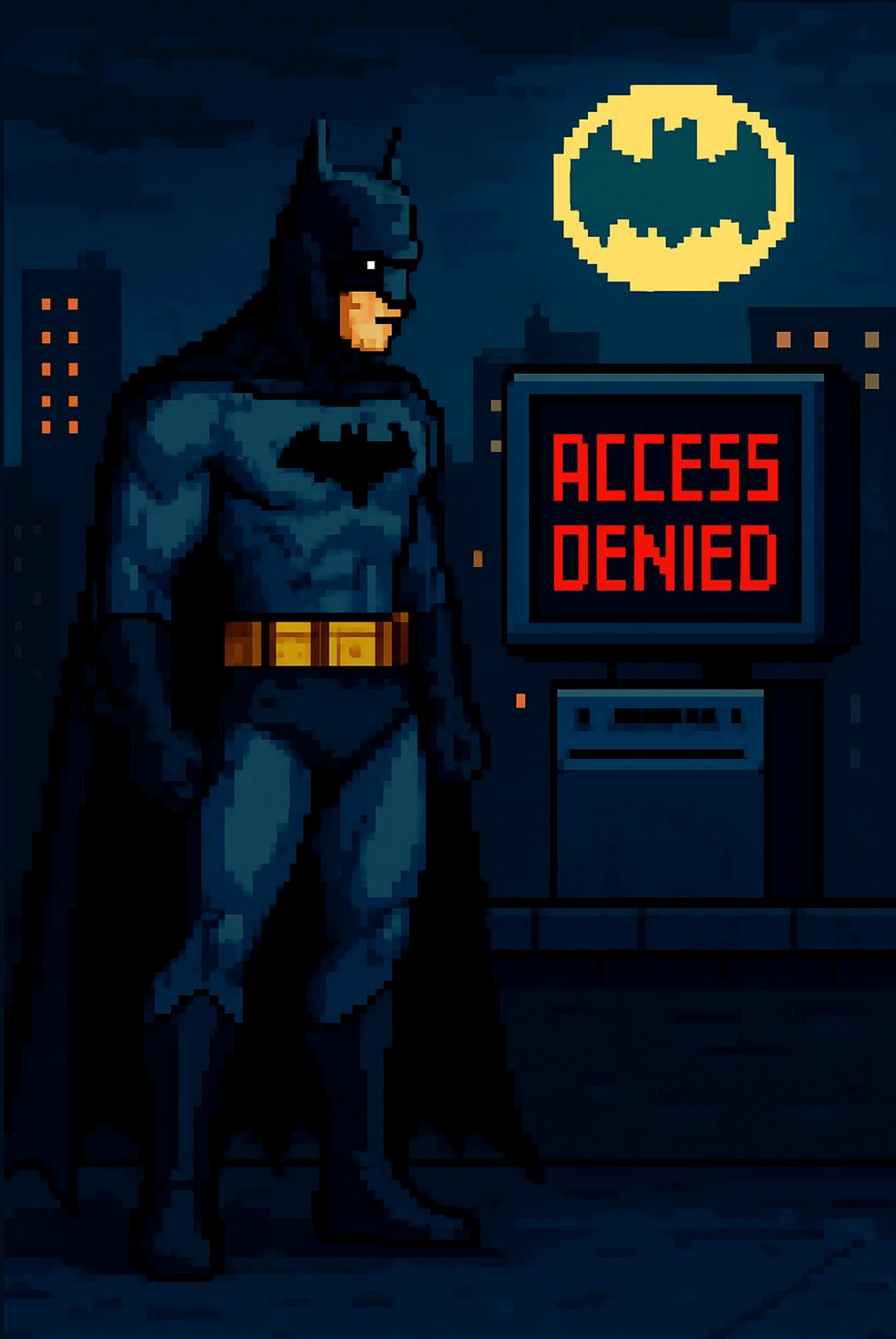


CÓDIGO SOMBRIO

LIÇÕES DO CAVALEIRO DA CODIFICAÇÃO CONTRA FALHAS NA AUTENTICAÇÃO



Luiz Vitorio Casagrande

01

O QUE SÃO FALHAS DE AUTENTICAÇÃO?

O que são falhas de autenticação?



Falhas de autenticação ocorrem quando um sistema não implementa corretamente os mecanismos que comprovam a identidade de um usuário. Se a autenticação for fraca, mal projetada ou vulnerável, um invasor pode assumir a identidade de outros usuários, inclusive administradores.

Impacto das falhas de autenticação:

- Acesso não autorizado a dados sensíveis.
- Sequestro de contas (Account Takeover).
- Escalonamento de privilégios.

Causas comuns:

- Implementação incorreta de login, tokens e sessões.
- Armazenamento e gerenciamento inseguro de senhas e tokens.
- Falta de proteção contra brute-force e credenciais vazadas.
- Uso de algoritmos de hash ou geração de tokens obsoletos.

02

SQL INJECTION EM FORMULÁRIOS – POR QUE WHITELIST AJUDA (MAS NÃO RESOLVE)

O que é SQL Injection?



SQL Injection ocorre quando um invasor consegue injetar comandos SQL maliciosos nos campos de entrada da aplicação, manipulando consultas ao banco de dados para obter, alterar ou excluir dados.

O que é Whitelist?

É uma técnica de validação que permite apenas valores previamente autorizados. Funciona bem quando o conjunto de valores válidos é conhecido e limitado. Não elimina SQL Injection por completo, mas reduz significativamente o risco em alguns contextos.

Exemplos práticos

Inseguro — Banco Relacional (MySQL)

```
php
$username = $_POST['username'];
$query = "SELECT * FROM users WHERE username = '$username'";
$result = mysqli_query($conn, $query);
```

Há interação direta entre o campo e a consulta ao banco de dados, ocasionando a vulnerabilidade.

Seguro — Com Whitelist



php

```
$allowed_users = ['admin', 'user1', 'user2'];  
  
if (!in_array($_POST['username'], $allowed_users)) {  
    die("Usuário inválido.");  
}  
  
$query = "SELECT * FROM users WHERE username = '$_POST['username']'";
```

Observação: A whitelist é útil para campos com um conjunto limitado (status, roles, categorias), mas não deve ser a única proteção para dados livres como login.

Inseguro — Banco Não Relacional (MongoDB)

javascript

```
const username = req.body.username;  
db.collection('users').find({ username: username });
```

Seguro — Com Whitelist

javascript

```
const allowedUsers = ["admin", "user1", "user2"];  
if (!allowedUsers.includes(req.body.username)) {  
    throw "Usuário inválido";  
}  
db.collection('users').find({ username: req.body.username });
```

Limitações da Whitelist

- Não funciona para campos dinâmicos como senhas ou e-mails.
- É eficaz apenas quando os valores válidos são conhecidos e restritos.

03

PROTEÇÃO REAL: BIND PARAMETERS EM AÇÃO

O que são Bind Parameters ?

Bind Parameters são placeholders na query SQL, onde os dados são enviados separadamente da instrução SQL, impedindo que o input do usuário seja interpretado como código SQL. É a proteção mais eficaz contra injeções SQL em bancos relacionais.

Exemplos práticos



Inseguro — Banco Relacional (MySQLi)

```
php

$conn = new mysqli("localhost", "usuario", "senha", "meubanco");

$username = $_GET['username']; // input direto do usuário
$query = "SELECT * FROM usuarios WHERE nome = '$username'";

$result = $conn->query($query);

while ($row = $result->fetch_assoc()) {
    echo $row['nome'];
}
```

Esse código concatena diretamente a entrada do usuário (\$username) dentro da string da query SQL. Isso significa que qualquer conteúdo que o usuário fornecer será interpretado literalmente como parte da instrução SQL. Falta de uso de prepared statements com bind parameters, que separariam os dados da estrutura da query, impedindo que a entrada do usuário seja executada como código SQL.

Inseguro — Banco Não Relacional (MongoDB)

```
php

$username = $_GET['username'];
$filter = ['nome' => $username];
```

Em PHP, se um parâmetro GET é enviado com colchetes (ex: username[\$ne]=), o valor será interpretado como um array associativo, não como uma string.

Seguro — Banco Relacional (MySQLi)

```
php
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $_POST['username']);
$stmt->execute();
$result = $stmt->get_result();
```

Seguro — Banco Não Relacional (MongoDB)

```
javascript
const username = req.body.username;
db.collection('users').findOne({ username: username });
```

MongoDB usa queries baseadas em objetos JSON, o que oferece proteção nativa contra SQL Injection, mas ainda é vulnerável a NoSQL Injection se operadores (\$ne, \$gt, \$or) não forem tratados. A manipulação de objetos JSON via requests pode permitir bypass completo de autenticação, quando não validado. É importante implementar validação de tipo (username, por exemplo, precisa ser string), e sanitizar contra operadores no input, de modo que não sejam tratados como objetos.

Erro Comum: Concatenar diretamente:

```
php
$query = "SELECT * FROM users WHERE username = '$username'";
```

04

RECOVER PASSWORD TOKEN

Cenário

Quando um usuário esquece sua senha, o sistema gera um token de recuperação e envia por e-mail. Este token permite redefinir a senha. A falha ocorre quando o token:

- São baseados em dados previsíveis (e-mail + timestamp).
- Não tem tempo de expiração.
- Não há invalidação após uso.
- Armazenamento inseguro.
- URL exposta sem HTTPS.



Exemplos práticos

Inseguro — PHP

```
php
$token = md5($email . time());
```


Um invasor pode gerar tokens se souber o e-mail.

Seguro — PHP

```
php
$token = bin2hex(random_bytes(32));
$tokenHash = password_hash($token, PASSWORD_DEFAULT);
```

- Aleatório, criptograficamente seguro e armazenado como hash.
- Definir validade (ex.: 15 minutos).
- Invalidar após o primeiro uso.


Inseguro — Java



```
public String generateRecoveryToken(String email) {
    String token = email + System.currentTimeMillis();
    tokenStorage.put(token, email);
    return token;
}
```

Um invasor pode prever ou fazer um brute-force em cima do horário em que o token foi gerado. O email é facilmente conhecido ou adivinhado. A concatenação com tempo previsível resulta em um token fácil de reproduzir.

Seguro — Java



```
public String generateRecoveryToken() {
    SecureRandom secureRandom = new SecureRandom();
    byte[] tokenBytes = new byte[32];
    secureRandom.nextBytes(tokenBytes);
    return Base64.getUrlEncoder().withoutPadding().encodeToString(tokenBytes);
}
```

Uso de SecureRandom gera números aleatórios com forte entropia criptográfica, ideal para segurança. “new byte[32]” gera 256 bits de entropia, adequado para tokens de autenticação ou recuperação.

05

FALHAS NA CRIAÇÃO DE UM TOKEN

Cenário

Tokens são gerados para autenticar usuários ou autorizar ações (como redefinir senha, acesso temporário etc). Quando gerados de forma previsível ou fraca, um invasor pode adivinhar ou forjar esses tokens. Falhas comuns:

- Uso de dados previsíveis (ID do usuário, timestamp).
- Algoritmos fracos (MD5, SHA-1).
- Comprimento pequeno do token.
- Geração não criptográfica.



Exemplos práticos

Inseguro — PHP

```
php
$token = sha1($userId . time());
```

Pode ser estimado com brute-force.

Seguro — PHP

```
php
$token = bin2hex(random_bytes(64));
```

Alternativas seguras:

- UUID v4 (ramsey/uuid no PHP).
- JWT com assinatura forte (HMAC SHA256 ou RSA).
- Use JWT apenas quando for necessário transportar informações no próprio token (auth claims). Para tokens de redefinição de senha, tokens opacos aleatórios armazenados no backend costumam ser mais seguros.

Inseguro — Java

```
Java

// Geração baseada em dados previsíveis
String token = user.getUsername() + System.currentTimeMillis();
token = Integer.toHexString(token.hashCode()); // inseguro
```



Seguro — Java

```
Java

import java.security.SecureRandom;
import java.util.Base64;

SecureRandom random = new SecureRandom();
byte[] bytes = new byte[32];
random.nextBytes(bytes);
String token = Base64.getUrlEncoder().withoutPadding().encodeToString(bytes);
```

Uso de biblioteca SecureRandom, comprimento forte e não determinístico.

06

EXPIRATION TIME TOKEN

Cenário

Tokens sem expiração representam risco, pois, se forem expostos, podem ser usados indefinidamente. Mesmo tokens longos e bem gerados precisam ter validade para mitigar esse risco. Falhas comuns:

- Não usar campo de expiração.
- Aceitar tokens expirados.
- Ignorar diferença de fuso horário ou clock skew.



Exemplos práticos

Inseguro — PHP

```
php

// Geração do token (sem expiração)
$token = bin2hex(random_bytes(16));
$_SESSION['recovery_token'] = $token;
// Validação do token (nunca expira)
if ($_GET['token'] === $_SESSION['recovery_token']) {
    echo "Token válido!";
}
```

Seguro — PHP

```
php

// Geração do token com expiração
$token = bin2hex(random_bytes(16));
$_SESSION['recovery_token'] = $token;
$_SESSION['recovery_token_expire'] = time() + 900; // 15 minutos
// Validação do token com expiração
if (
    isset($_GET['token'], $_SESSION['recovery_token'], $_SESSION['recovery_token_expire']) &&
    $_GET['token'] === $_SESSION['recovery_token'] &&
    time() <= $_SESSION['recovery_token_expire']
) {
    echo "Token válido!";
} else {
    echo "Token inválido ou expirado.";
}
```

Inseguro — Java

```
Java   
  
// Token sem expiração  
String token = generateToken(); // Aleatório, sem validade associada
```

Uma vez gerado, não há controle posterior de validade.

Seguro — Java

```
Java  
  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.SignatureAlgorithm;  
Date now = new Date();  
Date exp = new Date(now.getTime() + 15 * 60 * 1000); // 15 min  
String jwt = Jwts.builder()  
    .setSubject(userId.toString())  
    .setIssuedAt(now)  
    .setExpiration(exp)  
    .signWith(SignatureAlgorithm.HS256, secretKey.getBytes())  
    .compact();
```

- Backend deve validar exp ao processar o JWT, vale destacar que a simples presença do campo exp no token não significa que ele será validado automaticamente.
- Bibliotecas modernas (como System.IdentityModel.Tokens.Jwt, jjwt) geralmente realizam a verificação automática somente se configurado corretamente.
- Tokens expirados devem ser tratados explicitamente como inválidos.

07

INSECURE TOKEN HASH

Cenário

Tokens e senhas armazenados no banco de dados não devem ser guardados em texto plano, nem com algoritmos inseguros. O ideal é usar algoritmos de hash com salt e resistência à força bruta. Falhas comuns:

- Uso de MD5 ou SHA-1, ou ausência de salt.
- Hashes sem custo computacional (facilitam brute-force).
- Armazenamento de token em texto puro.



Exemplos práticos

Inseguro — PHP

```
php
$token = md5(random_bytes(16));
```

MD5 pode ser quebrado.

Seguro — PHP

```
php
$token = bin2hex(random_bytes(32));
```

Ou hash de senha:

```
php
$passwordHash = password_hash($password, PASSWORD_ARGON2ID);
```

Inseguro — Java



Java



```
import java.security.MessageDigest;

String token = "123456";
MessageDigest md = MessageDigest.getInstance("MD5"); // inseguro
byte[] hash = md.digest(token.getBytes());
```

MD5 pode ser quebrado.

Seguro — Java



Java

```
import org.mindrot.jbcrypt.BCrypt;

String token = "valorDoToken";
String hash = BCrypt.hashpw(token, BCrypt.gensalt());

// Validação:
boolean valid = BCrypt.checkpw(token, hash);
```

Uso de BCrypt é seguro para:

- Tokens sensíveis (redefinição de senha).
- Senhas de login.

Conclusão: A Jornada do Cavaleiro da Codificação



No combate às ameaças digitais, a autenticação segura é a primeira linha de defesa — e, como vimos ao longo deste eBook, é também uma das mais frequentemente negligenciadas. Quando sistemas falham em proteger adequadamente os mecanismos que comprovam a identidade dos usuários, as consequências podem ser devastadoras: invasões, sequestros de contas, roubo de dados e escalonamento de privilégios. Assim como o Cavaleiro das Trevas se prepara meticulosamente para qualquer cenário, o desenvolvedor consciente deve prever e neutralizar cada ponto fraco da autenticação:

- Validar entradas com rigor, especialmente em campos usados em queries SQL ou NoSQL.
- Utilizar Bind Parameters e evitar concatenar strings em instruções sensíveis.
- Gerar tokens criptograficamente seguros, com tempo de expiração bem definido e invalidação após o uso.
- Nunca armazenar senhas ou tokens em texto plano — prefira algoritmos como bcrypt.
- Rejeitar práticas obsoletas: MD5, SHA-1, GetHashCode, tokens previsíveis ou sem controle.

A segurança começa com a conscientização. Cada linha de código é uma decisão entre a proteção e a exposição. Ao adotar boas práticas de autenticação, você fortalece as defesas do seu sistema, protege seus usuários e impede que vilões explorem brechas invisíveis.

