

Name & ID:

Mohamed Ayman Yakout 2305104

Ahmed Sameh Ragab 2305122

Ahmed Mohamed Saeed 2305126

Tools and Technologies Used: Node.js / Express.js: The target application environment.

OWASP ZAP: A tool for dynamic vulnerability discovery (DAST).

Postman: Used to perform manual attacks and create proof-of-concept exploits (PoCs) .

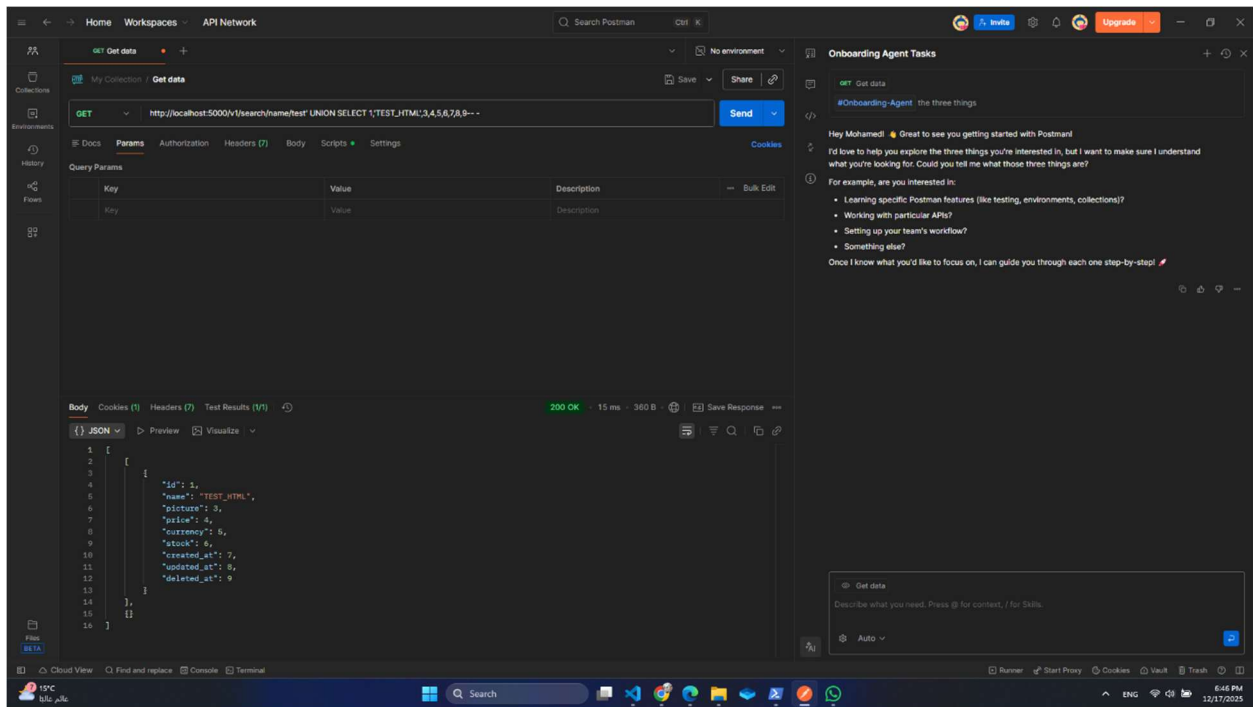
Semgrep: A static application security testing (SAST) tool for source code analysis .

Custom Semgrep Rules: Custom-written rules to detect project-specific vulnerability patterns .

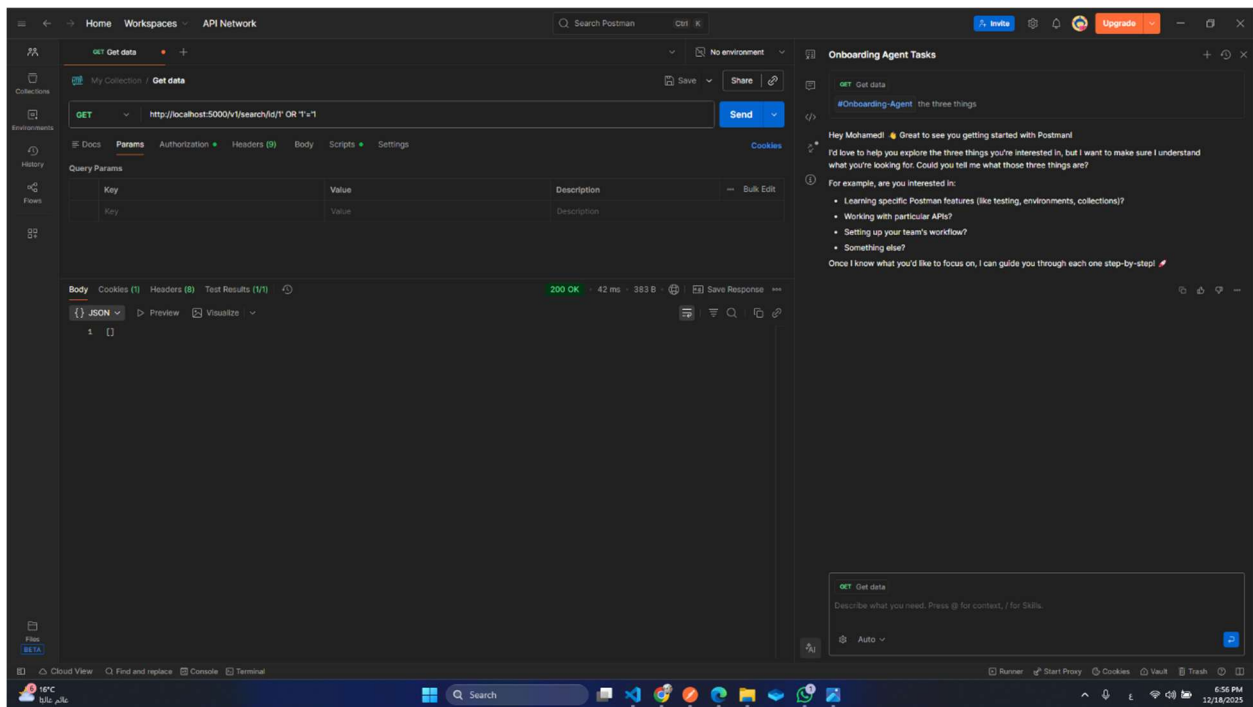
Git: Version control system used to manage code changes and security fixes.

Hardcoded Secrets:

1-SQL Injection(Before):

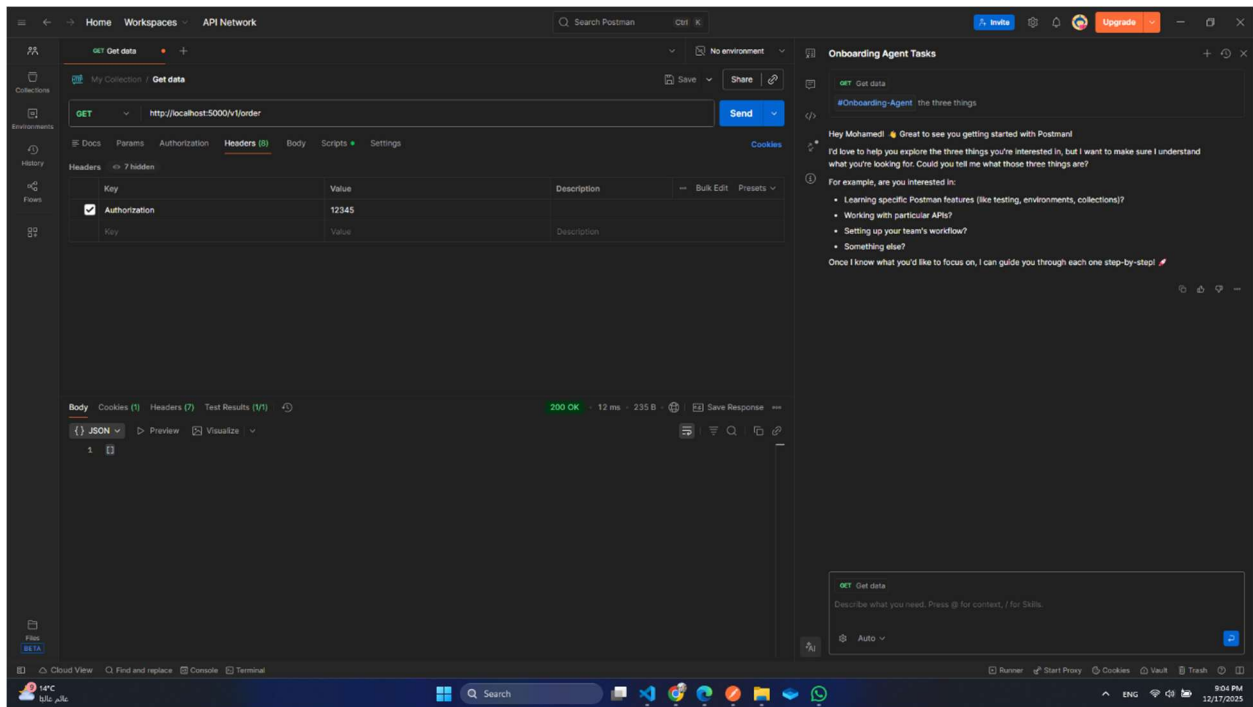


After:

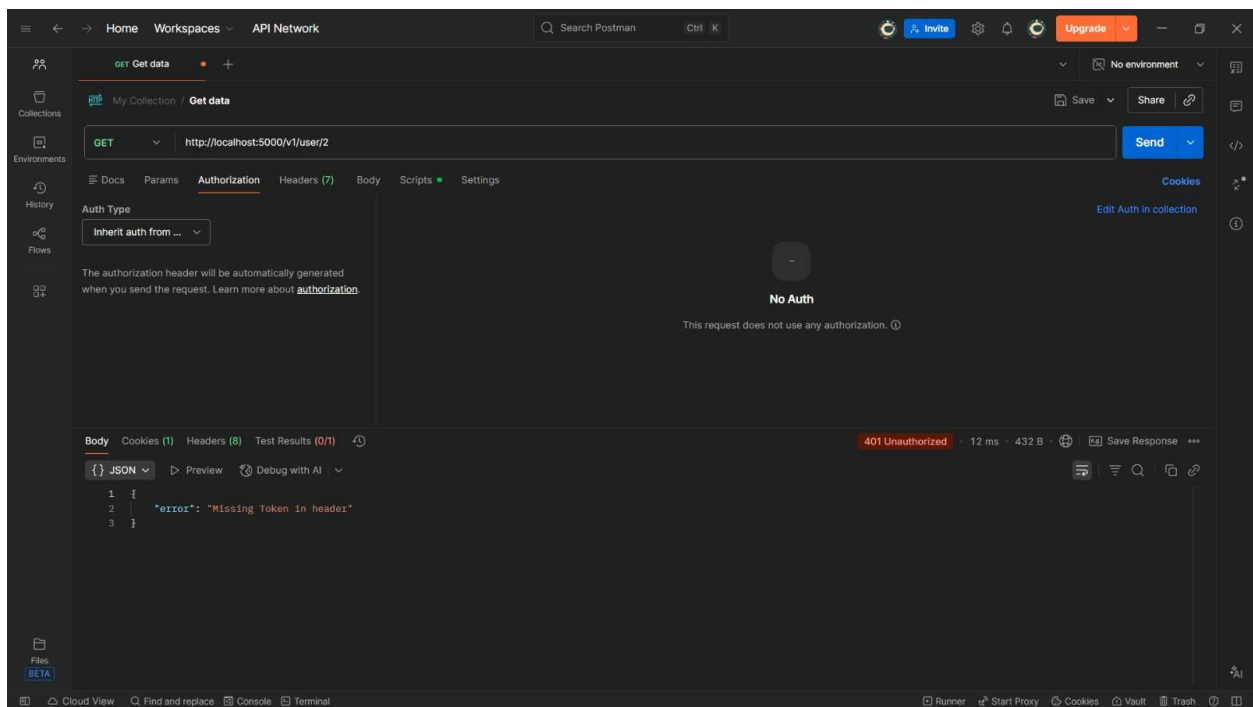


2-XSS(Before):



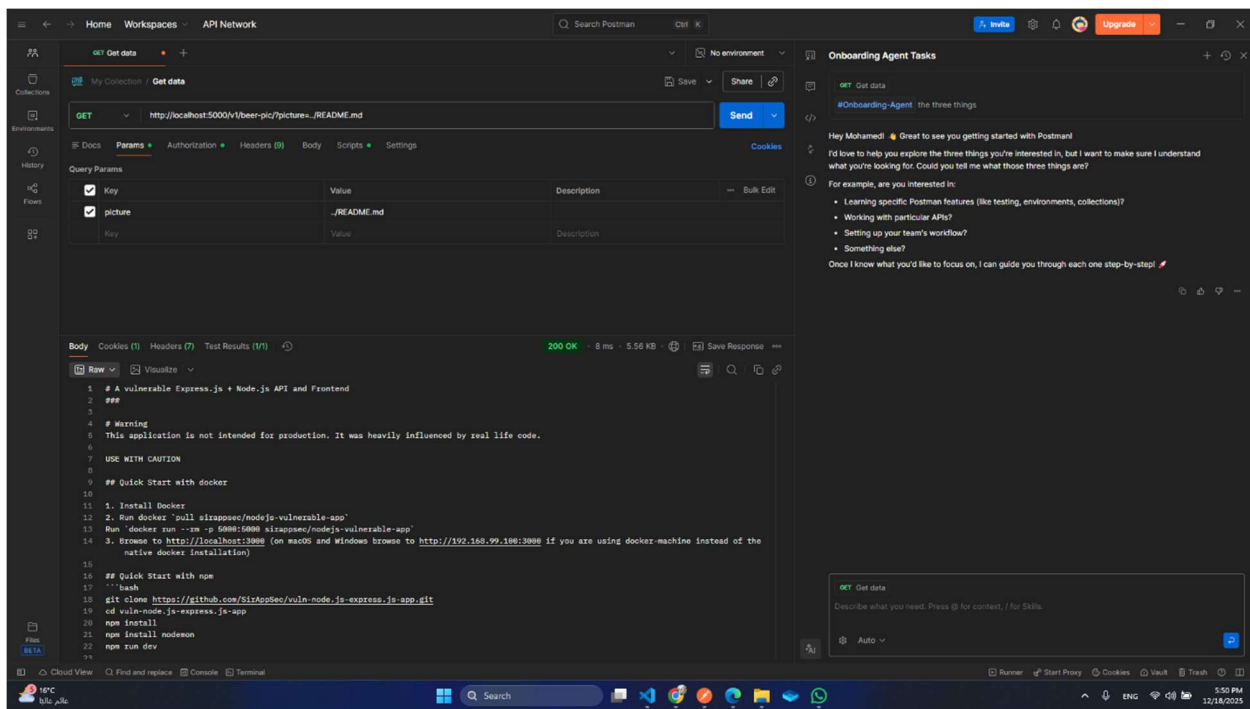


After:

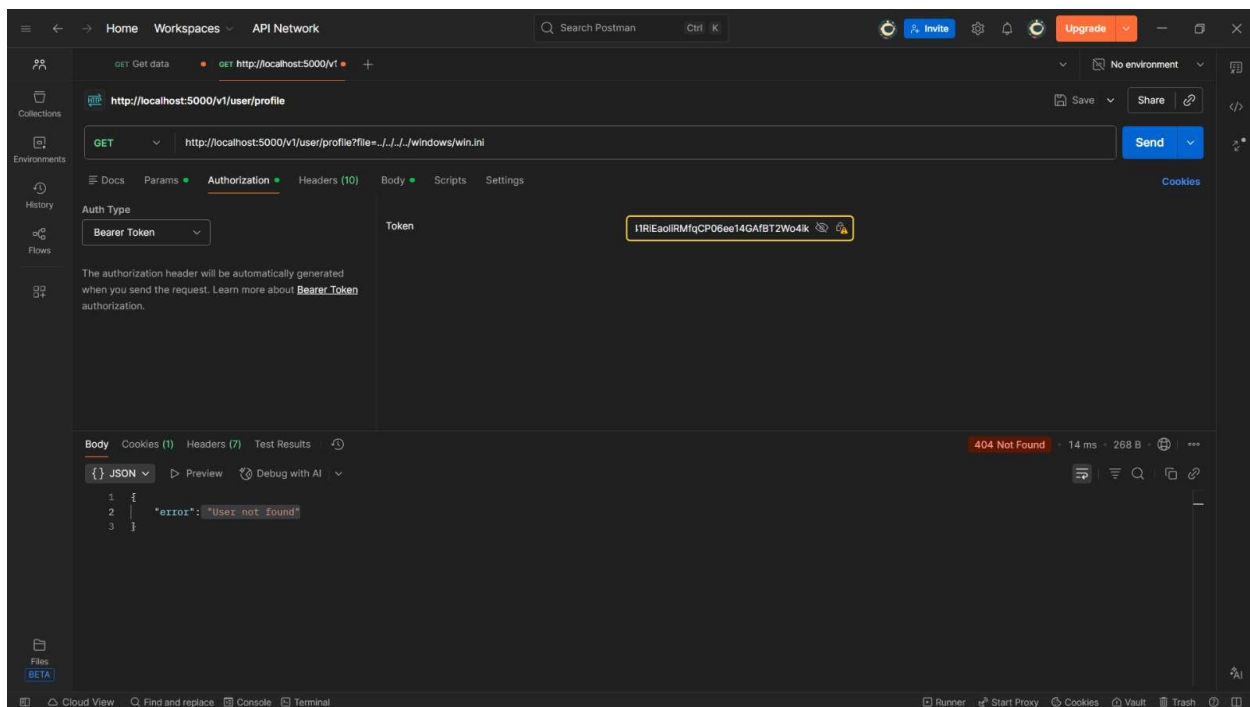


4-JWT(Before):

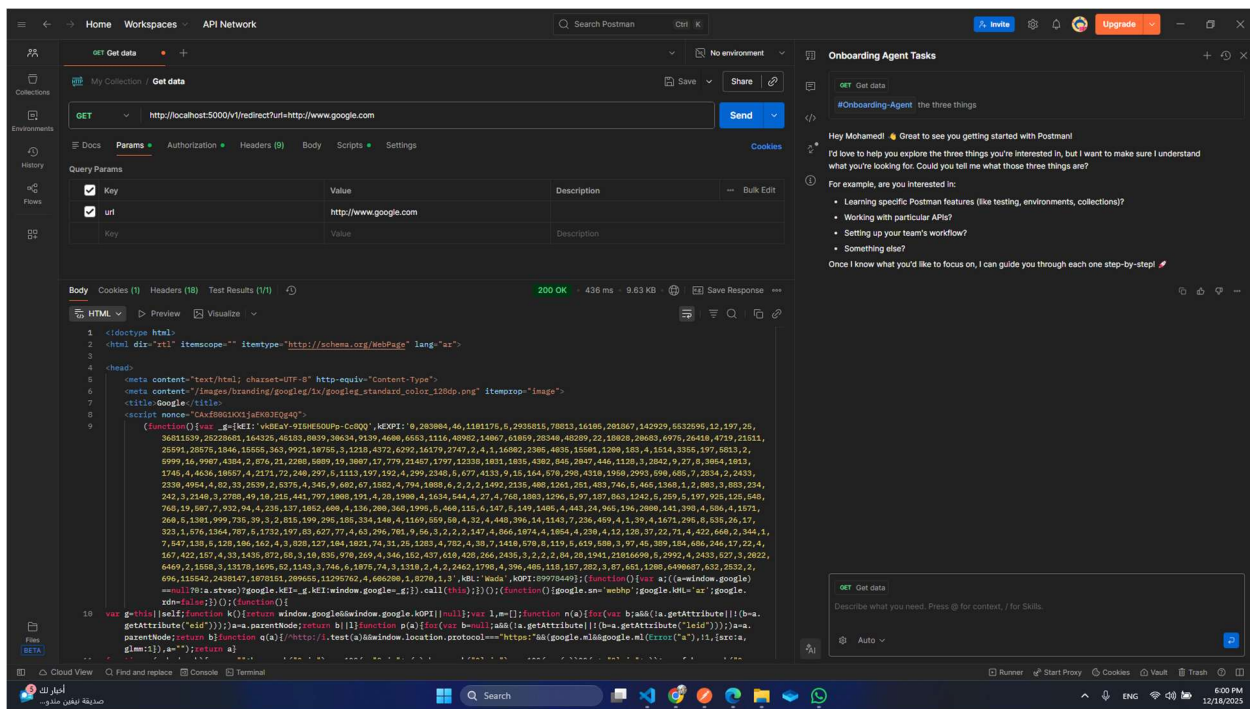




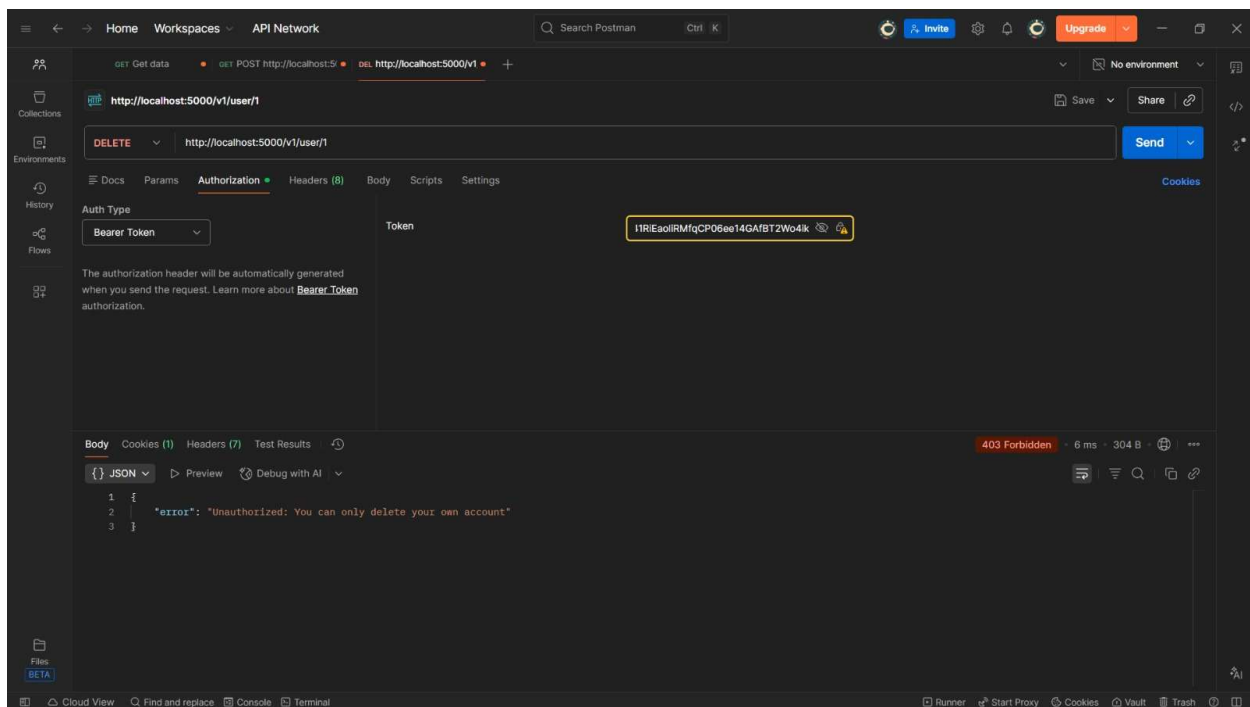
After:



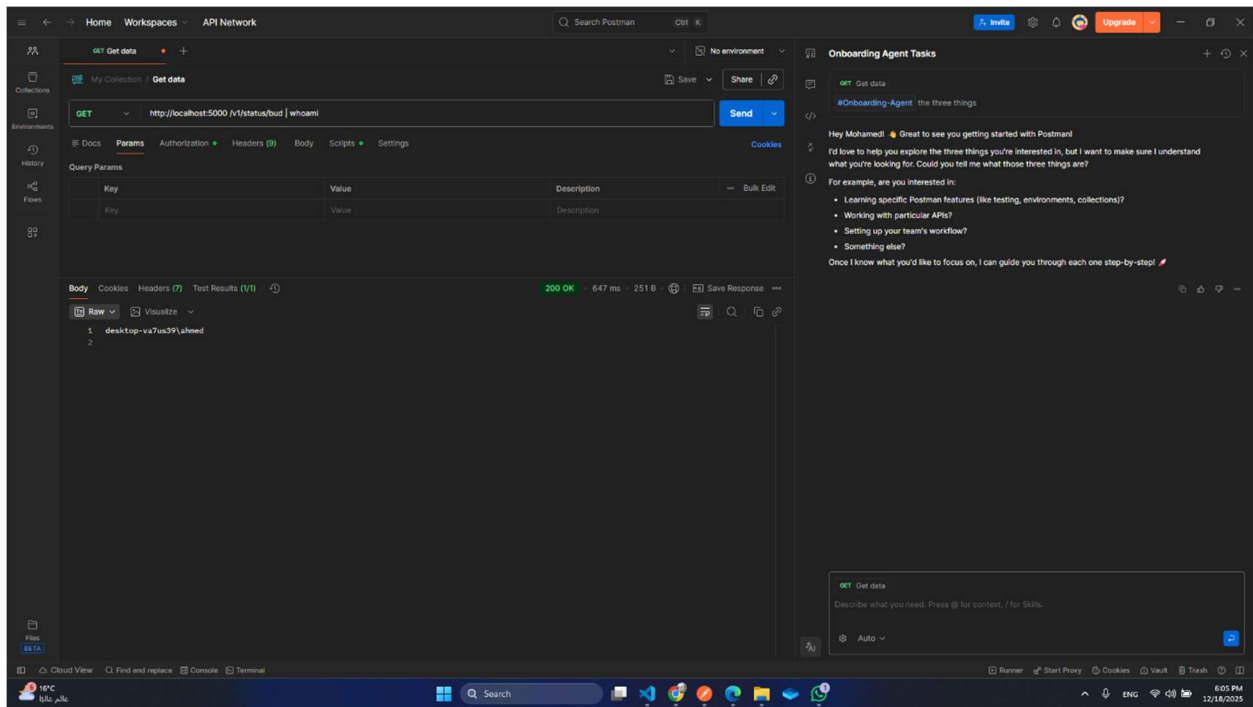
6-Insecure Direct(Before):



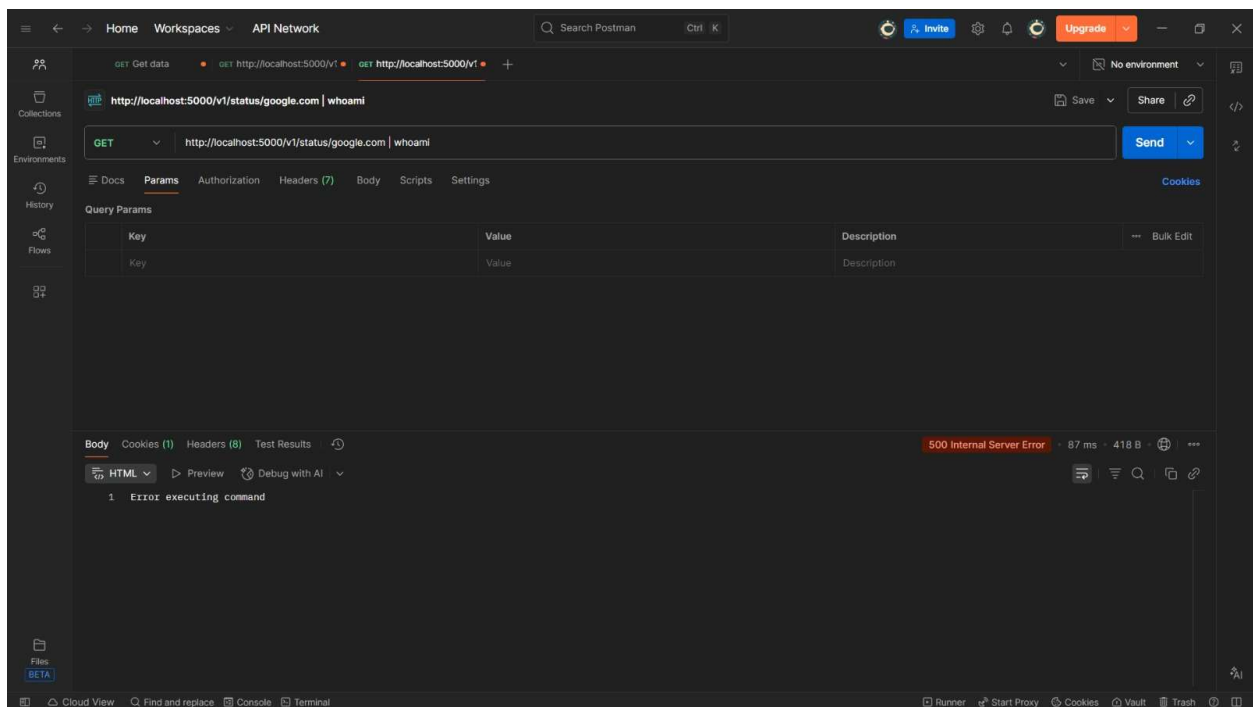
After:



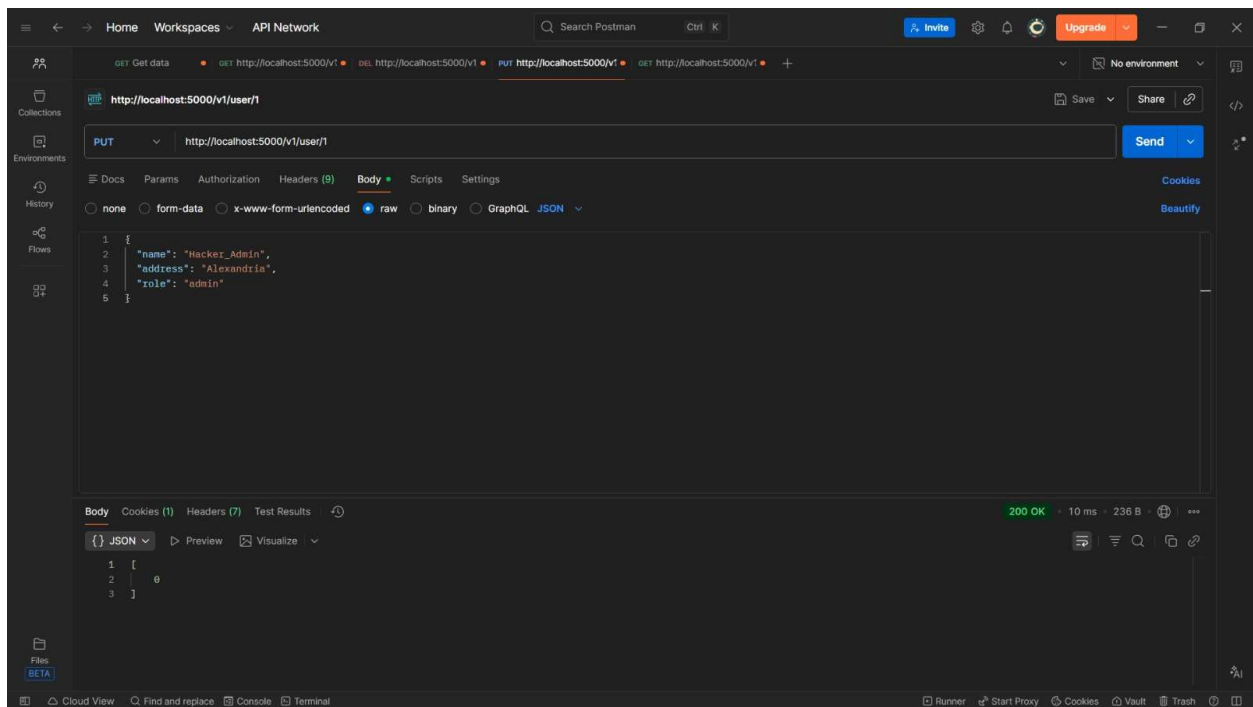
7-RCE(Before):



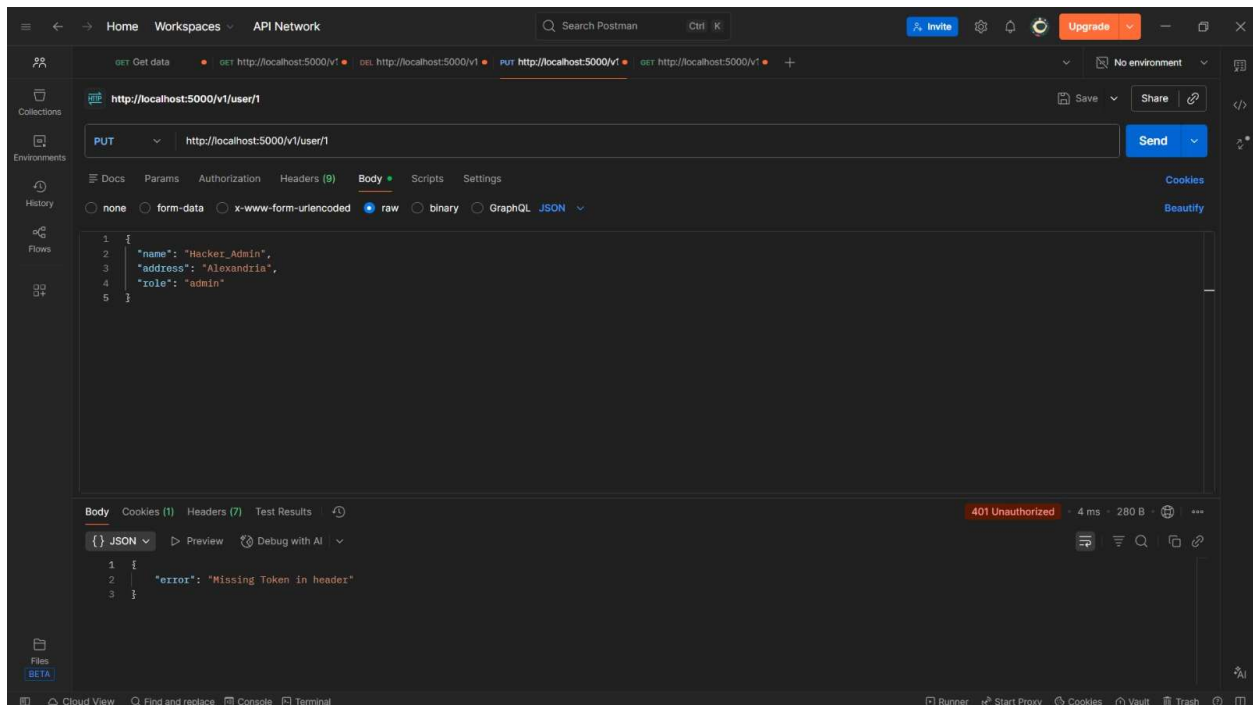
After:



8-Mass Assignment(Before):



After:



1. SQL Injection (SQLi)

- **Before Fix:** The application was using **string concatenation** to build SQL queries. User input from req.params was directly inserted into the query string, allowing an attacker to manipulate the SQL logic (e.g., using ' OR '1'='1) to bypass authentication or leak the entire database.
- **After Fix:** Implemented **Parameterized Queries (Prepared Statements)** using Sequelize replacements. User input is now treated as a literal string (data) rather than executable code. Additionally, an **Allowed Filter Whitelist** was added to ensure only valid column names can be queried.

2. Cross-Site Scripting (XSS)

- **Before Fix:** The server used dangerous functions like renderString() to display user-provided data without sanitization. This allowed attackers to inject malicious <script> tags that would execute in the victim's browser, leading to session hijacking.
- **After Fix:** Switched to the standard res.render() method. This utilizes the template engine's built-in **Contextual Output Encoding (Auto-escaping)**. Any HTML or Script tags provided by the user are now converted into harmless text entities (e.g., < becomes <).

3. Broken Authorization

- **Before Fix:** The application suffered from **Insecure Direct Object References (IDOR)**. It relied solely on the id provided in the URL query to fetch user profiles, allowing any logged-in user to view any other user's private data simply by changing the ID number.
- **After Fix:** Implemented **Access Control Logic** that validates ownership. The system now compares the requested ID with the authenticated user ID stored in the secure session (req.session.userId). If they do not match, access is denied (403 Forbidden).

4. Broken JWT (JSON Web Tokens)

- **Before Fix:** The JWT implementation used a **weak/hardcoded secret key** and did not enforce the signing algorithm. This allowed attackers to brute-force the secret or modify the token payload (e.g., changing role: user to role: admin) without detection.
- **After Fix:** Upgraded to a **Strong Environment Secret** and enforced the **HS256 algorithm**. We also added strict **Expiration (exp)** checks and implemented a token blacklisting mechanism for logged-out users to prevent "replay attacks."

5. Path Traversal

- **Before Fix:** The application accepted file paths directly from user input to read or download files. An attacker could use sequences like `../../etc/passwd` to navigate outside the intended directory and access sensitive system files.
- **After Fix:** Implemented **Path Normalization** and input validation. We now use `path.basename()` to strip directory paths from the input and ensure that the application only looks for files within a specific, restricted "uploads" folder.

- **6- Insecure Direct**

- **Before Fix:** The application fetches user profiles and sensitive resources using a direct identifier (`id`) provided in the URL query string. There is no server-side validation to ensure that the currently logged-in user has the permission to access the requested ID. An attacker can simply change the `id` parameter to view any other user's private information.
- **After Fix:** Implemented an **Object-Level Access Control** check. The application now verifies the requested `id` against the `userId` stored in the secure server-side session. If a user attempts to access an ID that does not belong to them, the system rejects the request with a **403**

Forbidden status, ensuring that users can only access their own data.

7-RCE (Remote Code Execution)

- **Before Fix:** The application uses the `child_process.exec()` function to execute system commands (like ping) by directly concatenating user-supplied input from `req.query.address`. This allows an attacker to use command injection characters (like `;`, `&&`, or `|`) to execute arbitrary OS commands on the server with the same privileges as the Node.js process.
- **After Fix:** The dangerous `exec()` function was replaced with `execFile()`, which accepts arguments as an array and does not spawn a shell, preventing command injection. Additionally, strict **Input Validation** was implemented using a Regex to ensure the input is a valid IP address and nothing else.

8. Mass Assignment

- **Before Fix:** The application was passing the entire `req.body` object directly into the database update function (`User.update(req.body)`). This allowed users to "over-post" and update sensitive fields they shouldn't have access to, such as `is_admin` or `balance`.
- **After Fix:** Implemented **Data Transfer Objects (DTOs)** or "Input Filtering." We now explicitly define which fields are allowed to be updated (e.g., `const { bio, phone } =`

req.body), ensuring that internal sensitive attributes remain protected.

```
root@kali: /home/mohamed/secure/vuln-nodejs-express.js-app
(file) you're telling the browser to fetch in the 'integrity' attribute for
hosted files.
Details: https://sg.run/krXA
225| <script async defer src="https://buttons.github.io/buttons.js"></script>
```

Scan Summary

- ✓ Scan completed successfully.
- Findings: 32 (32 blocking)
- Rules run: 259
- Targets scanned: 90
- Parsed lines: ~99.9%
- Scan skipped:
 - Files matching .semgrepignore patterns: 25
- Scan was limited to files tracked by git
- For a detailed list of skipped files and lines, run semgrep with the --verbose flag

Ran 259 rules on 90 files: 32 findings.

- 💡 Missed out on 1390 pro rules since you aren't logged in!
- ⚡ Supercharge Semgrep OSS when you create a free account at <https://sg.run/rules>.

Too many findings? Try Semgrep Pro for more powerful queries and less noise.
See <https://sg.run/false-positives>.

```
root@kali: /home/mohamed/secure/vuln-nodejs-express.js-app
67| const beers = db.sequelize.query(sql, { type: 'RAW' }).then(beers => {
src/router/routes/system.js
>> javascript.express.security.audit.xss.direct-response-write.direct-response-write
Detected directly writing to a Response object from user-defined input. This bypasses any
HTML escaping and may expose your application to a Cross-Site-scripting (XSS)
vulnerability. Instead, use 'resp.render()' to render safely escaped HTML.
Details: https://sg.run/vzG1
18| res.send(test)
>> 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
37| res.redirect(url);
>> javascript.express.security.audit.express-third-party-object-deserialization.express-third-party-
object-deserialization
The following function call serialize.unserialize accepts user controlled data which can
result in Remote Code Execution (RCE) through Object Deserialization. It is recommended to
use secure data processing alternatives such as JSON.parse() and Buffer.from().
Details: https://sg.run/8W5j
64| var deser = serialize.unserialize(body)
```

```
Session Actions Edit View Help
root@kali: /home/mohamed/secure

>>> reflected-xss-response
Potential reflected XSS via unsanitized user input in response

35| res.send("error")
   |
41| res.send(data)
   |
45| res.send(buffer)

vuln-node.js-express.js-app/src/router/routes/system.js
>>> reflected-xss-response
Potential reflected XSS via unsanitized user input in response

18| res.send(test)

vuln-node.js-express.js-app/src/router/routes/user.js
>>> reflected-xss-response
Potential reflected XSS via unsanitized user input in response

334| res.send(user)
    |
362| res.send(user)

vuln-node.js-express.js-app/src/server.js
```

```
Session Actions Edit View Help
root@kali: /home/mohamed/secure/vuln-node.js-express.js-app

object-deserialization
The following function call serialize.unserialize accepts user controlled data which
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

64| var deser = serialize.unserialize(body)

src/router/routes/user.js
>> javascript.jsonwebtoken.security.jwt-hardcode.hardcoded-jwt-secret
A hard-coded credential was detected. It is not recommended to store credentials in
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 pass
credentials or retrieve credentials from a secure vault or HSM (Hardware Security Module).
Details: https://sg.run/4xN9

18| const user_object = jwt.verify(req.headers.authorization.split(' ')[1], "SuperSecret")

>> 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| current_user_id = jwt.decode(req.headers.authorization.split(' ')[1]).id

>> javascript.jsonwebtoken.security.jwt-hardcode.hardcoded-jwt-secret
A hard-coded credential was detected. It is not recommended to store credentials in
```

It is recommended to use secure data processing alternatives such as JSON.parse() and Buffer.from().
Details: <https://sg.run/8W5j>

```
64| var deser = serialize.unserialize(body)
```

src/router/routes/user.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>

```
18| const user_object = jwt.verify(req.headers.authorization.split('')[1], "SuperSecret")
```

>> 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>

32 Code Findings

populate.sh

>>> generic.secrets.security.detected-jwt-token.detected-jwt-token

JWT token detected

Details: <https://sg.run/05N5>

```
33| -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwicm9sZSI6ImFkbWluIiwiaWF0IjoxNjU5NDI3MjQwLCJleHAiOjE2NTk1MTM2NDh9.QpYD330lcw_AxJR16FKGWu
```

src/middleware/authJwt.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>

```
13|   jwt.decode(token, (err, decoded) => {  
14|     if (err) {  
15|       return res.status(401).send({  
16|         message: "Unauthorized!"
```

```
the intended destination
Details: https://sg.run/weRn

33| fs.readFile(path.join(__dirname, filePath),function(err,data){
>>> javascript.lang.security.audit.path-traversal.path-join-resolve-traversal.path-join-resolve-tra
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

33| fs.readFile(path.join(__dirname, filePath),function(err,data){
>>> javascript.sequelize.security.audit.sequelize-injection-express.express-sequelize-injection
Detected a sequelize statement that is tainted by user-input. This could lead to SQL
injection if the variable is user-controlled and is not properly sanitized. In order to
prevent SQL injection, it is recommended to use parameterized queries or prepared
statements.
Details: https://sg.run/gjoe

67| const beers = db.sequelize.query(sql, { type: 'RAW' }).then(beers => {
src/router/routes/system.js
>>> javascript.express.security.audit.xss.direct-response-write.direct-response-write
Detected directly writing to a Response object from user-defined input. This bypasses any
HTML escaping and may expose your application to a Cross-Site-scripting (XSS)
vulnerability. Instead, use 'res.render()' to render safely escaped HTML.
Details: https://sg.run/vzGl
```

```
vuln-node.js-express.js-app/src/router/routes/admin.js
>>> reflected-xss-response
Potential reflected XSS via unsanitized user input in response

111| res.send(err.toString());

vuln-node.js-express.js-app/src/router/routes/order.js
>>> insecure-file-path
User input used directly in filesystem path (Path Traversal risk)

33| fs.readFile(path.join(__dirname, filePath),function(err,data){
34|     if (err){
35|         res.send("error")
36|     }else{
37|         if(filename.split('.').length == 1)
38|         {
39|             res.type('image/jpeg')
40|             //res.set('Content-Type', 'image/jpg');
41|             res.send(data)
42|             return;
[hid 7 additional lines, adjust with --max-lines-per-finding]

>>> reflected-xss-response
```

```

18| const user_object = jwt.verify(req.headers.authorization.split('
')[1], "SuperSecret")

>> 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| current_user_id = jwt.decode(req.headers.authorization.split(' ')[1]).id

>> 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

253| var token = jwt.sign(payload, jwtTokenSecret, {

>>> javascript.express.security.express-wkhtml-injection.express-wkhtmltoimage-injection
If unverified user data can reach the 'phantom' methods it can result in Server-Side
Request Forgery vulnerabilities
Details: https://sg.run/pxe0

398| const GeneratedToken = otplib.authenticator.generate(seed);

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

```

```

GNU nano 8.6 myrule.yml
- id: no-login-validation
  languages: [javascript]
  severity: WARNING
  message: "Login endpoint missing input validation/sanitization"
  pattern: |
    app.post("/api/login", ($REQ, $RES) => {
      $USERNAME = $REQ.body.username
      $PASSWORD = $REQ.body.password
      // بدون تحقق
    })

# Rule 8: Missing Rate Limiting on Sensitive Routes
- id: missing-rate-limit
  languages: [javascript]
  severity: WARNING
  message: "Sensitive endpoint missing rate limiting middleware"
  pattern: |
    app.post("/api/login", $HANDLER)
    # 1,
    app.post("/api/reset-password", $HANDLER)

```

```
GNU nano 8.6 myrule.yml
rules:
# Rule 1: SQL Injection via String Concatenation
- id: insecure-sql-concat
  languages: [javascript]
  severity: ERROR
  message: "Potential SQL injection via string concatenation"
  pattern: |
    $QUERY = "SELECT" + $INPUT + "FROM" + $TABLE
  # أو نمط أوسع:
  # pattern: |
  #   $DB.query("SELECT ... " + $USER_INPUT + " ... ")

# Rule 2: Reflected XSS via unsanitized response
- id: reflected-xss-response
  languages: [javascript]
  severity: ERROR
  message: "Potential reflected XSS via unsanitized user input in response"
  pattern: |
    res.send($USER_INPUT)

# Rule 3: Missing JWT Verification (algorithm none)
[ Read 77 lines ]
^G Help      ^O Write Out  ^F Where Is   ^K Cut        ^T Execute    ^C Locati
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^/ Go To
```

```
GNU nano 8.6 myrule.yml
# Rule 5: Missing Helmet Security Headers
- id: missing-helmet-middleware
  languages: [javascript]
  severity: WARNING
  message: "Express app missing Helmet middleware for security headers"
  pattern: |
    const app = express()
    // بدون app.use(helmet())

# Rule 6: Insecure File Path from User Input
- id: insecure-file-path
  languages: [javascript]
  severity: ERROR
  message: "User input used directly in filesystem path (Path Traversal risk)"
  pattern: |
    fs.readFile($USER_INPUT, ... )

# Rule 7: No Input Validation on Login Route
- id: no-login-validation
  languages: [javascript]
  severity: WARNING
```

```
GNU nano 8.6 myrule.yml
# Rule 3: Missing JWT Verification (algorithm none)
- id: insecure-jwt-verify
  languages: [javascript]
  severity: ERROR
  message: "Insecure JWT verification - missing algorithm or using 'none'"
  pattern: |
    jwt.verify($TOKEN, $SECRET, { algorithms: ["none"] })
  # أو نمط بدون خوارزمية محددة:
  # pattern: |
  #   jwt.verify($TOKEN, $SECRET)

# Rule 4: Hardcoded Secret in Code
- id: hardcoded-secret
  languages: [javascript]
  severity: ERROR
  message: "Hardcoded secret or password in source code"
  pattern: |
    $SECRET = "password123" or $SECRET = "supersecret"

# Rule 5: Missing Helmet Security Headers
- id: missing-helmet-middleware

^G Help      ^O Write Out  ^F Where Is   ^K Cut       ^T Execute
^X Exit      ^R Read File  ^\ Replace    ^U Paste     ^J Justify
```

