



CURSO DE PÓS-GRADUAÇÃO EM
CIBERSEGURANÇA - TURMA 2024.2

REVISÃO DE CÓDIGO E ANÁLISE ESTÁTICA AUTOMATIZADA



Alunos:

Adyellen Alves

Carolina Malinconico Vasconcelos

Fâbjo Campos

Rodrigo Almeida

Professor:

Carlos Eduardo Santos Barros Junior

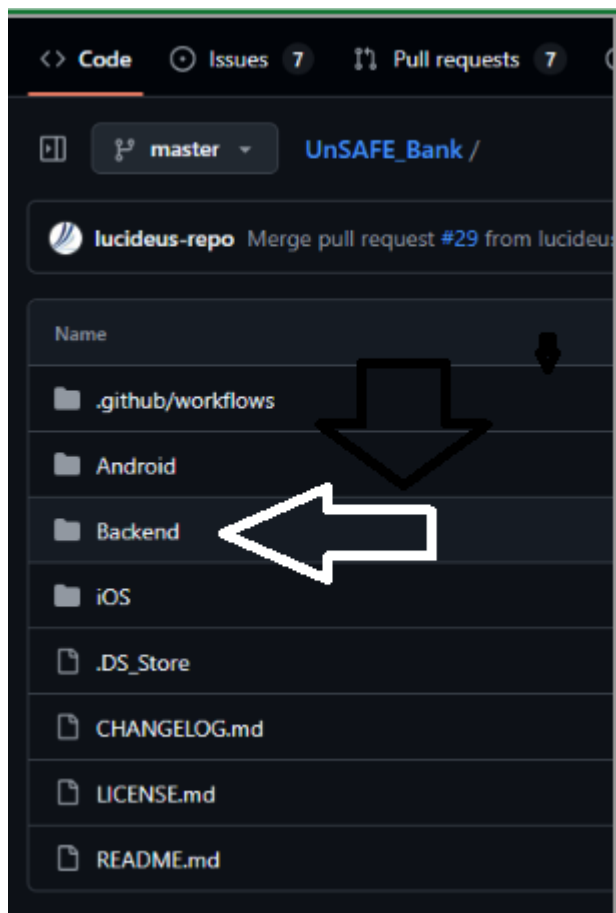
**ANAIISÃO DE PROOCCSESIAD AJÁTÁTICA
AUTOMATIZADA**

RECIFE 2025

Análise de Código – UnSAFE Bank (PHP)

1) Escopo e contexto

Projeto escolhido: **UnSAFE Bank** (PHP). O foco do review foi o **Backend** (CodeIgniter), cobrindo configuração, autenticação e acesso a dados.



2) Metodologia

1. **Mapeamento de pastas** e identificação de alvos: application/config/, application/controllers/, application/models/, helpers/, mysql/.
2. **Busca por palavras-chave** típicas: config.php, database.php, csrf, cookie, jwt, SELECT, UPDATE, \$_GET/\$_POST, echo.
3. **Inspeção manual** de arquivos prioritários: config.php, database.php, controllers/Login.php, helpers/jwt_helper.php, mysql/db.sql.

3) Análise dos arquivos:

3.1 Config.php

```
223  /*
224  |-----
225  | Global XSS Filtering
226  |-----
227  |
228  | Determines whether the XSS filter is always active when GET, POST or
229  | COOKIE data is encountered
230  |
231  | WARNING: This feature is DEPRECATED and currently available only
232  |         for backwards compatibility purposes!
233  |
234  */
235  $config['global_xss_filtering'] = FALSE;
236
```



No arquivo config.php foi identificado: `$config['global_xss_filtering'] = FALSE`. Isso significa que o CodeIgniter NÃO aplica o `xss_clean` automaticamente aos dados de GET/POST/COOKIE. Esse filtro global é, inclusive, de uso (deprecated). Na prática, qualquer dado vindo do usuário que for renderizado sem escape em uma view pode resultar em XSS refletido ou persistido.

Impacto: execução de JavaScript no contexto do usuário (roubo de sessão, defacement, movimentações indevidas).

Mitigações Propostas

Opcional: habilitar `$config['global_xss_filtering'] = TRUE`, ciente de que não substitui o escape de saída e pode afetar entradas válidas.

```
$config['cookie_secure'] = TRUE;
$config['cookie_httponly'] = TRUE;
$config['cookie_samesite'] = 'Strict';
$this->output->set_header("Content-Security-Policy: default-src 'self'; script-src 'self'");
```

Reduzir impacto: configurar cookies de sessão com Secure/HttpOnly/SameSite e adotar CSP.

```
$safe = $this->input->post('campo', TRUE);
$limpo = $this->security->xss_clean($valor);
```

Sanitização pontual: `$this->input->post('campo', TRUE)` ou `$this->security->xss_clean($valor)`.

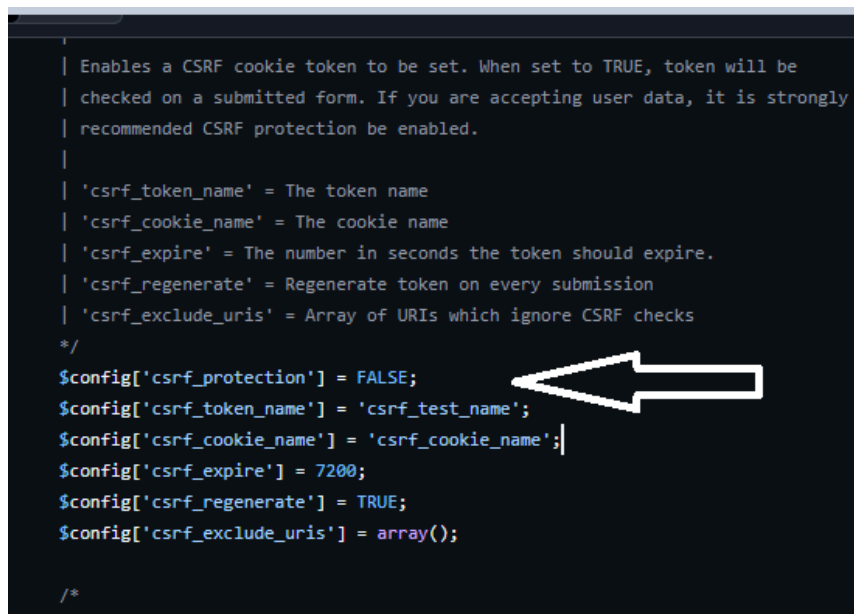
```
$this->load->library('form_validation');
$this->form_validation->set_rules('nome', 'Nome', 'required|trim|max_length[128]');
if (!$this->form_validation->run()) { /* 400 Bad Request */ }
```

- Validação/normatização no servidor (ex.: form_validation) e limites de tamanho/tipo.

```
<?= html_escape($var) ?>
var user = JSON.parse('<?= json_encode($user,
JSON_HEX_TAG|JSON_HEX_AMP|JSON_HEX_APOS|JSON_HEX_QUOT) ?>');
```

- Escape na saída: em views use <?= html_escape(\$var) ?>; para dados embutidos em JS use json_encode com flags.

CSRF :



```
| Enables a CSRF cookie token to be set. When set to TRUE, token will be
| checked on a submitted form. If you are accepting user data, it is strongly
| recommended CSRF protection be enabled.
|
| 'csrf_token_name' = The token name
| 'csrf_cookie_name' = The cookie name
| 'csrf_expire' = The number in seconds the token should expire.
| 'csrf_regenerate' = Regenerate token on every submission
| 'csrf_exclude_uris' = Array of URIs which ignore CSRF checks
|
| */
$config['csrf_protection'] = FALSE;
$config['csrf_token_name'] = 'csrf_test_name';
$config['csrf_cookie_name'] = 'csrf_cookie_name';
$config['csrf_expire'] = 7200;
$config['csrf_regenerate'] = TRUE;
$config['csrf_exclude_uris'] = array();


/*
```

Isso significa que **nenhum formulário** terá proteção contra ataques **Cross-Site Request Forgery (CSRF)**.

- Vulnerabilidade: um atacante pode forçar o usuário logado a executar ações sem consentimento.
- Mitigação: definir **TRUE** e usar tokens nos formulários.

COOKIES:

```
| -----  
|  
| 'cookie_prefix' = Set a cookie name prefix if you need to avoid collisions  
| 'cookie_domain' = Set to .your-domain.com for site-wide cookies  
| 'cookie_path'   = Typically will be a forward slash  
| 'cookie_secure' = Cookie will only be set if a secure HTTPS connection exists.  
| 'cookie_httponly' = Cookie will only be accessible via HTTP(S) (no javascript)  
|  
| Note: These settings (with the exception of 'cookie_prefix' and  
|       'cookie_httponly') will also affect sessions.  
|  
*/  
$config['cookie_prefix']   = '';  
$config['cookie_domain']   = '';  
$config['cookie_path']     = '/';  
$config['cookie_secure']   = FALSE;  
$config['cookie_httponly'] = FALSE;
```



Isso deixa os cookies mais expostos:

- Sem Secure, eles podem trafegar em **HTTP** simples.
- Sem HttpOnly, o **JavaScript** consegue ler o cookie (abrindo brecha para XSS roubar sessão).
- Mitigação: colocar como TRUE

Chave de criptografia exposta no código :

```
| -----  
| Encryption Key  
| -----  
|  
| If you use the Encryption class, you must set an encryption key.  
| See the user guide for more info.  
|  
| https://codeigniter.com/user\_guide/libraries/encryption.html  
|  
*/  
$config['encryption_key'] = '9bbc0d79e686e847bc305c9bd4cc2ea6';
```

Nunca é seguro manter a chave de criptografia em texto puro no repositório.

- Mitigação: armazenar em variáveis de ambiente ou em arquivos fora do versionamento.

3.2 database.php

https://github.com/lucideus-repo/UnSAFE_Bank/blob/master/Backend/src/api/application/config/database.php

```
| the query builder class.  
*/  
$active_group = 'default';  
$query_builder = TRUE;  
  
$db['default'] = array(  
    'dsn'        => '',  
    'hostname'   => 'database',  
    'username'   => 'phpmyadmin',  
    'password'   => '531486b2bf646636a6a1bba61e78ec4a4a54efbd',  
    'database'   => 'abstractwallet',  
    'dbdriver'   => 'mysqli',  
    'dbprefix'   => ''
```

Credenciais em texto-plano e usuário inadequado

Recomendação:

- **Rotacionar imediatamente** a senha exposta.
- Criar usuário de aplicação com **privilégios mínimos** (SELECT/INSERT/UPDATE/DELETE apenas).
- Mover segredos para **variáveis de ambiente** (DB_HOST, DB_USER, DB_PASS, DB_NAME)

Criptografia de transporte desativada

```
'dbcollat' => 'utf8_general_ci',  
'swap_pre' => '',  
'encrypt'  => FALSE,  
'compress' => FALSE,
```

Evidência: Encriptação como desabilitado

Risco/Impacto: se o BD não for local/mesma rede segura, o tráfego pode ser interceptado.

3.3 Login.php

https://github.com/lucideus-repo/UnSAFE_Bank/blob/master/Backend/src/api/application/controllers/Login.php

Lógica do código feita sem restrição de método HTTP (GET/POST)

```
{  
    $this->load->helper('request_response');  
    $status = new status_codes();  
    $parsed = request_parse();  
    if ($parsed['status_code'] != 'ALLOK1') {  
        echo prepare_response(  
            'Failed',  
            $parsed['status_code'],  
            $parsed['message'],  
            time(),  
            (object)array()  
        );  
        return false;  
    }
```

evidência abaixo

Falta de Sanitização e Validação de Input

- O código pega **\$parsed = request_parse();** e repassa diretamente para **validateLoginAPIParameter()** e **login_user()**.
- Se **request_parse()** não higienizar corretamente, pode haver SQL Injection, XSS ou manipulação de parâmetros críticos.

Falta de Proteção CSRF (Cross-Site Request Forgery)

- Mesmo sendo API, se não tiver tokens CSRF ou JWTs com antiforgery claims, pode ser explorado.

Sem HTTPS / Sem Segurança de Transporte

- Não aparece aqui, mas se esse login estiver exposto em HTTP simples, o tráfego (usuário/senha) pode ser capturado → CWE-319: Cleartext Transmission.

```
8  ✓      public function index()
9          {
10             $this->load->helper('request_response');
11             $status = new status_codes();
12             $parsed = request_parse();
13             if ($parsed['status_code'] != 'ALLOK1') {
14                 echo prepare_response(
15                     'Failed',
16                     $parsed['status_code'],
17                     $parsed['message'],
18                     time(),
19                     (object)array()
20                 );

```

Mitigações Propostas

Sanitização de Input

- Validar e higienizar `$parsed` antes de usar.
- Usar CodeIgniter Form Validation:

```
$this->form_validation->set_rules('username', 'Username', 'required|trim|alpha_numeric');
```

```
$this->form_validation->set_rules('password', 'Password', 'required|min_length[8]');
```

Prevenir SQL Injection

Garantir que `LoginModuleHandler->login_user()` use queries parametrizadas:

```
$this->db->where('username', $username);
```

- `$query = $this->db->get('users');`

Controle de Tentativas de Login (Brute Force)

- Implementar rate limiting (ex.: bloquear após 5 tentativas falhas em 5 minutos).
- Adicionar tempo de espera progressivo entre falhas.

Mensagens Genéricas de Erro

- Sempre retornar "**Credenciais inválidas**" sem dizer se o erro foi no usuário ou senha.
- Isso dificulta user enumeration.

Armazenamento Seguro de Senhas

Senhas devem ser armazenadas com `password_hash()` e verificadas com `password_verify()`:

```
if (password_verify($inputPassword, $storedHash)) {  
    // login success  
}
```

Proteção CSRF / JWT Seguro

- Se API → usar JWT com **exp**, **iat**, **aud** e **iss**.
- Se formulário web → habilitar CSRF Token no CodeIgniter.

Transport Layer Security

- Forçar HTTPS em todas as rotas de autenticação.

Remover Código Repetido

- A parte final duplicada deve ser eliminada para evitar falhas de manutenção.

3.4 db.sql

https://github.com/lucideus-repo/UnSAFE_Bank/blob/master/Backend/mysql/db.sql

```
LOCK TABLES `bank_master` WRITE;
/*!40000 ALTER TABLE `bank_master` DISABLE KEYS */;
INSERT INTO `bank_master` VALUES (1,'IFSC00001','HDFC Bank','Delhi'),(2,'IFSC00002','State Bank of India','Mumbai'),
/*!40000 ALTER TABLE `bank_master` ENABLE KEYS */;
UNLOCK TABLES;
```

1. Exposição de Dados Sensíveis

- O script insere informações reais de bancos (nome, IFSC, localidade).
- Caso esse script seja armazenado em repositório público, backup mal gerenciado ou logs de aplicação, pode expor dados sensíveis ou estratégicos.

2. Injeção de SQL (SQL Injection) se for concatenado dinamicamente

- Se esse trecho for executado como parte de uma query montada por aplicação (ex: **string concatenation**), há risco de que um atacante injete comandos maliciosos (DROP, UPDATE, UNION, etc.).
- O problema não está no dump em si, mas em como ele seria usado pela aplicação.

3. Uso de **LOCK TABLES** / **DISABLE KEYS** sem controle

- Em ambiente de produção, pode causar DoS (negação de serviço) temporária:
 - **LOCK TABLES** impede outros usuários/processos de acessarem a tabela.
 - **DISABLE KEYS** suspende índices, podendo afetar consultas críticas.

4. Ausência de Controle de Acesso / Privilégios

- Se o script for executado por um usuário de banco com permissões elevadas (**SUPER**, **ALL PRIVILEGES**), aumenta a superfície de ataque.
- Um invasor que consiga rodar esse script pode inserir, modificar ou substituir dados sem restrição.

5. Falta de Validação/Integridade

- Não há verificação se os registros já existem (poderia gerar duplicatas).
- Se usado em ambiente real, pode comprometer a integridade referencial ou causar inconsistência.

Mitigações Propostas

Proteção de Dados Sensíveis

- Nunca versionar scripts de carga contendo informações reais em repositórios públicos.
- Utilizar dados fictícios ou mascarados em ambientes de teste/dev.

Prevenção contra SQL Injection

- Se for necessário inserir dados via aplicação, usar queries parametrizadas/prepared statements, nunca concatenar strings SQL.
- Exemplo em PHP (PDO):

```
$stmt = $pdo->prepare("INSERT INTO bank_master (id, ifsc, name, city) VALUES (?, ?, ?, ?)");
```

```
$stmt->execute([1, "IFSC00001", "HDFC Bank", "Delhi"]);
```

Controle de Locks e Performance

- Evitar **LOCK TABLES** em produção salvo em migrações controladas.
- Usar transações (**BEGIN / COMMIT**) para manter atomicidade sem bloquear toda a tabela.

Princípio do Menor Privilégio

- Criar um usuário do banco apenas com permissão de INSERT para rodar scripts de carga.
- Restringir privilégios como **LOCK TABLES**, **ALTER**, **DROP** a administradores.

Validação de Integridade

- Usar **INSERT IGNORE** ou **INSERT ... ON DUPLICATE KEY UPDATE** para evitar duplicação.
- Garantir que a tabela tenha chaves primárias e constraints bem definidas.

3.5 Dockerfile

https://github.com/lucideus-repo/UnSAFE_Bank/blob/master/Backend/mysql/Dockerfile

Vulnerabilidades identificadas

1. **Uso de versão específica e desatualizada (mysql:8.0.19)**
 - Essa versão é de **2020**, já sem suporte.
 - Possui vulnerabilidades conhecidas (CVE de alta gravidade, como DoS, privilege escalation e falhas de autenticação).
 - Quanto mais antiga a imagem, maior a superfície de ataque.
2. **Exposição de dados sensíveis no build**
 - O **db.sql** pode conter **usuários, senhas, chaves ou dados de negócio reais**.
 - Como o arquivo é copiado para a imagem, ele pode ser extraído por qualquer pessoa que tenha acesso à imagem.
 - Risco alto em repositórios públicos (GitHub, Docker Hub, etc.).
3. **Uso de ADD ao invés de COPY**
 - **ADD** tem funcionalidades extras (como extração automática de arquivos **.tar.gz** ou download de URLs remotas).

- Isso aumenta a superfície de ataque e pode gerar comportamentos inesperados.
- O ideal é **COPY** para arquivos locais.

4. Script de inicialização exposto em **/docker-entrypoint-initdb.d**

- Tudo nesse diretório roda automaticamente quando o container sobe pela primeira vez.
- Se o **db.sql** não for sanitizado, pode incluir comandos perigosos (**DROP**, **GRANT ALL PRIVILEGES**, etc.).
- Caso alguém altere o **db.sql** (supply chain attack), a inicialização pode comprometer o banco.

5. Privilégios excessivos do container

- Se o container for rodado com usuário **root** (default em muitas imagens), somado a scripts maliciosos no init, aumenta o risco de **privilege escalation** no host.

Mitigações Propostas

Atualizar a versão da imagem

- Usar sempre uma versão suportada e atualizada (ex.: **mysql:8.0** ou **mysql:8.4 LTS**).
- Melhor ainda: **mysql:latest** para acompanhar patches críticos de segurança.

Proteger dados sensíveis

- Nunca colocar senhas ou dados reais no **db.sql**.
- Usar **variáveis de ambiente** (ex.: **MYSQL_ROOT_PASSWORD**, **MYSQL_USER**, **MYSQL_PASSWORD**, **MYSQL_DATABASE**) em vez de hardcode no SQL.
- Para secrets sensíveis → usar **docker secrets** ou soluções como Vault.

Trocar **ADD** por **COPY**

COPY ./db.sql /docker-entrypoint-initdb.d/

- Mais seguro, previsível e recomendado para Dockerfiles.

Sanitizar scripts de inicialização

- Garantir que `db.sql` contenha apenas o esquema necessário (CREATE TABLES, índices, constraints).
- Não incluir `GRANT ALL PRIVILEGES`, nem usuários com superpoderes.

Rodar com usuário não-root

Adicionar no Dockerfile:

`USER mysql`

-
- Isso limita o impacto se alguém explorar falhas na imagem.

Segregar ambientes

- Usar `db.sql` com dados **fictícios** em DEV/TEST.
- Produção deve inicializar com migrações controladas (Flyway, Liquibase, etc.), não com dump estático.

3.6 jwt_helper.php

https://github.com/lucideus-repo/UnSAFE_Bank/blob/master/Backend/src/api/application/helpers/jwt_helper.php

```

public static function encode($payload, $key, $algo = 'HS256')
{
    $header = array('typ' => 'JWT', 'alg' => $algo);
    $segments = array();
    $segments[] = JWT::urlsafeB64Encode(JWT::jsonEncode($header));
    $segments[] = JWT::urlsafeB64Encode(JWT::jsonEncode($payload));
    $signing_input = implode('.', $segments);

    $signature = JWT::sign($signing_input, $key, $algo);
    $segments[] = JWT::urlsafeB64Encode($signature);

    return implode('.', $segments);
}

public static function sign($msg, $key, $method = 'HS256')
{
    $methods = array(
        'HS256' => 'sha256',
        'HS384' => 'sha384',
        'HS512' => 'sha512',
    );
    if (empty($methods[$method])) {
        throw new DomainException('Algorithm not supported');
    }
    return hash_hmac($methods[$method], $msg, $key, true);
}

```



```

public static function jsonDecode($input)
{
    $obj = json_decode($input);
    if (function_exists('json_last_error') && $errno = json_last_error()) {
        JWT::_handleJsonError($errno);
    } else if ($obj === null && $input !== 'null') {
        throw new DomainException('Null result with non-null input');
    }
    return $obj;
}

/**
 * Encode a PHP object into a JSON string.
 *
 * @param object|array $input A PHP object or array
 *
 * @return string            JSON representation of the PHP object or array
 * @throws DomainException Provided object could not be encoded to valid JSON
 */
public static function jsonEncode($input)
{
    $json = json_encode($input);
    if (function_exists('json_last_error') && $errno = json_last_error()) {
        JWT::_handleJsonError($errno);
    } else if ($json === 'null' && $input !== null) {
        throw new DomainException('Null result with non-null input');
    }
    return $json;
}

```

vulnerabilidades

1. Ausência de validação de algoritmo seguro

- O método `decode()` aceita qualquer algoritmo informado no cabeçalho do JWT (`$header->alg`).
- Isso abre a **JWT Algorithm Confusion Attack** (CWE-347):
 - Exemplo: se o servidor espera **RS256** (chave pública/privada), mas o token vem com **HS256**, o atacante pode gerar um token válido usando apenas a chave pública como se fosse secreta.
- Também não impede o uso de `"alg": "none"`, que em algumas versões antigas permitia **token sem assinatura**.

2. Uso de chave simétrica para tudo

- A implementação só suporta **HMAC (HS256, HS384, HS512)**.
- Isso é inseguro para cenários em que o emissor e o validador são diferentes (ex.: SSO, OpenID Connect).
- Falta suporte a **RS256, ES256**, que usam chave pública/privada e são padrão em ambientes modernos.

3. Falta de verificação de Claims do JWT

- O método `decode()` retorna o **payload cru**, mas não valida campos críticos como:
 - `exp` (expiração)
 - `nbf` (not before)
 - `iss` (issuer)
 - `aud` (audience)
- Sem essa validação, tokens **expirados ou forjados** ainda são aceitos.

Mitigações Propostas

1. Forçar algoritmo seguro e fixo

Definir o algoritmo esperado no `decode()` e rejeitar tokens com outro:

```
$expectedAlg = 'HS256';  
if ($header->alg !== $expectedAlg) {  
    throw new DomainException('Invalid algorithm');  
}
```

-
- Nunca aceitar `"alg": "none"`.

2. Adicionar suporte a algoritmos assimétricos (RS256, ES256)

- Usar **chaves públicas/privadas** em vez de segredos compartilhados.

- Isso evita que o validador precise conhecer a chave privada do emissor.

3. Validar Claims obrigatórios

Após decodificar, verificar:

```
if (isset($payload->exp) && time() >= $payload->exp) {  
    throw new UnexpectedValueException('Token expired');  
}  
if (isset($payload->nbf) && time() < $payload->nbf) {  
    throw new UnexpectedValueException('Token not yet valid');  
}  
if ($payload->iss !== 'meu-servidor') {  
    throw new UnexpectedValueException('Invalid issuer');  
}  
if ($payload->aud !== 'meu-cliente') {  
    throw new UnexpectedValueException('Invalid audience');  
}
```

Comparação de assinatura com `hash_equals`

```
if (!hash_equals($sig, JWT::sign("$headb64.$bodyb64", $key, $header->alg))) {  
    throw new UnexpectedValueException('Signature verification failed');  
}
```

4. Mitigar Replay Attacks

- a. Usar **tokens curtos** (ex.: **expiração em 15 minutos**).
- b. Implementar **refresh tokens** para renovar credenciais.
- c. Opcional: manter **lista de revogação** (blacklist) em banco de dados/cache.

6. Atualizar dependência

- Substituir esse código por uma versão mantida do pacote `firebase/php-jwt`.
- Ele já trata `alg:none`, suporta múltiplos algoritmos, valida claims e usa `hash_equals`.

4- VULNERABILIDADES ENCONTRADAS

4.1 XSS (Cross-Site Scripting)

- **Descrição:** `$config['global_xss_filtering'] = FALSE` permite injeção de scripts em views.
- **Referências:** CWE-79, CVSS 6.1, CVE-2020-11022/11023.
- **Mitigação:** uso de `html_escape()`, sanitização com `$this->security->xss_clean()`, CSP, cookies seguros.

4.2 CSRF (Cross-Site Request Forgery)

- **Descrição:** Proteção CSRF desativada.
- **Referências:** CWE-352, CVSS 8.0, CVE-2018-1000119.
- **Mitigação:** ativar tokens CSRF, double-submit cookie, checar Origin/Referer.

4.3 Exposição de Credenciais

- **Descrição:** Credenciais em texto plano em *database.php*.
- **Referências:** CWE-798, CVSS 9.8, CVE-2019-9193.
- **Mitigação:** uso de variáveis de ambiente, rotação de senhas, privilégios mínimos.

4.4 Transmissão sem Criptografia

- **Descrição:** Login e DB podem trafegar sem TLS.
- **Referências:** CWE-319, CVSS 7.4, CVE-2016-2183.
- **Mitigação:** forçar HTTPS, habilitar SSL no MySQL.

4.5 JWT Inseguro

- **Descrição:** Algoritmo não fixado → JWT Algorithm Confusion.
- **Referências:** CWE-347, CVSS 9.1, CVE-2015-9235.
- **Mitigação:** fixar algoritmo (HS256/RS256), validar claims, libs atualizadas.

4.6 Uso de MD5 para Senhas

- **Descrição:** Senhas armazenadas com `md5()`.
- **Referências:** CWE-327, CVSS 9.1, CVE-2012-3287.
- **Mitigação:** uso de `password_hash()/password_verify()` (bcrypt/Argon2).

4.7 Dockerfile Desatualizado

- **Descrição:** Uso de *mysql:8.0.19* com CVEs conhecidas.
- **Referências:** CWE-1104, CVSS 7.5, CVE-2020-2574/2579.
- **Mitigação:** atualizar para versão LTS, evitar `db.sql` sensível, rodar como não-root.

5 CONCLUSÃO E PRÓXIMOS PASSOS

O sistema **UnSAFE Bank** apresenta vulnerabilidades críticas que podem comprometer toda a aplicação e os dados sensíveis.

Impactos Principais

- XSS: sequestro de sessão, exfiltração de dados, ações não autorizadas.
- CSRF: execução de ações críticas sem consentimento do usuário.
- Exposição de credenciais: acesso não autorizado ao banco.
- JWT inseguro: quebra de autenticação.
- MD5: senhas vulneráveis a ataques de dicionário.

5.1 Mitigações Propostas

Categoria	Mitigação	Objetivo	Exemplo (curto)
XSS	Escape na saída (principal)	Evitar execução de scripts ao renderizar dados do usuário	<code><?= htmlspecialchars(\$var) ?></code>
XSS	Dados em JS via <code>json_encode</code> com flags	Preservar contexto JS com segurança	<code><?= json_encode(\$user, JSON_HEX_TAG</code>
XSS	Validação/normalização no servidor	Reduzir payloads maliciosos	<code>form_validation</code> com <code>required</code>
XSS	Sanitização pontual	Camada extra (não substitui escape)	<code>\$this->input->post('campo', TRUE) / \$this->security->xss_clean(\$v)</code>
XSS	Cookies <code>Secure</code> , <code>HttpOnly</code> , <code>SameSite</code>	Diminuir impacto se houver XSS	<code>cookie_secure=TRUE; cookie_httponly=TRUE; cookie_samesite='Strict'</code>
XSS	CSP (Content-Security-Policy)	Restringir fontes de script	<code>default-src 'self'; script-src 'self' (via header)</code>

CSRF	Token CSRF em formulários (Web)	Impedir submissões forjadas	Habilitar CSRF no CI; campo hidden com token
CSRF (API)	Double-submit cookie + checar Origin/Referer + POST + JSON	Proteger endpoints sem sessão HTML	Exigir Content-Type: application/json e validar Origin/Referer
CSRF	Rate limiting e logging	Reduzir tentativas automatizadas	5–10 req/min por IP/usuário; logs de falha
SQLi	Queries parametrizadas / binds	Evitar injeção	<code>\$this->db->where('id', \$id);</code> <code>\$this->db->get('tabela');</code>
SQLi	Proibir concatenação de parâmetros em SQL	Eliminar vetores de injeção	❌ "... WHERE id=\$id" → ✅ usar <code>?/Active Record</code>
SQLi	Validar tipo/tamanho	Hardening dos parâmetros	<code>is_numeric(\$id)</code> ; limites de <code>max_length</code>

5.2 Plano de Ação

- **Imediato (0–7 dias):** ativar CSRF, revisar views com `html_escape()`, rotacionar credenciais, configurar cookies seguros.
- **Curto Prazo (até 15 dias):** migrar armazenamento de senhas para bcrypt/Argon2, corrigir JWT, forçar HTTPS/TLS.
- **Médio Prazo (até 30 dias):** atualizar Docker, implementar rate limiting e auditoria, adotar CSP e segregar ambientes.

6. REFERÊNCIAS

- CWE - Common Weakness Enumeration. MITRE.
- CVE Details Database.
- OWASP Secure Coding Practices.
- Documentação oficial CodeIgniter.
- NIST SP 800-63 (Digital Identity Guideli