Bloomberg Hackathon Capture the flag

Report by Sheniese Aracena-Baez 06/27/25-06/30/25 Team Wall-E

The purpose of this Capture the Flag (CTF) challenge was to test both offensive (red team) and defensive (blue team) web security skills. The event focused on finding, exploiting, and defending against common web-based vulnerabilities. Many of the challenges were solved using tools available through a regular web browser, combined with knowledge of password weaknesses, open-source intelligence (OSINT) basic Linux commands such as curl, along with crafted injection payloads.

The capture the flag was hosted over the weekend of June 27th. The competition started at 9 am on Friday June 27th, and ended at 12 pm on Sunday the 29th. Our team however completed all 13 challenges within five and a half hours, earning 1st place among six competing teams.

Note:
Certain details, screenshots, and challenge artifacts have been intentionally omitted at the request of Bloomberg which hosted this CTF. All information included here reflects general concepts and personal methodology without disclosing proprietary material or sensitive challenge content.

Level 1 – Entry-Level Logic Challenge

Challenge

This challenge presented a simple webpage with no obvious input boxes or forms to log in. Instead, the clue was hidden in the web address (URL) itself. The goal was to figure out how to "log in" by logically manipulating the URL.

Enumeration

When first loading the page, the URL ended in something similar to:

/level1?username=&password=

This format suggested that:

- The website expected a username and a password
- But there was no form to submit them normally
- The only way to provide the login info was directly through the URL

Since this was an introductory level, it was reasonable to test simple, commonly used default credentials.

Exploitation

To test this theory, I edited the end of the URL manually and added:

username=admin&password=admin

This is one of the most common default login combinations for routers and simple systems. After updating the URL with these values and pressing Enter, the page reloaded and displayed the flag. No tools, no scripts, just adjusting the URL to include the expected information.

This worked because the challenge intentionally relied on:

- Recognizing the hint in the URL
- Knowing that “admin/admin” is a typical default credential
- Manually supplying those values through the URL

Validation

To confirm the solution:

- The modified URL successfully authenticated the session
- The webpage was updated to display the correct flag
- The flag was saved/documented to complete Level 1

Level 2 – HTTP Headers Challenge

Challenge

This level didn’t display any obvious clues on the webpage itself. The challenge expected the user to look “behind the scenes” of how a browser communicates with a website. The goal was to inspect the website’s hidden metadata — specifically the HTTP headers — to locate information that would point to the next level.

Enumeration

To begin examining what the browser was sending and receiving, I used the built-in Developer Tools:

1. Right-clicked the page → “Inspect”
2. Navigated through several tabs, refreshing after each to capture new data:
    a. Console
    b. Sources
    c. Network

The Network tab is especially useful because it shows every request the browser makes to the server. Each request contains headers, which act like labels or instructions attached to the request and response.

After selecting the main page request, I clicked on **“Headers”**.

Exploitation

Inside the Response Headers section, I found clues that were not visible anywhere on the webpage. Specifically, one of the headers contained:

- A reference to Level 3
- The URL needed to access the next challenge

This works because some developers accidentally leave internal notes, debug info, or sensitive paths inside these headers a small but common mistake in real-world systems.

The challenge relied on recognizing:

- That HTTP headers can leak unintended information
- That Developer Tools reveal everything the server sends, even if the page doesn't show it

No external tools needed, just browser developer tools.

Validation

To confirm the solution:

- I copied the Level 3 URL found inside the response header
- Opened it in the browser
- Verified that it successfully loaded the next stage of the challenge

---

Level 3 – Client-Side JavaScript & XSS

Challenge

This level hinted that the answer was hidden somewhere on the client side, meaning inside the code that runs in the browser. Nothing obvious appeared on the webpage itself, so the challenge required inspecting the site's JavaScript to uncover a variable storing the key needed to move to the next level.

Enumeration

The words “client” and “window object” stood out immediately.

Those usually refer to:

- The browser (client-side code)
- JavaScript variables stored in the browser’s global scope
- Logic that runs before the server ever gets involved

Since this level was hinting at testing basic JavaScript knowledge, I opened the browser’s Developer Tools again and focused on the Console tab that lets you directly interact with anything the webpage exposes.

Because many web apps keep small pieces of state in the browser (like progress markers or keys), I tried querying the most suspicious variable name shown in hints:
nextLevelKey

Exploitation

Once I typed the variable name into the JavaScript console, the browser returned the stored value.

This worked because:

- The application stored a value directly in the browser
- There was no protection preventing users from seeing or modifying it
- Browsers trust whatever is on the client side, which means client-side data can always be altered or inspected

This level also provided a small hint toward Cross-Site Scripting (XSS) concepts. Since you can run JavaScript in the Console, even a simple payload like: alert("test")

Validation

To validate completion:

- Retrieved the value by inspecting the nextLevelKey variable

- Confirmed that it matched the expected key format
- Used the key to unlock Level 4 or the next challenge stage

Security Lesson

Web applications often share small pieces of information with the browser so the site can function smoothly. But unless the data is *cryptographically signed* or verified, it must always be assumed to be visible and tamperable by anyone.

Client-side = not secure storage.

Level 4 – Hidden Form Fields Challenge

Challenge

This challenge used a webpage with a form that looked simple on the surface, but it included hidden fields with pieces of data the user doesn't normally see. One of those fields controlled whether the user had "accepted" the required EULA. Even though the field was hidden visually, it was not protected, and the challenge expected you to discover and change it.

Enumeration

To explore what the form was doing behind the scenes, I opened the browser's Developer Tools again and inspected the page's HTML.

There were two ways to locate the hidden field:

1. Right-click anywhere → "Inspect" → Search for has_accepted_eula
    a. This allows you to search the page's HTML for specific keywords.
2. Right-click directly on the form or submit button → "Inspect"
    a. This jumps straight to the exact portion of code that contains the form's fields, including the hidden input.

Once I expanded the form's HTML structure, I saw something like:

```
<input type="hidden" name="has_accepted_eula" value="yes">
```

This tells the browser to quietly send information when the form is submitted, even if it is never seen on the page.

Exploitation

Since hidden fields can be edited just like any other text, I changed the value from:

value="yes" **to** value="no"

After modifying it directly in the HTML, I typed “no” into the main input and clicked Submit.

The website accepted the altered value and granted access to the next level.

This worked because:

- Hidden fields are NOT a security measure
- Browsers trust whatever the user’s machine sends
- If the server doesn’t validate the field, it can be completely bypassed

Tools like Burp Suite can also intercept and modify hidden field values, but the challenge intentionally showed how simple it is to do with just the browser.

Validation

To confirm the success:

- The modified form submission was accepted
- The page advanced to the next challenge
- The expected success message or next-level URL was displayed

---

Level 5 – Cookie Manipulation Challenge

Challenge

This level tested awareness of how websites use **cookies** to track whether a user is logged in. Instead of performing a real login, the challenge required creating a fake authentication cookie manually. If the cookie named isAuthenticated existed and had the value 1, the site would treat the user as logged in and reveal the next flag.

Enumeration

To figure out how the site tracked authentication, I inspected the browser's stored data using Developer Tools.

Here's what I checked:

1. Opened DevTools → Console tab
    a. This allows running JavaScript commands directly in the browser.
2. Checked existing cookies by typing document.cookie
    a. I could see whether any authentication-related cookies were already present.
3. Explored Storage/Application tabs
    a. These tabs show all cookies stored by the website. In this challenge, there was no authentication cookie set by default, which hinted that the challenge wanted me to create one manually.

Enumeration

There were two ways to solve this challenge:

Method 1 (Quickest): Use the Console

I created a new cookie by running:

```
document.cookie = "isAuthenticated=1";
```

After pressing Enter, the cookie was instantly stored in the browser. I then refreshed the page, and the website read the cookie and assumed I was authenticated

---

Method 2 (Longer but Visual): Use Storage/Application Tabs

1. Open Developer Tools
2. Go to Application (or Storage, depending on browser)
3. Select Cookies under the left sidebar
4. Add a new entry:
    - Name: isAuthenticated
    - Value: 1
    - Domain autofills automatically

This method is more click-based and easier to see what's happening.

Validation

To confirm the cookie was added, I typed again:

- document.cookie

The response showed

- isAuthenticated=1

After refreshing the page, the site accepted the cookie and granted access to the next level.

Security Lesson

This challenge demonstrates a common misconception:

If a website trusts a cookie without verifying it on the server, anyone can fake being authenticated.

Real applications must:

- Sign cookies
- Validate them server-side
- Never trust client-controlled data

Level 6 – JWT Logic

Challenge

This level introduced JSON Web Tokens (JWTs) a very common method used by websites to store user session information. The challenge required understanding how JWTs work, recognizing that they're only encoded (not encrypted), and modifying their contents to bypass the site's logic and access the next level.

You don't need the secret key to read the token. You only need to modify the payload in a way the challenge accepts.

Enumeration

A JWT looks like this: Header.Payload.Signature

Each part is Base64-encoded text. Even non-technical readers can think of it like this:

- **Header:** Describes how the token is formatted
- **Payload:** Contains the actual data (like username or permissions)
- **Signature:** A checksum that verifies nothing was tampered with

In real systems, modifying the payload would break the signature and make the token invalid. But this challenge intentionally did NOT check the signature, which is why modification was possible.

To inspect the token, I used a JWT decoder site: https://jwt.tplant.com.au/

This allowed me to:

- Paste in the original JWT the site provided
- Decode it into readable text

- Edit the contents of the payload
- Re-encode it automatically

Inside the payload, I could clearly see fields describing the user state, such as authentication level or permissions.

Exploitation

The goal was to modify the token's payload so that the challenge would treat the user as authorized.

Steps taken:

1. Copied the original JWT from the challenge webpage.
2. Pasted it into the JWT decoder website.
3. Edited the payload values (e.g., changing a role from "user" to "admin" or flipping a boolean flag to true).
4. The decoder automatically re-assembled the modified token.
5. Submitted the modified token back into the challenge's input box.

Because the system did not verify the signature properly, it accepted the forged JWT as valid.

Validation

To confirm the solution:

- The modified JWT was entered into the site
- The challenge responded with success and advanced to the next level
- The displayed message confirmed that the tampered token was accepted

---

Security Lesson

This challenge demonstrates a major real-world security pitfall:

- JWTs are encoded, not encrypted. Anyone can read their contents.

- If a system does not check the token’s signature, users can modify the payload and impersonate anyone they want.
- Proper JWT use requires verifying the signature with the server’s secret key.

Never trust the contents of a JWT unless it has been cryptographically validated.

Level 7 – Spoofed Requests Using curl

Challenge

This level required accessing the same webpage but under two specific conditions:

1. The URL must include ?flag=true
2. The request must appear to come from localhost

Both conditions had to be true at the same time for the server to reveal the next flag.

The challenge demonstrated how servers sometimes trust information sent through HTTP headers and how that trust can be abused if not validated properly.

Enumeration

The browser alone wasn’t enough to solve this level.

The hints provided including the Network tab and the suggestion to use curl implied that:

- The server was checking headers (specifically the Origin or Host header)
- The request needed to appear as if it came from localhost, even though we were not actually running the challenge locally
- The browser does not let you fake certain headers for security reasons (which explains why we had to switch to the terminal)

By inspecting a normal request in the Network tab, it became clear which headers the server was using. From there, we could recreate and modify that request manually with curl.

Exploitation

To spoof the request properly, two things had to be added:

- The query parameter: ?flag=true
- A fake header indicating we are localhost: -H "Origin: http://localhost" or sometimes -H "X-Forwarded-For: 127.0.0.1"
- A typical curl command looked like: curl -H "Origin: http://localhost" "http://example.com/level7?flag=true"

After running the command in the Linux terminal, the server accepted the spoofed request and returned the output containing the URL for the next challenge. This worked because the server trusted whatever value was placed in the header even though it wasn't actually true.

Validation

To confirm success:

- The curl command returned an HTTP response containing the next-level link
- The link was copied into the browser
- The page successfully loaded the next challenge

Security Lesson

This challenge demonstrates two important ideas:

- Many servers trust headers like Origin or X-Forwarded-For, even though users can fake them.
- The browser protects these headers for safety but tools like curl can modify them freely.

In real applications, servers should never trust client-supplied headers without verification.

Level 8 – Binary Reverse Engineering (Strings Extraction)

Challenge

This level provided a downloadable executable file and required discovering a hidden password stored inside it. The challenge focused on a simple but extremely common reverse engineering technique: searching for readable text (strings) inside a binary file.

Enumeration

After downloading the file onto a Linux machine, the next step was to inspect its contents.

Instead of running the file, I examined it using the Linux command-line tool strings, which pulls out anything inside a binary that resembles human-readable text.

Why this works:
Executables often contain hardcoded values like error messages, debug text, and if poorly designed passwords or secret keys.

Process:

1. Downloaded the file onto a Linux environment
2. Opened a terminal
3. Navigated to the directory containing the binary

This allowed for analyzing the file without executing it.

Exploitation

To extract readable text from the binary, I ran: strings <filename>

This printed a long list of words, fragments, and text sequences found inside the executable.

By scrolling through the output (or piping it through grep), I located a clear text value that resembled a password or key. This was intentionally placed in the binary as the “hidden” secret needed to solve the challenge.

Validation

To validate:

- I copied the discovered password
- Entered it into the challenge page
- The webpage accepted the password and provided the link or flag for the next level

Security Lesson

This challenge shows why developers should never hardcode secrets inside compiled programs.
Attackers can always extract those values using basic tools.

Level 9 – Client-Side Hash Matching

Challenge

This level presented a password field on the webpage, but instead of sending your password to the server, the website contained a hash value inside the page itself. The challenge expected you to find the correct password by matching it to the hash that the browser was checking.

The flaw:
All the "security" happened client-side (inside the browser), where the attacker can see everything.

Enumeration

By opening Developer Tools (right-click → Inspect), I was able to view the JavaScript being used for password validation.

Inside the code, the challenge exposed:

- The hash value of the correct password
- The hashing method being used (e.g., SHA-256 or MD5)
- The client-side logic that compared your input's hash to the expected one

Once I located the hash, the next step was to figure out which password produced that exact hash.

Exploitation

There were two main ways to solve this challenge:

Method 1 (Intended): Run a Python Script with a Password List

1. Downloaded the **password list** provided by the challenge (e.g., a custom wordlist or rockyou.txt).
2. Created a small Python script to:
    - Read each password from the list
    - Hash it using the same hash function
    - Compare it to the expected hash
    - Print the matching password

A simplified example:

```
import hashlib

target_hash = "EXAMPLE_HASH"

with open("passwordlist.txt", "r") as f:

    for word in f:

        word = word.strip()

        if hashlib.sha256(word.encode()).hexdigest() == target_hash:

            print("Password found:", word)

            break
```

- Ran the script in Linux
- The script returned the matching password

Method 2 (Shortcut / "Cheating"): Use crackstation.net

Since the challenge used a weak hash or common password:

1. Copy the hash
2. Visit: https://crackstation.net/
3. Paste the hash
4. CrackStation instantly returned the password

This works because large databases exist mapping common passwords → hashes.

Validation

After identifying the correct password:

- I returned to the challenge webpage
- Entered the discovered password
- The browser computed the hash
- Since the hash matched the expected value, the page revealed the flag / next-level URL

Security Lesson

This level highlights a classic security mistake:

- Never perform password validation in client-side JavaScript.
- Never rely on client-side hash comparison.
- Attackers can always see the hash and the validation logic.
- Hashes are not secrets — they can be reversed through dictionary attacks or online lookup tables.

Real systems must validate passwords server-side, never in the browser.

Level 9 – Client-Side Hash Matching

Challenge

This level dealt with an open redirect flaw a security issue that occurs when a website accepts a URL from the user and redirects them there without checking whether it's safe or allowed.

The challenge required finding a way to bypass the website's "allowed URL" checks and successfully redirect to an external domain. The site filtered obvious patterns, so the solution depended on tricking the validation by using encoding and obfuscation techniques.

Enumeration

First, I inspected how the site handled redirects by testing: ?redirect=http://example.com

The server rejected this, meaning it was doing some level of validation — likely checking:

- Whether the redirect starts with "http"
- Whether it contains //
- Whether it goes to an external domain
- Whether it looks like a known redirect pattern

To understand the restrictions, I experimented with multiple bypass techniques used in real-world open redirect exploits.

Exploitation

I tested common redirect bypass techniques one by one.
Each one interferes with simple filters that only look for very specific patterns.

1. Some of the attempts included: ?redirect=http://malicious.com Rejected.
2. Scheme-less URL ?redirect=//malicious.com also rejected — the filter detected the //.
3. Extra slashes ?redirect=////malicious.com still rejected
4. URL Encoding ?redirect=%2F%2Fmalicious.com Blocked — filter probably decoded once and noticed //
5. Double URL Encoding (successful technique)
    a. This method **double-encodes** characters so basic filters fail to detect them.
        i. ?redirect=%252F%252Fmalicious.com

Why this works:

- The server decodes once → sees %2F%2Fmalicious.com
- Browser decodes again → interprets //malicious.com
- The redirect succeeds

This breaks naive filters that only decode *once* and don't validate deeply.

Final Solution (Double URL-Encoding)

The working exploit was:

?redirect=%2f%2fwww%252E******%252Ecom (Not the real one used but an example)

This fooled the server into allowing the redirect, and the browser correctly interpreted it as an external site.

Validation

To confirm the challenge was solved:

- The crafted URL was loaded in the browser
- The server performed the redirect
- The browser navigated to the external domain
- The challenge responded with the flag or next-level link

Security Lesson

This level teaches an important real-world principle:

- Input validation must be strict, consistent, and performed after full decoding.
- Open redirect filters that rely on simple string checks are easy to bypass.
- Attackers can use open redirects for:
    - Phishing
    - Session hijacking
    - Redirecting to malware
    - OAuth token theft

Proper protections include:

- Enforcing allowlists
- Validating final resolved URLs (after decoding)
- Rejecting user-controlled redirect destinations entirely

Level 11 – Stack Traces & NoSQL Injection (NoSQLi)

Challenge

This level focused on exploiting server errors to reveal backend information and crafting an injection that would cause the application to behave unexpectedly. The challenge involved manipulating a query parameter (?q=) to trigger internal errors, gather clues from stack traces, and ultimately use NoSQL injection to access the next level.

Enumeration

The page contained a URL parameter structured like: ?q=<value>

This suggested that the server was performing a search, lookup, or filter based on user input. Query parameters like q are commonly tied to database queries, making them good candidates for injection attempts.

To inspect how the server handled requests, I used Burp Suite:

1. Enabled the proxy to intercept the request
2. Captured the page's GET request
3. Sent the request to Repeater for controlled testing

Before any injection was performed, simply sending the original request through Repeater revealed helpful information in the response:

- The server returned an error message referencing Express.js
- Express is part of the Node.js ecosystem, which often uses NoSQL databases like MongoDB

This was an important clue:
If the backend is Node + Express, the database may be MongoDB - meaning NoSQL injection may apply.

Exploitation

The goal was to intentionally trigger a 500 Internal Server Error. This can expose stack traces detailed server error logs that reveal:

- Function calls
- File paths
- Database usage
- Internal logic flow

These details help determine what type of injection will work.

I began by modifying the q parameter in Burp Repeater to send unexpected input: ?q[]=

This caused a 500 error, confirming that the backend was trying to treat the parameter as something other than a simple string. The presence of a stack trace validated the existence of an exploitable input-handling flaw.

At first, I attempted traditional SQL injection, but it had no effect, further supporting the likelihood that the backend was NoSQL-based.

Next, I tested NoSQL injection, which worked.
The payload exploited how MongoDB queries interpret objects and arrays, allowing the request to bypass normal logic and reveal the next-level URL.

In short:

- Breaking the parameter - produced a stack trace
- Stack trace - confirmed Express.js / Node.js backend
- Node.js backend - likely NoSQL
- Testing NoSQLi payloads - success

Validation

To confirm the exploit:

1. The NoSQL injection payload returned a different response than normal
2. The response contained the URL for the next challenge
3. Navigating to the provided URL successfully loaded Level 12

Level 11 was completed using a combination of error-based information gathering and NoSQL injection.

Security Lesson

This level demonstrates several real-world security lessons:

- Stack traces should never be exposed in production.
  They give attackers detailed insight into backend frameworks, technologies, and logic.
- Query parameters require strict validation.
  Treating them blindly as objects or arrays can lead directly to NoSQL injection.
- Error-based exploitation is powerful.
  Simply triggering an error can reveal enough information to craft a targeted attack.

Level 12 – Obfuscated JavaScript & Client-Side Secret Extraction

Challenge

This level required analyzing a webpage whose logic was hidden inside a minified and partially obfuscated JavaScript file. The challenge involved locating an encrypted value and understanding the client-side code used to decrypt it. The goal was to reverse engineer the logic to extract the real key needed for the next level.

Enumeration

Opening Developer Tools and navigating to: Inspect → Sources revealed a compact, hard-to-read JavaScript file.

This file contained:

- A stored encrypted value
- A small, minified decryption function

Minified JavaScript is extremely common in real-world sites. Developers compress their code to reduce file size and speed up loading. While it appears difficult to read, minification is not real security it only hides formatting, not functionality.

With features like the a debugger or simply by stepping through the code, I examined how the function manipulated the encrypted value.

Exploitation

The extraction process involved:

1. Finding the encrypted key inside the JavaScript source code.
2. Identifying the decryption function, even though it was compressed into a single line.
3. Using the browser's console or debugger to run the function manually on the encrypted value.
4. Capturing the decrypted string that the challenge expected.

Because the decryption logic was executed on the client side (in the browser), nothing was actually hidden and everything was available to copy, run, and analyze.

In short:

- The site stored a secret in the JS
- The JS contained the code to decode it
- Running the function revealed the real key

Validation

Once the decrypted value was obtained:

- It was entered into the challenge page
- The site validated it
- The next-level URL was displayed

Security Lesson

This level reinforces a critical principle:

> Anything stored or executed client-side is not secure.
> Minifying JavaScript may look like obfuscation, but it does not protect secrets.

Secrets, keys, and validation logic must always be handled server-side. If the browser can see it, the attacker can see it.

Level 13 – Hidden Assets & Non-Rendered Media

Challenge

This level focused on identifying resources that a webpage loads but does **not** visibly display. Even if an image, script, or file is hidden in the user interface, the browser still downloads it — which means it can be viewed through Developer Tools.

The task was to locate a hidden image file that contained the next clue.

Enumeration

To inspect all assets the page was loading, I opened Developer Tools and navigated to: Inspect - Network

Then I refreshed the page so the Network tab would repopulate with everything the browser downloaded — images, JavaScript files, CSS, fonts, etc.

Most of these files were marked red or triggered errors, but one PNG file loaded successfully. This was a strong sign that the image was important, especially since it did not appear anywhere on the visible webpage.

For non-technical readers:
Even if a picture doesn't show up on the screen, the browser might still download it in the background. The Network tab exposes everything, even invisible elements.

Exploitation

Once I saw the successfully loaded PNG file in the Network list:

1. I clicked the file name.
2. The preview panel displayed the image.
3. The image contained the information needed for the next level (flag, key, or URL).

No special decoding or tools were needed — simply viewing the non-rendered asset revealed the answer.

Validation

To confirm the challenge was completed:

- The hidden PNG file was opened in the Network tab
- The image clearly showed the next-level information
- Entering that information into the challenge page successfully progressed the challenge

---

Security Lesson

This level demonstrates an important principle:

> If a webpage loads a resource, users can access it — even if it’s not displayed.

Developers sometimes leave sensitive data inside hidden images, unused scripts, or “invisible” elements. But anything loaded by the browser is inherently accessible.

Conclusion

This CTF walkthrough demonstrated how seemingly simple web challenges can reveal important lessons about real-world security. Across the levels, we explored a wide range of vulnerabilities from hidden fields, exposed JavaScript logic, and cookie manipulation to open redirects, NoSQL injection, and improper handling of JWTs. Each challenge reinforced a core principle: anything executed or stored on the client side can be viewed, modified, or exploited.

Through hands on investigation with browser tools, curl, Burp Suite, Linux utilities, and basic scripting, the path from initial prompt to final flag showcased how attackers think and how developers must defend. More importantly, it highlighted how small oversights like trusting URL parameters, revealing stack traces, or embedding secrets in JavaScript can lead to major weaknesses.

Overall, this CTF served as both a learning experience and a practical reminder that secure applications require careful validation, server-side checks, and an understanding of how users can interact with unintended parts of a system. Each level built confidence in analyzing, testing, and exploiting common patterns, helping develop the mindset needed for penetration testing and web application security.

Note:

Certain details, screenshots, and challenge artifacts have been intentionally omitted at the request of Bloomberg which hosted this CTF. All information included here reflects general concepts and personal methodology without disclosing proprietary material or sensitive challenge content.

Glossary of Terms

JavaScript
A programming language that runs inside your web browser. It controls how interactive parts of a website behave.

Minified JavaScript
A compressed form of JavaScript where all spaces and line breaks are removed. It makes code harder to read but does not actually secure it.

JWT (JSON Web Token)
A small data package used by websites to store user identity information. It contains three parts: a header, a payload, and a signature. The payload is only encoded, not encrypted, meaning anyone can read its contents.

Base64 Encoding
A method of turning data into a text format using only readable characters. Used in web applications to safely store or transmit information.

cURL
A command-line tool used to send custom web requests without using a browser. It is often used for security testing, sending headers, and accessing URLs directly.

Burp Suite
A professional web security testing tool. It allows you to intercept, view, and edit traffic between your browser and a server. Its Repeater tool is commonly used to resend and modify requests.

HTTP Headers
Extra pieces of information attached to a web request. Examples include the type of browser you're using, where the request came from, and authentication data. Developers sometimes leak internal information here by mistake.

Cookie
A small piece of information saved in your browser. Websites use cookies to remember who you are, whether you are logged in, and your preferences.

Origin Header
A header that tells the server where the request came from. It can be faked using tools like cURL if the server does not properly validate it.

Stack Trace
A detailed error message that shows the internal steps the server took before failing. Exposing stack traces is dangerous because they reveal backend details like file paths and programming languages.

NoSQL Injection
A type of injection attack used against NoSQL databases such as MongoDB. It works by sending unexpected data that changes the way the database handles a query.

SQL Injection
An attack where a user inserts malicious text into a field or URL to alter how a database query is executed. It can expose, modify, or delete data if the server does not validate inputs.

URL Parameter
Information added to the end of a web address to control what the server does. For example: q=search or page=2.

Open Redirect
A flaw where a website accepts a user-provided link and redirects the user to it without checking if it is safe. Attackers may use it to redirect victims to harmful websites.

URL Encoding
A way to represent special characters in a URL using percent symbols. For example, a forward slash can be encoded as %2F. Double encoding means encoding it twice to bypass filters.

Network Tab (DevTools)
A browser tool that shows all the files and resources a webpage loads. Useful for spotting hidden images, scripts, or API requests.

Strings Command
A Linux terminal tool that extracts readable text from a compiled program. Often used to find passwords or clues inside binary files.

Client-Side Logic
Any code that runs in the user's browser, such as JavaScript. It should never be used for storing sensitive information because users can always view and modify it.

Server-Side Logic
Code that runs on the server. It handles secure tasks like authentication, data validation, and database operations, because users cannot tamper with it.

Hash
A one-way transformation used to turn text into a fixed-length fingerprint. Commonly used to store passwords. Hashes cannot be reversed mathematically, but weak ones can be guessed through dictionary attacks.

Dictionary Attack
A technique where an attacker tries many possible passwords from a prepared list until one matches the target.

Payload
Any specially crafted input used to test or exploit a vulnerability. This can include encoded URLs, injection strings, or JavaScript snippets.

Debugger
A browser tool that allows you to pause and step through JavaScript code line by line to see how it behaves.
