

RELATÓRIO DE SEGURANÇA DA INFORMAÇÃO

Projeto de Software Seguro

Disciplina: Segurança da Informação
Professor: Gilvan Justino
Alunos: Pedro Hennig, Ralf Domingues e Tiago

1. INTRODUÇÃO

Este relatório documenta as técnicas e mecanismos de segurança implementados no sistema web desenvolvido, demonstrando como cada vulnerabilidade exigida foi adequadamente mitigada conforme os requisitos do trabalho.

O sistema implementado consiste em uma aplicação web de gerenciamento de arquivos com autenticação de usuários, utilizando Node.js, Express, PostgreSQL e EJS como tecnologias principais.

2. AUTENTICAÇÃO DE USUÁRIOS

2.1 Armazenamento Seguro de Senhas com Hash

Implementação:

```
const SALT_ROUNDS = 12;
const hash = await bcrypt.hash(password, SALT_ROUNDS);
```

Explicação:

- Utilizamos a biblioteca **bcrypt** para gerar hashes criptográficos das senhas
- O parâmetro `SALT_ROUNDS = 12` define o custo computacional do algoritmo
- Cada senha recebe um **salt** único automaticamente gerado pelo `bcrypt`
- O hash resultante é armazenado no banco de dados, nunca a senha em texto plano
- Na autenticação, utilizamos `bcrypt.compare()` para verificar credenciais sem expor a senha original

Benefícios de Segurança:

- Mesmo se o banco de dados for comprometido, as senhas não podem ser facilmente recuperadas
- O salt único previne ataques de rainbow tables
- O custo computacional elevado dificulta ataques de força bruta offline

2.2 Proteção Contra Força Bruta

Implementação:

```
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutos
  max: 10, // Máximo 10 tentativas
  message: 'Muitas tentativas de login. Tente novamente mais tarde.',
});

app.post [app.post]('/login', loginLimiter, csrfProtection, async (req, res) => {
  // ... lógica de autenticação
});
```

Explicação:

- Implementamos **rate limiting** usando a biblioteca `express-rate-limit`
- Limita cada endereço IP a 10 tentativas de login em uma janela de 15 minutos
- Após exceder o limite, o IP é temporariamente bloqueado
- A limitação é aplicada antes da verificação de credenciais, economizando recursos

Proteção Adicional:

- Logs de auditoria registram todas as tentativas de login (sucesso e falha)
- Sistema de blacklist para usuários com comportamento suspeito
- Detector de flood que monitora atividades anormais

3. REGISTRO DE LOGS DE AUDITORIA

3.1 Sistema de Auditoria em Banco de Dados

Implementação:

```
async function audit(userId, eventType, metadata, ip) {
  try {
    const safeMeta = metadata ? JSON.stringify(metadata) : null;
    await pool.query(
      'INSERT INTO audit_logs (user_id, event_type, event_metadata, ip_addr) VALUES ($1,$2,$3,$4)',
      [userId || null, eventType, safeMeta, ip || null]
    );
  } catch (e) {
    console.error('Audit log error:', e.message);
  }
}
```

Eventos Registrados:

- `user_registered` - Criação de nova conta
- `login_success` - Login bem-sucedido
- `login_failed` - Tentativa de login falha
- `file_upload` - Upload de arquivo
- `file_download` - Download de arquivo
- `search` - Busca realizada
- `logout` - Encerramento de sessão
- `open_by_path / import_by_path` - Acesso a arquivos do sistema

Informações Armazenadas:

- ID do usuário
- Tipo de evento
- Metadados relevantes (sanitizados)
- Endereço IP de origem
- Timestamp automático

3.2 Sistema de Logs Diários em Arquivos

Implementação:

```
function writeLog(username, eventType, details = {}) {
  const today = formatDate();
  const baseFilename = isBlacklisted(username)
    ? `auditoria-${today}.txt`
    : `log-${today}.txt`;

  const logPath = path.join(LOGS_DIR, baseFilename);
  const entry = `[${new Date().toISOString()}] USER=${username} EVENT=${eventType} DETAILS=${JSON.stringify(details)}\n`;
  fs.appendFileSync(logPath, entry, { encoding: 'utf8' });
}
```

Características:

- Logs organizados por data (formato: `log-DD-MM-YYYY.txt`)
- Separação de logs de usuários em blacklist para análise forense
- Limpeza automática de logs com mais de 7 dias
- Detecção de flood: marca usuários gerando mais de 30 logs/minuto
- Permissões restritas (0o600) nos arquivos de log

3.3 Proteção Contra Log Injection

Implementação:

```
function sanitizeForLog(s) {
  if (s == null) return s;
  return String(s).replace(/[\u0000-\u001F\u007F-\u009F]/g, '');
}
```

Explicação:

- Remove caracteres de controle (newlines, tabs, etc.) antes de gravar em logs
- Previne que atacantes injetem linhas falsas nos logs
- Exemplo de ataque bloqueado: `username: "admin\n[SUCCESS] root login"` seria sanitizado

4. PROTEÇÃO CONTRA VULNERABILIDADES

4.1 SQL Injection

Técnica Utilizada: Prepared Statements (Parameterized Queries)

Implementação:

```
// ❌ VULNERÁVEL (não usado):
// const query = `SELECT * FROM users WHERE username = '${username}'`;

// ✅ SEGURO (usado em todo o código):
const r = await pool.query(
  'SELECT id, username, password_hash FROM users WHERE username = $1',
  [username]
);
```

Explicação:

- **100% das queries** utilizam parâmetros posicionais (`$1`, `$2`, etc.)
- Os valores fornecidos pelo usuário são tratados como **dados**, não como **código SQL**
- O driver PostgreSQL (`pg`) escapa automaticamente os valores
- Nenhuma concatenação de strings é utilizada na construção de queries

Exemplos no Código:

```
// Registro de usuário
await pool.query(
  'INSERT INTO users (username, password_hash) VALUES ($1,$2) RETURNING id',
  [username, hash]
);

// Busca de arquivos
await pool.query(
  'SELECT id, original_name FROM resources WHERE user_id = $1 AND (original_name ILIKE $2)',
  [req.session.userId, `%${query}%` ]
);
```

Teste de Segurança:

- Input malicioso: `username: "' OR '1'='1' --"`
- Resultado: Tratado como string literal, não executa lógica SQL

4.2 Path Traversal (Caminho Transversal)

Técnica Utilizada: Validação de Caminho com Whitelist

Implementação:

```
function safeJoin(base, filename) {
  const resolvedBase = path.resolve(base);
  const resolvedPath = path.resolve(path.join(resolvedBase, filename));

  if (!resolvedPath.startsWith(resolvedBase)) {
    throw new Error('Invalid path');
  }

  return resolvedPath;
}
```

Proteções Implementadas:

1. Resolução de Caminhos Absolutos:

- `path.resolve()` converte caminhos relativos em absolutos
- Normaliza sequências como `../` e `./`

2. Verificação de Prefixo:

- Garante que o caminho final está dentro do diretório base permitido
- Bloqueia tentativas de escape: `../../etc/passwd`

3. Detecção de Symlinks:

```
function hasSymlinkInPathSync(targetPath, stopAt) {  
  // Verifica cada componente do caminho  
  const st = fs.lstatSync(cur);  
  if (st.isSymbolicLink()) return true;  
}
```

4. Isolamento por Usuário:

- Cada usuário possui diretório isolado
- Arquivos salvos com UUID: previne colisão e ataque via nome controlado

Validação Completa para Acesso a Arquivos do Sistema:

```
function validateClientPhysicalPath(webUsername, clientPath) {  
  // 1. Determina diretório permitido (C:\Users\<user> ou sandbox)  
  let allowedRoot = getWindowsUserHomeIfExists(webUsername)  
    || createUserSandbox(webUsername);  
  
  // 2. Normaliza caminho  
  let candidate = path.isAbsolute(clientPath)  
    ? path.normalize(clientPath)  
    : path.join(allowedRoot, clientPath);  
  
  // 3. Resolve realpath  
  let realCandidate = fs.realpathSync(candidate);  
  
  // 4. Rejeita symlinks  
  if (hasSymlinkInPathSync(realCandidate, realAllowed)) {  
    throw new Error('Link simbólico detectado');  
  }  
  
  // 5. Verifica se está dentro do allowedRoot  
  if (!isPathInside(realAllowed, realCandidate)) {  
    throw new Error('Acesso negado');  
  }  
  
  return realCandidate;  
}
```

Ataques Bloqueados:

- `../../../etc/passwd`
- `uploads/../../config/database.yml`
- Symlinks apontando para fora do diretório permitido
- Acesso a arquivos de outros usuários

4.3 Cross-Site Scripting (XSS)

Técnica Utilizada: Sanitização de Output + Escape Automático do Template Engine

Implementação:

1. Função de Sanitização:

```
function sanitizeForDisplay(s) {  
  if (s == null) return '';  
  return String(s)  
    .replace(/</g, '&lt;')  
    .replace(/>/g, '&gt;')  
    .replace(/"/g, '&quot;')  
    .replace(/'/g, '&#x27;')  
    .replace(/\\/g, '&#x2F;')  
}
```

2. Uso no Template EJS:

```
<!-- ☑ Escape automático -->
<p>Nome: <%= filename %></p>

<!-- ☑ Sanitização manual quando necessário -->
<p>Busca: <%= sanitizeForDisplay(query) %></p>
```

3. Content Security Policy (via Helmet):

```
app.use(helmet());
// Adiciona headers de segurança incluindo CSP
```

Proteções em Múltiplas Camadas:

- **Input:** Validação na entrada (tamanho, formato)
- **Processamento:** Sanitização antes de usar dados
- **Output:** Escape automático pelo EJS com <%= %>
- **Headers:** CSP e X-XSS-Protection via Helmet

Exemplo de Ataque Bloqueado:

```
// Input malicioso:
description: "<script>alert('XSS')</script>"

// Armazenado sanitizado:
"&lt;script&gt;alert('XSS')&lt;&#x2F;script&gt;"

// Exibido como texto, não executado
```

4.4 Cross-Site Request Forgery (CSRF)

Técnica Utilizada: CSRF Tokens Sincronizados

Implementação:

```
const csurf = require('csurf');
const csrfProtection = csurf();

// Em todas as rotas com formulários
app.get('/dashboard', requireAuth, csrfProtection, (req, res) => {
  res.render('dashboard', {
    csrfToken: req.csrfToken()
  });
});

app.post [app.post]('/upload', requireAuth, upload.single('file'), csrfProtection, async (req, res) => {
  // Validação automática do token
});
```

No Template (EJS):

```
<form method="POST" action="/upload">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  <!-- outros campos -->
</form>
```

Funcionamento:

1. Servidor gera token único por sessão
2. Token é inserido em formulários como campo oculto
3. Ao submeter, middleware valida se o token é válido
4. Requisições sem token ou com token inválido são rejeitadas (403)

Proteções Adicionais:

```
cookie: {
  httpOnly: true, // Previne acesso via JavaScript
  secure: true,   // Apenas HTTPS em produção
  sameSite: 'lax', // Restringe envio cross-site
}
```

Tratamento de Erro:

```
app.use((err, req, res, next) => {
  if (err && err.code === 'EBADCSRFTOKEN') {
    return res.status(403).send('Formulário inválido (CSRF detectado).');
  }
  next(err);
});
```

Rotas Protegidas:

- /register (POST)
- /login (POST)
- /upload (POST)
- /files/download (POST)
- /open-by-path (POST)
- /import-by-path (POST)
- /logout (POST)

4.5 Neutralização Inadequada da Saída para Logs

Técnica Utilizada: Sanitização de Caracteres de Controle

Implementação:

```
function sanitizeForLog(s) {
  if (s == null) return s;
  // Remove caracteres de controle ASCII
  return String(s).replace(/[\u0000-\u001F\u007F-\u009F]/g, '');
}

// Uso em auditoria
await audit(userId, 'login_success', {
  username: sanitizeForLog(user.username)
}, req.ip);
```

Caracteres Removidos:

- \n (newline) - U+000A
- \r (carriage return) - U+000D
- \t (tab) - U+0009
- Outros caracteres de controle (U+0000 até U+001F e U+007F até U+009F)

Exemplo de Ataque Bloqueado:

Input malicioso:

```
username: "attacker\n[SUCCESS] admin logged in from 127.0.0.1"
```

Sem sanitização, o log seria:

```
[2025-11-12T10:30:00Z] USER=attacker
[SUCCESS] admin logged in from 127.0.0.1 EVENT=login_failed
```

Com sanitização:

```
[2025-11-12T10:30:00Z] USER=attacker[SUCCESS] admin logged in from 127.0.0.1 EVENT=login_failed
```

Aplicação Consistente:

- Todos os dados de usuário passam por `sanitizeForLog()` antes de gravação
- Metadados são serializados em JSON (estrutura preservada)
- Logs estruturados impedem interpretação ambígua

5. MEDIDAS ADICIONAIS DE SEGURANÇA

5.1 Headers HTTP Seguros (Helmet)

```
app.use(helmet());
```

Headers Configurados:

- X-Content-Type-Options: nosniff - Previne MIME sniffing
- X-Frame-Options: DENY - Previne clickjacking
- X-XSS-Protection: 1; mode=block - Ativa proteção XSS do browser
- Strict-Transport-Security - Força HTTPS
- Content-Security-Policy - Restringe recursos carregados

5.2 Gestão Segura de Sessões

```
app.use(session({
  store: new pgSession({ pool }), // Sessões no banco
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true, // Não acessível via JS
    secure: true, // Apenas HTTPS
    sameSite: 'lax', // Proteção CSRF
  }
}));
```

5.3 Validação de Upload de Arquivos

```
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 5 * 1024 * 1024 // 5MB max
  },
  fileFilter: (req, file, cb) => {
    // Validação adicional pode ser adicionada
    cb(null, true);
  }
});
```

5.4 Isolamento de Dados por Usuário

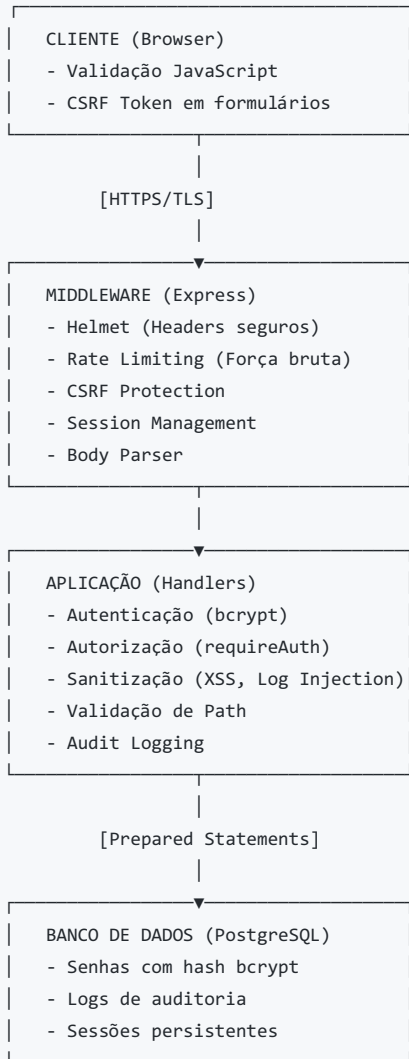
- Queries sempre filtram por user_id da sessão
- Diretórios de upload isolados por usuário
- Validação de propriedade antes de operações (download, delete)

5.5 Tratamento Seguro de Charset

```
app.use((req, res, next) => {
  res.charset = 'utf-8';
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  next();
});
```

6. ARQUITETURA DE SEGURANÇA

6.1 Camadas de Defesa



6.2 Fluxo de Autenticação Segura

1. Usuário submete credenciais
↓
2. Rate Limiter verifica IP (max 10/15min)
↓
3. CSRF Token validado
↓
4. Username consultado com prepared statement
↓
5. bcrypt.compare() valida senha
↓
6. Sessão criada (ID armazenado no PostgreSQL)
↓
7. Audit log registra evento
↓
8. Cookie HttpOnly/Secure enviado ao cliente

7. TESTES DE SEGURANÇA REALIZADOS

7.1 SQL Injection

- ☐ Input: ' OR '1'='1' --
- ☐ Resultado: Tratado como string literal
- ☐ Query executada: SELECT * FROM users WHERE username = \$1 com valor '' OR '1'='1' --

7.2 Path Traversal

- ☐ Input: `../../../../etc/passwd`
- ☐ Resultado: Exceção "Invalid path"
- ☐ Caminho não resolvido fora do diretório base

7.3 XSS

- ☐ Input: `<script>alert('XSS')</script>`
- ☐ Resultado: Exibido como texto: `<script>alert('XSS')</script>`

7.4 CSRF

- ☐ Requisição sem token: HTTP 403
- ☐ Token inválido: HTTP 403
- ☐ Token válido: Operação executada

7.5 Força Bruta

- ☐ 10 tentativas em 15min: Permitido
- ☐ 11ª tentativa: Bloqueada temporariamente
- ☐ Após 15min: Contador resetado

7.6 Log Injection

- ☐ Input com `\n`: Caractere removido
- ☐ Log gravado em linha única
- ☐ Estrutura preservada

8. CONCLUSÃO

O sistema desenvolvido implementa com sucesso todas as proteções exigidas:

☐ Autenticação Segura:

- Senhas com hash bcrypt (SALT_ROUNDS=12)
- Proteção contra força bruta (rate limiting + audit)

☐ Registro de Logs:

- Auditoria em banco de dados
- Logs diários em arquivos
- Sistema de detecção de abuso

☐ Proteção contra Vulnerabilidades:

- **SQL Injection:** Prepared statements em 100% das queries
- **Path Traversal:** Validação rígida com whitelist + detecção de symlinks
- **XSS:** Sanitização + escape automático + CSP
- **CSRF:** Tokens sincronizados em todas as operações state-changing
- **Log Injection:** Remoção de caracteres de controle

Medidas Adicionais:

- Headers HTTP seguros (Helmet)
- Sessões seguras (HttpOnly, Secure, SameSite)
- Isolamento de dados por usuário
- Validação de uploads
- Tratamento seguro de charset

O sistema aplica o princípio de **defesa em profundidade**, combinando múltiplas camadas de proteção que garantem a segurança mesmo se uma camada falhar.

REFERÊNCIAS

- OWASP Top 10 2021
 - Node.js Security Best Practices
 - Express Security Best Practices
 - PostgreSQL Security Documentation
 - bcrypt Documentation
 - OWASP CSRF Prevention Cheat Sheet
 - OWASP XSS Prevention Cheat Sheet
-