Segurança e Armazenamento de Credenciais

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

Segurança em autenticação não é apenas importante, é absolutamente crítica. Um vazamento de dados ou vulnerabilidade pode comprometer todos os usuários do seu aplicativo, destruir a confiança conquistada, e potencialmente expor sua empresa a consequências legais sérias. Nesta seção, vamos explorar os princípios e práticas que garantem que os dados de autenticação dos seus usuários estejam protegidos adequadamente.

O que torna segurança particularmente desafiadora é que você precisa acertar em todas as camadas. Uma única falha em qualquer ponto da cadeia pode comprometer todo o sistema. Por isso, vamos adotar uma abordagem de defesa em profundidade, onde múltiplas camadas de proteção trabalham juntas para criar um sistema verdadeiramente seguro.

É importante entender que segurança não é um destino, mas um processo contínuo. Novas vulnerabilidades são descobertas, novos ataques são desenvolvidos, e as melhores práticas evoluem. O que vamos aprender aqui representa o estado atual do conhecimento, mas você deve manter-se atualizado e sempre questionar se suas implementações seguem as práticas mais recentes.

A Regra de Ouro: Nunca Armazene Senhas

Esta é uma regra absoluta, sem exceções, que você deve gravar profundamente: NUNCA armazene a senha do usuário no dispositivo. Não em texto plano, não criptografada, não com hash, não de forma alguma. A senha deve existir no aplicativo apenas durante o tempo necessário para enviá-la ao servidor durante o login, e deve ser descartada imediatamente após.

Vamos entender por que esta regra é tão importante e por que até mesmo abordagens aparentemente seguras são problemáticas.

Por Que Não Armazenar em Texto Plano

Esta deveria ser óbvia, mas vale enfatizar. Se você armazena a senha em texto plano em qualquer lugar do sistema de arquivos do dispositivo, seja em SharedPreferences, em um arquivo de configuração, ou em um banco de dados local, você está criando uma vulnerabilidade crítica. Qualquer pessoa com acesso ao dispositivo, qualquer aplicativo malicioso com permissões adequadas, ou qualquer atacante que consiga fazer backup dos dados do app pode ler essas senhas diretamente.

Considere que dispositivos com root no Android ou jailbreak no iOS removem muitas das proteções do sistema operacional. Um atacante com acesso físico ao dispositivo pode extrair todos os arquivos do seu aplicativo e ler qualquer dado armazenado em texto plano. Isto não é um cenário teórico, acontece regularmente.

Por Que Não Criptografar Localmente

Alguns desenvolvedores pensam que podem contornar o problema criptografando a senha antes de armazená-la. Isto parece razoável inicialmente, mas cria um problema fundamental: se você criptografa a senha, precisa armazenar a chave de criptografia em algum lugar para poder decriptografar quando necessário. Mas onde você armazena esta chave?

Se você coloca a chave no código do aplicativo, um atacante que descompila seu app pode extrair essa chave e usá-la para decriptografar todas as senhas. Se você deriva a chave de algo único ao dispositivo, como o ID do hardware, ainda há o problema que um atacante com acesso ao dispositivo pode fazer o mesmo processo para derivar a chave. Você essencialmente criou segurança por obscuridade, que não é verdadeira segurança.

Além disso, há o problema de sincronização. Se o usuário muda a senha em outro dispositivo ou na web, o dispositivo com a senha antiga armazenada fica desincronizado. Você teria que implementar mecanismos complexos para invalidar senhas armazenadas quando elas mudam externamente, adicionando mais complexidade e mais pontos de falha.

A Solução Correta: Tokens de Sessão

Ao invés de armazenar a senha, armazenamos tokens de sessão que o servidor emite após validar as credenciais. Tokens têm propriedades importantes que os tornam muito mais seguros que senhas. Primeiro, eles expiram automaticamente após um período definido, limitando a janela de oportunidade se forem comprometidos. Segundo, podem ser revogados pelo servidor a qualquer momento, algo impossível com senhas armazenadas localmente. Terceiro, não revelam a senha original, então mesmo se um token vazar, o atacante não obtém a senha que o usuário pode estar usando em outros serviços.

A senha flui apenas uma vez do aplicativo para o servidor através de uma conexão HTTPS criptografada durante o login. O servidor valida, gera um token, e retorna este token ao cliente. O cliente armazena apenas o token, nunca a senha. Em requisições subsequentes, o cliente envia o token para provar sua identidade. Se precisar fazer login novamente, o usuário digita a senha novamente, mas ela nunca é armazenada.

Este padrão é universalmente aceito e testado em milhões de aplicações. Não tente inventar algo diferente pensando que pode ser mais seguro ou conveniente. Siga o padrão estabelecido.

Compreendendo JSON Web Tokens (JWT)

JSON Web Token, ou JWT, é o padrão de facto para tokens de acesso em aplicações modernas. Entender como JWTs funcionam e suas propriedades é essencial para trabalhar com autenticação de forma segura e eficaz.

Anatomia de um JWT

Um JWT é uma string que parece aleatória mas na verdade tem estrutura específica. Ele consiste em três partes separadas por pontos: header, payload e signature. Vamos examinar cada parte para entender o que ela contém e por quê.

O header é um objeto JSON que especifica o tipo de token e o algoritmo de assinatura usado. Um header típico se parece com isto quando decodificado:

```json
{
 "alg": "HS256",
 "typ": "JWT"
}
```

Este header indica que o token é um JWT assinado usando o algoritmo HMAC-SHA256. O header é codificado em base64 e forma a primeira parte do token.

O payload é outro objeto JSON que contém as claims, que são declarações sobre o usuário e o token em si. Claims padrão incluem subject que identifica o usuário, issued at que registra quando o token foi criado, e expiration time que define quando ele expira. Você também pode adicionar claims customizadas com informações específicas da sua aplicação:

```json
{
 "sub": "user-123-abc",
 "email": "usuario@exemplo.com",
 "name": "João Silva",
 "role": "user",
 "iat": 1699900800,
 "exp": 1699987200
}
```

Este payload é também codificado em base64 e forma a segunda parte do token. Note que qualquer um que tenha o token pode decodificar o payload e ler essas informações, pois base64 é codificação, não criptografia. Por isso, nunca coloque informações verdadeiramente sensíveis como senhas ou números de cartão de crédito em um JWT.

A signature é criada pegando o header codificado e o payload codificado, concatenando-os com um ponto, e então aplicando o algoritmo de assinatura usando uma chave secreta que só o servidor conhece. Esta assinatura é o que torna o JWT seguro, pois permite ao servidor verificar que o token não foi adulterado.

Quando um cliente envia um JWT ao servidor, o servidor refaz o processo de assinatura usando a mesma chave secreta. Se a assinatura calculada coincide com a assinatura no token, o servidor sabe que o token é legítimo e não foi modificado. Se alguém tentar alterar o payload para dar a si mesmo permissões de administrador, por exemplo, a assinatura não vai mais validar e o servidor rejeitará o token.

Propriedades de Segurança dos JWTs

JWTs têm várias propriedades que os tornam adequados para autenticação em aplicações modernas. Primeiro, eles são stateless, significando que o servidor não precisa manter um registro de tokens emitidos em um banco

de dados. Toda informação necessária para validar o token está contida nele mesmo. Isto permite escalar horizontalmente seu backend facilmente, pois qualquer servidor pode validar qualquer token independentemente de qual servidor o emitiu originalmente.

Segundo, eles são autocontidos, carregando informações sobre o usuário dentro do próprio token. Isto significa que o servidor pode identificar o usuário e suas permissões sem consultar o banco de dados para cada requisição, melhorando performance.

Terceiro, eles têm expiração embutida. Cada JWT carrega sua própria data de expiração, após a qual ele automaticamente se torna inválido. Isto limita o dano se um token for roubado, pois ele só funciona até expirar.

Limitações e Considerações

Apesar de suas vantagens, JWTs também têm limitações importantes que você precisa entender. Uma vez emitido, um JWT é válido até expirar, a menos que você implemente uma lista de revogação no servidor. Isto significa que se você muda as permissões de um usuário ou o suspende, tokens JWT já emitidos continuam válidos até expirarem. Por isso, é importante usar tempos de expiração curtos para tokens de acesso.

Também é crucial entender que JWT não é criptografia. O payload pode ser lido por qualquer um que tenha o token. A assinatura apenas garante integridade, não confidencialidade. Se você precisa transmitir informações verdadeiramente confidenciais, criptografe-as antes de colocar no JWT, ou melhor ainda, não as coloque no JWT.

Armazenamento Seguro com Flutter Secure Storage

Uma vez que recebemos um token do servidor após login bem-sucedido, precisamos armazená-lo de forma segura no dispositivo para que o usuário não precise fazer login novamente toda vez que abre o aplicativo. O Flutter fornece o pacote flutter_secure_storage especificamente para este propósito.

Por Que Não Usar SharedPreferences

SharedPreferences é uma opção tentadora para armazenar dados persistentes no Flutter porque é simples de usar e familiar para muitos desenvolvedores. Porém, SharedPreferences armazena dados em texto plano em arquivos que podem ser acessados por outros aplicativos em dispositivos com root ou jailbreak, ou através de backups do dispositivo. Para dados sensíveis como tokens de autenticação, isto é inaceitável.

Um atacante que obtém acesso aos arquivos de SharedPreferences pode simplesmente ler o token e usá-lo para se autenticar como aquele usuário. Não há proteção alguma. SharedPreferences é apropriado para preferências do usuário como tema escolhido ou idioma preferido, mas nunca para dados de autenticação.

Como Flutter Secure Storage Funciona

Flutter Secure Storage usa mecanismos específicos de cada plataforma projetados para armazenar informações sensíveis de forma segura. No iOS, ele usa o Keychain, que é um serviço do sistema operacional que armazena dados criptografados e os protege com políticas de acesso rigorosas. No Android, ele usa o KeyStore, que oferece proteção similar através de criptografia baseada em hardware quando disponível.

Quando você salva um valor usando flutter_secure_storage, a biblioteca automaticamente criptografa o dado usando chaves gerenciadas pelo sistema operacional. Estas chaves nunca saem do ambiente seguro do sistema e são protegidas por mecanismos de hardware quando o dispositivo suporta. Mesmo com acesso root ao dispositivo, é significativamente mais difícil extrair dados do Keychain ou KeyStore comparado a arquivos de texto plano.

Além disso, dados armazenados através de flutter_secure_storage são isolados por aplicativo. Outros aplicativos no mesmo dispositivo não podem acessar os dados do seu app, criando uma camada adicional de proteção.

Implementação Prática

Usar flutter_secure_storage é direto. Primeiro, adicione a dependência ao seu pubspec.yaml:

```yaml
dependencies:
  flutter_secure_storage: ^9.0.0
```

Então, crie uma instância do storage e use-a para salvar, recuperar e deletar dados:

```dart
```

```dart
import 'package:flutter_secure_storage/flutter_secure_storage.dart';

class SecureStorageService {
  // Cria instância do secure storage com opções específicas
  final _storage = FlutterSecureStorage(
    aOptions: AndroidOptions(
      encryptedSharedPreferences: true,
    ),
    iOptions: IOSOptions(
      accessibility: KeychainAccessibility.first_unlock,
    ),
  );

  // Chaves para diferentes tipos de dados
  static const _keyAccessToken = 'access_token';
  static const _keyRefreshToken = 'refresh_token';
  static const _keyUserId = 'user_id';

  // Salva tokens após login
  Future<void> saveTokens({
    required String accessToken,
    required String refreshToken,
  }) async {
    await Future.wait([
      _storage.write(key: _keyAccessToken, value: accessToken),
      _storage.write(key: _keyRefreshToken, value: refreshToken),
    ]);
  }

  // Recupera access token
  Future<String?> getAccessToken() async {
    return await _storage.read(key: _keyAccessToken);
  }

  // Recupera refresh token
  Future<String?> getRefreshToken() async {
    return await _storage.read(key: _keyRefreshToken);
  }

  // Limpa todos os dados de autenticação (logout)
  Future<void> clearAll() async {
    await _storage.deleteAll();
  }

  // Verifica se há tokens salvos
  Future<bool> hasTokens() async {
```

```
    final accessToken = await getAccessToken();
    final refreshToken = await getRefreshToken();
    return accessToken != null && refreshToken != null;
  }
}
```

Esta implementação encapsula toda lógica de armazenamento seguro em um único service. Note como usamos constantes para as chaves ao invés de strings literais espalhadas pelo código. Isto previne erros de digitação e facilita mudanças futuras.

As opções passadas ao construtor do FlutterSecureStorage customizam o comportamento em cada plataforma. No Android, encryptedSharedPreferences garante uma camada adicional de proteção. No iOS, a opção de acessibilidade define quando os dados podem ser acessados, neste caso após o primeiro unlock do dispositivo.

Considerações de Performance

Operações no secure storage são mais lentas que leitura de SharedPreferences normal porque envolvem criptografia e descriptografia. Por isso, evite fazer múltiplas leituras do mesmo valor. Leia uma vez quando o aplicativo inicia, guarde em memória se necessário, e reutilize. Não leia em cada build de um widget, por exemplo.

Se você precisa armazenar múltiplos valores relacionados, considere serializá-los em JSON e armazenar como uma única entrada ao invés de múltiplas entradas separadas. Isto reduz o número de operações de criptografia necessárias:

dart

```
// Múltiplas operações (mais lento)
await _storage.write(key: 'user_id', value: userId);
await _storage.write(key: 'user_email', value: email);
await _storage.write(key: 'user_name', value: name);

// Uma operação (mais rápido)
final userData = jsonEncode({
  'id': userId,
  'email': email,
  'name': name,
});
await _storage.write(key: 'user_data', value: userData);
```

Gestão de Tokens: Access e Refresh

Um sistema robusto de autenticação baseada em tokens tipicamente usa dois tipos de tokens trabalhando em conjunto: access tokens de curta duração e refresh tokens de longa duração. Entender como estes tokens

interagem é crucial para implementar autenticação que equilibra segurança com experiência do usuário.

Por Que Dois Tokens

O problema que estamos tentando resolver é um trade-off fundamental. Se usarmos apenas um token de longa duração, simplificamos a implementação e o usuário permanece logado indefinidamente sem fricção. Porém, isto cria um risco de segurança significativo. Se este token for roubado, talvez através de um aplicativo malicioso que consegue ler o armazenamento do seu app em um dispositivo comprometido, o atacante tem acesso permanente à conta até que o usuário explicitamente faça logout ou mude sua senha.

Por outro lado, se usarmos apenas um token de curta duração, digamos que expira em quinze minutos, minimizamos a janela de vulnerabilidade se um token for roubado. Porém, isto forçaria o usuário a fazer login novamente a cada quinze minutos, criando uma experiência terrível.

A solução elegante é usar dois tokens com propósitos diferentes. O access token é de curta duração, tipicamente expirando em uma hora ou menos. Este é o token que você inclui em cada requisição para endpoints protegidos da API. Se for roubado, o dano é limitado porque expira rapidamente.

O refresh token é de longa duração, podendo durar dias, semanas ou até meses. Sua única finalidade é obter novos access tokens quando eles expiram. Você não usa o refresh token para acessar endpoints protegidos, apenas para o endpoint específico de renovação de token. Isto significa que o refresh token é transmitido pela rede muito menos frequentemente, reduzindo as oportunidades de interceptação.

Fluxo de Uso de Tokens

Vamos traçar o fluxo completo de como estes tokens funcionam em prática para entender como as peças se encaixam.

Quando o usuário faz login pela primeira vez, ele envia email e senha ao servidor. O servidor valida as credenciais e, se corretas, gera dois tokens: um access token com expiração curta e um refresh token com expiração longa. Ambos são retornados ao cliente e salvos em secure storage.

Quando o cliente precisa fazer uma requisição para um endpoint protegido, ele inclui o access token no header Authorization. O servidor valida o token, verifica que não expirou, e processa a requisição. Isto acontece perfeitamente enquanto o access token está válido.

Eventualmente, o access token expira. Na próxima requisição, o servidor retorna um erro 401 indicando que o token expirou. O cliente detecta este erro e automaticamente usa o refresh token para solicitar um novo access token do endpoint de renovação. O servidor valida o refresh token, verifica que ainda é válido, e emite um novo access token. O cliente salva este novo access token e tenta novamente a requisição original, que agora funciona com o token renovado.

Este processo de renovação é invisível para o usuário. Do ponto de vista dele, ele simplesmente continua usando o aplicativo sem interrupções. Apenas se o refresh token também expirar ou for revogado é que o usuário precisa fazer login novamente.

Implementando Renovação Automática

A renovação de tokens deve acontecer automaticamente nos bastidores sem envolver o usuário ou interromper a experiência. Vamos implementar um interceptor que detecta quando tokens expiram e os renova transparentemente:

dart

```dart
import 'package:dio/dio.dart';

class AuthInterceptor extends Interceptor {
  final Dio _dio;
  final SecureStorageService _storage;

  AuthInterceptor(this._dio, this._storage);

  @override
  Future<void> onRequest(
    RequestOptions options,
    RequestInterceptorHandler handler,
  ) async {
    // Adiciona access token em todas as requisições
    final accessToken = await _storage.getAccessToken();
    if (accessToken != null) {
      options.headers['Authorization'] = 'Bearer $accessToken';
    }
    handler.next(options);
  }

  @override
  Future<void> onError(
    DioException err,
    ErrorInterceptorHandler handler,
  ) async {
    // Se erro é 401 e temos refresh token, tenta renovar
    if (err.response?.statusCode == 401) {
      final refreshToken = await _storage.getRefreshToken();

      if (refreshToken != null) {
        try {
          // Tenta renovar o access token
          final newTokens = await _refreshAccessToken(refreshToken);

          // Salva novos tokens
          await _storage.saveTokens(
            accessToken: newTokens['access_token'],
            refreshToken: newTokens['refresh_token'],
          );

          // Tenta novamente a requisição original com novo token
          final options = err.requestOptions;
          options.headers['Authorization'] =
            'Bearer ${newTokens['access_token']}';
```

```
          final response = await _dio.fetch(options);
          return handler.resolve(response);

        } catch (refreshError) {
          // Renovação falhou, usuário precisa fazer login
          await _storage.clearAll();
          // Emitir evento de sessão expirada para UI
          return handler.reject(err);
        }
      }
    }

    handler.next(err);
  }

  Future<Map<String, dynamic>> _refreshAccessToken(
    String refreshToken,
  ) async {
    final response = await _dio.post(
      '/auth/refresh',
      data: {'refresh_token': refreshToken},
    );
    return response.data;
  }
}
```

Este interceptor funciona de forma transparente. Ele automaticamente adiciona o access token em todas as requisições, detecta quando o servidor retorna 401 indicando token expirado, tenta renovar usando o refresh token, e retenta a requisição original com o novo token. Se a renovação falhar, limpa os tokens salvos e propaga o erro para que a UI possa redirecionar para login.

Revogação de Tokens

Um aspecto importante da segurança de tokens é a capacidade de revogá-los quando necessário. Quando um usuário faz logout, você deve invalidar o refresh token no servidor para que não possa mais ser usado para obter novos access tokens. Isto previne que alguém que obteve o refresh token continue acessando a conta após o logout.

dart

```dart
Future<void> logout() async {
  final refreshToken = await _storage.getRefreshToken();

  if (refreshToken != null) {
    try {
      // Informa o servidor para revogar o refresh token
      await _dio.post(
        '/auth/revoke',
        data: {'refresh_token': refreshToken},
      );
    } catch (e) {
      // Mesmo se a revogação falhar, continua com logout local
      print('Erro ao revogar token: $e');
    }
  }

  // Limpa tokens localmente
  await _storage.clearAll();

  // Atualiza estado de autenticação
  // ... navega para tela de login
}
```

HTTPS: A Base da Segurança em Trânsito

Toda discussão sobre segurança de autenticação assume que a comunicação entre cliente e servidor é criptografada. HTTPS não é opcional ou algo para implementar depois. É absolutamente fundamental e deve estar presente desde o primeiro dia de desenvolvimento.

Por Que HTTPS é Crítico

Quando você faz uma requisição HTTP normal sem criptografia, todos os dados trafegam em texto plano pela internet. Isto inclui credenciais de login, tokens de acesso, e qualquer outra informação que você envia ou recebe. Qualquer pessoa em qualquer ponto entre seu dispositivo e o servidor pode interceptar e ler esses dados.

Considere um usuário conectado ao WiFi público de um café. Sem HTTPS, um atacante na mesma rede pode usar ferramentas simples para capturar todo tráfego de rede e extrair credenciais, tokens e dados pessoais. Este tipo de ataque, conhecido como man-in-the-middle, é trivialmente fácil de executar em redes não confiáveis.

HTTPS resolve isto criptografando toda comunicação entre cliente e servidor usando TLS (Transport Layer Security). Mesmo que alguém intercepte o tráfego, tudo que vêem são dados criptografados que são computacionalmente impraticáveis de decifrar sem a chave privada do servidor.

Implementando HTTPS no Flutter

No Flutter, usar HTTPS é tão simples quanto usar HTTP. Você apenas muda o esquema da URL de http:// para https://. Os pacotes de networking como dio e http automaticamente lidam com toda complexidade do TLS nos bastidores:

```dart
// ❌ NUNCA faça isto
final response = await dio.post(
  'http://api.exemplo.com/auth/login',
  data: credentials,
);

// ✅ SEMPRE use HTTPS
final response = await dio.post(
  'https://api.exemplo.com/auth/login',
  data: credentials,
);
```

A maior parte do trabalho de HTTPS acontece no lado do servidor, onde você precisa obter um certificado SSL/TLS válido e configurar seu servidor web para usá-lo. Serviços de backend modernos como Supabase, Firebase e outros já fornecem HTTPS por padrão, então você não precisa se preocupar com configuração de certificados.

Validação de Certificados

O Flutter automaticamente valida certificados SSL para garantir que você está realmente conectando ao servidor correto e não a um impostor. Por padrão, ele só aceita certificados assinados por autoridades certificadoras reconhecidas.

Nunca desabilite a validação de certificados em produção, mesmo que facilite desenvolvimento ou resolva algum problema temporário. Isto deixa seus usuários completamente vulneráveis a ataques man-in-the-middle:

```dart
// ❌ NUNCA faça isto em produção
(_dio.httpClientAdapter as DefaultHttpClientAdapter).onHttpClientCreate =
  (client) {
    client.badCertificateCallback = (cert, host, port) => true;
    return client;
  };
```

Se você precisa usar certificados auto-assinados durante desenvolvimento local, use essa configuração apenas em builds de debug e certifique-se de que está completamente desabilitada em builds de release.

Validação no Backend: A Última Linha de Defesa

Um princípio fundamental de segurança é nunca confiar em dados vindos do cliente. Toda validação que você faz no Flutter é para melhorar a experiência do usuário mostrando erros rapidamente, mas não é segurança real. A validação verdadeira sempre acontece no servidor.

Por Que Validação Cliente-Side Não é Suficiente

Um usuário malicioso pode modificar o código do seu aplicativo, remover todas as validações, e enviar qualquer dado que quiser diretamente para sua API. Ele pode usar ferramentas como Postman ou curl para fazer requisições HTTP sem sequer usar seu aplicativo. Se você confia apenas em validação cliente-side, você está completamente vulnerável a esses ataques.

Considere uma validação de comprimento mínimo de senha. No Flutter, você verifica que a senha tem pelo menos oito caracteres antes de permitir envio. Mas se um atacante faz uma requisição POST direta para seu endpoint de cadastro com uma senha de três caracteres, o que acontece? Se o backend não valida também, a conta é criada com senha fraca, violando suas políticas de segurança.

Ou imagine que você valida no cliente que um usuário não pode deletar recursos de outros usuários. Um atacante pode modificar seu app ou fazer requisições diretas tentando deletar recursos com IDs arbitrários. Se o backend não verifica que o usuário autenticado realmente possui aquele recurso, você tem uma vulnerabilidade grave.

Implementando Validação Redundante

A abordagem correta é validar tanto no cliente quanto no servidor, mas com propósitos diferentes. No cliente, validação melhora UX dando feedback imediato. No servidor, validação garante segurança real e integridade dos dados.

Isto significa que você essencialmente duplica lógica de validação. Por exemplo, se você exige que senhas tenham pelo menos oito caracteres com letra e número, esta regra deve estar tanto no código Flutter quanto no código backend:

```dart
```

```
// Flutter (validação de UX)
String? validatePassword(String? value) {
  if (value == null || value.isEmpty) {
    return 'Por favor, informe uma senha';
  }
  if (value.length < 8) {
    return 'A senha deve ter no mínimo 8 caracteres';
  }
  if (!value.contains(RegExp(r'[a-zA-Z]'))) {
    return 'A senha deve conter pelo menos uma letra';
  }
  if (!value.contains(RegExp(r'[0-9]'))) {
    return 'A senha deve conter pelo menos um número';
  }
  return null;
}
```

javascript

```
// Backend (validação de segurança)
function validatePassword(password) {
  if (!password || password.length < 8) {
    throw new Error('Senha deve ter no mínimo 8 caracteres');
  }
  if (!/[a-zA-Z]/.test(password)) {
    throw new Error('Senha deve conter pelo menos uma letra');
  }
  if (!/[0-9]/.test(password)) {
    throw new Error('Senha deve conter pelo menos um número');
  }
}
```

Esta duplicação é necessária e não viola o princípio DRY (Don't Repeat Yourself) porque são camadas diferentes com propósitos diferentes. O cliente não pode acessar o código do servidor e vice-versa, então não há como compartilhar a lógica mesmo que quiséssemos.

Autorização no Backend

Além de validar formato de dados, o backend deve sempre verificar autorização. Isto significa checar que o usuário autenticado tem permissão para realizar a ação solicitada.

Por exemplo, se um endpoint permite atualizar perfil de usuário, o backend deve verificar que o token JWT pertence ao mesmo usuário cujo perfil está sendo atualizado. Não confie que o cliente só enviará requisições válidas. Um atacante pode tentar enviar requisição para atualizar o perfil de outro usuário.

javascript

```javascript
// Backend - exemplo de verificação de autorização
async function updateUserProfile(req, res) {
  // Token foi validado e informações do usuário extraídas
  const authenticatedUserId = req.user.id;
  const profileUserId = req.params.userId;

  // Verifica se está tentando atualizar próprio perfil
  if (authenticatedUserId !== profileUserId) {
    return res.status(403).json({
      error: 'Você não tem permissão para atualizar este perfil'
    });
  }

  // Prosseguir com atualização
  // ...
}
```

Esta verificação acontece no servidor e não pode ser contornada pelo cliente. Mesmo que um atacante modifique o app para tentar atualizar perfil de outro usuário, o backend rejeitará a requisição.

Rate Limiting e Proteção Contra Ataques

Além de garantir que dados são transmitidos e armazenados de forma segura, precisamos proteger contra ataques automatizados que tentam comprometer contas através de força bruta ou outras técnicas maliciosas.

Ataques de Força Bruta

Um ataque de força bruta contra login envolve um atacante tentando milhares ou milhões de combinações de senha até encontrar a correta. Sem proteções, um atacante pode fazer centenas de tentativas por segundo, potencialmente comprometendo contas com senhas fracas em minutos ou horas.

A defesa primária contra isto é implementar rate limiting, que limita o número de tentativas de login que podem ser feitas em um período de tempo. Isto pode ser implementado tanto no cliente quanto no servidor, com o servidor sendo a proteção real.

No cliente, você pode desabilitar o botão de login por alguns segundos após uma tentativa falhada, forçando um delay entre tentativas. Isto previne que usuários façam cliques rápidos repetidos acidentalmente, mas não protege contra ataques automatizados já que um atacante pode contornar facilmente esta limitação.

dart

```dart
class LoginPage extends StatefulWidget {
  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  bool _isLoading = false;
  DateTime? _lastFailedAttempt;

  bool get _canAttemptLogin {
    if (_lastFailedAttempt == null) return true;

    final now = DateTime.now();
    final difference = now.difference(_lastFailedAttempt!);

    // Permite nova tentativa apenas após 3 segundos
    return difference.inSeconds >= 3;
  }

  Future<void> _handleLogin() async {
    if (!_canAttemptLogin) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('Aguarde alguns segundos antes de tentar novamente'),
        ),
      );
      return;
    }

    setState(() => _isLoading = true);

    try {
      await authService.login(
        _emailController.text,
        _passwordController.text,
      );
      // Login bem-sucedido
    } catch (e) {
      // Login falhou
      setState(() {
        _lastFailedAttempt = DateTime.now();
        _isLoading = false;
      });

      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Email ou senha incorretos')),
```

```
      );
    }
  }
}
```

No servidor, rate limiting é mais eficaz. Você pode rastrear tentativas de login por endereço IP ou por email e bloquear temporariamente após um número de falhas. Por exemplo, permitir cinco tentativas por endereço IP por minuto, e bloquear por quinze minutos após cinco falhas consecutivas para o mesmo email.

CAPTCHAs e Desafios

Para proteção adicional contra bots, considere implementar CAPTCHAs após múltiplas tentativas falhadas. Isto força o atacante a provar que é humano antes de continuar tentando. Serviços como Google reCAPTCHA podem ser integrados em aplicativos Flutter.

Porém, use CAPTCHAs com moderação. Eles prejudicam a experiência do usuário, especialmente em dispositivos móveis onde resolver um CAPTCHA é mais trabalhoso. Uma boa estratégia é mostrar CAPTCHA apenas após detectar comportamento suspeito como múltiplas tentativas falhadas ou padrões de tráfego automatizado.

Princípio do Menor Privilégio

Cada token deve ter apenas as permissões mínimas necessárias para realizar sua função. Isto limita o dano potencial se um token for comprometido.

Por exemplo, um token usado para recuperação de senha deveria apenas permitir redefinir senha, nada mais. Não deveria permitir fazer login, atualizar perfil, ou acessar dados sensíveis. Se este token específico vazar, o atacante pode no máximo redefinir a senha, o que o usuário pode reverter.

Da mesma forma, se seu aplicativo tem múltiplos níveis de usuários como administradores e usuários comuns, os tokens devem refletir essas permissões. Um token de usuário comum não deveria permitir operações administrativas mesmo que o atacante tente.

Implemente isto usando claims nos JWTs que especificam permissões ou roles do usuário:

json

```
{
  "sub": "user-123",
  "email": "usuario@exemplo.com",
  "role": "user",
  "permissions": ["read:own_profile", "update:own_profile"],
  "iat": 1699900800,
  "exp": 1699904400
}
```

O servidor então verifica estas claims antes de processar operações sensíveis, garantindo que o token tem a permissão necessária.

Conclusão da Seção

Nesta seção, exploramos os aspectos fundamentais de segurança em autenticação. Aprendemos por que nunca devemos armazenar senhas localmente, como JWTs funcionam e suas propriedades de segurança, como usar flutter_secure_storage para proteger tokens, a importância de gestão adequada de access e refresh tokens, por que HTTPS é não-negociável, e porque validação no backend é a verdadeira linha de defesa.

Segurança não é um recurso que você adiciona, é um requisito fundamental que deve estar presente em cada decisão de design e implementação. Cada camada de proteção que implementamos adiciona profundidade à nossa defesa, criando um sistema onde uma única falha não compromete tudo.

Na próxima seção, vamos aplicar esses princípios de segurança enquanto implementamos validação de formulários robusta e eficaz, garantindo que dados estão corretos antes mesmo de chegar ao servidor.

Fim da Seção 5: Segurança e Armazenamento de Credenciais