Documentação da Biblioteca SecureIO

1. Tecnologias Utilizadas

- **Java JDK 21**: A biblioteca foi desenvolvida utilizando a versão mais recente do Java, garantindo acesso a recursos e melhorias de desempenho.
- **IDE IntelliJ**: Utilizada como ambiente de desenvolvimento integrado, o IntelliJ oferece ferramentas avançadas para desenvolvimento em Java, incluindo suporte a Spring Boot e funcionalidades de refatoração.
- **Spring Boot**: Todo o projeto é baseado no ecossistema Spring Boot, que facilita a configuração e desenvolvimento de aplicações Java, oferecendo uma arquitetura modular e práticas recomendadas para construção de APIs.
- **Postman**: Utilizado para realizar testes de API, tanto com entradas em formato HTML quanto JSON, simulando ataques de SQL Injection e XSS. O Postman permitiu a validação do comportamento da biblioteca sob diferentes cenários de entrada.

Necessidade da Biblioteca SecureIO

Ataque via SQL Injection

Em fevereiro de 2024, o grupo de hackers chamado "ResumeLooters" realizou um ataque massivo usando SQL injection contra 65 sites de empregos e lojas de varejo. O ataque, focado principalmente na região da Ásia-Pacífico (APAC), resultou no roubo de dados pessoais de mais de 2 milhões de candidatos a emprego. Esses dados incluíam nomes, endereços de e-mail, números de telefone e históricos de emprego. O grupo utilizou ferramentas de código aberto, como SQLmap, para automatizar a exploração de vulnerabilidades de SQL injection, permitindo controle total sobre os servidores de banco de dados comprometidos. Os dados roubados foram então colocados à venda em canais do Telegram.

Ataque via Cross-Site Scripting (XSS)

Em julho de 2024, a Netgear identificou uma vulnerabilidade grave de cross-site scripting (XSS) em vários modelos de seus roteadores WiFi 6. A falha permitia que atacantes injetassem scripts maliciosos nas interfaces web dos dispositivos, o que poderia levar à execução remota de código, se explorada corretamente. A Netgear rapidamente lançou uma atualização de firmware para mitigar o risco. O ataque poderia ser explorado para roubar credenciais de login ou comprometer a rede doméstica de forma mais ampla

Ambos os ataques destacam a importância da correção rápida de vulnerabilidades e da implementação de boas práticas de segurança em sites e dispositivos conectados.

Em um cenário onde ataques cibernéticos estão se tornando cada vez mais sofisticados, a segurança das aplicações web é uma prioridade crucial. **SQL Injection** e **Cross-Site**

Scripting (XSS) são algumas das vulnerabilidades mais comuns e perigosas que podem comprometer dados sensíveis e a integridade das aplicações.

A biblioteca **SecureIO** foi desenvolvida com a necessidade de proporcionar uma solução prática e eficaz para mitigar esses riscos. Com um foco em segurança, a biblioteca auxilia desenvolvedores a implementar medidas de proteção em suas aplicações Java de forma simples e eficiente.

Objetivo

O objetivo do SecureIO é fornecer ferramentas de validação, sanitização de entradas e geração de relatórios que permitam detectar e prevenir tentativas de ataque, contribuindo assim para a segurança das aplicações.

Instalação do SecureIO

Para utilizar a biblioteca SecureIO em um projeto Java web, siga estas etapas:

1. **Clone o repositório do GitHub**: A biblioteca está disponível para download no GitHub.

git clone https://github.com/Ricardo200021/SecureIO.git

Adicione a dependência no seu `pom.xml`: Para integrar o SecureIO ao seu projeto Maven, adicione a seguinte dependência ao arquivo `pom.xml`:

```
<dependency>
    <groupId>br.com.darkscreen</groupId>
    <artifactId>SecureIO</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

Configure a biblioteca: Certifique-se de que as classes de configuração e interceptores estão corretamente integrados no seu aplicativo Spring Boot. Isso pode incluir ajustes na configuração de segurança e nas rotas da API.

Resumo das Classes e Funcionalidades

A seguir, uma descrição detalhada de cada classe da biblioteca, suas funções principais e trechos de código relevantes:

1. Sanitizer

Descrição: Esta classe é responsável por sanitizar as entradas de dados, evitando ataques de XSS e SQL Injection através de um processo de limpeza de dados.

Trecho principal:

```
public String sanitize(String input) {  1 usage
    if (input == null || input.trim().isEmpty()) {
        return ""; // Retorna string vazia se a entrada for nula ou vazia
    }

    // Exemplo de sanitização básica para prevenir XSS e SQL Injection
    String sanitizedInput = input.replaceAll( regex: "<", replacement: "&lt;")  // Converte "<" para "&lt;"
            .replaceAll( regex: ">", replacement: "&gt;") // Converte ">" para "&gt;"
            .replaceAll( regex: "&", replacement: "&amp;") // Converte "&" para "&amp;"
            .replaceAll( regex: "'", replacement: "&#39;") // Converte "'" para "&#39;"
            .replaceAll( regex: "\"", replacement: "&quot;"); // Converte "" para "&quot;"

    logger.info( msg: "Input sanitized: " + sanitizedInput); // Loga a entrada sanitizada
    return sanitizedInput; // Retorna a entrada sanitizada
}
```

Funcionalidade: Converte caracteres especiais em entidades HTML, impedindo a execução de scripts maliciosos.

SanitizerController

Descrição: Controlador REST que gerencia as requisições para sanitização de entradas, utilizando a classe `Sanitizer`.

Trecho principal:

```
@PostMapping("/") no usages
public ResponseEntity<String> sanitize(@RequestBody String input) {
    // Verifica se a entrada é suspeita de SQL Injection
    if (SQLInjectionValidator.isSQLInjectionSuspected(input)) {
        logger.warning( msg: "SQL Injection attempt detected: " + input);
        return ResponseEntity.badRequest().body("SQL Injection attempt detected: " + input);
    }

    // Verifica se a entrada é suspeita de XSS
    if (XssValidator.isXSSSuspected(input)) {
        logger.warning( msg: "XSS attempt detected: " + input);
        return ResponseEntity.badRequest().body("XSS attempt detected: " + input);
    }

    // Se a entrada for segura, sanitiza e retorna o resultado
    String sanitizedInput = sanitizer.sanitize(input);
    return ResponseEntity.ok( body: "Input sanitized: " + sanitizedInput);
}
```

Funcionalidade: Recebe dados em formato JSON ou texto simples, aplica a sanitização e retorna o resultado ao usuário.

SecureApplication

Descrição: Contém métodos auxiliares para validar a entrada e verificar a suspeita de ataques XSS.

Trecho principal:

```
public static boolean isXSSSuspected(String input) {  no usages
    if (input == null || input.trim().isEmpty()) {
        return false;
    }

    for (Pattern pattern : XSS_PATTERNS) {
        if (pattern.matcher(input).find()) {
            logger.warning( msg: "XSS pattern detected: " + input);

            // Gerar relatório em PDF
            ReportGenerator reportGenerator = new ReportGenerator();
            reportGenerator.generateReport( filePath: "XSS Attack Detected",  reportContent: "Suspicious input: " + input);

            return true;
        }
    }
    return false;
}
}
```

Funcionalidade: Avalia se a entrada possui padrões que indicam a presença de um ataque XSS.

SQLInjectionInterceptor

Descrição: Interceptor que analisa parâmetros de requisição HTTP em busca de possíveis tentativas de SQL Injection antes da execução.

Trecho principal:

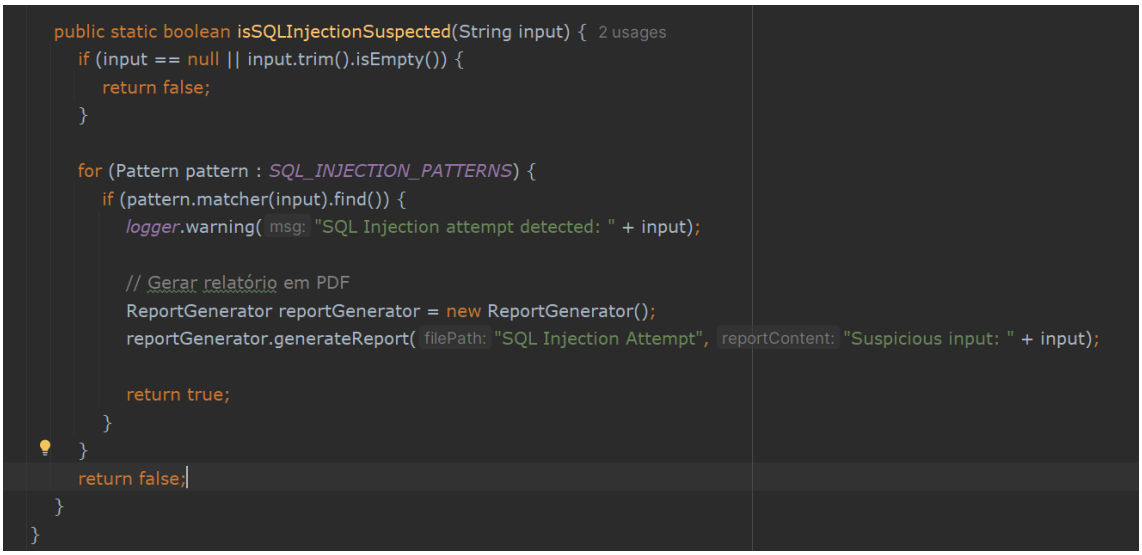
```
@Override  no usages
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
    // Iterar sobre todos os parâmetros da requisição
    Enumeration<String> parameterNames = request.getParameterNames();

    while (parameterNames.hasMoreElements()) {
        String paramName = parameterNames.nextElement();
        String paramValue = request.getParameter(paramName);

        // Verificar se o valor do parâmetro contém SQL Injection
        if (SQLInjectionValidator.isSQLInjectionSuspected(paramValue)) {
            logger.warning( msg: "SQL Injection attempt detected in parameter '" + paramName + "': " + paramValue);

            // Gerar relatório em PDF
            ReportGenerator reportGenerator = new ReportGenerator();
            reportGenerator.generateReport( filePath: "SQL Injection detected", reportContent: "Parameter: " + paramName + ", Value: " + paramValue);

            // Bloqueia a requisição e retorna erro
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, s: "Tentativa de SQL Injection detectada!");
            return false;
        }
    }
    return true; // Permite a requisição se não houver problemas
}
}
```

Funcionalidade: Interrompe a requisição caso um ataque seja detectado, enviando um erro 400 ao cliente.

SQLInjectionValidator

Descrição: Esta classe valida as entradas para identificar padrões de SQL Injection.

Trecho principal:



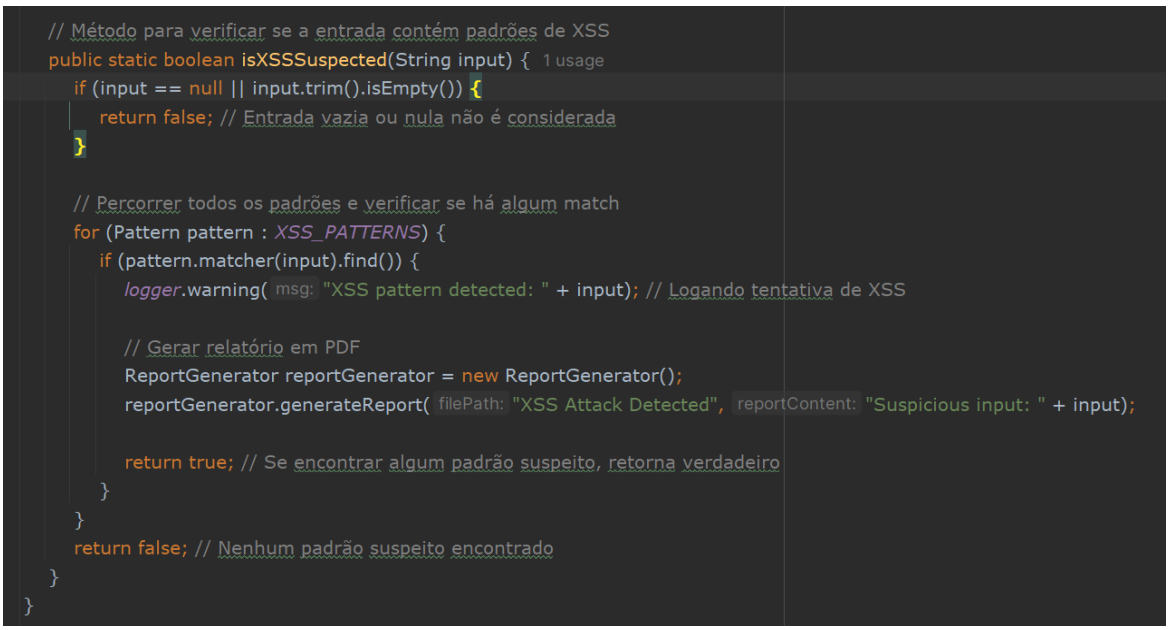
```
public static boolean isSQLInjectionSuspected(String input) {  2 usages
    if (input == null || input.trim().isEmpty()) {
        return false;
    }

    for (Pattern pattern : SQL_INJECTION_PATTERNS) {
        if (pattern.matcher(input).find()) {
            logger.warning( msg: "SQL Injection attempt detected: " + input);

            // Gerar relatório em PDF
            ReportGenerator reportGenerator = new ReportGenerator();
            reportGenerator.generateReport( filePath: "SQL Injection Attempt",  reportContent: "Suspicious input: " + input);

            return true;
        }
    }
    return false;
}
}
```

Funcionalidade: Verifica se a entrada contém comandos SQL suspeitos, retornando um booleano indicando a presença de um ataque.

XssValidator

Descrição: Similar ao `SQLInjectionValidator`, mas focada na detecção de XSS.

Trecho principal:



```
// Método para verificar se a entrada contém padrões de XSS
public static boolean isXSSSuspected(String input) {  1 usage
    if (input == null || input.trim().isEmpty()) {
        return false; // Entrada vazia ou nula não é considerada
    }

    // Percorrer todos os padrões e verificar se há algum match
    for (Pattern pattern : XSS_PATTERNS) {
        if (pattern.matcher(input).find()) {
            logger.warning( msg: "XSS pattern detected: " + input); // Logando tentativa de XSS

            // Gerar relatório em PDF
            ReportGenerator reportGenerator = new ReportGenerator();
            reportGenerator.generateReport( filePath: "XSS Attack Detected",  reportContent: "Suspicious input: " + input);

            return true; // Se encontrar algum padrão suspeito, retorna verdadeiro
        }
    }
    return false; // Nenhum padrão suspeito encontrado
}
}
```

Funcionalidade: Identifica padrões que podem ser utilizados para injetar scripts maliciosos na aplicação.

ReportGenerator

Descrição: Gera relatórios em PDF sobre tentativas de ataques detectadas.

Trecho principal:

```java
public void generateReport(String filePath, String reportContent) {  4 usages
    // Validações de entrada
    if (filePath == null || filePath.isEmpty()) {
        throw new IllegalArgumentException("Caminho do arquivo não pode ser nulo ou vazio.");
    }

    if (reportContent == null || reportContent.isEmpty()) {
        throw new IllegalArgumentException("Conteúdo do relatório não pode ser nulo ou vazio.");
    }

    // Tenta gerar o relatório
    try (PDDocument document = new PDDocument()){...} catch (IOException e) {
        // Trata a exceção de forma adequada
        throw new RuntimeException("Erro ao gerar o relatório em PDF: " + e.getMessage(), e);
    }
}
}
```

Funcionalidade: Documenta os incidentes de segurança, criando registros que podem ser usados para auditoria e análise.

Conclusão

A biblioteca **SecureIO** oferece uma solução prática e robusta para fortalecer a segurança de aplicações web em Java. Com funcionalidades que vão desde a sanitização de entradas até a geração de relatórios, o SecureIO ajuda desenvolvedores a implementar práticas de segurança eficazes. Recomendamos que você explore o repositório no GitHub, teste a biblioteca em seus projetos e contribua para o seu desenvolvimento contínuo.