**Результаты статического анализа**

Анализ проводился через терминал, файлы были сохранены на компьютер. В ходе статического анализа с помощью Semgrep в конфигурации auto было проанализировано 995 файлов, в которых были обнаружены 89 уязвимостей.

Результат анализа выглядит следующим образом:

89 Code Findings │

└──────────────────┘

data/static/codefixes/dbSchemaChallenge\_1.ts

❯❯❱ **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

5┆ models.sequelize.query("SELECT \* FROM Products

WHERE ((name LIKE '%"+criteria+"%' OR description

LIKE '%"+criteria+"%') AND deletedAt IS NULL) ORDER

BY name")

❯❯❱ **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

5┆ models.sequelize.query("SELECT \* FROM Products

WHERE ((name LIKE '%"+criteria+"%' OR description

LIKE '%"+criteria+"%') AND deletedAt IS NULL) ORDER

BY name")

data/static/codefixes/dbSchemaChallenge\_3.ts

❯❯❱ **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

11┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

❯❯❱ **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

11┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

data/static/codefixes/restfulXssChallenge\_2.ts

❱ **javascript.audit.detect-replaceall-sanitization.detect-replaceall-**

**sanitization**

Detected a call to `replaceAll()` in an attempt to HTML

escape the string `tableData[i].description`. Manually

sanitizing input through a manually built list can be

circumvented in many situations, and it's better to use a

well known sanitization library such as `sanitize-html` or

`DOMPurify`.

Details: https://sg.run/AzoB

59┆ tableData[i].description =

tableData[i].description.replaceAll('<',

'&lt;').replaceAll('>', '&gt;')

⋮┆----------------------------------------

❱ **javascript.audit.detect-replaceall-sanitization.detect-replaceall-**

**sanitization**

Detected a call to `replaceAll()` in an attempt to HTML

escape the string `tableData[i].description.replaceAll('<',

'&lt;')`. Manually sanitizing input through a manually built

list can be circumvented in many situations, and it's better

to use a well known sanitization library such as `sanitize-

html` or `DOMPurify`.

Details: https://sg.run/AzoB

59┆ tableData[i].description =

tableData[i].description.replaceAll('<',

'&lt;').replaceAll('>', '&gt;')

data/static/codefixes/unionSqlInjectionChallenge\_1.ts

❯❯❱ **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

6┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

❯❯❱ **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

6┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

data/static/codefixes/unionSqlInjectionChallenge\_3.ts

❯❯❱ **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

10┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

❯❯❱ **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

10┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`)

data/static/users.yml

❯❯❱ **generic.secrets.security.detected-generic-secret.detected-generic-**

**secret**

Generic Secret detected

Details: https://sg.run/l2o5

150┆ totpSecret: IFTXE3SPOEYVURT2MRYGI52TKJ4HC3KH

data/staticData.ts

❯❱ **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

7┆ const filePath = path.resolve('./data/static/' +

file + '.yml')

frontend/src/app/app.guard.spec.ts

❯❯❱ **generic.secrets.security.detected-jwt-token.detected-jwt-token**

JWT token detected

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

40┆ localStorage.setItem('token', 'eyJhbGciOiJIUzI1NiI

sInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ

SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ

SMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c')

frontend/src/app/last-login-ip/last-login-ip.component.spec.ts

❯❯❱ **generic.secrets.security.detected-jwt-token.detected-jwt-token**

JWT token detected

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

50┆ localStorage.setItem('token', 'eyJhbGciOiJIUzI1NiI

sInR5cCI6IkpXVCJ9.eyJkYXRhIjp7Imxhc3RMb2dpbklwIjoiM

S4yLjMuNCJ9fQ.RAkmdqwNypuOxv3SDjPO4xMKvd1CddKvDFYD

BfUt3bg')

⋮┆----------------------------------------

56┆ localStorage.setItem('token', 'eyJhbGciOiJIUzI1NiI

sInR5cCI6IkpXVCJ9.eyJkYXRhIjp7fX0.bVBhvll6IaeR3aUd

oOeyR8YZe2S2DfhGAxTGfd9enLw')

frontend/src/app/search-result/search-result.component.ts

❯❱ **typescript.angular.angular-route-bypass-security-trust.angular-route-**

**bypass-security-trust**

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. A popular library used to

prevent XSS is DOMPurify. You can also use libraries and

frameworks such as Angular, Vue, and React, which offer

secure defaults when rendering input.

Details: https://sg.run/JpBW

151┆ this.searchValue =

this.sanitizer.bypassSecurityTrustHtml(queryParam

) // vuln-code-snippet vuln-line localXssChallenge

xssBonusChallenge

frontend/src/hacking-instructor/helpers/helpers.ts

❯❱ **javascript.lang.security.audit.prototype-pollution.prototype-pollution-**

**loop.prototype-pollution-loop**

Possibility of prototype polluting function detected. By

adding or modifying attributes of an object prototype, it is

possible to create attributes that exist on every object, or

replace critical attributes with malicious ones. This can be

problematic if the software depends on existence or non-

existence of certain attributes, or uses pre-defined

attributes of object prototype (such as hasOwnProperty,

toString or valueOf). Possible mitigations might be:

freezing the object prototype, using an object without

prototypes (via Object.create(null) ), blocking

modifications of attributes that resolve to object

prototype, using Map instead of object.

Details: https://sg.run/w1DB

38┆ replacementValue = replacementValue[property]

frontend/src/hacking-instructor/index.ts

❯❯❱ **javascript.browser.security.insecure-document-method.insecure-document-**

**method**

User controlled data in methods like `innerHTML`,

`outerHTML` or `document.write` is an anti-pattern that can

lead to XSS vulnerabilities

Details: https://sg.run/LwA9

111┆ textBox.innerHTML = snarkdown(hint.text)

frontend/src/index.html

❯❱ **html.security.audit.missing-integrity.missing-integrity**

This tag is missing an 'integrity' subresource integrity

attribute. The 'integrity' attribute allows for the browser

to verify that externally hosted files (for example from a

CDN) are delivered without unexpected manipulation. Without

this attribute, if an attacker can modify the externally

hosted resource, this could lead to XSS and other types of

attacks. To prevent this, include the base64-encoded

cryptographic hash of the resource (file) you’re telling the

browser to fetch in the 'integrity' attribute for all

externally hosted files.

Details: https://sg.run/krXA

14┆ <link rel="stylesheet" type="text/css" href="//cdn

js.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/co

okieconsent.min.css" />

⋮┆----------------------------------------

15┆ <script src="//cdnjs.cloudflare.com/ajax/libs/cook

ieconsent2/3.1.0/cookieconsent.min.js"></script>

⋮┆----------------------------------------

16┆ <script src="//cdnjs.cloudflare.com/ajax/libs/jque

ry/2.2.4/jquery.min.js"></script>

lib/codingChallenges.ts

❯❱ **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

24┆ files.map(file => path.resolve(currPath, file))

⋮┆----------------------------------------

24┆ files.map(file => path.resolve(currPath, file))

❯❱ **javascript.lang.security.audit.detect-non-literal-regexp.detect-non-**

**literal-regexp**

RegExp() called with a `challengeKey` function argument,

this might allow an attacker to cause a Regular Expression

Denial-of-Service (ReDoS) within your application as RegExP

blocks the main thread. For this reason, it is recommended

to use hardcoded regexes instead. If your regex is run on

user-controlled input, consider performing input validation

or use a regex checking/sanitization library such as

https://www.npmjs.com/package/recheck to verify that the

regex does not appear vulnerable to ReDoS.

Details: https://sg.run/gr65

76┆ if (new RegExp(`vuln-code-snippet vuln-

line.\*${challengeKey}`).exec(lines[i]) != null) {

⋮┆----------------------------------------

78┆ } else if (new RegExp(`vuln-code-snippet neutral-

line.\*${challengeKey}`).exec(lines[i]) != null) {

lib/insecurity.ts

❯❱ **javascript.lang.hardcoded.strings.detected-private-key.detected-**

**private-key**

A secret is hard-coded in the application. Secrets stored in

source code, such as credentials, identifiers, and other

types of sensitive data, can be leaked and used by internal

or external malicious actors. Use environment variables to

securely provide credentials and other secrets or retrieve

them from a secure vault or Hardware Security Module (HSM).

Details: https://sg.run/2dRY

23┆ const privateKey = '-----BEGIN RSA PRIVATE KEY----

-\r\nMIICXAIBAAKBgQDNwqLEe9wgTXCbC7+RPdDbBbeqjdbs4k

OPOIGzqLpXvJXlxxW8iMz0EaM4BKUqYsIa+ndv3NAn2RxCd5ubV

dJJcX43zO6Ko0TFEZx/65gY3BE0O6syCEmUP4qbSd6exou/F+WT

ISzbQ5FBVPVmhnYhG/kpwt/cIxK5iUn5hm+4tQIDAQABAoGBAI+

8xiPoOrA+KMnG/T4jJsG6TsHQcDHvJi7o1IKC/hnIXha0atTX5A

UkRRce95qSfvKFweXdJXSQ0JMGJyfuXgU6dI0TcseFRfewXAa/s

sxAC+iUVR6KUMh1PE2wXLitfeI6JLvVtrBYswm2I7CtY0q8n5AG

imHWVXJPLfGV7m0BAkEA+fqFt2LXbLtyg6wZyxMA/cnmt5Nt3U2

dAu77MzFJvibANUNHE4HPLZxjGNXN+a6m0K6TD4kDdh5HfUYLWW

RBYQJBANK3carmulBwqzcDBjsJ0YrIONBpCAsXxk8idXb8jL9aN

Ig15Wumm2enqqObahDHB5jnGOLmbasizvSVqypfM9UCQCQl8xIq

y+YgURXzXCN+kwUgHinrutZms87Jyi+D8Br8NY0+Nlf+zHvXAom

D2W5CsEK7C+8SLBr3k/TsnRWHJuECQHFE9RA2OP8WoaLPuGCyFX

axzICThSRZYluVnWkZtxsBhW2W8z1b8PvWUE7kMy7TnkzeJS2LS

naNHoyxi7IaPQUCQCwWU4U+v4lD7uYBw00Ga/xt+7+UqFPlPVdz

1yyr4q24Zxaw0LgmuEvgU5dycq8N7JxjTubX0MIRR+G9fmDBBl8

=\r\n-----END RSA PRIVATE KEY-----'

❯❱ **javascript.lang.security.audit.hardcoded-hmac-key.hardcoded-hmac-key**

Detected a hardcoded hmac key. Avoid hardcoding secrets and

consider using an alternate option such as reading the

secret from a config file or using an environment variable.

Details: https://sg.run/K9bn

44┆ export const hmac = (data: string) =>

crypto.createHmac('sha256', 'pa4qacea4VK9t9nGv7yZt

wmj').update(data).digest('hex')

❯❱ **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

56┆ export const authorize = (user = {}) =>

jwt.sign(user, privateKey, { expiresIn: '6h',

algorithm: 'RS256' })

❯❱ **javascript.lang.security.audit.hardcoded-hmac-key.hardcoded-hmac-key**

Detected a hardcoded hmac key. Avoid hardcoding secrets and

consider using an alternate option such as reading the

secret from a config file or using an environment variable.

Details: https://sg.run/K9bn

152┆ const hmac = crypto.createHmac('sha256',

privateKey)

❯❱ **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

195┆ res.cookie('token', token)

lib/startup/restoreOverwrittenFilesWithOriginals.ts

❯❱ **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

28┆ await copyFile(filename, path.resolve('i18n/',

filename.substring(filename.lastIndexOf('/') +

1)))

lib/startup/validatePreconditions.ts

❯❱ **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

120┆ return await access(path.resolve(pathRelativeToPro

jectRoot)).then(() => {

lib/utils.ts

❯❱ **javascript.jssha.jssha-sha1.jssha-sha1**

The SHA1 hashing algorithm is considered to be weak. If this

is used in any sensitive operation such as password hashing,

or is used to ensure data integrity (collision sensitive)

then you should use a stronger hashing algorithm. For

passwords, consider using `Argon2id`, `scrypt`, or `bcrypt`.

For data integrity, consider using `SHA-256`.

Details: https://sg.run/ERyN

90┆ const shaObj = new jsSHA('SHA-1', 'TEXT') //

eslint-disable-line new-cap

models/index.ts

❯❱ **javascript.sequelize.node-sequelize-hardcoded-secret-argument.node-**

**sequelize-hardcoded-secret-argument**

A secret is hard-coded in the application. Secrets stored in

source code, such as credentials, identifiers, and other

types of sensitive data, can be leaked and used by internal

or external malicious actors. Use environment variables to

securely provide credentials and other secrets or retrieve

them from a secure vault or Hardware Security Module (HSM).

Details: https://sg.run/E7ZB

29┆ const sequelize = new Sequelize('database',

'username', 'password', {

routes/b2bOrder.ts

❯❱ **javascript.express.security.audit.express-detect-notevil-usage.express-**

**detect-notevil-usage**

Detected usage of the `notevil` package, which is

unmaintained and has vulnerabilities. Using any sort of

`eval()` functionality can be very dangerous, but if you

must, the `eval` package is an up to date alternative. Be

sure that only trusted input reaches an `eval()` function.

Details: https://sg.run/W70E

22┆ vm.runInContext('safeEval(orderLinesData)',

sandbox, { timeout: 2000 })

routes/captcha.ts

❯❱ **javascript.browser.security.eval-detected.eval-detected**

Detected the use of eval(). eval() can be dangerous if used

to evaluate dynamic content. If this content can be input

from outside the program, this may be a code injection

vulnerability. Ensure evaluated content is not definable by

external sources.

Details: https://sg.run/7ope

23┆ const answer = eval(expression).toString() //

eslint-disable-line no-eval

routes/chatbot.ts

❯❱ **javascript.express.security.injection.raw-html-format.raw-html-format**

User data flows into the host portion of this manually-

constructed HTML. This can introduce a Cross-Site-Scripting

(XSS) vulnerability if this comes from user-provided input.

Consider using a sanitization library such as DOMPurify to

sanitize the HTML within.

Details: https://sg.run/5DO3

198┆ body: bot.training.state ?

bot.greet(`${user.id}`) :

`${config.get<string>('application.chatBot.name')}

isn't ready at the moment, please wait while I set

things up`

routes/dataErasure.ts

❯❱ **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

69┆ const filePath: string =

path.resolve(req.body.layout).toLowerCase()

❯❱ **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

69┆ const filePath: string =

path.resolve(req.body.layout).toLowerCase()

routes/dataExport.ts

❯❯❱ **javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli**

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

61┆ db.ordersCollection.find({ email: updatedEmail

}).then((orders: Array<{

routes/fileServer.ts

❯❱ **javascript.express.security.audit.express-res-sendfile.express-res-**

**sendfile**

The application processes user-input, this is passed to

res.sendFile which can allow an attacker to arbitrarily read

files on the system through path traversal. It is

recommended to perform input validation in addition to

canonicalizing the path. This allows you to validate the

path against the intended directory it should be accessing.

Details: https://sg.run/7DJk

33┆ res.sendFile(path.resolve('ftp/', file))

❯❱ **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

33┆ res.sendFile(path.resolve('ftp/', file))

routes/fileUpload.ts

❯❱ **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

29┆ const tempFile = path.join(os.tmpdir(), filename)

⋮┆----------------------------------------

39┆ const absolutePath =

path.resolve('uploads/complaints/' + fileName)

❯❱ **javascript.express.security.audit.express-libxml-vm-noent.express-**

**libxml-vm-noent**

Detected use of parseXml() function with the `noent` field

set to `true`. This can lead to an XML External Entities

(XXE) attack if untrusted data is passed into it.

Details: https://sg.run/n8Ag

80┆ const xmlDoc =

vm.runInContext('libxml.parseXml(data, { noblanks:

true, noent: true, nocdata: true })', sandbox, {

timeout: 2000 })

routes/keyServer.ts

❯❱ **javascript.express.security.audit.express-res-sendfile.express-res-**

**sendfile**

The application processes user-input, this is passed to

res.sendFile which can allow an attacker to arbitrarily read

files on the system through path traversal. It is

recommended to perform input validation in addition to

canonicalizing the path. This allows you to validate the

path against the intended directory it should be accessing.

Details: https://sg.run/7DJk

14┆ res.sendFile(path.resolve('encryptionkeys/',

file))

❯❱ **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

14┆ res.sendFile(path.resolve('encryptionkeys/',

file))

❯❱ **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

14┆ res.sendFile(path.resolve('encryptionkeys/',

file))

routes/likeProductReviews.ts

❯❯❱ **javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli**

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

18┆ db.reviewsCollection.findOne({ \_id: id

}).then((review: Review) => {

⋮┆----------------------------------------

25┆ { \_id: id },

⋮┆----------------------------------------

31┆ db.reviewsCollection.findOne({ \_id: id

}).then((review: Review) => {

⋮┆----------------------------------------

42┆ { \_id: id },

routes/logfileServer.ts

❯❱ **javascript.express.security.audit.express-res-sendfile.express-res-**

**sendfile**

The application processes user-input, this is passed to

res.sendFile which can allow an attacker to arbitrarily read

files on the system through path traversal. It is

recommended to perform input validation in addition to

canonicalizing the path. This allows you to validate the

path against the intended directory it should be accessing.

Details: https://sg.run/7DJk

14┆ res.sendFile(path.resolve('logs/', file))

❯❱ **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

14┆ res.sendFile(path.resolve('logs/', file))

❯❱ **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

14┆ res.sendFile(path.resolve('logs/', file))

routes/login.ts

❯❯❱ **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

36┆ models.sequelize.query(`SELECT \* FROM Users WHERE

email = '${req.body.email || ''}' AND password =

'${security.hash(req.body.password || '')}' AND

deletedAt IS NULL`, { model: UserModel, plain:

true }) // vuln-code-snippet vuln-line

loginAdminChallenge loginBenderChallenge

loginJimChallenge

❯❯❱ **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

36┆ models.sequelize.query(`SELECT \* FROM Users WHERE

email = '${req.body.email || ''}' AND password =

'${security.hash(req.body.password || '')}' AND

deletedAt IS NULL`, { model: UserModel, plain: true

}) // vuln-code-snippet vuln-line

loginAdminChallenge loginBenderChallenge

loginJimChallenge

routes/order.ts

❯❱ **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

45┆ const fileWriter =

doc.pipe(fs.createWriteStream(path.join('ftp/',

pdfFile)))

routes/orderHistory.ts

❯❯❱ javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

17┆ const order = await ordersCollection.find({ email:

updatedEmail })

⋮┆----------------------------------------

36┆ await ordersCollection.update({ \_id: req.params.id

}, { $set: { delivered: deliveryStatus, eta } })

routes/profileImageFileUpload.ts

❯❱ **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

28┆ fs.open(`frontend/dist/frontend/assets/public/imag

es/uploads/${loggedInUser.data.id}.${uploadedFileTy

pe.ext}`, 'w', function (err, fd) {

routes/profileImageUrlUpload.ts

❯❱ **javascript.express.request.ssrf-deepsemgrep.ssrf-deepsemgrep**

Untrusted input might be used to build an HTTP request,

which can lead to a Server-side request forgery (SSRF)

vulnerability. SSRF allows an attacker to send crafted

requests from the server side to other internal or external

systems. SSRF can lead to unauthorized access to sensitive

data and, in some cases, allow the attacker to control

applications or systems that trust the vulnerable service.

To prevent this vulnerability, avoid allowing user input to

craft the base request. Instead, treat it as part of the

path or query parameter and encode it appropriately. When

user input is necessary to prepare the HTTP request, perform

strict input validation. Additionally, whenever possible,

use allowlists to only interact with expected, trusted

domains.

Details: https://sg.run/Wxn4

23┆ .get(url)

❯❱ **javascript.express.security.audit.express-ssrf.express-ssrf**

The following request request.get() was found to be crafted

from user-input `req` which can lead to Server-Side Request

Forgery (SSRF) vulnerabilities. It is recommended where

possible to not allow user-input to craft the base request,

but to be treated as part of the path or query parameter.

When user-input is necessary to craft the request, it is

recommeneded to follow OWASP best practices to prevent

abuse.

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

23┆ .get(url)

❯❱ **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

31┆ imageRequest.pipe(fs.createWriteStream(`frontend/d

ist/frontend/assets/public/images/uploads/${loggedI

nUser.data.id}.${ext}`))

routes/quarantineServer.ts

❯❱ **javascript.express.security.audit.express-res-sendfile.express-res-**

**sendfile**

The application processes user-input, this is passed to

res.sendFile which can allow an attacker to arbitrarily read

files on the system through path traversal. It is

recommended to perform input validation in addition to

canonicalizing the path. This allows you to validate the

path against the intended directory it should be accessing.

Details: https://sg.run/7DJk

14┆ res.sendFile(path.resolve('ftp/quarantine/',

file))

❯❱ **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

14┆ res.sendFile(path.resolve('ftp/quarantine/',

file))

❯❱ **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

14┆ res.sendFile(path.resolve('ftp/quarantine/',

file))

routes/redirect.ts

❯❱ **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

19┆ res.redirect(toUrl)

❯❱ **javascript.express.security.audit.express-open-redirect.express-open-**

**redirect**

The application redirects to a URL specified by user-

supplied input `query` 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

19┆ res.redirect(toUrl)

routes/search.ts

❯❯❱ **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

23┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`) // vuln-code-snippet vuln-line

unionSqlInjectionChallenge dbSchemaChallenge

❯❯❱ **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

23┆ models.sequelize.query(`SELECT \* FROM Products

WHERE ((name LIKE '%${criteria}%' OR description

LIKE '%${criteria}%') AND deletedAt IS NULL) ORDER

BY name`) // vuln-code-snippet vuln-line

unionSqlInjectionChallenge dbSchemaChallenge

routes/showProductReviews.ts

❯❯❱ **javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli**

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

34┆ db.reviewsCollection.find({ $where: 'this.product

== ' + id }).then((reviews: Review[]) => {

routes/trackOrder.ts

❯❯❱ **javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli**

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

17┆ db.ordersCollection.find({ $where: `this.orderId

=== '${id}'` }).then((order: any) => {

routes/updateProductReviews.ts

❯❯❱ **javascript.express.mongodb.express-mongo-nosqli.express-mongo-nosqli**

Detected a `../data/mongodb` statement that comes from a

`req` argument. This could lead to NoSQL injection if the

variable is user-controlled and is not properly sanitized.

Be sure to properly sanitize the data if you absolutely must

pass request data into a mongo query.

Details: https://sg.run/Qqxq

18┆ { \_id: req.body.id }, // vuln-code-snippet vuln-

line noSqlReviewsChallenge forgedReviewChallenge

routes/userProfile.ts

❯❱ **javascript.browser.security.eval-detected.eval-detected**

Detected the use of eval(). eval() can be dangerous if used

to evaluate dynamic content. If this content can be input

from outside the program, this may be a code injection

vulnerability. Ensure evaluated content is not definable by

external sources.

Details: https://sg.run/7ope

36┆ username = eval(code) // eslint-disable-line no-

eval

❯❱ **javascript.express.security.express-insecure-template-usage.express-**

**insecure-template-usage**

User data from `req` is being compiled into the template,

which can lead to a Server Side Template Injection (SSTI)

vulnerability.

Details: https://sg.run/b49v

56┆ const fn = pug.compile(template)

routes/videoHandler.ts

❯❱ **javascript.lang.security.audit.unknown-value-with-script-tag.unknown-**

**value-with-script-tag**

Cannot determine what 'subs' is and it is used with a

'<script>' tag. This could be susceptible to cross-site

scripting (XSS). Ensure 'subs' is not externally controlled,

or sanitize this data.

Details: https://sg.run/1Zy1

57┆

challengeUtils.solveIf(challenges.videoXssChallenge

, () => { return utils.contains(subs,

'</script><script>alert(`xss`)</script>') })

⋮┆----------------------------------------

69┆ compiledTemplate =

compiledTemplate.replace('<script

id="subtitle"></script>', '<script id="subtitle"

type="text/vtt" data-label="English" data-

lang="en">' + subs + '</script>')

routes/vulnCodeFixes.ts

❯❱ **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

79┆ if (fs.existsSync('./data/static/codefixes/' + key

+ '.info.yml')) {

⋮┆----------------------------------------

80┆ const codingChallengeInfos =

yaml.load(fs.readFileSync('./data/static/codefixes

/' + key + '.info.yml', 'utf8'))

routes/vulnCodeSnippet.ts

❯❱ **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

93┆ if (fs.existsSync('./data/static/codefixes/' + key

+ '.info.yml')) {

⋮┆----------------------------------------

94┆ const codingChallengeInfos =

yaml.load(fs.readFileSync('./data/static/codefixes

/' + key + '.info.yml', 'utf8'))

server.ts

❱ **javascript.express.security.audit.express-check-csurf-middleware-**

**usage.express-check-csurf-middleware-usage**

A CSRF middleware was not detected in your express

application. Ensure you are either using one such as `csurf`

or `csrf` (see rule references) and/or you are properly

doing CSRF validation in your routes with a token or

cookies.

Details: https://sg.run/BxzR

105┆ const app = express()

❱ **javascript.lang.security.audit.unsafe-formatstring.unsafe-formatstring**

Detected string concatenation with a non-literal variable in

a util.format / console.log function. If an attacker injects

a format specifier in the string, it will forge the log

message. Try to use constant values for the format string.

Details: https://sg.run/7Y5R

148┆ console.error('Error in timed startup function: '

+ name, err)

❯❱ **javascript.express.security.audit.express-check-directory-**

**listing.express-check-directory-listing**

Directory listing/indexing is enabled, which may lead to

disclosure of sensitive directories and files. It is

recommended to disable directory listing unless it is a

public resource. If you need directory listing, ensure that

sensitive files are inaccessible when querying the resource.

Details: https://sg.run/DX2G

260┆ app.use('/ftp', serveIndexMiddleware,

serveIndex('ftp', { icons: true })) // vuln-code-

snippet vuln-line directoryListingChallenge

⋮┆----------------------------------------

264┆ app.use('/.well-known', serveIndexMiddleware,

serveIndex('.well-known', { icons: true, view:

'details' }))

⋮┆----------------------------------------

268┆ app.use('/encryptionkeys', serveIndexMiddleware,

serveIndex('encryptionkeys', { icons: true, view:

'details' }))

⋮┆----------------------------------------

272┆ app.use('/support/logs', serveIndexMiddleware,

serveIndex('logs', { icons: true, view: 'details'

})) // vuln-code-snippet vuln-line

accessLogDisclosureChallenge

views/promotionVideo.pug

❯❱ **javascript.express.security.audit.xss.pug.explicit-unescape.template-**

**explicit-unescape**

Detected an explicit unescape in a Pug template, using

either '!=' or '!{...}'. If external data can reach these

locations, your application is exposed to a cross-site

scripting (XSS) vulnerability. If you must do this, ensure

no external data can reach this location.

Details: https://sg.run/3xbe

79┆ if (splitted.length != 2) {

┌──────────────┐

│ Scan Summary │

└──────────────┘

Some files were skipped or only partially analyzed.

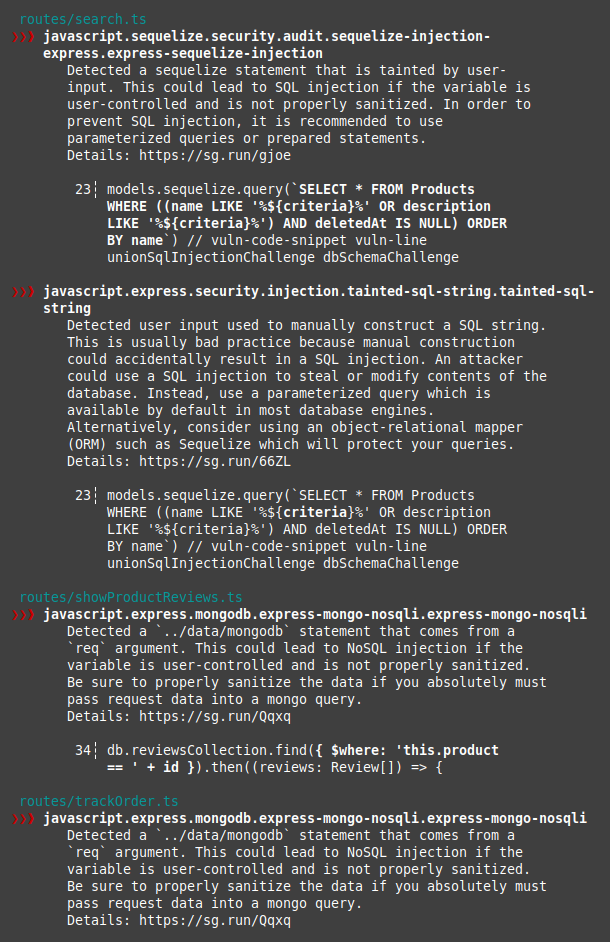
Partially scanned: 30 files only partially analyzed due to parsing or internal Semgrep errors

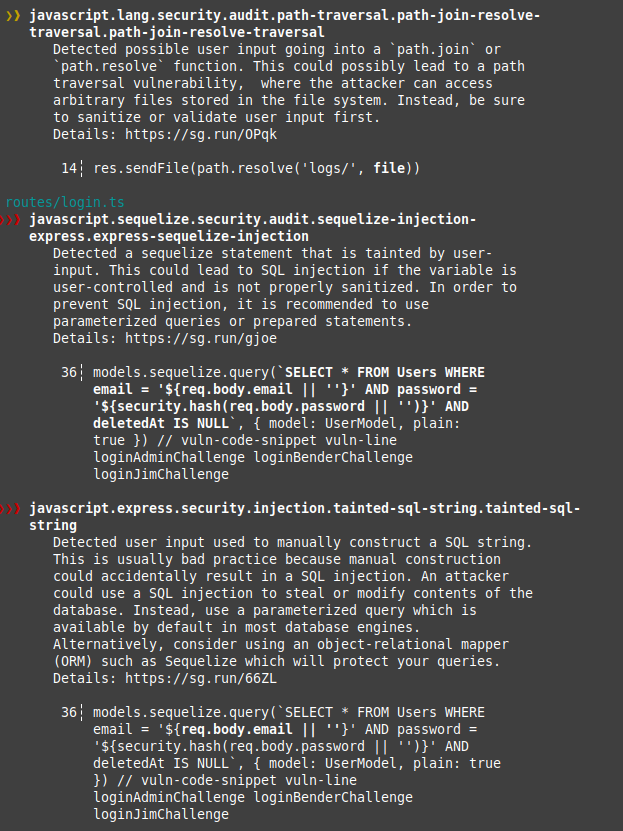
Scan skipped: 8 files larger than 1.0 MB, 138 files matching .semgrepignore patterns

For a full list of skipped files, run semgrep with the --verbose flag.

Ran 373 rules on 995 files: 89 findings.

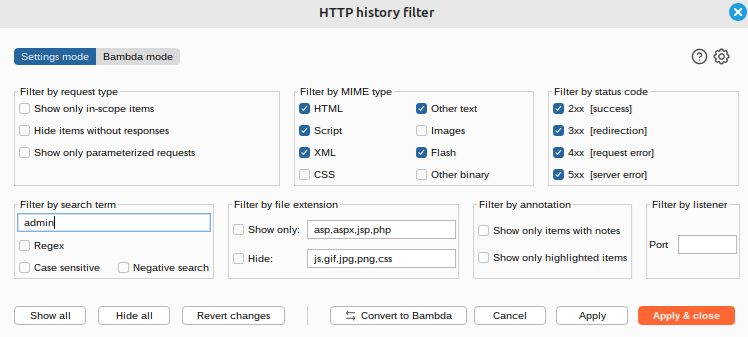
В качестве скриншотов приведу пару случайных участков вывода, на которых встречаются уязвимости типа sql injection:

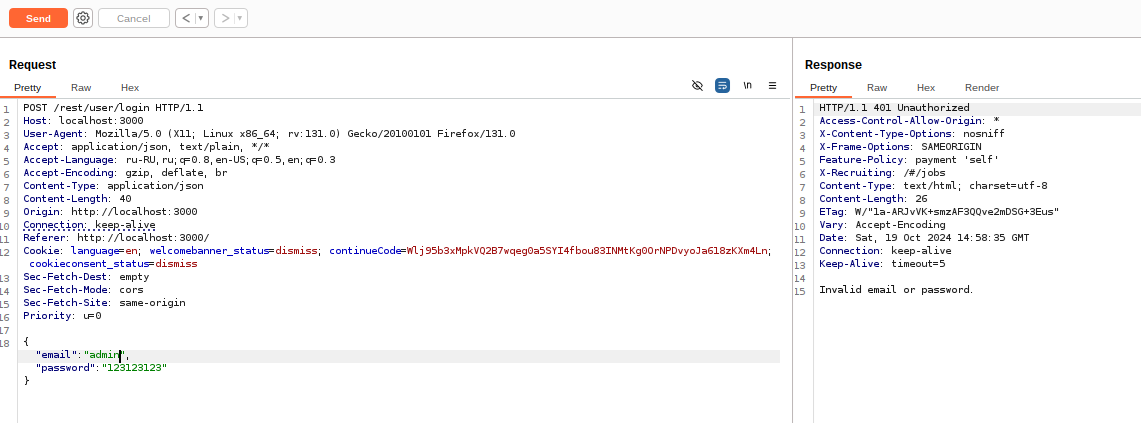




**Эксплуатация уязвимостей**

В поле авторизации есть возможность использовать спецсимволы. Я перехватил пакет авторизации, нашёл его через фильтр и отправил его в модуль repeater для изучения.

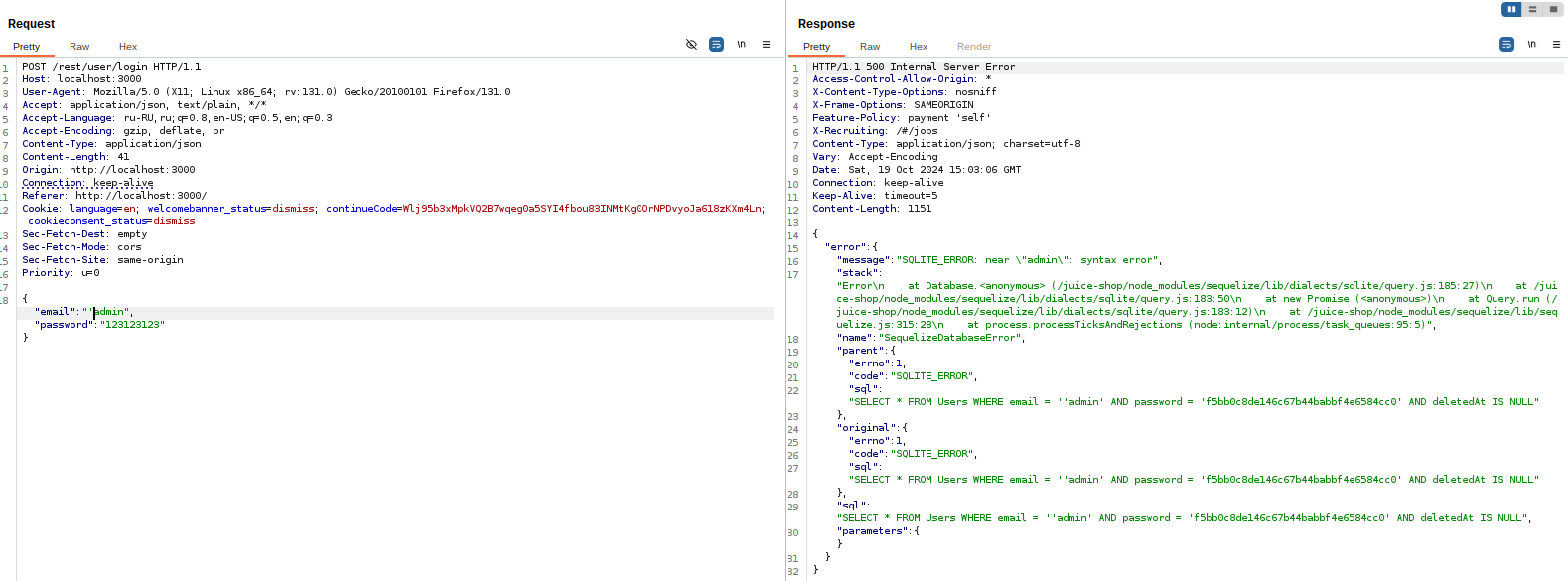




Если добавить в логин ‘ , то можно увидеть сам код проверки введённых данных, он выглядит следующим образом:

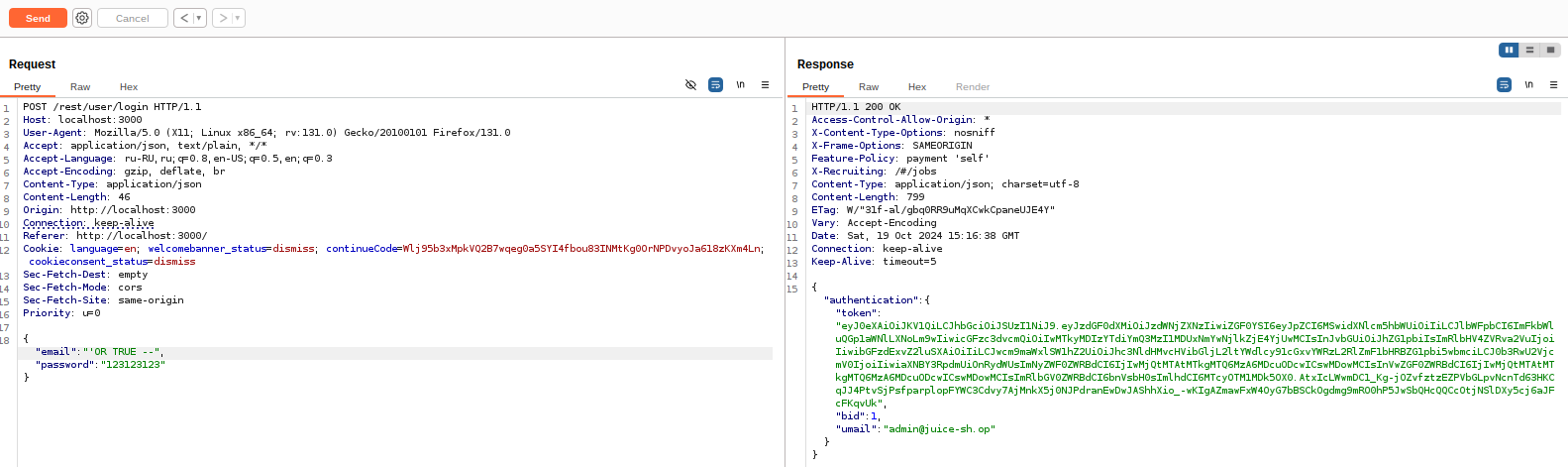
SELECT \* FROM Users WHERE email = ''admin' AND password = 'f5bb0c8de146c67b44babbf4e6584cc0' AND deletedAt IS NULL

Электронный адрес сравнивается с явно хранящимся значением в базе данных, пароль предварительно преобразуется и сравнивается с неявным значением, и есть некое булево поле, предположительно отвечающее за удалена или нет данная комбинация адреса и пароля.



Эту проверку можно обмануть, добавив в sql запрос вместо электронного адреса код ‘OR TRUE - -. Тогда запрос получится следующего вида:

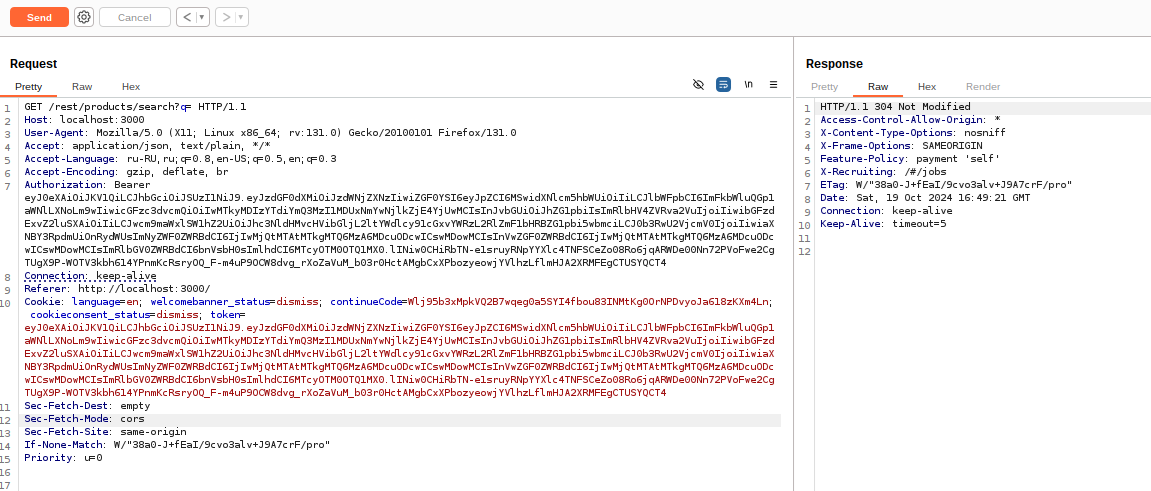
SELECT \* FROM Users WHERE email = ' ' OR TRUE. Первый апостроф и добавленный мной образовали значение пустой строки, OR TRUE – всегда истинное выражение, которое даст запросу результат истины, а - - закомментирует остальную часть с проверкой пароля и актуальности данных.



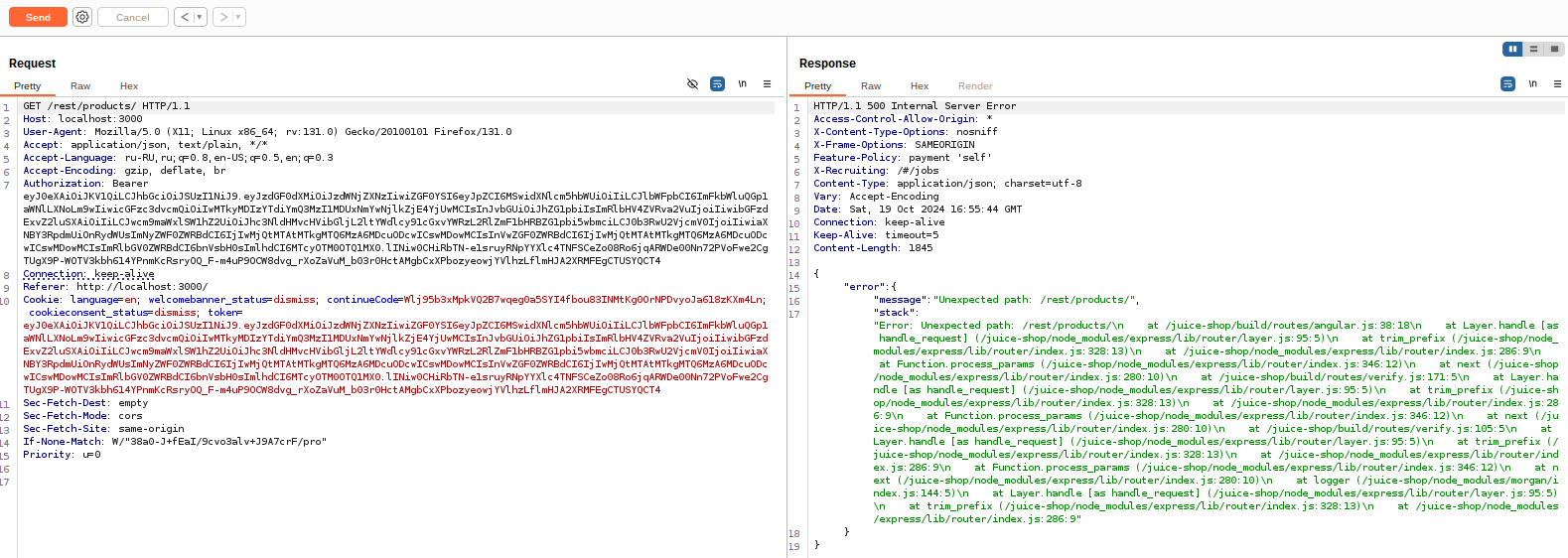
Я вижу, что система выдала мне токен авторизации, и логин (электронный адрес) пользователя admin@juice-sh.op.

Уязвимость типа SQL Injection. Её можно закрыть, используя параметризированные запросы с предварительной проверкой пользовательского ввода.

Запрос поиска товара выглядит следующим образом:

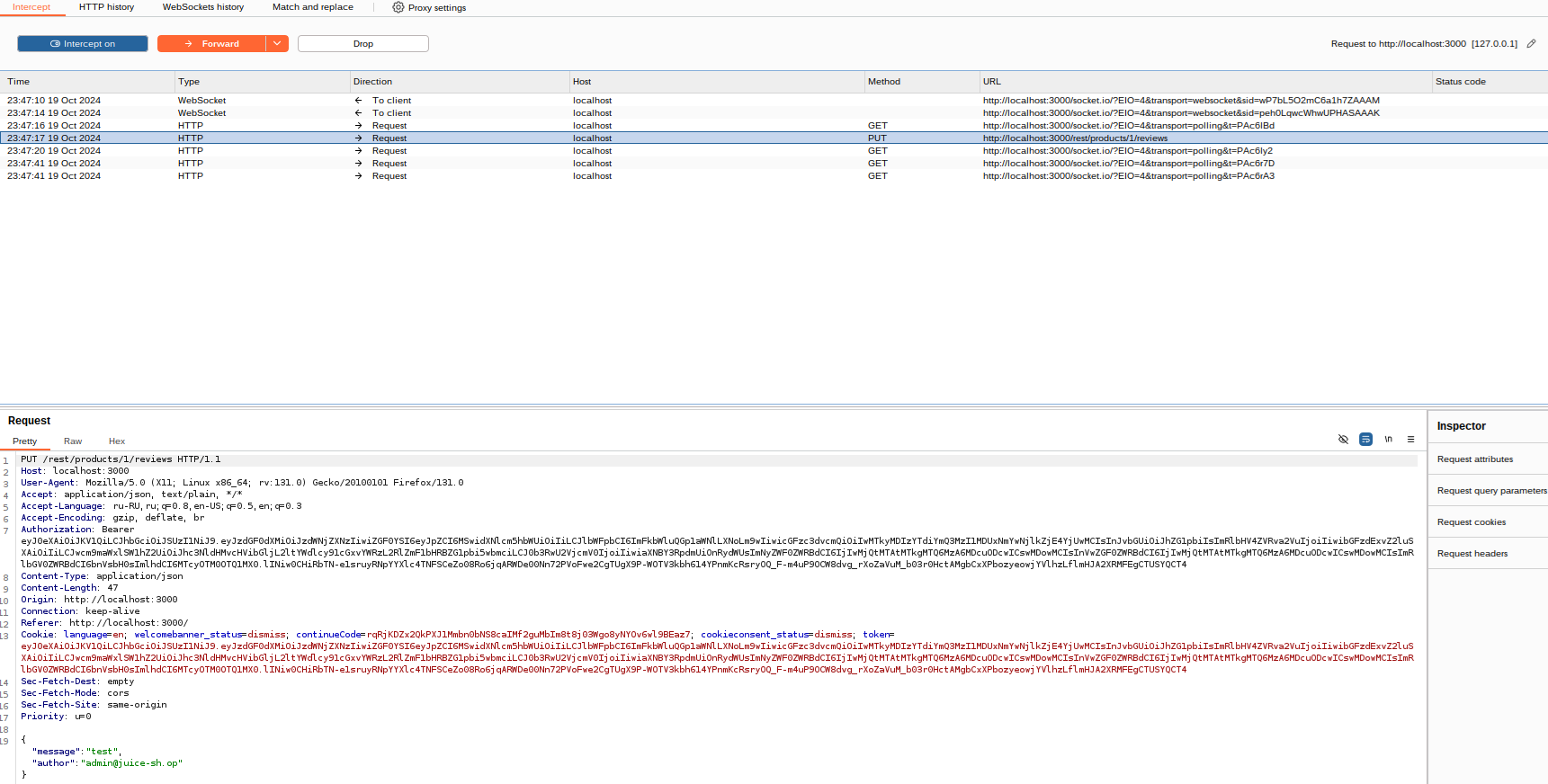


Если нарушить строку поиска, то можно вызвать ошибку стэка.

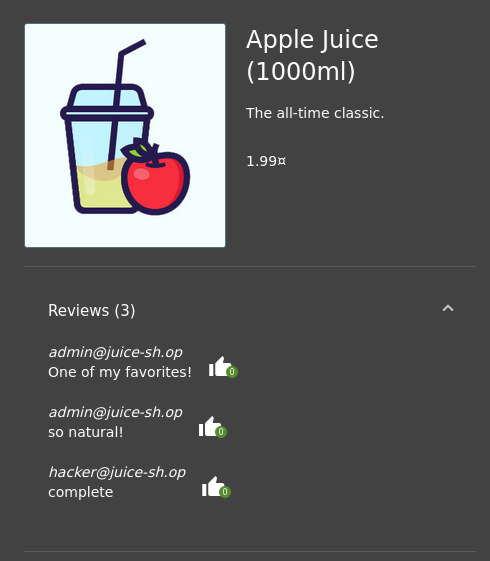


Уязвимость типа Security Misconfiguration. Для защиты можно использовать автоматизированные средства обнаружения и исправления ошибок конфигурации.

Перехватив пакет с отзывом с помощью модуля Intercept, можно заменить текст отзыва или адрес автора данного отзыва.



Я изменил message на complete, author на hacker@juice-sh.op и получил уже изменённый отзыв



Уязвимость типа Broken access control. Можно передавать информацию в зашифрованном виде.