**#The challenge**

https://github.com/kaufland-ecommerce-security/web-challenge

Uncover the flag hidden within the protected /admin page by using your hacking skills to exploit vulnerabilities in the system.  
  
Document your methodology and the underlying causes of each vulnerability found during your penetration testing.  
Can you outsmart the security measures and claim victory?

**#The walkthrough**

### I ran the container, built the image provided by Kaufland E-commerce, and run the service that is available on my host machine at [http://localhost:5001](http://localhost:5001/)

### Docker Compose configuration file written in YAML format defines two services, "web" and "nginx", which are containers created and managed by Docker.

### The "web" service is defined with a build context of "./web", meaning Docker looks for a Dockerfile in the "web" directory to build the container image. The "restart" property is set to "on-failure", which means the container automatically restarts if it fails for any reason.

### The "nginx" service is defined with a build context of "nginx", meaning Docker looks for a Dockerfile in the "nginx" directory to build the container image. The "command" property is set to "nginx -g "daemon off;", which means the container starts the Nginx server and runs it in the foreground. The "container\_name" property sets a custom name for the container. The "ports" property maps port 5001 on the host to port 5001 in the container, allowing the Nginx server to be accessed through [http://localhost:5001/](http://localhost:5001/" \t "_new).

### I opened the web browser and navigate to [http://localhost:5001/](http://localhost:5001/I)

### 

### Then I manually checked if there is a login page ([http://localhost:5001/](http://localhost:5001/I)login)

### 

**## 1**

### I checked default credentials like “admin:admin” to understand the authentication mechanism. In BurpSuite I got the HTTP request with *Authorization: Basic and encoded “admin:admin”* in base64 format

GET /login/ HTTP/1.1

Host: localhost:5001

Cache-Control: max-age=0

**Authorization: Basic YWRtaW46YWRtaW4=**

{truncated}

Authorization: Basic is an HTTP header that is used to send authentication credentials in plaintext, encoded in base64 in a request. HTTP response with ‘WWW-Authenticate: Basic realm="must login"’ indicates that the server is using HTTP Basic Authentication, which is a simple authentication scheme where the client sends a username and password in the HTTP request headers. The realm parameter specifies the authentication realm, which is a string that identifies the domain or context of the authentication. In this case, the realm is "must login", indicating that the client needs to provide valid credentials to access the resource. When a server receives Authorization: Basic header, it can decode the base64 string and extract the username and password for authentication purposes.

Because the authorization header had to be sent in the header of each HTTP request, the web browser needed to cache credentials for a reasonable period of time to avoid constantly prompting the user for their username and password. Caching policy differs between browsers.

In the case of Basic Authentication, the Brute-Force protection cannot be solved with a CAPTCHA solution, only with temporary or permanent user or IP address lockouts. An attacker can lock out a user if the username is revealed and the application implements a user lockout. If no user lockout is implemented, an attacker might be able to perform a brute-force attack. However, an attacker requires network-level access to perform the above-described attack scenario.

Authorization Basic is a simple authentication scheme that uses a username and password to authenticate a user. It does not provide a session ID or a token to identify a session, which means that it is stateless and does not require session invalidation or expiration.

However, this also means that the server cannot revoke a session or log out a user because there is no session ID or token to invalidate. Once the client has been authenticated with the username and password, it can continue to make requests to the server until the session is terminated.

Lack of session invalidation and expiration is not an issue when using Authorization Basic because it does not use sessions or tokens. However, it is important to note that Authorization Basic is not a secure authentication scheme as the username and password are sent in plain text and can be intercepted by attackers

### #Recomendation:

It is recommended to implement modern authentication solution with CAPTCHA protection to prevent the Brute-Force attacks and temporary account lockouts. . It is recommended to use stronger authentication mechanisms such as OAuth or OpenID Connect.

**## 2**

### Meantime I ran “dirb” tool with “big.txt” wordlist to discover folders and pages with the command:

dirb http://localhost:5001/ /Users/sdemchyk/Downloads/PayloadsAllTheThings-master/big.txt

### 

### “Dirb” discovered “.htpasswd” file that is used by web servers to store user credentials for Basic Authentication. It is typically used to protect web pages or resources that require authentication, such as an administration dashboard or an API. The file contains a list of usernames and hashed passwords, where each line corresponds to one user. When a user tries to access a protected resource, the web server checks the .htpasswd file to authenticate the user by matching the supplied username and password with those stored in the file.

### Then I navigated it to <http://localhost:5001/.htpasswd> and the file was downloaded to my host as following screenshot shows:

### 

### This file contains credentials where username is “kaufland” and password “pAssWordo&@dzoo”

### 

### I could access “.htpasswd” file because the web server is configured to serve files from a directory that includes the .htpasswd file, and the web server is not configured to prevent access to this file. So just guessing the path to the .htpasswd file, gives access file directly through the browser

### #Recomendation:

### To prevent this type of vulnerability, the web server should be properly configured to restrict access to sensitive files, such as the .htpasswd file, and to ensure that these files cannot be accessed through the browser. This can be done by placing the sensitive files in a directory that is not accessible to the public or by using access control mechanisms, such as .htaccess files, to restrict access to the files. Additionally, web servers should be regularly updated and patched to ensure they are not vulnerable to known exploits.

### To restrict access to sensitive files like the .htpasswd file, configure the webserver to deny access to these files from external requests. Here are some ways to do it:

### Move the sensitive files outside the web root directory: By moving sensitive files outside the web root directory, they will not be accessible through a web browser.

### Use the 'Deny from all' directive: This directive can be added to the .htaccess file to deny access to a file or directory from external requests. For example, to deny access to the .htpasswd file, add the following line to the .htaccess file in the directory containing the file:

<Files .htpasswd>

Deny from all

</Files>

### Use server configuration files: Instead of using .htaccess files, use server configuration files to restrict access to sensitive files. This is generally considered more secure because it's harder for an attacker to modify the server configuration file. For example, in Node.js, you can use middleware to protect sensitive files, such as the .htpasswd file, from being accessed by unauthorized users. One common approach is to use the express-basic-auth middleware, which provides basic authentication for routes and paths.

### Here's an example of how to use express-basic-auth to restrict access to the .htpasswd file:

const express = require('express');

const basicAuth = require('express-basic-auth');

const app = express();

// Protect the .htpasswd file with basic authentication

app.use('/.htpasswd', basicAuth({

users: { kaufland: ' pAssWordo&@dzoo ' },

challenge: true,

realm: 'must login'

}));

// Start the server

app.listen(5000, () => {

console.log('Server started on port 5000');

});

### In this example, the basicAuth middleware is used to protect the .htpasswd file. The middleware is configured with a single user (username: “kaufland”, password: “pAssWordo&@dzoo”), a challenge prompt, and a realm of must login. This means that when a user attempts to access the .htpasswd file, they will be prompted to enter a username and password, and will only be able to access the file if they provide the correct credentials.

**## 3**

### Also would like to mention that web application has HTTP cleartext vulnerability. HTTP cleartext protocol vulnerability refers to the risk associated with using the HTTP protocol to transmit sensitive information over the internet. HTTP is an unencrypted, clear-text protocol, which means that any data transmitted over it can be intercepted and read by an attacker with access to the network traffic. As a result, sensitive information, such as usernames, passwords, credit card details, or other personal data, transmitted over HTTP can be easily intercepted and compromised, potentially leading to identity theft, financial loss, or other malicious activities.

### #Recomendation:

### To address this vulnerability, use HTTPS protocol, which provides encryption and secure communication over the internet, making it much more difficult for attackers to intercept and read the transmitted data. Additionally, it is also important to ensure that web applications and servers are properly configured to enforce the use of HTTPS and to redirect all HTTP requests to HTTPS to prevent any potential vulnerabilities.

**## 4**

### The web application login form doesn’t have brute-force protection. I performed manual checks by trying to log in with a valid username but a wrong password. After several unsuccessful attempts we successfully logged in with the proper password. At the same time there was no CAPTCHA displayed after the failed login attempts. Without proper brute-force protection, an attacker may gain access to the application by trying possible passwords for valid usernames (e.g., based on a password dictionary).

### #Recomendation:

### Implement CAPTCHA protection by considering the following security aspects:

### Implement an account lockout and bind the “failed login attempts” counter to the username (at server side), because:

### If the application binds the counter to the session identifier, the attacker can force the application to regenerate the session ID (or just reset it manually) then the application could count all attempts as the first one.

### If the application binds the counter to the client IP address, the attacker might be able to spoof the client IP address via HTTP headers (e.g., X-Forwarded-For, X-Real-IP).

### The CAPTCHA verification doesn’t have to be enforced for the first login attempt for usability reasons. The first (or the first two) login attempts do not have to be protected with CAPTCHA but if the user fails twice in a row, he would have to provide the proper CAPTCHA value for the following login attempts.

**## 5**

### So we were able to login and get the form with url field.

### When a GET request is made to this endpoint, the web app checks if the request has a query parameter called url. If the url parameter is present, the web app parses the URL and checks if it's a local URL (i.e., it's pointing to the same machine where the web app is running). If it's not a local URL, the web app uses the Axios library to make a GET request to the URL specified in the url parameter and returns the response data to the client. If it's a local URL, the web app sends a "Not good" response to the client.

### 

### The “uri” parameter on the application’s page used to redirect to the address provided as the parameter value outside of the domain. For example, the following URL redirects the user to the “http://attacker.com” site.

### 

### The application did not filter input on the “uri“ parameter value. Thus redirection to external domains was possible. A possible attack scenario of this vulnerability is that an attacker creates a special link, which redirects the user to a domain similar to the Kaufland sites. On this malicious site, the attacker displays a login screen similar to the original one and convinces the user to provide his/her credentials. The attacker could capture the provided credential as he controls this malicious site. The client’s trust in the Kaufland domains may increase the possibility of a successful attack. A successful attack can lead to the theft of valid user credentials by an attacker. Furthermore, an attacker can use this vulnerability to mount further attacks (e.g., phishing attacks).

### #Recomendation:

### It is recommended to use the affected parameter as a relative path and not allowing full URLs as input. If the application has to take full URLs from parameters, it is recommended to create a whitelist, which contains all valid redirection URLs. The application should accept URL format only, which are present on the whitelist.

**## 6**

The application did not set the following HTTP headers.

HTTP/1.1 200 OK

Server: nginx/1.23.4

Date: Sat, 15 Apr 2023 10:25:19 GMT

Content-Type: text/html; charset=utf-8

Content-Length: 14886

Connection: close

X-Powered-By: Express

ETag: W/"3a26-UlYB+LZer0/LTyPYZJbwICsQoEs"

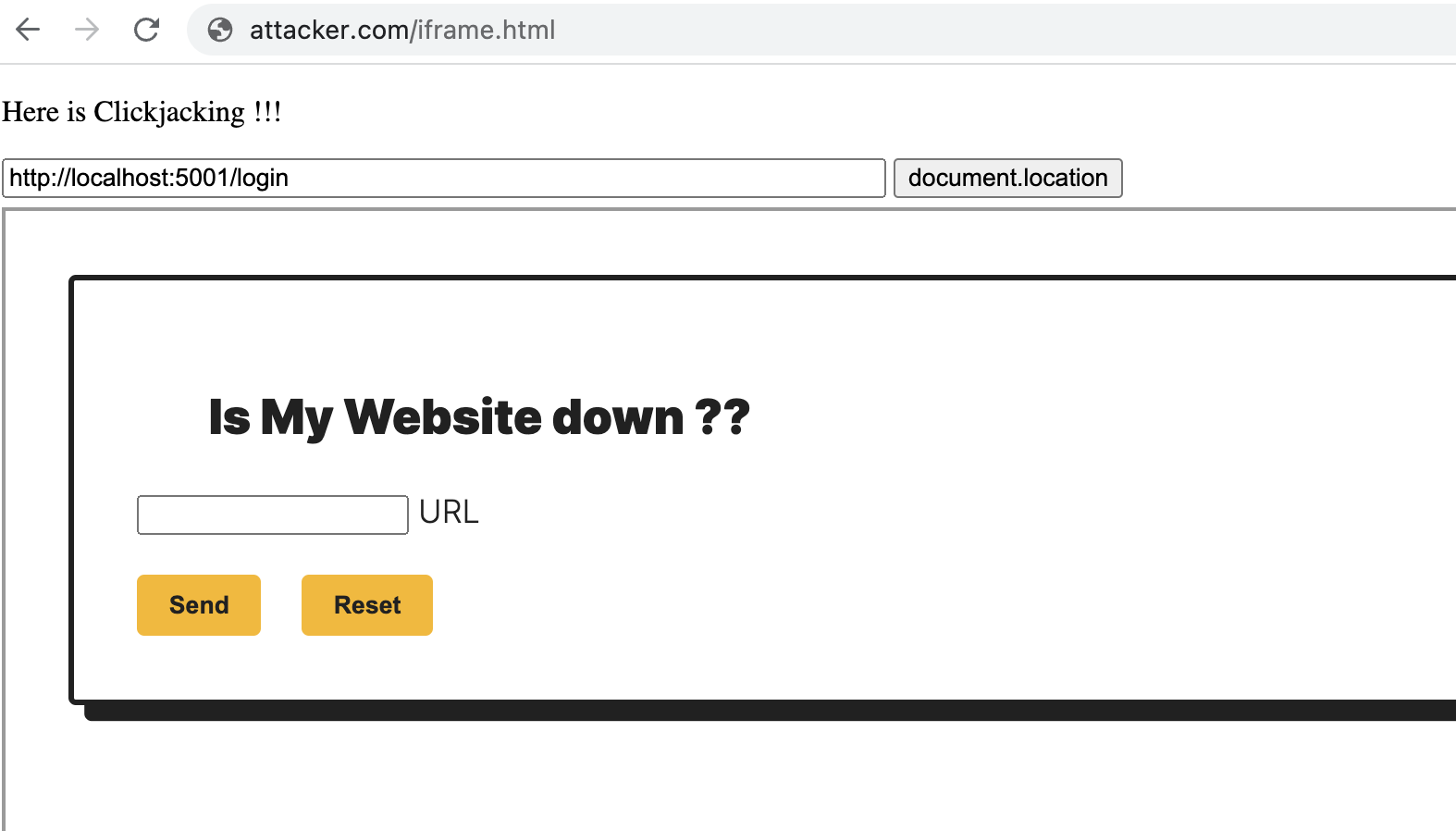
Without the “Content-Security-Policy” header, an attacker might be able to access sensitive information or perform actions in the application by using “Clickjacking” or Cross-Site Scripting (XSS) attacks, and these attacks might remain undetected. Furthermore, without the “X-Frame-Options” header, the application might not be fully protected against “Clickjacking” attacks (if the user is using an older or unsupported browser).

If the HTTPS communication is not enforced by the “Strict-Transport-Security” header, an attacker might be able to change the HTTPS links in the network traffic to cleartext HTTP links during a Man-in-the-Middle (MitM) attack and force the user to use a cleartext communication channel. Furthermore, an attacker may carry out an SSL MitM against the users if they accept the attacker’s invalid certificate.

Without the X-Content-Type-Options header set to “nosniff”, browsers may perform MIME sniffing to guess the correct MIME type if certain conditions are met. If an attacker is able to inject their malicious code (e.g., HTML, JavaScript) into the response, the browser may “sniff” the content (e.g., “text/html”), identify it as an executable, and run the malicious payload.

Without the “Referrer-Policy” header, the browser will send the full URL of the page that made the request as the referrer by default. This can potentially reveal sensitive information and compromise the security and privacy of the website and its users.

Thought I discovered that web application is vulnerable to Clickjacking attacks. It was possible to load the webpage into an iframe, as the following screenshot demonstrates:



Even though the URL field doesn’t have input validation, HTML form doesn't have any JavaScript code (no necessary elements, such as the <form> tag, <input> fields), so there was XSS by clicking on the Send button. XSS attacks require malicious JavaScript code to be executed in the victim's browser to inject and execute malicious code.

#Recomendation:

It is recommended to set the “Strict-Transport-Security” header in order to mitigate the risk of Man-in-the-Middle (MitM) attacks, set the max-age parameter of the "Strict-Transport-Security" header to at least 10,368,000 seconds (120 days) and ideally to 31,536,000 (one year) to mitigate the risk of Man-in-the-Middle (MitM) attacks.

Also, it is recommended to use the Content Security Policy (CSP) Level 2 (http://www.w3.org/TR/CSP2/) HTTP header by considering the following security decisions:

* Use with frame-ancestors directive to prevent “Clickjacking” attacks. However, we note that versions of Internet Explorer older than IE 11 do not support the CSP header, therefore, we also recommend setting the “X-Frame-Options” HTTP header with the “SAMEORIGIN” or “DENY” values as a fallback measure.
* Use with default-src directive set to none and allow only those sites, which are necessary for the application functionality (white-list based approach).
* Use with script-src directive to prevent script execution from untrusted sources and mitigate the Cross-Site Scripting (XSS) attacks. For adequate protection, this directive should not contain the unsafe-inline or unsafe-eval keywords.

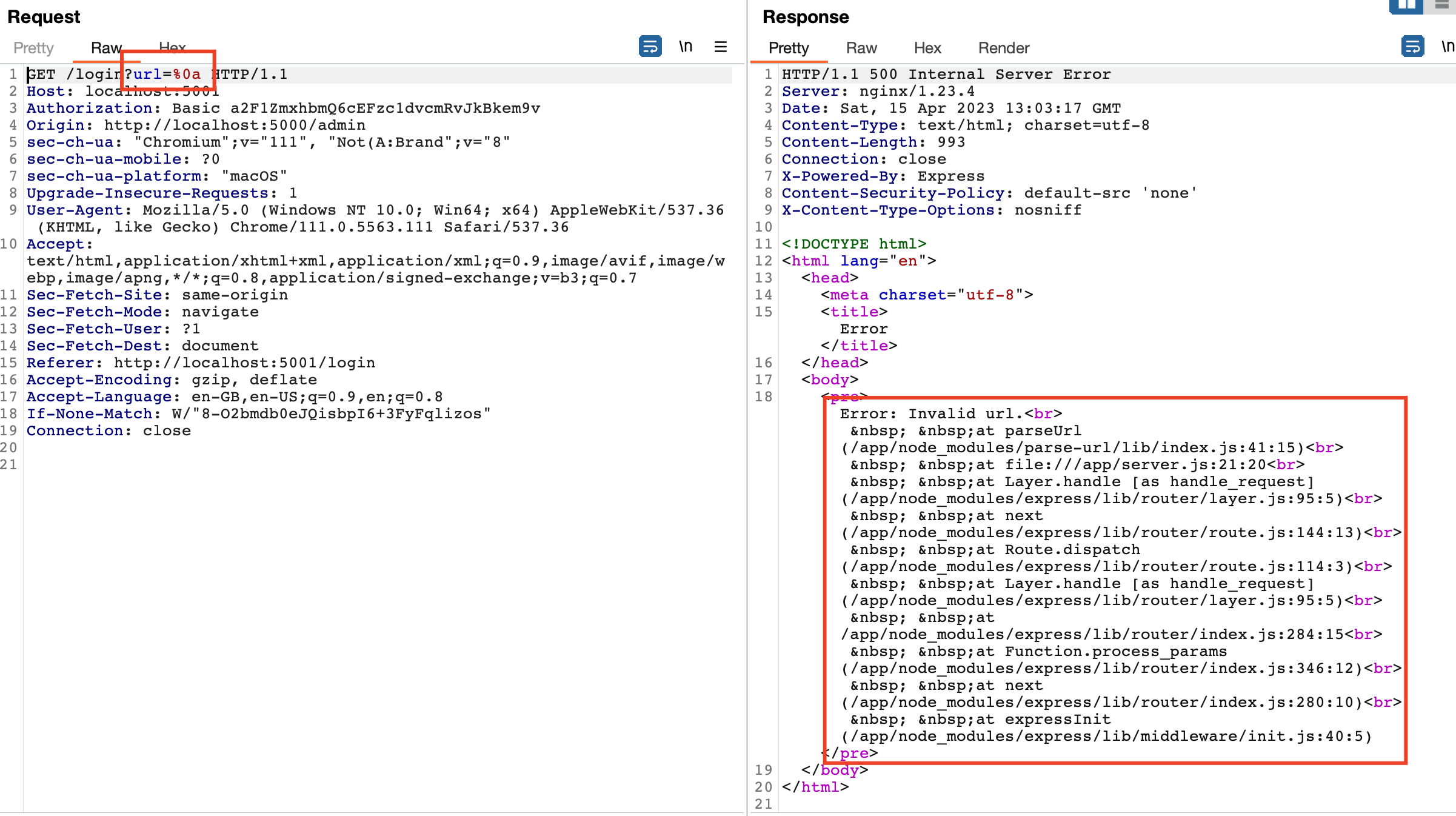
Furthermore, in case if the web application will be developed with other features and javascript will be implemented, it is recommended to set the "X-Content-Type-Options" header to "nosniff" on the web server side. In the case of web application development, it’s recommended to set the "Referrer-Policy" header to an appropriate value, such as "no-referrer" or "strict-origin-when-cross-origin".

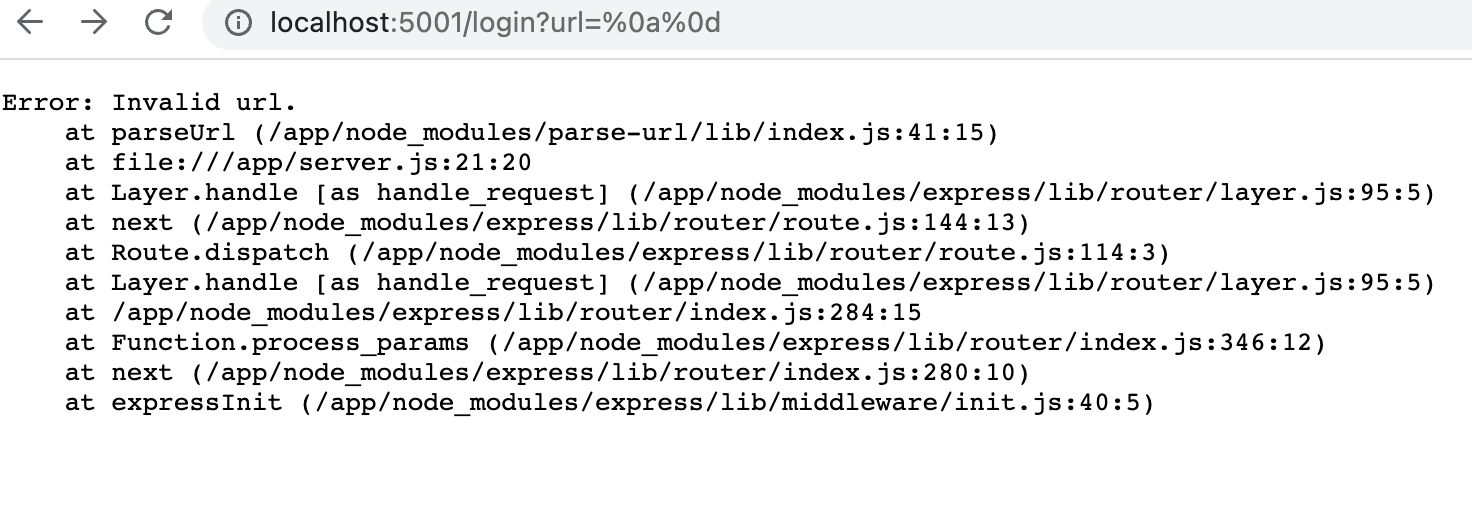
## Note:

### URL HTTP request doesn't use a database or doesn't use user input in SQL queries. It uses the Axios library. Therefore, the URL parameter is not vulnerable to SQL injection. Axios is a popular JavaScript library for making HTTP requests from a web browser or Node.js.

**## 7**

The application sent back error messages to the client, which contained valuable information for an authenticated attacker. For example, the following error message was sent back during the testing:



This verbose error message contains information that provides an insight into the application's internal architecture, the applied software components, the developer’s variable nomenclature scheme etc. As a result, they could help an attacker formulate further attacks. As well as server version as following HTTP response shows:

HTTP/1.1 200 OK

**Server: nginx/1.23.4**

Date: Sat, 15 Apr 2023 10:25:19 GMT

Content-Type: text/html; charset=utf-8

Content-Length: 14886

Connection: close

X-Powered-By: Express

ETag: W/"3a26-UlYB+LZer0/LTyPYZJbwICsQoEs"

#Recomendation:

It is recommended to remove technical and detailed version information from HTTP response headers and use standard error messages without exposing any details about the application and the deployment system.

The application should use custom error pages for HTTP error codes (e.g., HTTP 404, HTTP 403, HTTP 500) and should display standardized error messages in case of application errors as well. Detailed error messages should be logged at server side.

**## 8**

So come back to the original task “Uncover the flag hidden within the protected /admin page by using your hacking skills to exploit vulnerabilities in the system.”

I need to access /admin page. However even being login as a user I still can’t access it.

Reviewing code, this is a Node.js web application that uses the Express framework to create HTTP endpoints. The web app listens on port 5000 and has two endpoints: /login and /admin. From the server.js it is clear when a GET request is made to this endpoint, the web app sends a response with the string.

"FLAG : o&@dzooSeFZ34ml6M0Pd5YZb%".

The web app also has a middleware function called isLocal that checks if the client is making the request from the same machine where the web app is running. If the client is not making the request from the same machine, the middleware sends a response to the client with the message "You're not locally". However, this middleware function is not used in the current code.

So thinking logically accessing the /admin panel of web application hosted on http://localhost:5001 depends on how the authentication and authorization logic for the admin panel is implemented.

Assuming it is implemented in a way a login system that verifies the user's credentials and grants them access to the admin panel, we could navigate to <http://localhost:5001/admin> in the browser. If I am not logged in, I should be redirected to the login page. However, /the admin page doesn’t have this redirection to the login page, so I assume it was not implemented as an authentication and authorization system for the admin panel but for some network access limitations.

Come back to the code enumeration.

Docker Compose configuration file written in YAML format defines two services, "web" and "nginx", which are containers that will be created and managed by Docker.

The "web" service is defined with a build context of "./web", meaning Docker looks for a Dockerfile in the "web" directory to build the container image.

The "nginx" service is defined with a build context of "nginx", meaning Docker looks for a Dockerfile in the "nginx" directory to build the container image. So, it means that it has to be "nginx.conf” file. The "command" property is set to "nginx -g "daemon off;", which means the container starts the Nginx server and runs it in the foreground. The "container\_name" property sets a custom name for the container. The "ports" property maps port 5001 on the host to port 5001 in the container, allowing the Nginx server to be accessed through http://localhost:5001/.

Reading the configuration file for the Nginx web server code I got understanding that it sets up a reverse proxy to balance the load between a group of servers that run a web application.

The file includes the following settings:

The upstream directive defines a group of servers with the least\_conn load balancing method. The group includes one server named web running on port 5000.

The server directive defines the server block for the Nginx web server. It listens on port 5001 and uses a regular expression pattern to match any server name.

The location directive defines the location block for handling requests that match the specified URL paths. The /login location requires basic authentication with the specified .htpasswd file and proxies the request to the upstream server group defined earlier.

upstream loadbalance {

least\_conn;

server web:5000;

}

server {

listen 5001;

server\_name ~^(.+)$;

root /etc/nginx;

location /login {

auth\_basic "must login";

auth\_basic\_user\_file /etc/nginx/.htpasswd;

proxy\_pass http://loadbalance;

}

**location ~\* /admin {**

**deny all;**

**}**

}

The location directive for the /admin location block is configured to deny all requests to the /admin URL path.

If the "deny all" rule set is defined in the nginx.conf file, it is intended to block **all** incoming requests.

To be able to access the /admin page, I need to change the configuration of the Nginx server by removing the “deny all” directive or adding a new directive that allows access to the page. So I can change the regular expression in the server\_name directive to match a specific domain or IP address, but this would require changes to the client-side code to send requests to that domain or IP address.

The reason is that the location /login block is only authenticating the user, and proxying the request to http://loadbalance which is the upstream server block defined in the upstream directive. Since the location ~\* /admin block is still denying all requests, the user won't be able to access the /admin resource even after authentication.

To allow authenticated users to access the /admin resource, I need to update the location ~\* /admin block to include a valid allow directive, and also ensure that the user is authenticated by setting the auth\_basic and auth\_basic\_user\_file directives appropriately.

So to get the Flag I modified the nginx.conf file with following code to allow any requests:

…

location ~\* /admin {

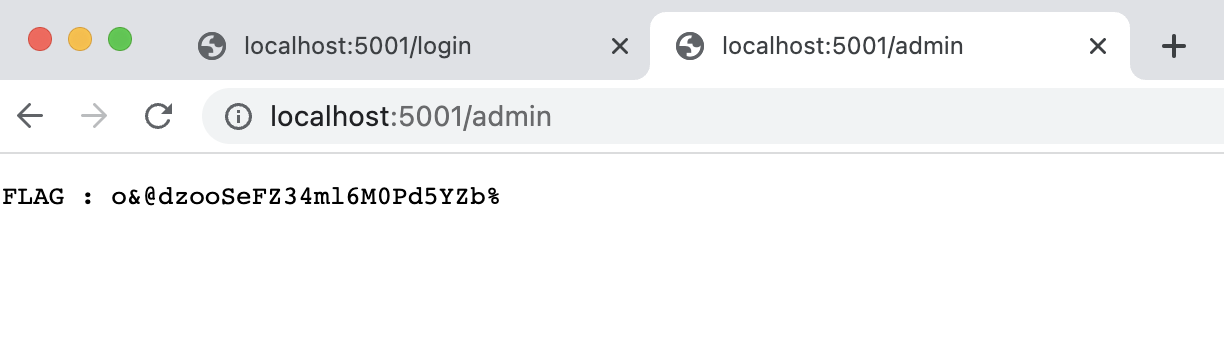
allow all;

proxy\_pass http://loadbalance;

}

…

Then Build the image and run the container and get the Flag via vising /admin page.

****#Recomendation:**

Hovewer it’s not recommended to set up “allow all”.

To allow access to specific IP in nginx.conf file the server block contains the server\_name line. Modify the regular expression in the server\_name directive to match the specific domain or IP address that only admin has to access with.

For example, if use the domain example.com, you can change the server\_name line to:

server\_name kaufland.com;

Or, if use the specific admin IP address 192.168.1.100, change the server\_name line to:

server\_name 192.168.1.100;

Note: also need to update the client-side code to send requests to the new domain or IP address.

To allow access to authenticated users, use the satisfy any directive along with the allow directive in the location block for /admin. Here's an example configuration:

upstream loadbalance {

least\_conn;

server web:5000;

}

server {

listen 5001;

server\_name ~^(.+)$;

root /etc/nginx;

location /login {

auth\_basic "Restricted";

auth\_basic\_user\_file /etc/nginx/.htpasswd;

proxy\_pass http://loadbalance;

}

**location /admin {**

**auth\_basic "Restricted";**

**auth\_basic\_user\_file /etc/nginx/.htpasswd;**

**satisfy any;**

**allow 192.168.1.0/24;**

**deny all;**

**proxy\_pass http://loadbalance;**

**}**

}

In this example, the satisfy any directive allows access if the user is authenticated OR if their IP address is in the allowed range. The allow directive specifies the IP address range that is allowed to access the /admin location. The deny all directive denies access to all other IP addresses.

### ## General recommendations:

### Here are some tips on how to protect the application:

### Use strong authentication: Ensure that users are authenticated before allowing them to access any sensitive resource. Use strong password policies and encourage users to choose unique, strong passwords.

### Use HTTPS: Always use HTTPS to secure communications between the client and server. HTTPS ensures that data exchanged between the client and server is encrypted, preventing attackers from intercepting and reading sensitive information.

### Use access control: Use access control mechanisms to limit access to sensitive resources only to authorized users. Make sure that users are only able to access the resources they are authorized to access.

### Use secure coding practices: Ensure that your code is secure and free from vulnerabilities that could be exploited by attackers. Use secure coding practices and follow best practices when writing code.

### Perform regular security audits: Regularly perform security audits of your application to identify vulnerabilities and fix them before they can be exploited by attackers.

### Keep your software up to date: Keep your software and dependencies up to date to ensure that any known security vulnerabilities are patched.

### Use security tools: Use security tools such as firewalls, intrusion detection systems, and anti-virus software to help protect your application from attacks.