite_Shop/BuyProduct-1/3L/wp-content/plugins/flexible-custom-post-type/edit-post.php?id=FUZZ
http://testphp.vulnweb.com/ipp/params.php?tp=FUZZ&pp=FUZZ
http://testphp.vulnweb.com/trk=FUZZ
http://testphp.vulnweb.com/Mod_Rewrite_Shop/Details/web-camera-a4tech/2/wp-content/plugins/flexible-custom-post-type/edit-post.php?id=FUZZ
http://testphp.vulnweb.com/AJAX/infoartist.php?id=FUZZ&lhns=FUZZ
http://testphp.vulnweb.com/AJAX/infoartist.php?id=FUZZ
http://testphp.vulnweb.com/product.php?pid=FUZZ&hkh1=FUZZ
http://testphp.vulnweb.com/search.php?cat=FUZZ&pid=FUZZ&test=FUZZ
http://testphp.vulnweb.com/Mod_Rewrite_Shop/BuyProduct-1/3L/wp-content/plugins/flexible-custom-post-type/edit-post.php?id=FUZZ
http://testphp.vulnweb.com/Mod_Rewrite_Shop/Details/web-camera-a4tech/2/wp-content/plugins/flexible-custom-post-type/edit-post.php?id=FUZZ
http://testphp.vulnweb.com/showimage.php?FEXC281le=FUZZ
http://testphp.vulnweb.com/product.php?pid=FUZZ&tdRkI=FUZZ
```

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?qXf=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)-[~] Analy... social-engi... repos
└─# 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
=====
00000003: 200      109 L  388 W  4958 Ch  "23"
00000015: 200      109 L  388 W  4958 Ch  "15"
00000024: 200      109 L  388 W  4958 Ch  "24"
00000028: 200      109 L  388 W  4958 Ch  "28"
00000025: 200      109 L  388 W  4958 Ch  "25"
00000020: 200      109 L  388 W  4958 Ch  "26"
00000027: 200      109 L  388 W  4958 Ch  "27"
00000007: 200      109 L  388 W  4958 Ch  "7"
00000003: 200      109 L  388 W  4958 Ch  "3"
00000001: 200      109 L  388 W  4958 Ch  "1"
00000022: 200      109 L  388 W  4958 Ch  "22"
00000021: 200      109 L  388 W  4958 Ch  "21"
00000020: 200      109 L  388 W  4958 Ch  "20"
00000019: 200      109 L  388 W  4958 Ch  "19"
00000018: 200      109 L  388 W  4958 Ch  "18"
00000017: 200      109 L  388 W  4958 Ch  "17"
00000016: 200      109 L  388 W  4958 Ch  "16"
00000014: 200      109 L  388 W  4958 Ch  "14"
00000013: 200      109 L  388 W  4958 Ch  "13"
00000012: 200      109 L  388 W  4958 Ch  "12"
00000011: 200      109 L  388 W  4958 Ch  "11"
00000010: 200      109 L  388 W  4958 Ch  "10"
00000009: 200      109 L  388 W  4958 Ch  "9"
00000006: 200      109 L  388 W  4958 Ch  "6"
```

POST does not protect preset URL parameters.:.

```
[root@kali] ~ [~]
└─* wfuzz --threads=1 -w /usr/share/wfuzz/lists/words.txt --method=POST --delay=0.1 --threads=1 -d "dummy-data" --uri=http://testphp.vulnweb.com/listproducts.php?cat=FUZZ --hh=4699
  "http://testphp.vulnweb.com/listproducts.php?cat=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/listproducts.php?cat=FUZZ
Total requests: 50
=====
ID      Response   Lines   Word    Chars   Payload
=====
0000000001: 200       107 L   526 W   7880 Ch   "1"
0000000002: 200       104 L   394 W   5311 Ch   "2"

Total time: 0
Processed Requests: 50
Filtered Requests: 48
Requests/sec.: 0
```

- 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:

The screenshot shows the Burp Suite interface. In the Request tab, a POST request to `/guestbook.php` is displayed with the following headers:
Host: testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
Cache-Control: max-age=0
Accept-Language: en-US,en;q=0.9
Origin: http://testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://testphp.vulnweb.com/guestbook.php
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Name: anonymous+user&text=Aditya&submit=add+message

In the Response tab, the response is shown as a plain HTML document with the title "guestbook".

Step3: Baseline request (sending without changing anything).

The screenshot shows the Burp Suite interface. In the Request tab, a POST request to `/guestbook.php` is displayed with the following headers:
Host: testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
Cache-Control: max-age=0
Accept-Language: en-US,en;q=0.9
Origin: http://testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://testphp.vulnweb.com/guestbook.php
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Name: anonymous+user&text=Aditya&submit=add+message

In the Response tab, the response is shown as a plain HTML document with the title "guestbook".

HTTP 200 OK. This confirms The requests works normally.

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

The screenshot shows the Burp Suite interface. In the Request tab, a POST request to `/guestbook.php` is displayed with the following headers:
Host: testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 112
Cache-Control: max-age=0
Accept-Language: en-US,en;q=0.9
Origin: http://testphp.vulnweb.com
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Name: anonymous+user&text=Aditya&submit=add+message

In the Response tab, the response is shown as a plain HTML document with the title "guestbook".

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.

The screenshot shows two panels from the NetworkMiner tool. The left panel, titled 'Request', displays a POST request to 'http://testphp.vulnweb.com/guestbook.php'. The right panel, titled 'Response', shows the resulting HTML page. The response includes standard headers like HTTP/1.1 200 OK, Server: nginx/1.19.0, and various content types and meta tags. The body of the response contains the HTML code for a guestbook form, including fields for name, user, text, and submit.

```
Request
Pretty Raw Hex
1 POST /guestbook.php HTTP/1.1
2 Host: testphp.vulnweb.com
3 Content-Length: 463
4 Cache-Control: max-age=0
5 Accept-Language: en-US,en;q=0.9
6
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,image/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 10:33:45 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: 5391
8
9 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
10 "http://www.w3.org/TR/html4/loose.dtd">
11 <html>
12   <!-- InstanceBegin template="/Templates/main_dynamic_template.dwt.php"
codeOutsideHTMLIsLocked="false" -->
13   <head>
14     <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
15     <!-- InstanceBeginEditable name="document_title_rgn" -->
16     <title>
17       guestbook
18     </title>
19     <!-- InstanceEndEditable -->
20     <link rel="stylesheet" href="style.css" type="text/css">
21     <!-- InstanceBeginEditable name="headers_rgn" -->
22     <!-- InstanceEndEditable -->
23   <!-- InstanceBeginEditable name="Footer_rgn" -->
24   <!-- InstanceEndEditable -->
```

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

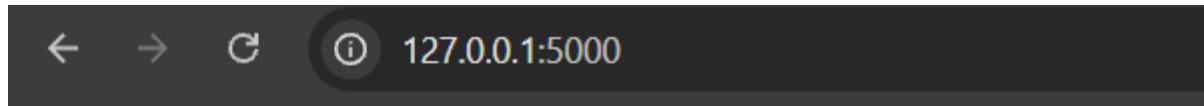
Opaque Data Manipulation:

App.py: we created locally.

```
app.py > ...
  1  from flask import Flask, request, render_template_string
  2  from cryptography.fernet import Fernet
  3  app = Flask(__name__)
  4  # Hardcoded key (bad practice, intentional for learning)
  5  key = Fernet.generate_key()
  6  cipher = Fernet(key)
  7  # Fake product database
  8  PRODUCTS = {
  9    "cheap": 100,
 10   "expensive": 1000
 11 }
 12 @app.route("/")
 13 def index():
 14   product = request.args.get("product", "expensive")
 15   price = PRODUCTS[product]
 16   # Encrypt the price (opaque data)
 17   enc_price = cipher.encrypt(str(price).encode()).decode()
 18   return render_template_string("""
 19     <h2>Product: {{product}}</h2>
 20     <form action="/order" method="post">
 21       Quantity: <input name="qty" value="1"><br><br>
 22       <input type="hidden" name="enc" value="{{enc}}">
 23       <input type="submit" value="Buy">
 24     </form>
 25   """, enc=enc_price, product=product)
 26 @app.route("/order", methods=["POST"])
 27 def order():
 28   qty = int(request.form["qty"])
 29   enc = request.form["enc"]
 30   # Decrypt opaque value
 31   price = int(cipher.decrypt(enc.encode()).decode())
 32   total = price * qty
 33   return f"<h3>Total charged: ${total}</h3>"
 34 if __name__ == "__main__":
 35   app.run(debug=True)
```

Step1: Run the application using Python app.py

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



Product: expensive

Quantity:

Buy

Step3: See the page source.

```
<h2>Product: expensive</h2>
<form action="/order" method="post">
    Quantity: <input name="qty" value="1"><br><br>
    <input type="hidden" name="enc" value="gAAAAABpX8D_p0nr35Yb8xXdcdeRzr8tWcRS5R-CkzwUD8oJfn09eH07SKsgP_9tHDiobP6hIv5my9-4I-woYvMEAfvsiOfHOg==">
    <input type="submit" value="Buy">
</form>
```

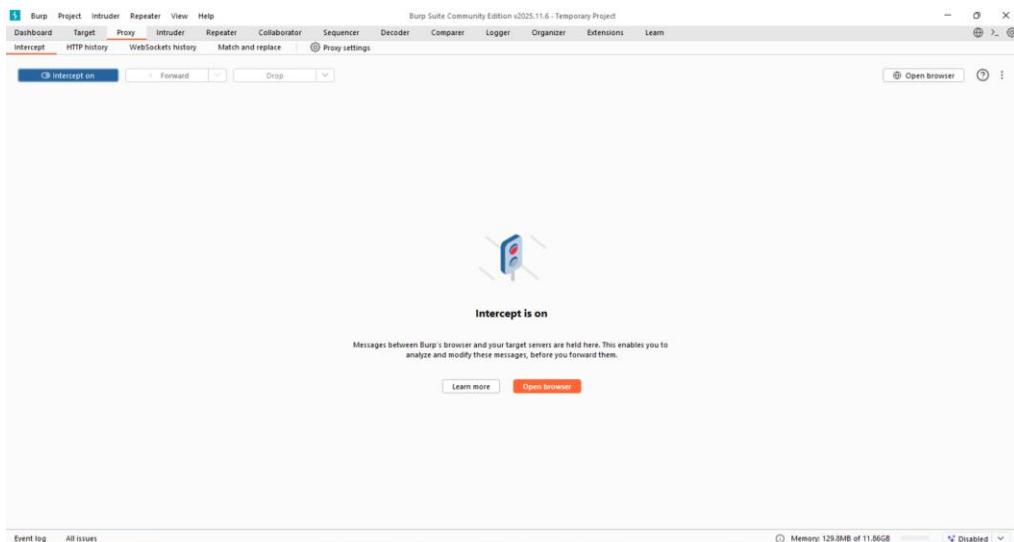
This is opaque

- 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:

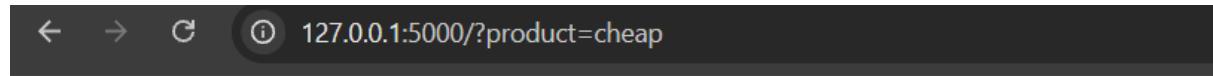
The screenshot shows the Burp Suite interface with the "Proxy" tab selected. A single request is listed in the timeline. The request details show a POST method to the "/order" endpoint. The "Raw" tab displays the following request body:

```
POST /order HTTP/1.1
Host: 127.0.0.1:5000
Content-Language: en-US
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: 70
sec-ch-ua-platform: "windows"
Accept-Language: en-US,en;q=0.9
Origin: http://127.0.0.1:5000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:5000/
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 1024
Content-Type: application/x-www-form-urlencoded
Content-Encoding: gzip
Content-MD5: gAAAAABpX8RmQy6TawmzGe2P9IS4ydlLvWiwvp6MY950TX1ZPtM06_IkxbtgH4wVNT92C6SYCaWFBRxu_n5nITFFA8VGNV3VdSA==
```

The "Inspector" panel on the right shows various request details such as attributes, query parameters, body parameters, cookies, and headers.

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:



Product: cheap

Quantity:

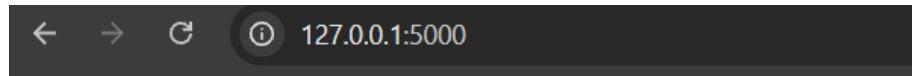
Source code: see the difference in the value.

The screenshot shows the source code of the "cheap" product page. The key difference is in the form's hidden input field:

```
<form action="/order" method="post">
    Quantity: <input name="qty" value="1"><br><br>
    <input type="hidden" name="enc" value="gAAAAABpX8RmQy6TawmzGe2P9IS4ydlLvWiwvp6MY950TX1ZPtM06_IkxbtgH4wVNT92C6SYCaWFBRxu_n5nITFFA8VGNV3VdSA==">
    <input type="submit" value="Buy">
</form>
```

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:



Product: expensive

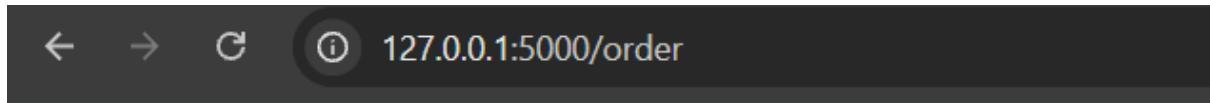
Quantity:

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

The screenshot shows the Burp Suite interface with the "Proxy" tab selected. A single request is listed in the timeline. The request details show a POST method to the URL `http://127.0.0.1:5000/order`. The request payload is very long and contains many lines of encrypted ViewState data.

Change the enc.

Step8: Then forward it.



Total charged: \$100

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:

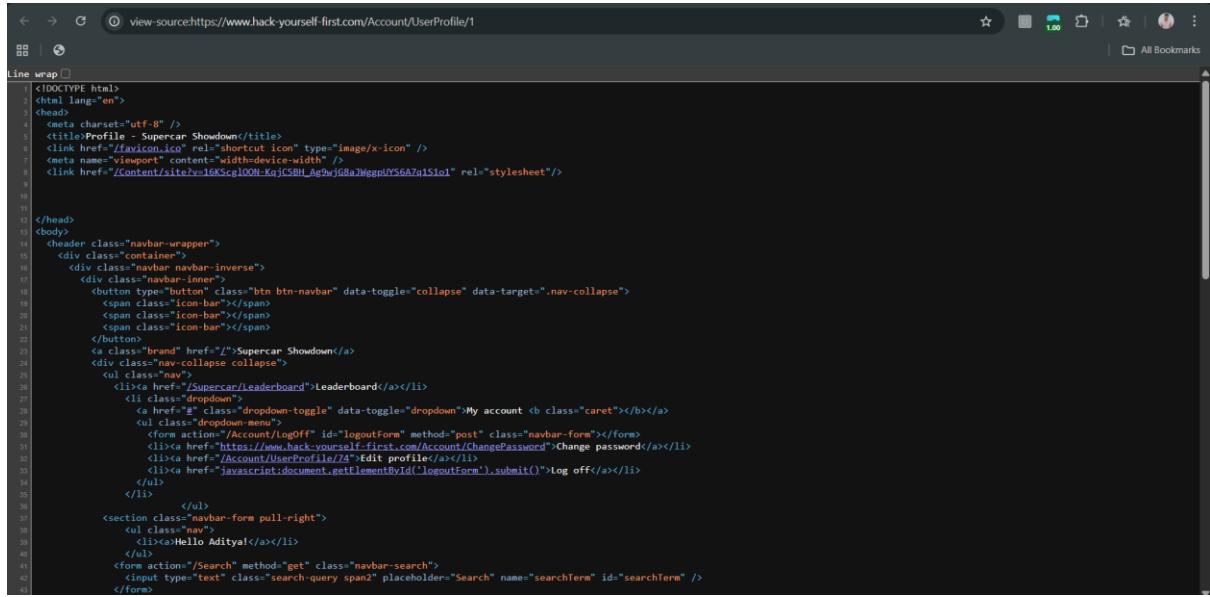
URL: <https://www.hack-yourself-first.com/Account/UserProfile/1>

Step1: Open the URL, following page will open:

Browser:

The screenshot shows a web browser window with the URL <https://www.hack-yourself-first.com/Account/UserProfile/1>. The page title is "Profile.". It displays a form titled "Edit your profile." with two input fields: "First name" containing "Elijah" and "Last name" containing "kweetnihelemaal". Below the inputs is a "Save" button. At the bottom of the page, there is a small footer note: "© 2026 - Hack Yourself First - [troyhunt.com](#)".

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



The screenshot shows the browser's developer tools with the "View Source" tab selected. The page source code is displayed, showing the HTML structure of the login page. The code includes the header, navigation bar, and a search bar. A dropdown menu is shown, revealing options like "Logout" and "Change Password". The code uses Bootstrap classes for styling.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Profile - Supercar Showdown</title>
    <link href="/favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    <link href="/Content/site1e06Scg1OON-KoJCG0H_Ag9wjd8aJWggpJY56A7q1S1o1" rel="stylesheet"/>
</head>
<body>
    <header class="navbar-wrapper">
        <div class="container">
            <div class="navbar navbar-inverse">
                <div class="navbar-inner">
                    <button type="button" class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="brand" href="/">Supercar Showdown</a>
                    <div class="nav-collapse collapse">
                        <ul class="nav">
                            <li><a href="/Supercar/leaderboard">Leaderboard</a></li>
                            <li><a href="#">Logout</a></li>
                            <li><a href="#">My account <b>caret</b></a></li>
                                <ul class="dropdown-menu">
                                    <form action="/Account/logout" id="logoutForm" method="post" class="navbar-form"></form>
                                    <li><a href="https://www.hack-yourself-first.com/Account/changePassword">Change password</a></li>
                                    <li><a href="/Account/UserProfile/74>Edit profile</a></li>
                                    <li><a href="javascript:document.getElementById('logoutForm').submit()>Log off</a></li>
                                </ul>
                            </li>
                        </ul>
                    </div>
                </div>
                <ul class="nav">
                    <li><a href="#">Hello Aditya!</a></li>
                </ul>
            </div>
            <form action="/Search" method="get" class="navbar-search">
                <input type="text" class="search-query span2" placeholder="Search" name="searchTerm" id="searchTerm" />
            </form>
        </div>
    </header>
```

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 maxlen 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:

The image contains two screenshots of a web browser window. Both screenshots show a search results page from the URL <https://www.hack-yourself-first.com/Search?searchTerm=His+mother+had+always+taught+him+not+to+ever+think+of+himself+as+better+than+others.+He%27d+tried...>.
The top screenshot shows a large block of text starting with "You searched for "His mother had always taught him not to ever think of himself as better than others. He'd tried to live by this motto. He never looked down on those who were less fortunate or who had less money than him. But the stupidity of the group of people he was talking to made him change his mind. The bridge spanning a 100-foot gully stood in front of him as the last obstacle blocking him from reaching his destination. While people may have called it a "bridge", the reality was it was nothing more than splintered wooden planks held together by rotting ropes. It was questionable whether it would hold the weight of a child, let alone the weight of a grown man. The problem was there was no other way across the gully, and this played into his calculations of whether or not it was worth the risk of trying to cross it. It was the first day of the rest of her life. This wasn't the day she was actually born, but she knew that nothing would be the same from this day forward. Although this was a bit scary to her, it was also extremely freeing. Her past was no longer a burden or something that she needed to be concerned about and".
The bottom screenshot shows a similar large block of text starting with "a day and he always received the same answer. It had become such an ingrained part of his daily routine that he had to step back and actively think when he heard the little girl's reply. Nobody had before answered the question the way that she did, and David didn't know how he should respond. He couldn't move. His head throbbed and spun. He couldn't decide if it was the flu or the drinking last night. It was probably a combination of both. Finding the red rose in the mailbox was a pleasant surprise for Sarah. She didn't have a boyfriend or know of anyone who was interested in her as anything more than a friend. There wasn't even a note attached to it. Although it was a complete mystery, it still made her heart jump and race a little more than usual. She wished that she could simply accept the gesture and be content knowing someone had given it to her, but that wasn't the way Sarah did things. Now it was time to do a little detective work and try to figure who had actually left the red rose."

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 })
```

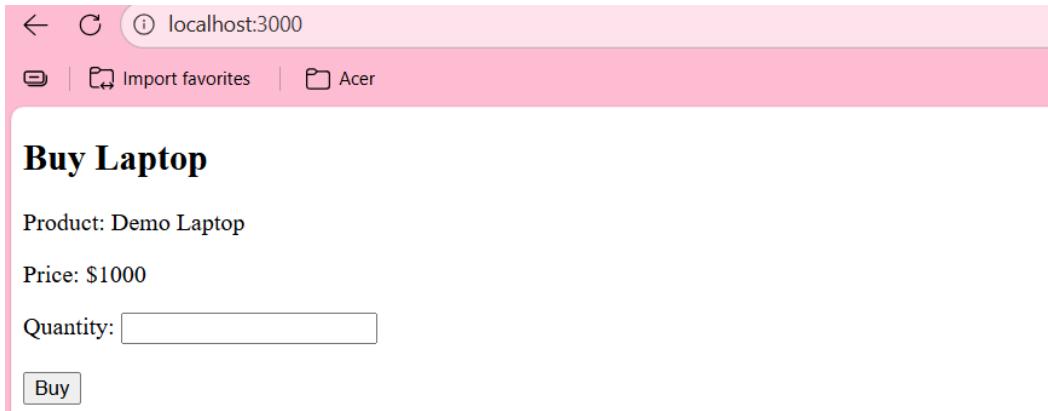
Public/index.html:

```
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      
25      <input type="hidden" name="price" value="1000">
26      <input type="submit" value="Buy">
27    </form>
28  </body>
29 </html>
```

Terminal:

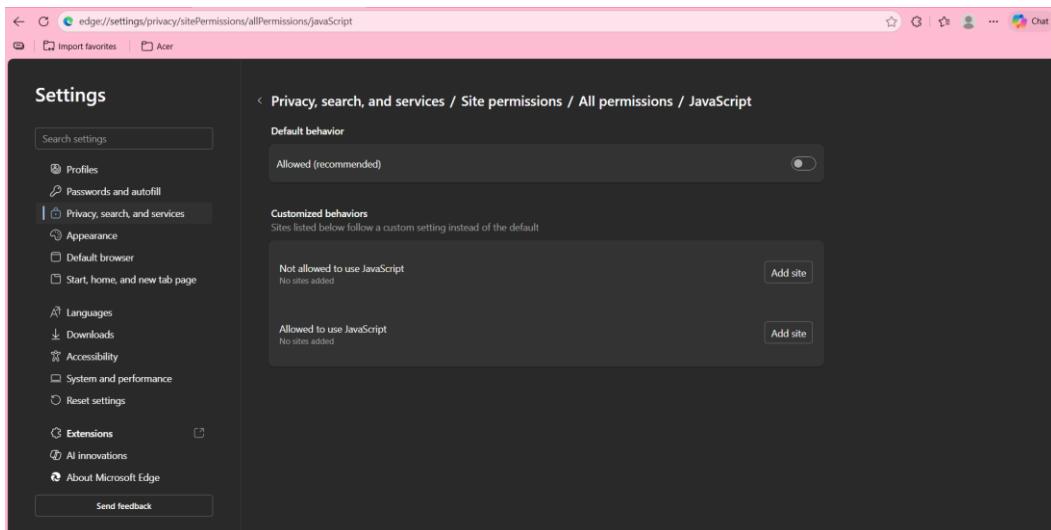
```
PS E:\Tounderstnad\vulnerable-shop> node server.js
vulnerable app running on http://localhost:3000
```

Browser:



Check1: Close the JS from the browser.

Settings:



Enter any negative value:



Click on buy:

localhost:3000/order

Import favorites Acer

Order Confirmation

Quantity ordered: -5
Price per item: 1000
Total cost: -5000

Server accepted it.

Check2: Manipulating from the inspect section.

Originally: See 1000.

localhost:3000

Welcome Elements Console Sources Network Performance Memory

Buy Laptop

Product: Demo Laptop

Price: \$1000

Quantity:

Buy

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <h2>Buy Laptop</h2>
    <p>Product: Demo Laptop</p>
    <p>Price: $1000</p>
    <form action="/order" method="POST" onsubmit="return validateForm()">
      <input type="text" id="quantity" name="quantity">
      <br>
      <br>
      <!-- X Hidden field controlled by client -->
      <input type="hidden" name="price" value="1000"> == $0 ⓘ
      <input type="submit" value="Buy">
    </form>
    <iframe src="chrome-extension://iidlbfakcgbmmcpfbhmkndacikdm/audio-devices.html" allow="microphone" style="display: none;"></iframe>
    <input type="file" id="file" style="display: none;">
    <div data-v-5f0b9ba1 class="container_selected_area" style="cursor: url('chrome-extension://iidlbfakcgbmmcpfbhmkndacikdm/assets/images/cursor-imagen.svg') 9 9, crosshair;"></div>
  </body>
```

Manipulated:

localhost:3000

Welcome Elements Console Sources Network Performance Memory

Buy Laptop

Product: Demo Laptop

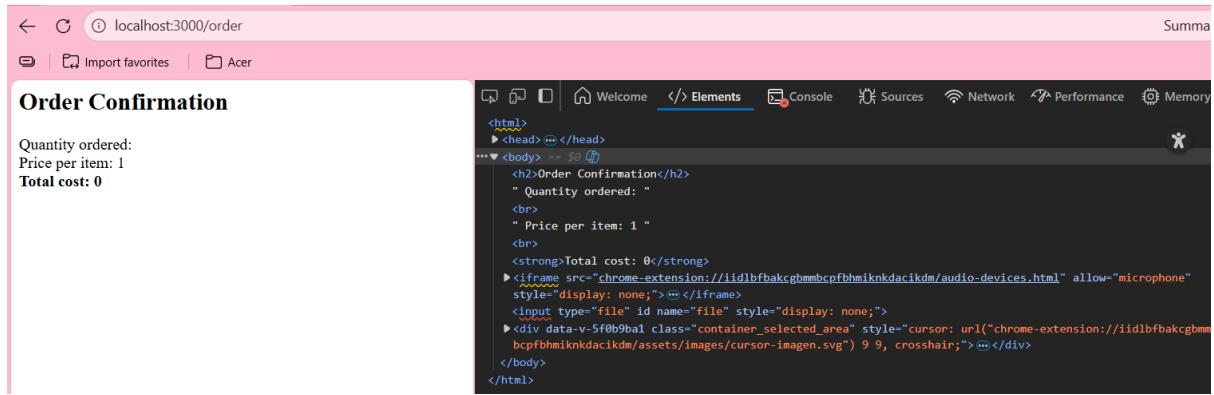
Price: \$1000

Quantity:

Buy

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <h2>Buy Laptop</h2>
    <p>Product: Demo Laptop</p>
    <p>Price: $1000</p>
    <form action="/order" method="POST" onsubmit="return validateForm()">
      <input type="text" id="quantity" name="quantity">
      <br>
      <br>
      <!-- X Hidden field controlled by client -->
      <input type="hidden" name="price" value="1"> == $0 ⓘ
      <input type="submit" value="Buy">
    </form>
    <iframe src="chrome-extension://iidlbfakcgbmmcpfbhmkndacikdm/audio-devices.html" allow="microphone" style="display: none;"></iframe>
    <input type="file" id="file" style="display: none;">
    <div data-v-5f0b9ba1 class="container_selected_area" style="cursor: url('chrome-extension://iidlbfakcgbmmcpfbhmkndacikdm/assets/images/cursor-imagen.svg') 9 9, crosshair;"></div>
  </body>
```

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



The screenshot shows a browser window with the URL "localhost:3000/order". The developer tools are open, specifically the Elements tab. The DOM tree displays the following HTML structure:

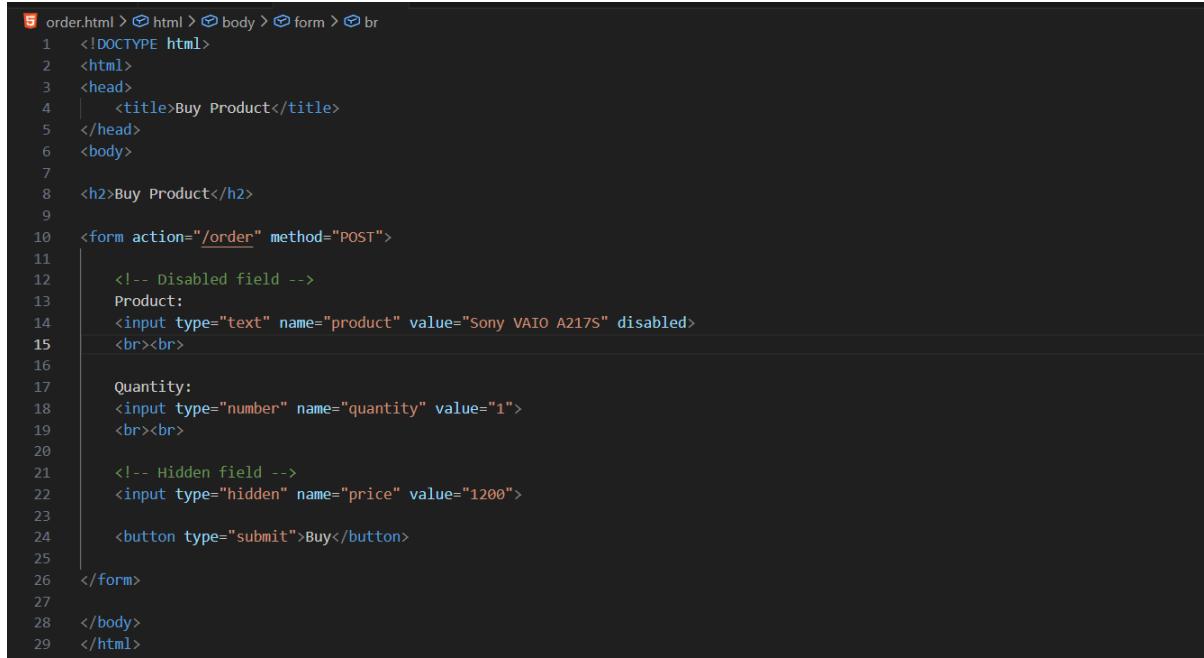
```
<html>
  <head> ... </head>
  <body> ...
    <h2>Order Confirmation</h2>
    " Quantity ordered: "
    <br>
    " Price per item: 1 "
    <br>
    <strong>Total cost: 0</strong>
    <iframe src="chrome-extension://iidlbfbakcgbmmbcpcfblmiknkdacikdm/audio-devices.html" allow="microphone" style="display: none;"></iframe>
    <input type="file" id="file" style="display: none;">
    <div data-v-5f0b9ba1 class="container_selected_area" style="cursor: url('chrome-extension://iidlbfbakcgbmmbcpcfblmiknkdacikdm/assets/images/cursor-imagen.svg') 9 9, crosshair;"></div>
  </body>
</html>
```

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:

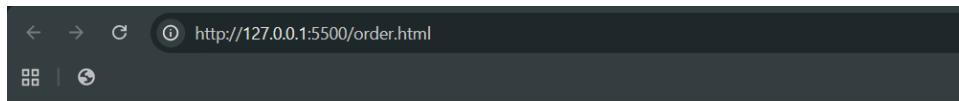


```
order.html > ⚡ html > ⚡ body > ⚡ form > ⚡ br
1  <!DOCTYPE html>
2  <html>
3  <head>
4  |   <title>Buy Product</title>
5  </head>
6  <body>
7
8  <h2>Buy Product</h2>
9
10 <form action="/order" method="POST">
11
12   <!-- Disabled field -->
13   Product:
14   <input type="text" name="product" value="Sony VAIO A217S" disabled>
15   <br><br>
16
17   Quantity:
18   <input type="number" name="quantity" value="1">
19   <br><br>
20
21   <!-- Hidden field -->
22   <input type="hidden" name="price" value="1200">
23
24   <button type="submit">Buy</button>
25
26 </form>
27
28 </body>
29 </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:



Buy Product

Product:

Quantity:

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

A screenshot of a browser developer tools window. The main view shows the "Buy Product" form. The "Elements" tab is selected in the bottom-left corner. In the bottom-right corner, the "Styles" panel is open, showing the CSS styles for various elements. A specific rule for "input:disabled" is highlighted, showing its properties: cursor: default; background-color: #light-dark; border-color: #light-dark; color: #light-dark; user agent stylesheet. The "disabled" attribute is also visible in the element's computed style. The URL in the address bar is "http://127.0.0.1:5500/order.html".

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

The screenshot shows the Burp Suite interface with the "Proxy" tab selected. The "HTTP History" tab is active. There are five entries in the history list:

- Host: http://localhost:3000, Method: GET, URL: /, Status code: 200, Length: 833, MIME type: HTML, Title: Buy Product, Time: 14:03:05 11.J.., IP: 127.0.0.1, TLS: 127.0.0.1, Cookies: None, Listener port: 8080, Start response: 3.
- Host: http://127.0.0.1:5500, Method: POST, URL: /order, Status code: 405, Length: 271, MIME type: HTML, Title: Buy Product, Time: 14:00:57 11.J.., IP: 127.0.0.1, TLS: 127.0.0.1, Cookies: None, Listener port: 8080, Start response: 39.
- Host: http://localhost:3000, Method: POST, URL: /order, Status code: 200, Length: 366, MIME type: HTML, Title: Buy Product, Time: 14:03:07 11.J.., IP: 127.0.0.1, TLS: 127.0.0.1, Cookies: None, Listener port: 8080, Start response: 39.
- Host: http://127.0.0.1:5500, Method: GET, URL: /order.html, Status code: 200, Length: 2358, MIME type: HTML, Title: Buy Product, Time: 14:00:54 11.J.., IP: 127.0.0.1, TLS: 127.0.0.1, Cookies: None, Listener port: 8080, Start response: 4.
- Host: http://127.0.0.1:5500, Method: GET, URL: /order.html/ws, Status code: 101, Length: 129, MIME type: HTML, Title: Buy Product, Time: 14:00:55 11.J.., IP: 127.0.0.1, TLS: 127.0.0.1, Cookies: None, Listener port: 8080, Start response: 2.

The selected request (entry 7) is shown in the Request and Response panes. The Request pane contains the raw POST data:

```

POST /order HTTP/1.1
Host: localhost:3000
Content-Length: 21
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Origin: http://localhost:3000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
quantity=1&price=1200

```

The Response pane shows the raw response:

```

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 137
ETag: W/"09-DahWhGmEnWh7nsXtx/vIrcLLOCc"
Date: Sun, 11 Jan 2026 08:33:07 GMT
Connection: keep-alive
Keep-Alive: timeout=5

```

The Inspector pane shows the request attributes and headers.

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

The screenshot shows the Burp Suite interface with the "Repeater" tab selected. The "HTTP/1" tab is active. The repeater pane displays the same captured request and response as the previous screenshot. The request is identical to the one in Step 2:

```

POST /order HTTP/1.1
Host: localhost:3000
Content-Length: 21
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Origin: http://localhost:3000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
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
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
quantity=1&price=1200

```

The response is identical to the one in Step 2:

```

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 137
ETag: W/"09-DahWhGmEnWh7nsXtx/vIrcLLOCc"
Date: Sun, 11 Jan 2026 08:33:07 GMT
Connection: keep-alive
Keep-Alive: timeout=5

```

The Inspector pane shows the request attributes and headers.

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,image/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.

The screenshot shows the Burp Suite interface with the following details:

Request:

```
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,image/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|
```

Response:

```
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: text/html; charset=utf-8
4 Content-Length: 143
5 ETAG: W/"8f-3vhUpQFQ16gFbTHyxEPnDNE61Y"
6 Date: Sun, 11 Jan 2026 08:36:42 GMT
7 Connection: keep-alive
8 Keep-Alive: timeout=5
9
10
11 <h2>
12   Order Details
13 </h2>
14
15 Product: Sony VAIO A217S<br>
16 Quantity: 1<br>
17 Price: $1200<br>
18 Total: $1200
19
20
21
```

Inspector:

- Request attributes: 2
- Request query parameters: 0
- Request body parameters: 3
- Request cookies: 0
- Request headers: 19
- Response headers: 7

Notes:

Custom actions:

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.