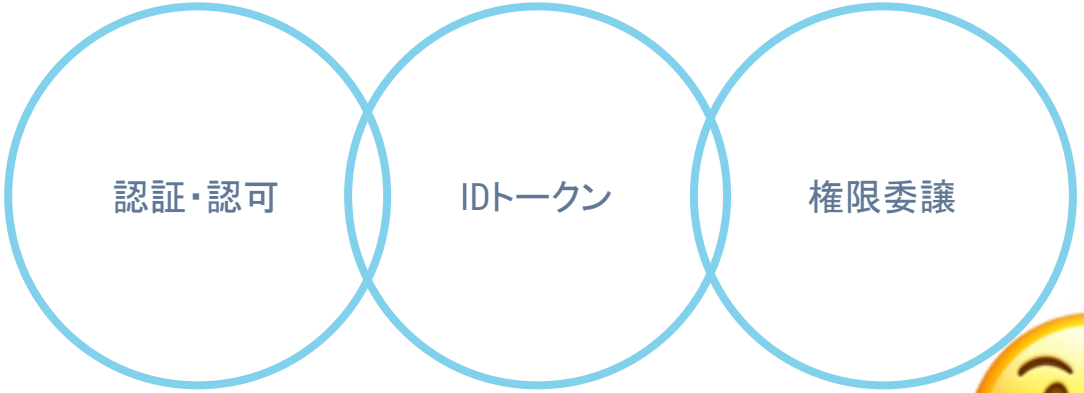
30分で OpenID Connect 完全に理解したと言えるようになる勉強会

OAuthとOAuth認証とOpenID Connect

- OAuthは権限移譲のプロトコル
- 「OAuth認証」はプロフィールAPIによる認証
- OpenID ConnectはIDトークンによる認証

いきなり言われてもわからん...



OAuthとOpenID Connectの関係

基礎となるOAuthを理解していないと
OpenID Connectを理解するのは難しい

お品書き

- そもそもOAuthってなに？
- OAuth認証ってなにが問題なの？
- OpenID ConnectはOAuthとなにが違うの？
- ネイティブアプリやSPAではどう使うべき？

今日しない話

- OAuthの歴史
- OAuth2.0の詳細
- IDトークンの具体的な検証方法

※注意点

わかりやすさを優先しているため、一部説明や前提を省略しています。

あらかじめご了承ください。

0.

認証・認可

そもそも認証・認可って？

認証（AuthN）

- 通信の相手が誰であるかを確認すること
- 個人の身元の確認

認可（AuthZ）

- リソースアクセス権限をサードパーティのアプリに移譲すること
- 誰かに許可を与えること

1.

OAuth

OAuthとは？

- 権限委譲のプロトコル
- 認可のプロトコル

OAuthのメリット

権限委譲

ユーザのリソースへのアクセス権限を
サードパーティのアプリに移譲することが
できる

安全性

サードパーティアプリにはリソースオー
ナーのID・パスワードを教える必要がない

OAuthのメリット

権限委譲

ユーザのリソースへのアクセス権限をサードパーティのアプリに移譲することができる

安全性

サードパーティアプリにはリソースオーナーのID・パスワードを教える必要がない

正直なに言ってるかわからん…


OAuthの流れを追ってみる

OAuthの登場人物

リソースオーナー

（ユーザー）

クライアント

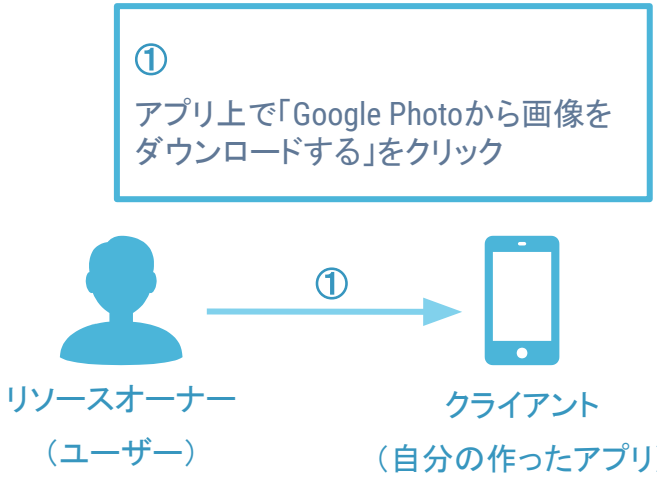
（自分の作ったアプリ）

認可サーバー

（Google OAuth）

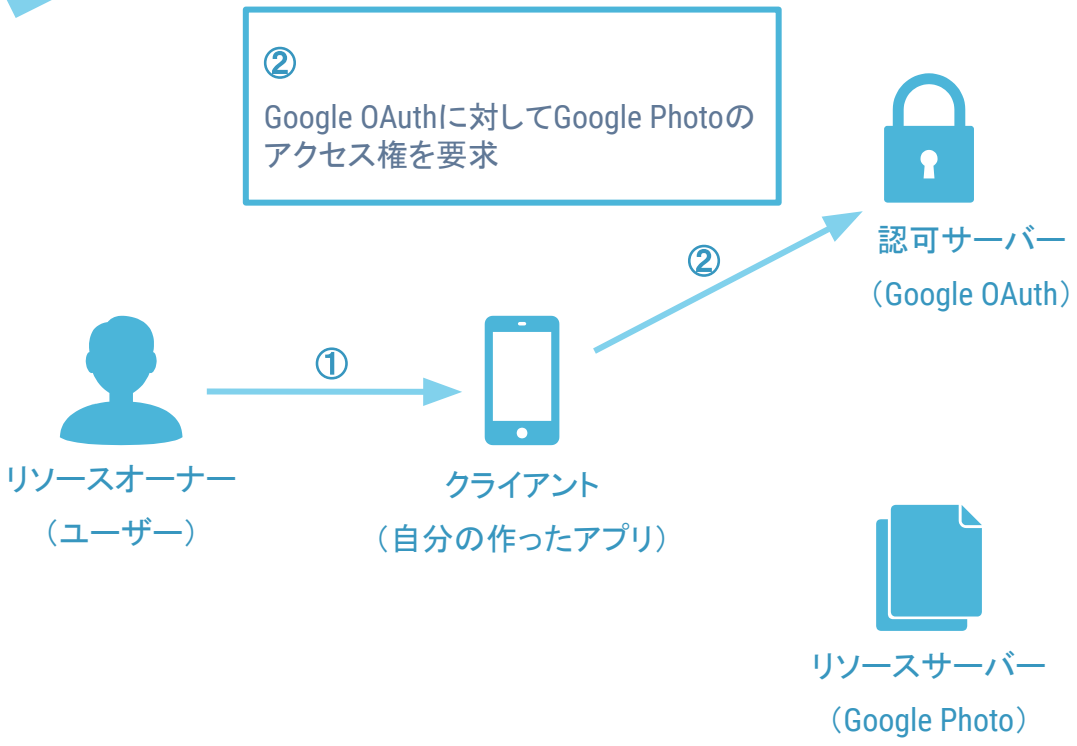
リソースサーバー

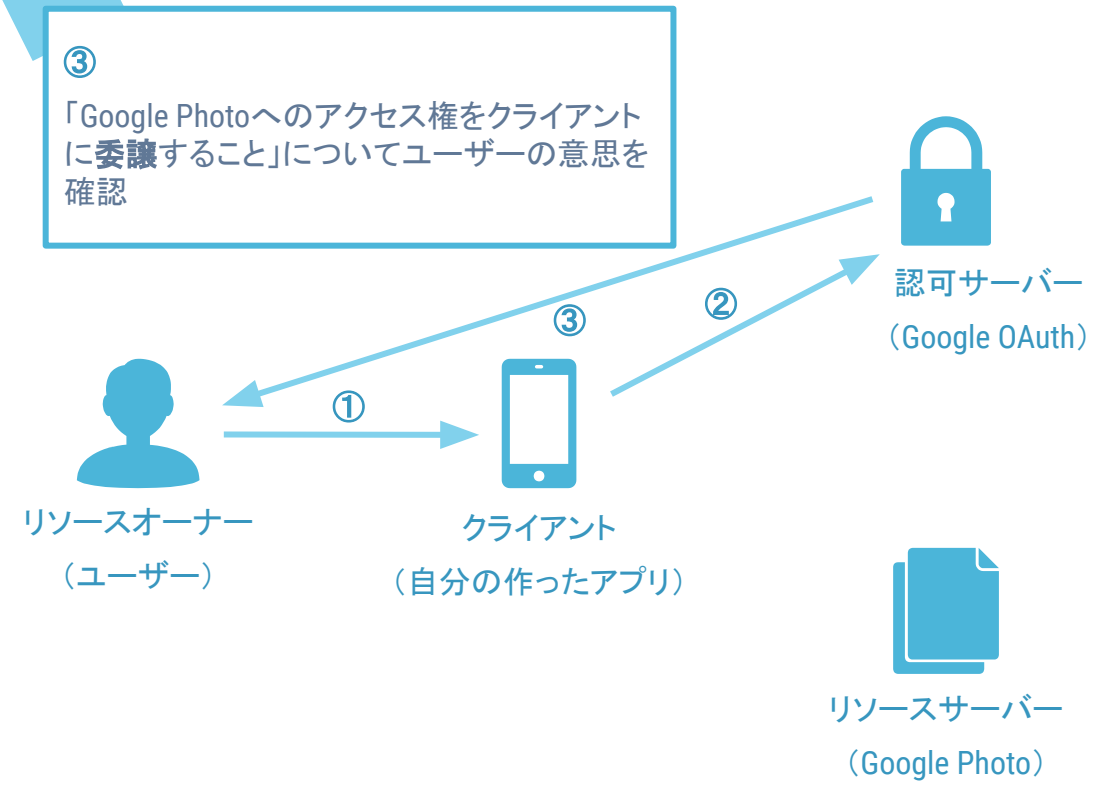
（Google Photo）

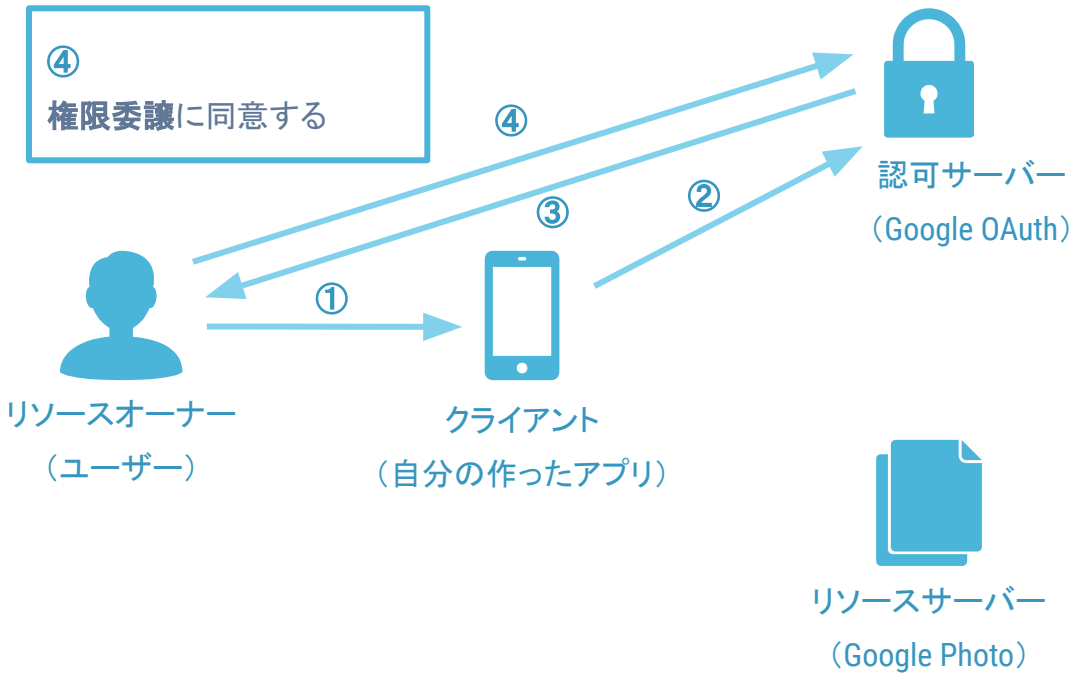


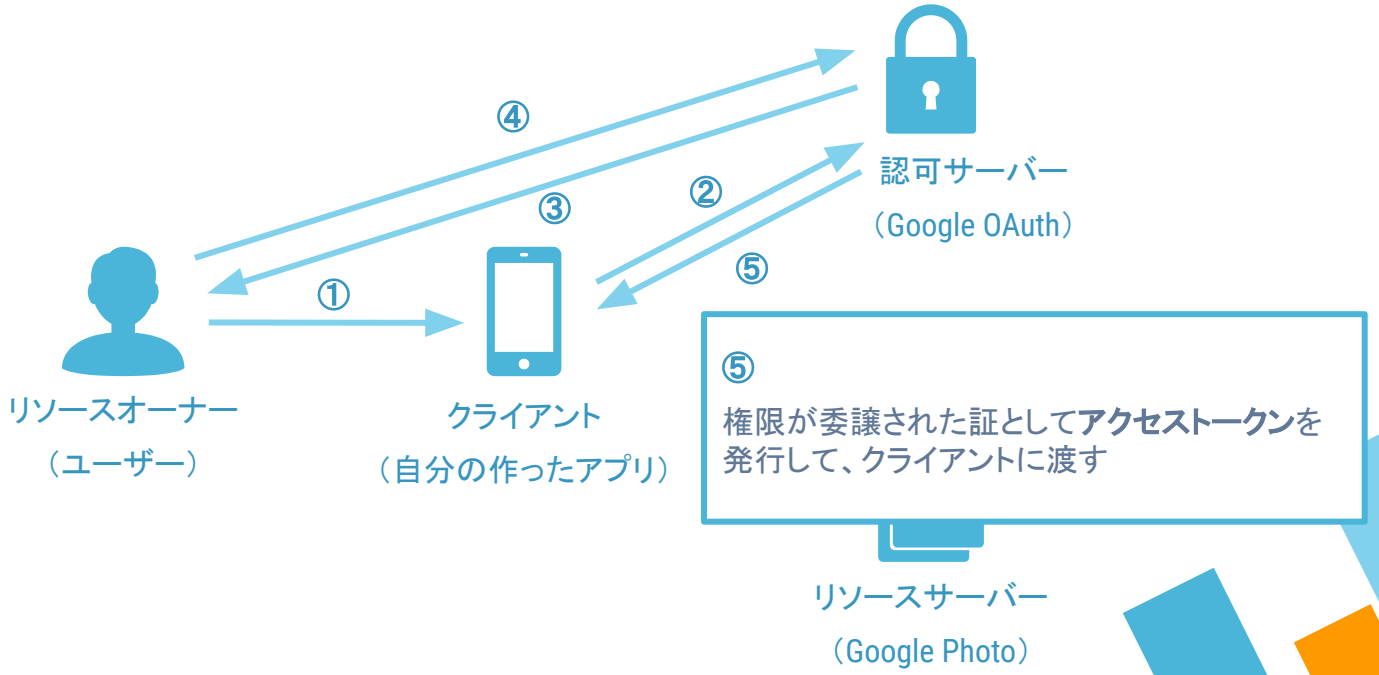
認可サーバー
（Google OAuth）

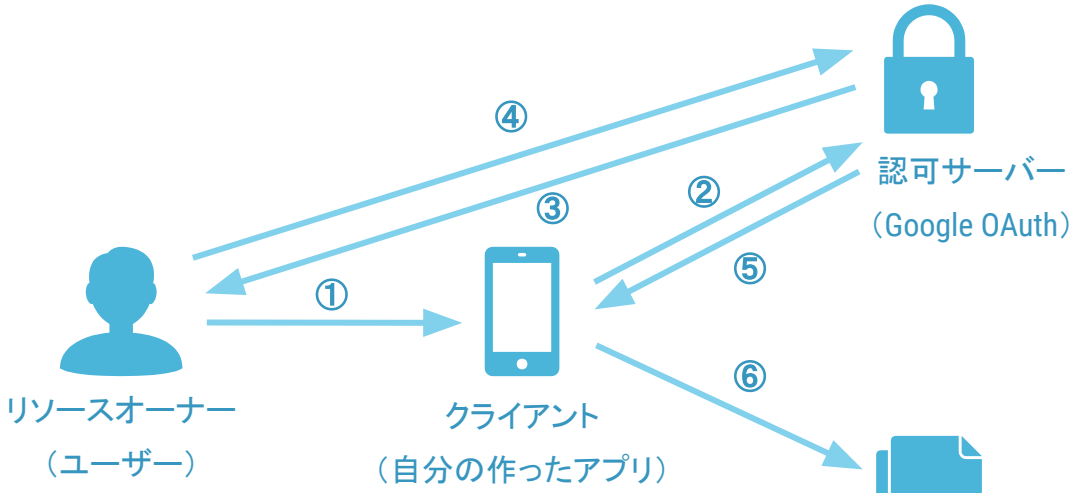
リソースサーバー
（Google Photo）


②
Google OAuthに対してGoogle Photoの
アクセス権を要求
認可サーバー
（Google OAuth）
②
①
リソースオーナー
（ユーザー）
クライアント
（自分の作ったアプリ）
リソースサーバー
（Google Photo）


③
「Google Photoへのアクセス権をクライアントに委譲すること」についてユーザーの意思を確認
認可サーバー
（Google OAuth）
③
②
①
リソースオーナー
（ユーザー）
クライアント
（自分の作ったアプリ）
リソースサーバー
（Google Photo）

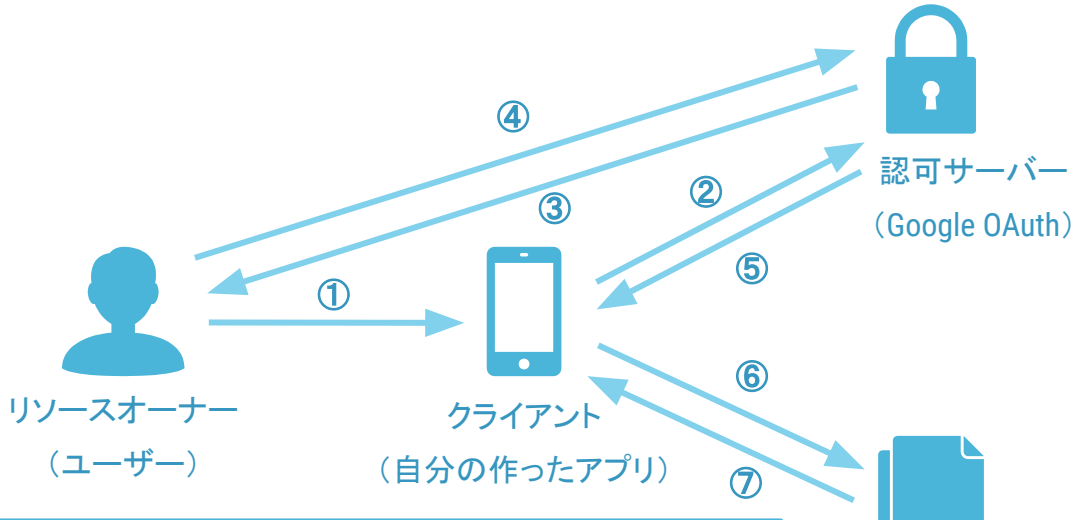

④
権限委譲に同意する
④
③
②
①
認可サーバー
（Google OAuth）
リソースオーナー
（ユーザー）
クライアント
（自分の作ったアプリ）
リソースサーバー
（Google Photo）


④
③
②
①
⑤
認可サーバー
（Google OAuth）
リソースオーナー
（ユーザー）
クライアント
（自分の作ったアプリ）
⑤
権限が委譲された証として**アクセストークン**を
発行して、クライアントに渡す
リソースサーバー
（Google Photo）



⑥

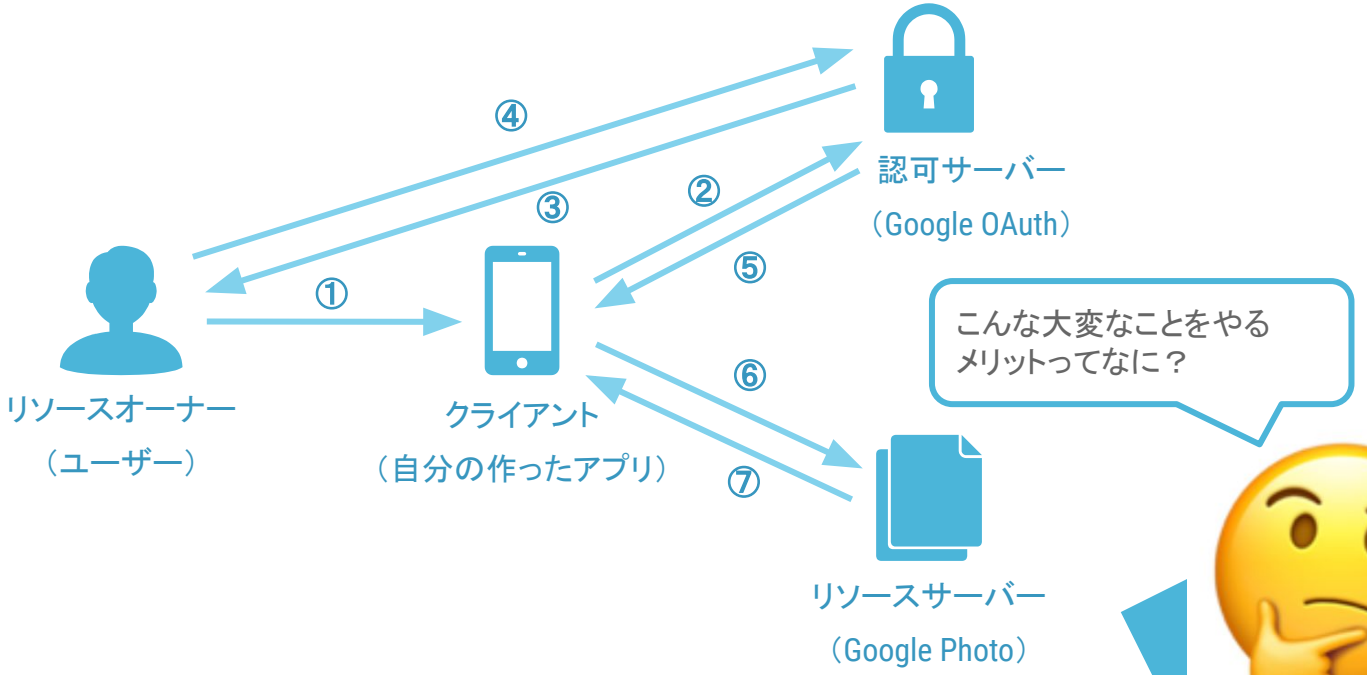
クライアントはアクセストークンをもってGoogle PhotoのAPIにアクセスし、ユーザーの画像を要求

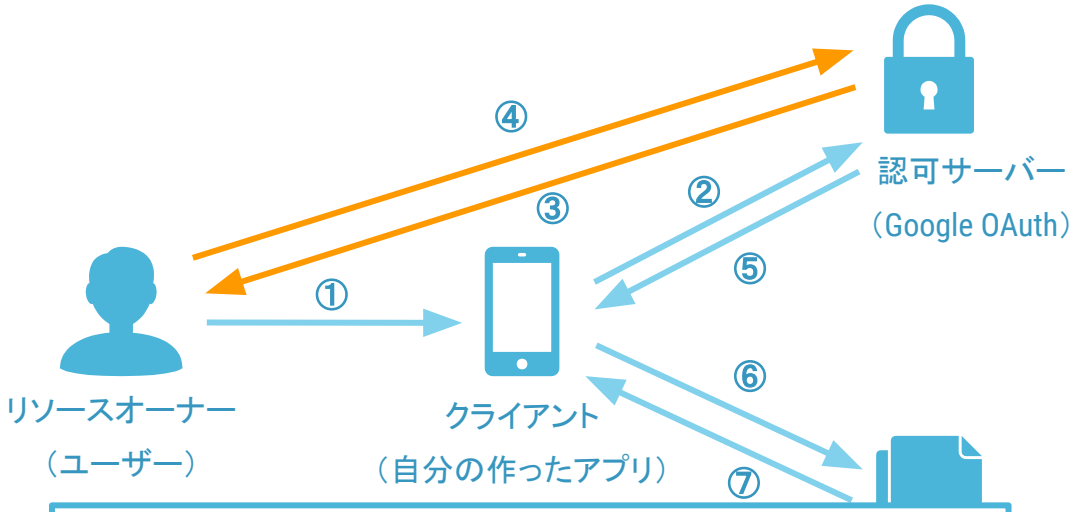


⑦

Google Photoはアクセストークンの有効性と紐づく権限を確認

問題がなければユーザーの画像をクライアントに渡す

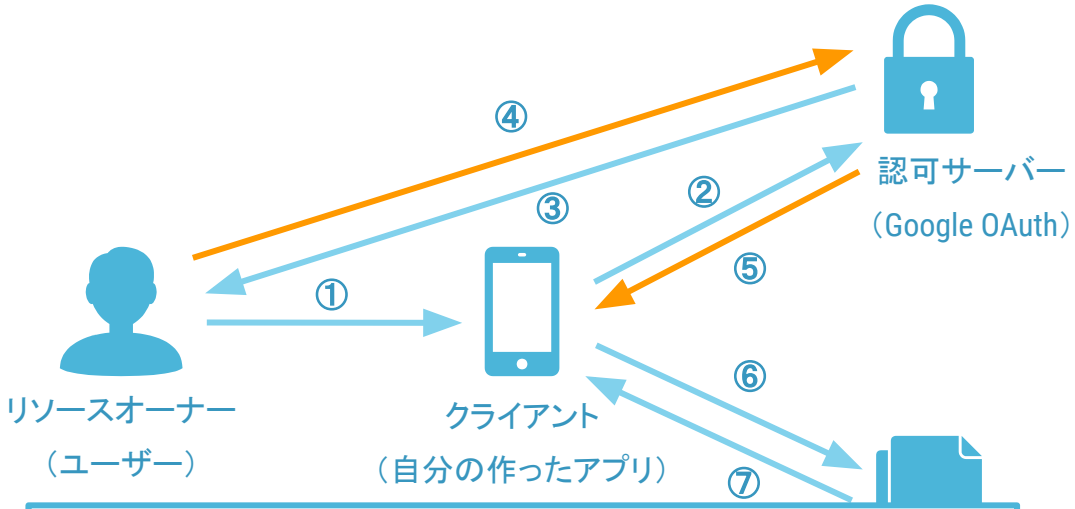

④
③
②
⑤
①
⑥
⑦
認可サーバー
（Google OAuth）
リソースオーナー
（ユーザー）
クライアント
（自分の作ったアプリ）
こんな大変なことをやる
メリットってなに？
リソースサーバー
（Google Photo）



Point 1

ユーザーのID・パスワードがクライアントに伝わっていない

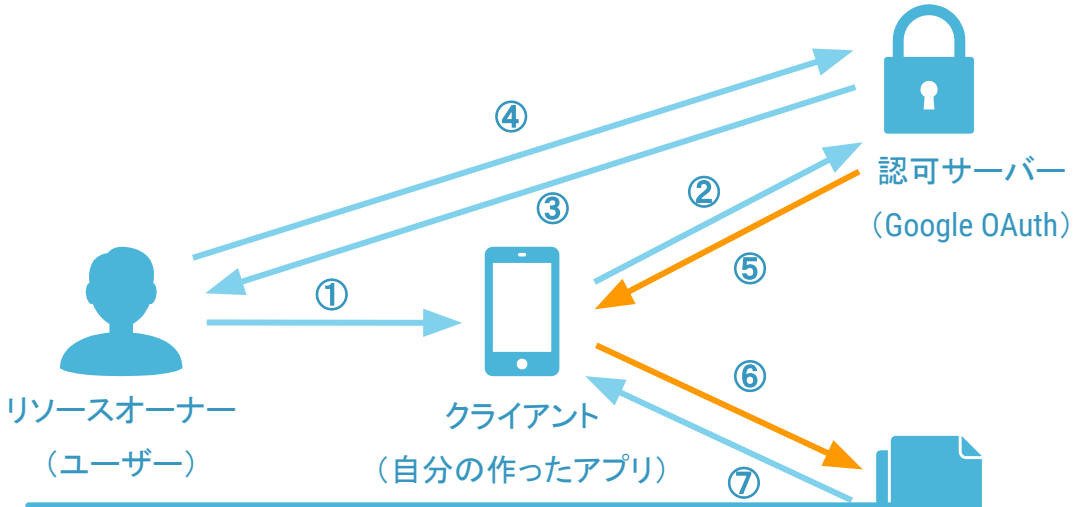
→ID・パスワード入力に起因する脆弱性を防げる



Point 2

ユーザーが許可してはじめて権限が委譲される

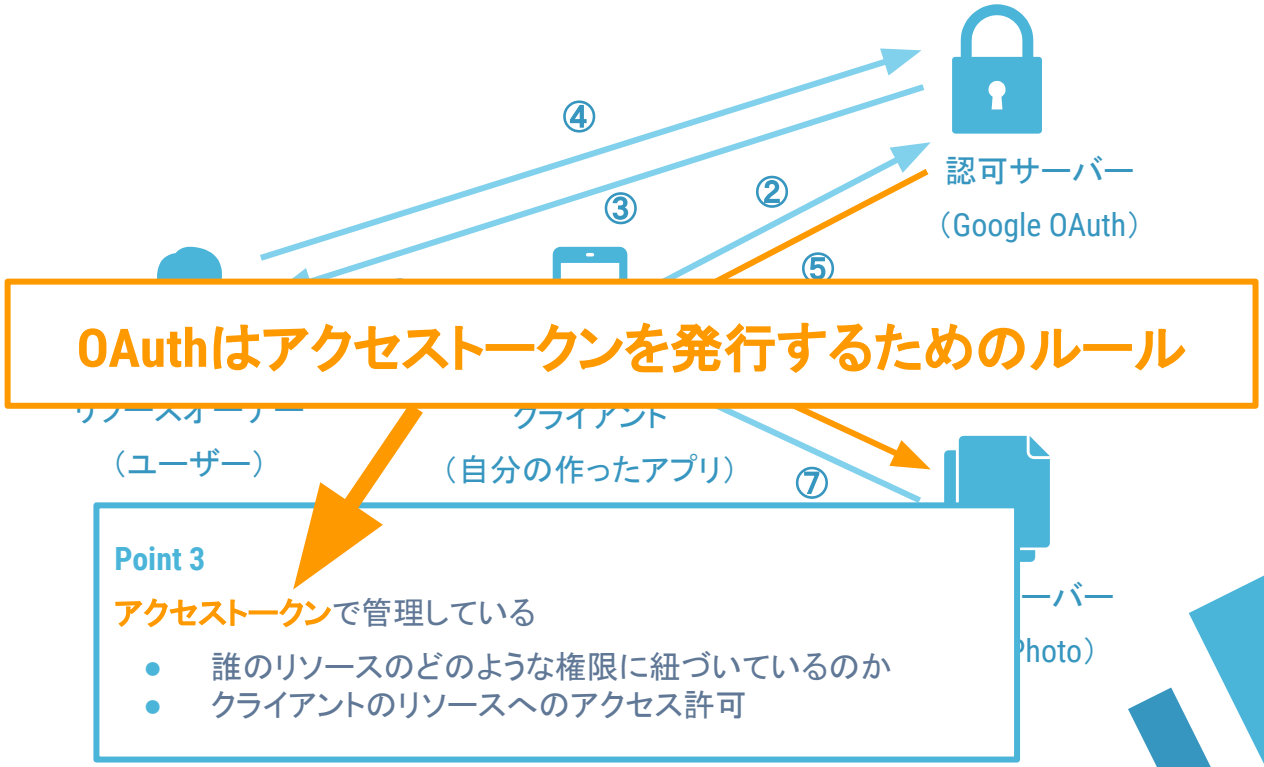
- 勝手にユーザー情報へのアクセスが許可されることはない
- 必要最小限の権限のみ委譲



Point 3

アクセストークンで管理している

- 誰のリソースのどのような権限に紐づいているのか
- クライアントのリソースへのアクセス許可



OAuthはアクセストークンを発行するためのルール

Point 3

アクセストークンで管理している

- 誰のリソースのどのような権限に紐づいているのか
- クライアントのリソースへのアクセス許可

OAuthのまとめ

クライアント
（自分の作ったアプリ）

が

リソースサーバー
（Google Photo）

（あるユーザーへの必要最小限の範囲の）
への**限定的なアクセス**を

可能にするための

アクセストークンの発行方法

2.

OAuth認証

認証のためのプロトコルという勘違い

OAuth

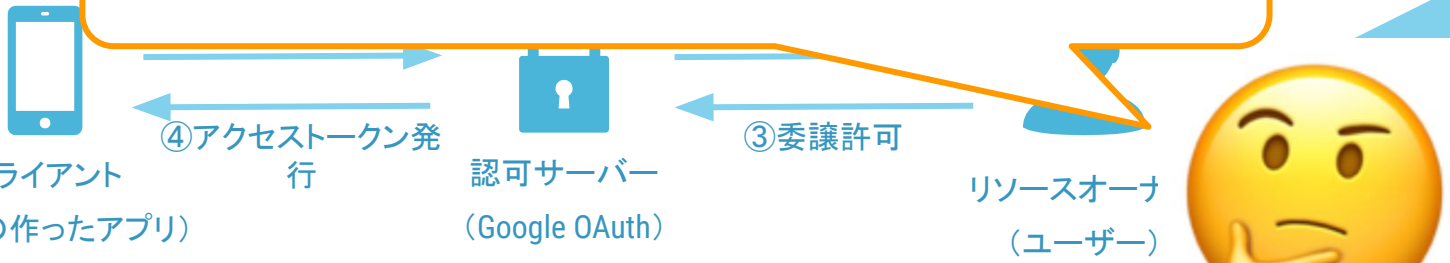
- 権限委譲のためのプロトコル
- アプリのユーザー認証の話は出てこない

認証のためのプロトコルという勘違い

OAuth

- 権限委譲のためのプロトコル
-

じゃあなんでOAuth認証なんて言葉があるの？



OAuth + プロフィールAPI

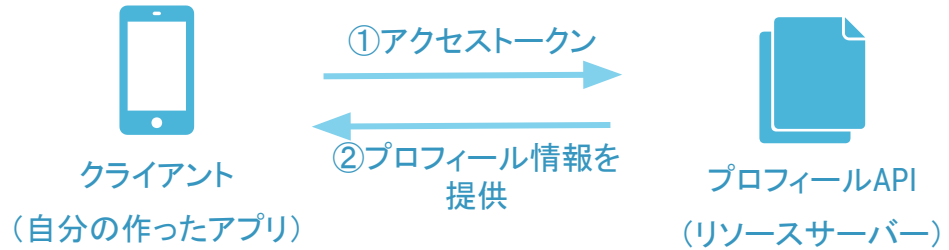
だいたいこいつのせい

プロフィールAPI
（リソースサーバー）

OAuth + プロフィールAPI

プロフィールAPI

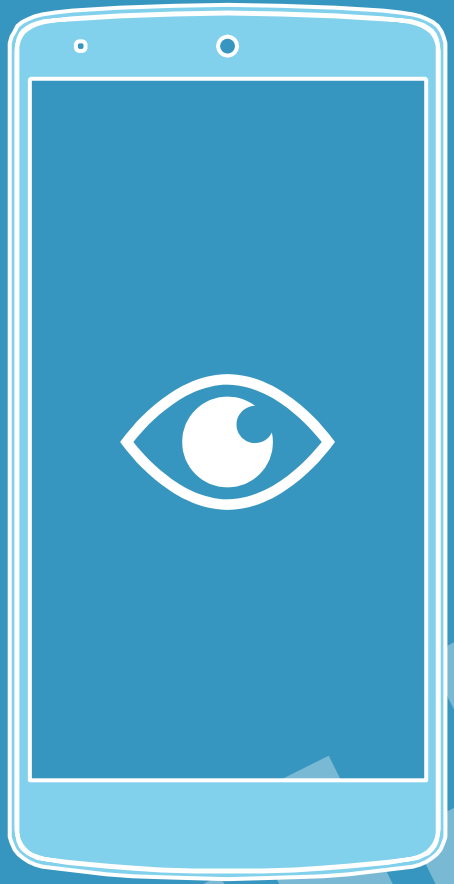
» プロフィール情報を提供するAPI



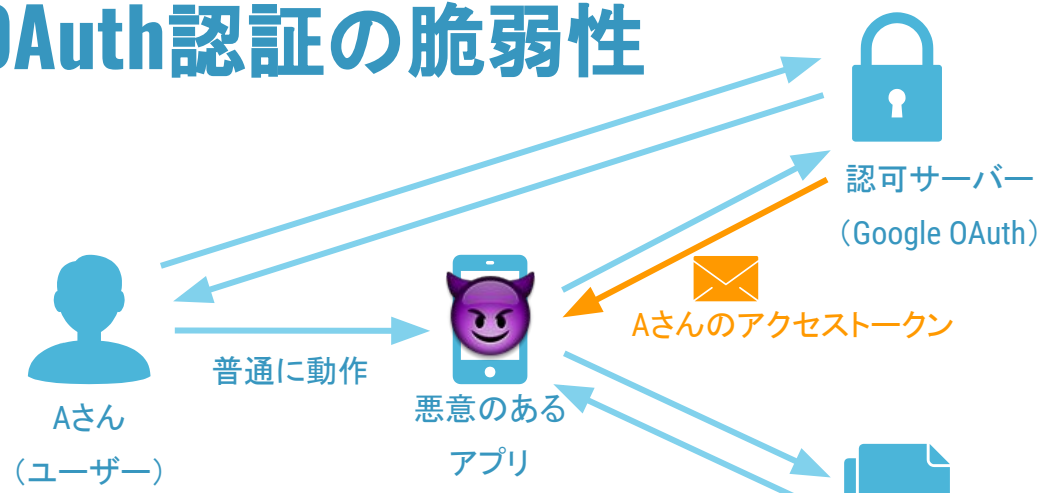
OAuth認証の仕組み

☆
アクセストークンに紐づくプロフィール情報を要求
→返されたプロフィール情報のデータで認証

OAuth認証の
脆弱性



OAuth認証の脆弱性



Step 1

悪意あるアプリが**他人のアクセストークン**を取得

→見た目上は普通に動作

OAuth認証の脆弱性

Step 2

盗んだアクセストークンに差し替え

→別のユーザーとしてログインできてしまう！

OAuth認証の脆弱性

Step 2

盗んだアクセストークンに差し替え

→別のユーザーとしてログインできてしまう！

補足：OAuth認証

FacebookのOAuth認証

- アクセストークン検証用エンドポイント
  - /debug_token
  - 検証用データを取得
- アクセストークンを検証
  - すり替えられていないことを担保

OAuth認証のまとめ

プロフィールAPI

から取得したプロフィールの情報を
使用して認証&ログインする方法

悪意あるアプリによってアクセストークンを盗まれた場合、
別のユーザーになりすましができる脆弱性がある

3.

OpenID Connect

OpenID Connectとは？

OpenID Connect(OIDC)

- **クライアント**が認証できるプロトコル
- リソースアクセス（認可）も可能
- OAuthの拡張仕様

OpenID Connect (OIDC)
= OAuth + IDトークン + UserInfoエンドポイント

OpenID Connectとは？

OpenID Connect(OIDC)

- **クライアント**が認証できるプロトコル
- リソースアクセス（認可）も可能
- OAuthの拡張仕様

OpenID Connect (OIDC)
= OAuth + **IDトークン** + UserInfoエンドポイント

Point!

OpenID Connectとは？

三種類のフロー

- 認可フロー
  - 安全にclient secretを保存できるクライアント用（サーバー）
- インプリシットフロー
  - 安全にclient secretを保存できないクライアント用（アプリ）
- ハイブリットフロー
  - 安全にアクセストークンや IDトークンを保存できない
    クライアント用（SPA）

OpenID Connectとは？

三種類のフロー

- 認可フロー
  - 安全にclient secretを保存できるクライアント用（サーバー）
- インプリシットフロー
  - 安全にclient secretを保存できないクライアント用（アプリ）
- ハイブリットフロー
  - 安全にアクセストークンやIDトークンを保存できない
    クライアント用（SPA）

あとで説明

OpenID Connectとは？

OAuthのグラントとOIDCのフロー

OAuth	OIDC
認可コードグラント	認可コードフロー
インプリシットグラント	インプリシットフロー
クライアント クレデンシャルグラント	-
リソースオーナパスワード クレデンシャルグラント	-
ハイブリッドグラント	ハイブリッドフロー

OpenID Connectの流れを追ってみる

OIDCの登場人物

IDプロバイダ

エンドユーザー

リライング・パーティ

UserInfoエンド

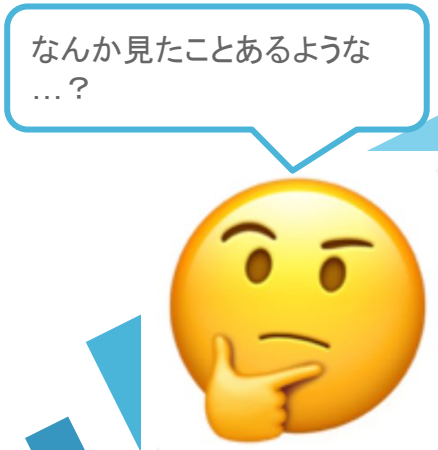
OIDCの登場人物

IDプロバイダ

エンドユーザー

リライング・パーティ

UserInfoエンド



OIDCの登場人物

OAuth：認可サーバー

OIDC：IDプロバイダ

OAuth：リソースオーナー

OIDC：エンドユーザー

OAuth：クライアント

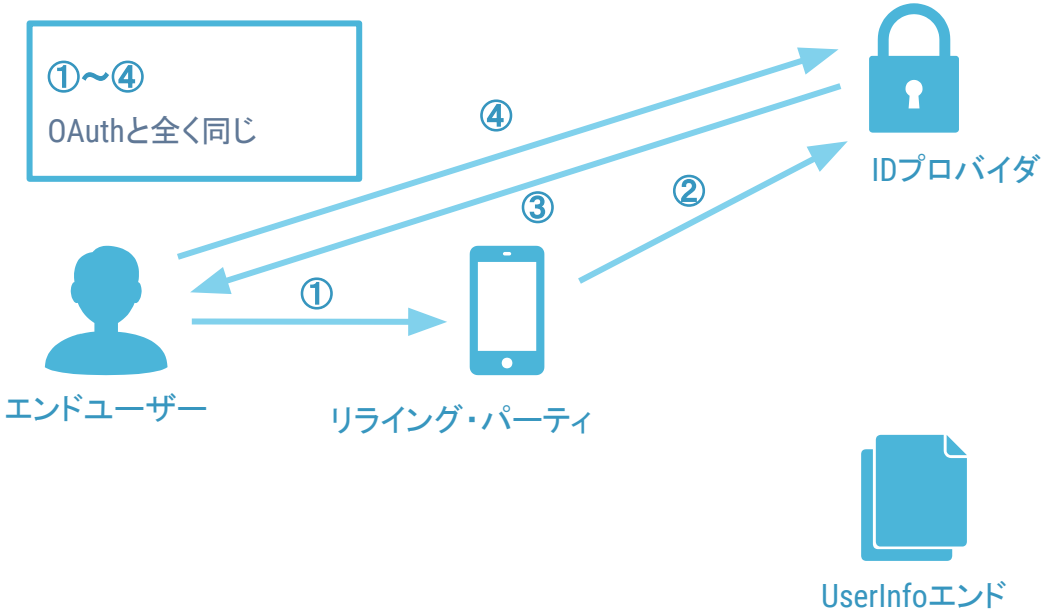
OIDC：リライング・パーティ

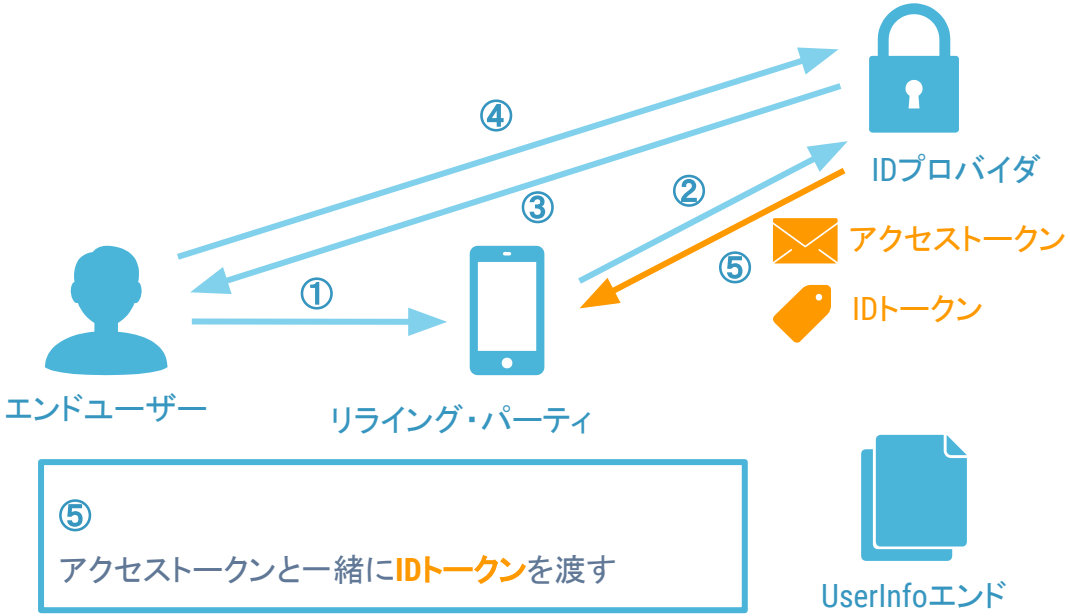
OAuth：リソースサーバー

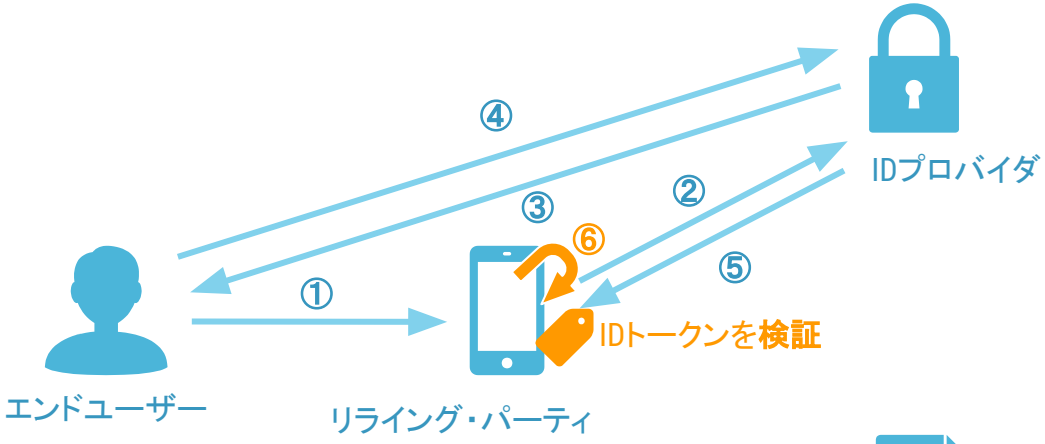
OIDC：UserInfoエンド

Point

ロールの呼び方が違うだけで、役割はOAuthと同じ！

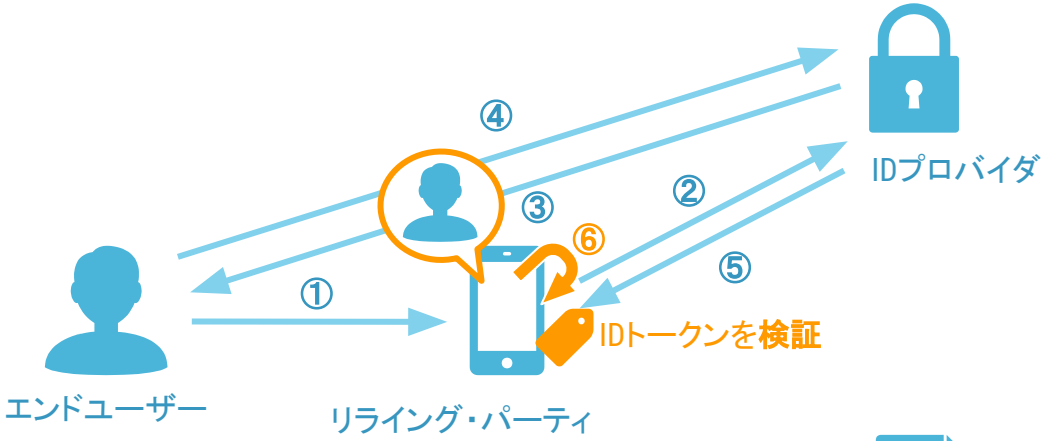

①〜④
OAuthと全く同じ
④
③
②
①
IDプロバイダ
エンドユーザー
リライング・パーティ
UserInfoエンド


④
③
②
①
⑤
IDプロバイダ
アクセストークン
IDトークン
エンドユーザー
リライング・パーティ
⑤
アクセストークンと一緒にIDトークンを渡す
UserInfoエンド



⑥
IDトークンを検証し、ユーザーを認証する

UserInfoエンド

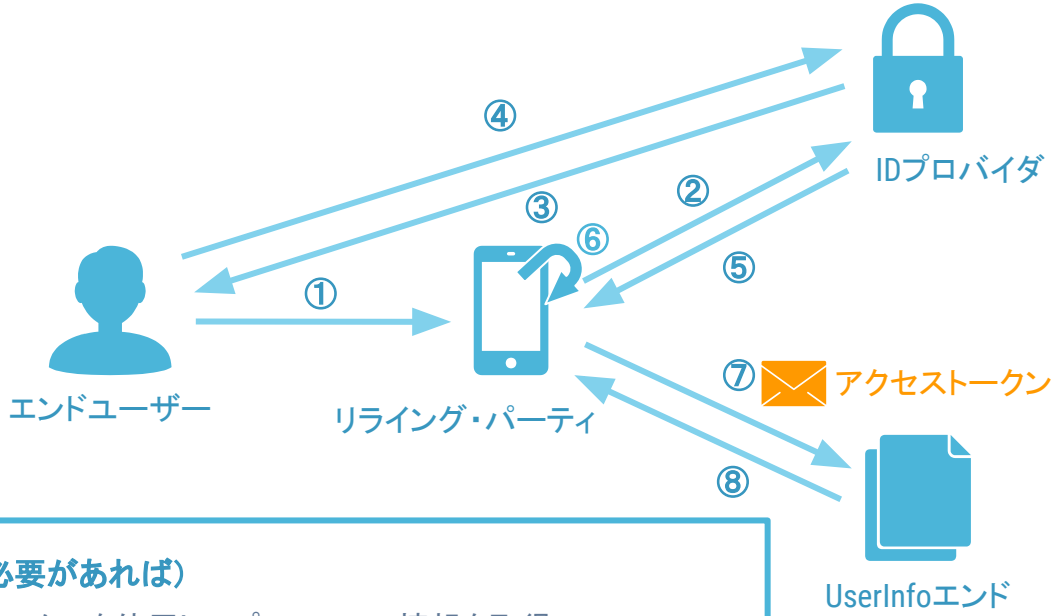


⑥

IDトークンを検証し、ユーザーを認証する

→ログイン完了！

UserInfoエンド



⑦～⑧（必要があれば）

アクセストークンを使用してプロフィール情報を取得

例：名前、アイコン画像、その他個人情報

IDトークンが
なぜ必要なのか？

IDトークンとは？

IDトークン

- リライング・パーティがエンドユーザを認証するために使用
- 署名付きのJSON Web Token(JWT)
- ユーザーIDや検証に必要な情報、署名などが含まれる

IDトークンとは？

IDトークンの中身

- 誰が、いつ、どの範囲で、と言った（IdPから見た）一連の認証イベントしての情報が含まれている
  - ユーザーIDや検証に必要な情報、署名
  - 発行日時、有効期限
  - 許可された権限

適切に検証することで意図するやりとりが行われたことを確認できる

IDトークンとは？

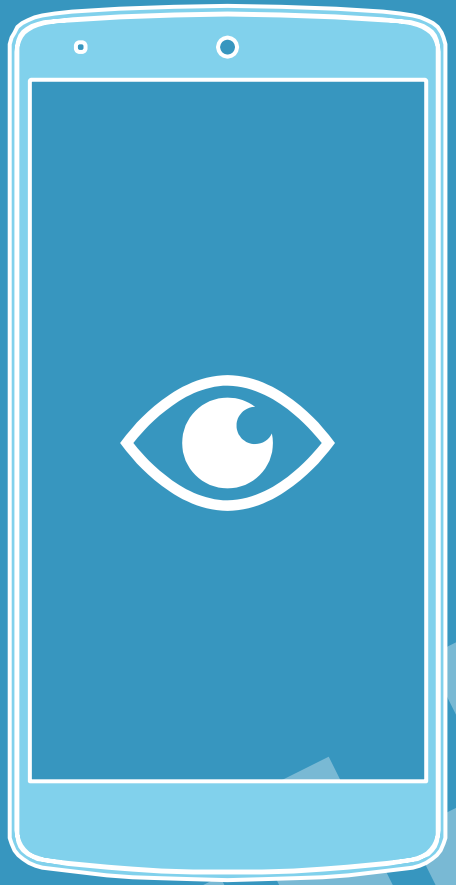
IDトークンの中身

- 誰が、いつ、どの範囲で、と言った（IdPから見た）一連の認証イベントしての情報が含まれている
  - ユーザーIDや検証に必要な情報、署名
  - 発行日時、有効期限
  - 許可された権限

適切に検証することで意図するやりとりが行われたことを確認できる

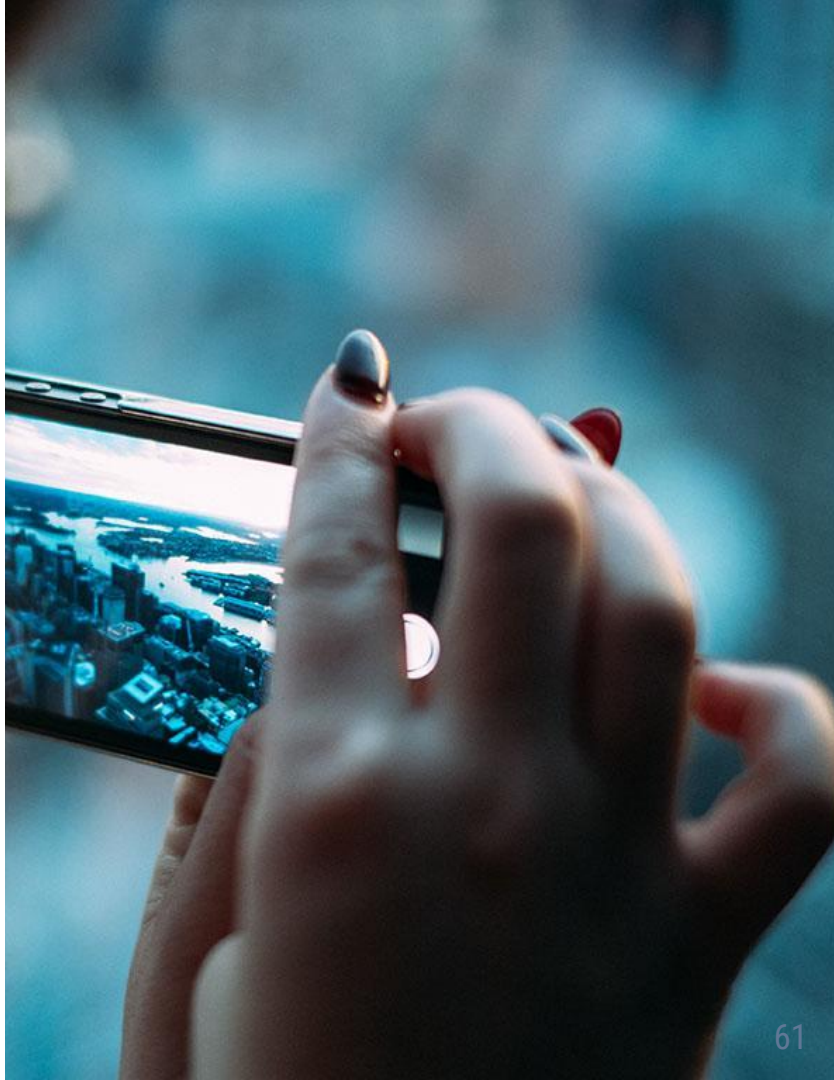
Point!

不正ログイン
を防ぐ仕組み



注意！

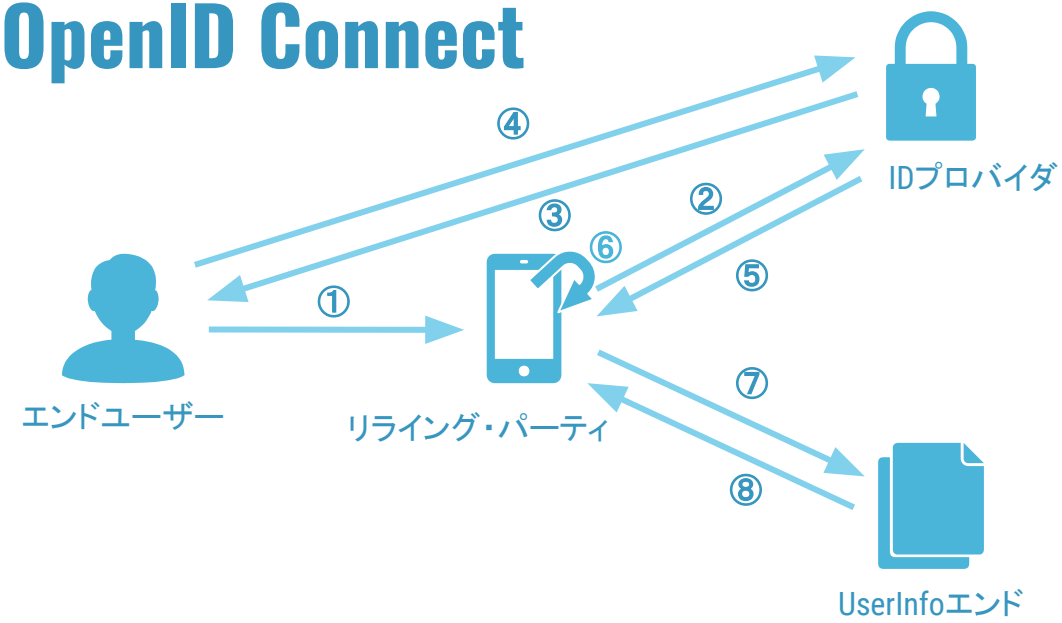
IDトークン自体の具体的な検証方法については触れません



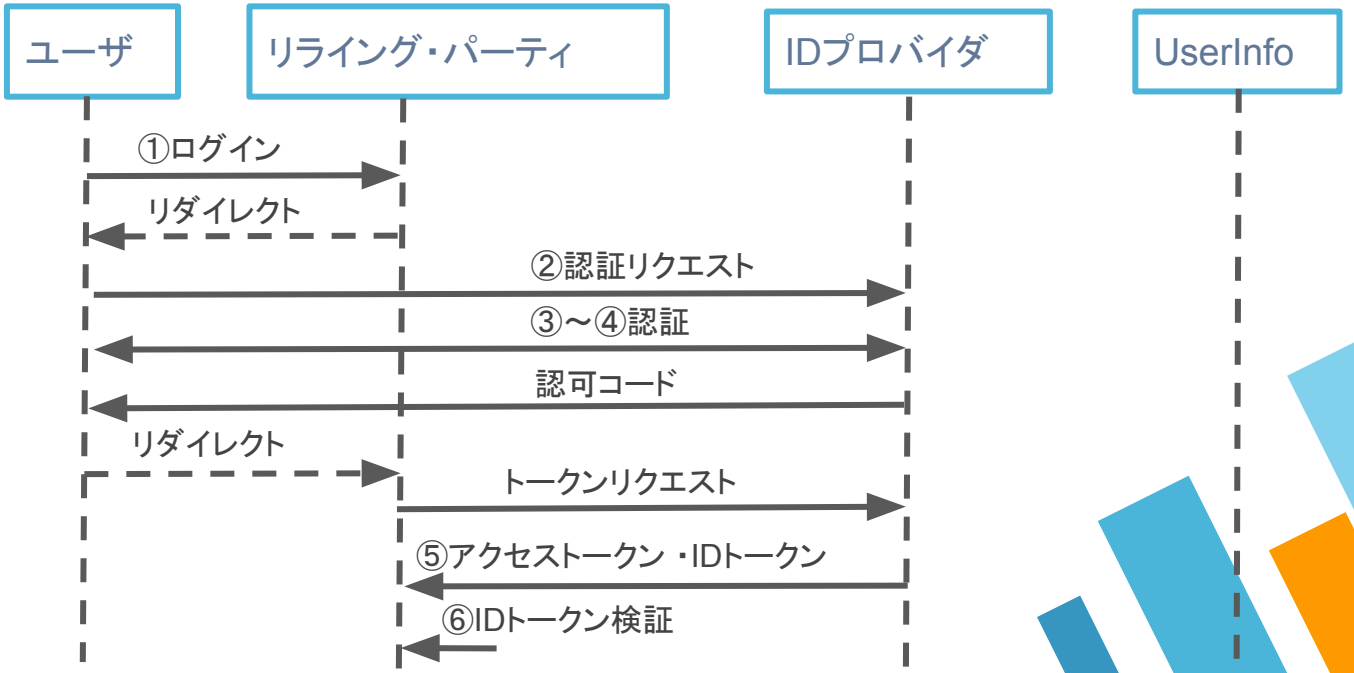
nonceパラメータ

- リプレイアタックを防ぐためのパラメータ
- IDトークンに含まれる
- 毎回異なるランダムな文字列を使用する

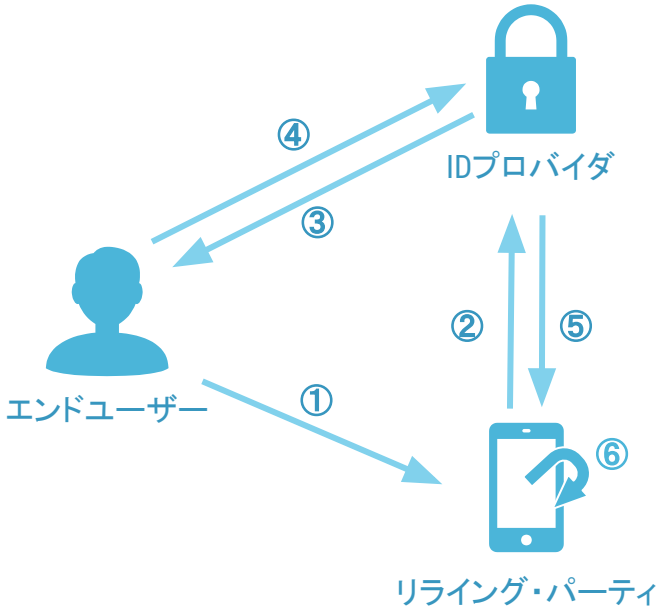
OpenID Connect

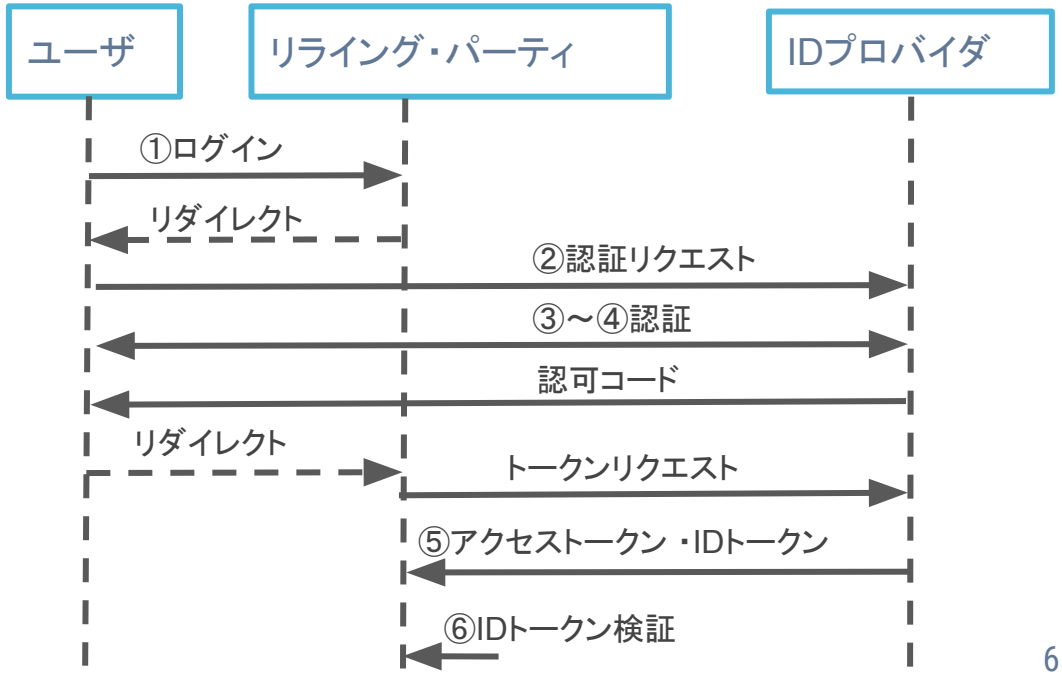


OpenID Connectのシーケンス図

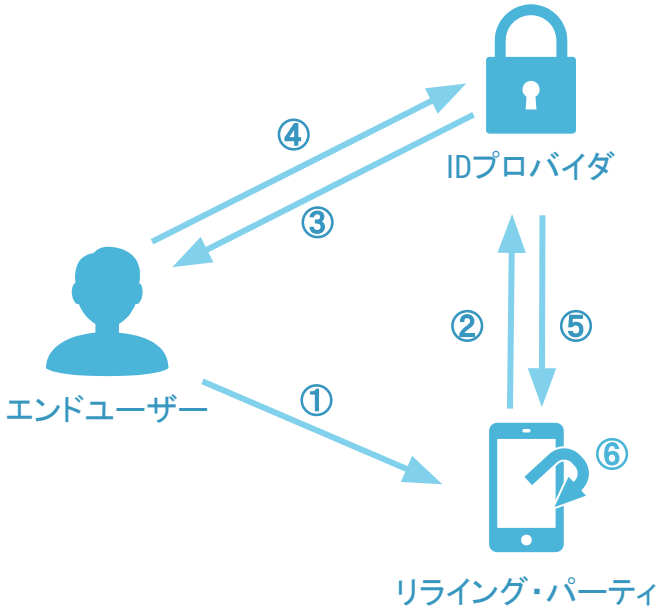


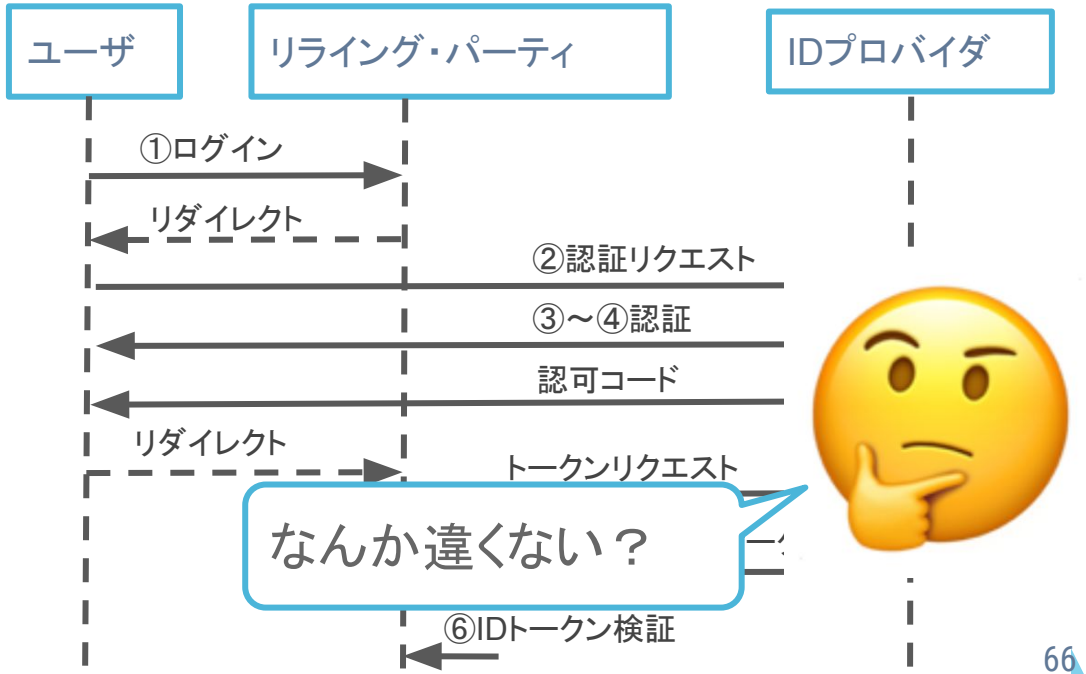
OpenID Connectのシーケンス図



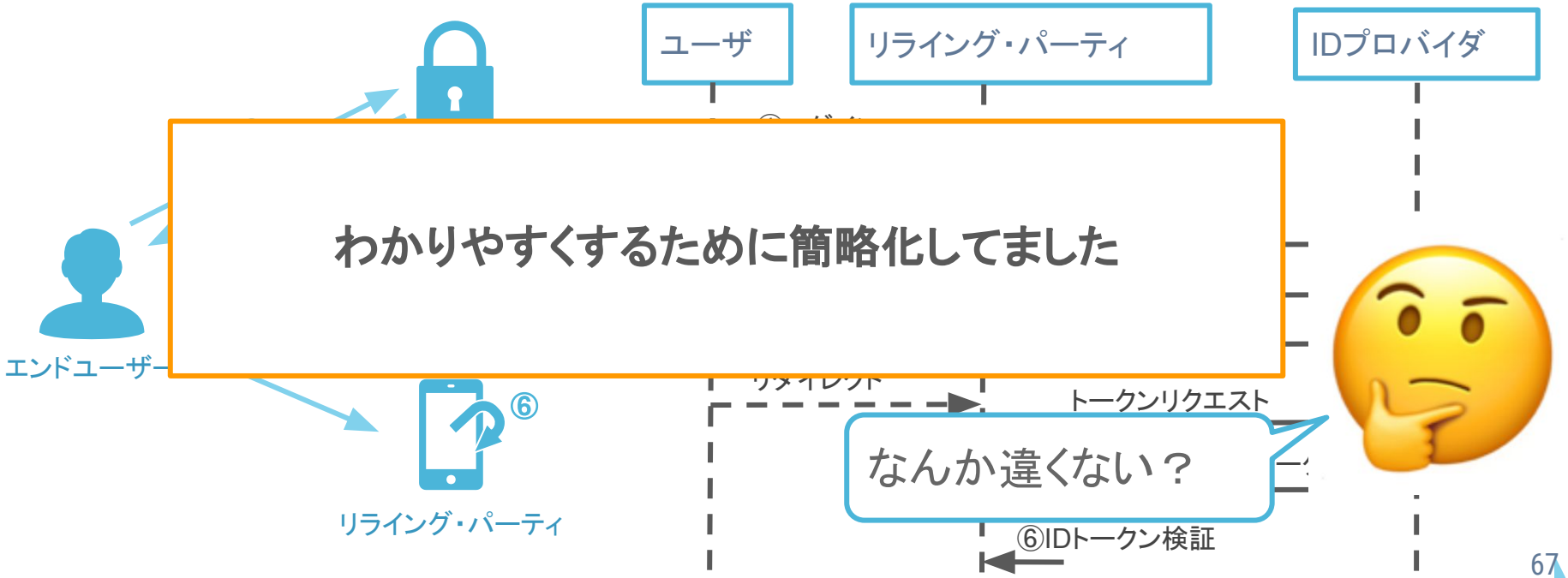


OpenID Connectのシーケンス図





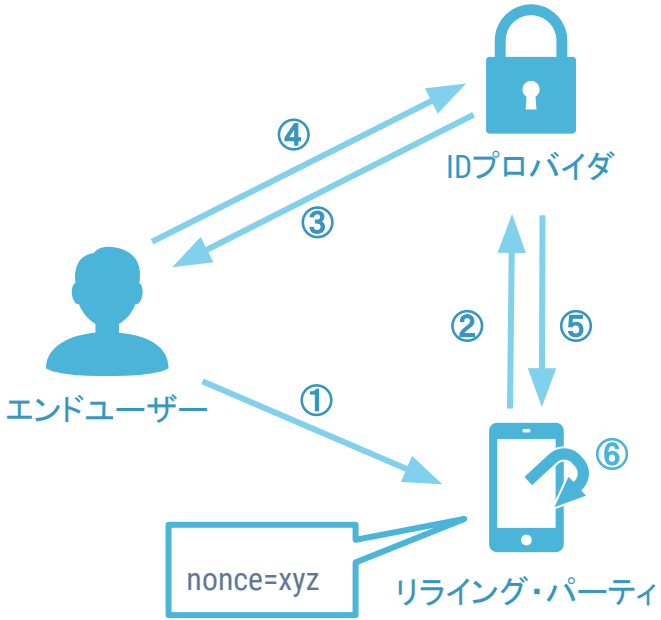
OpenID Connectのシーケンス図

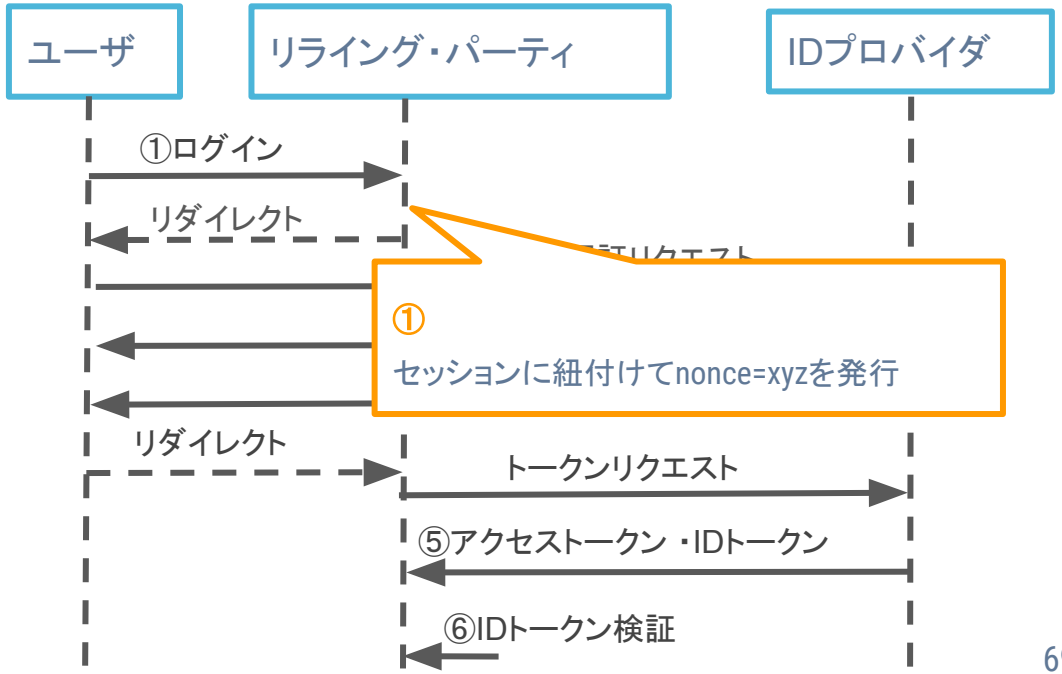


わかりやすくするために簡略化してました

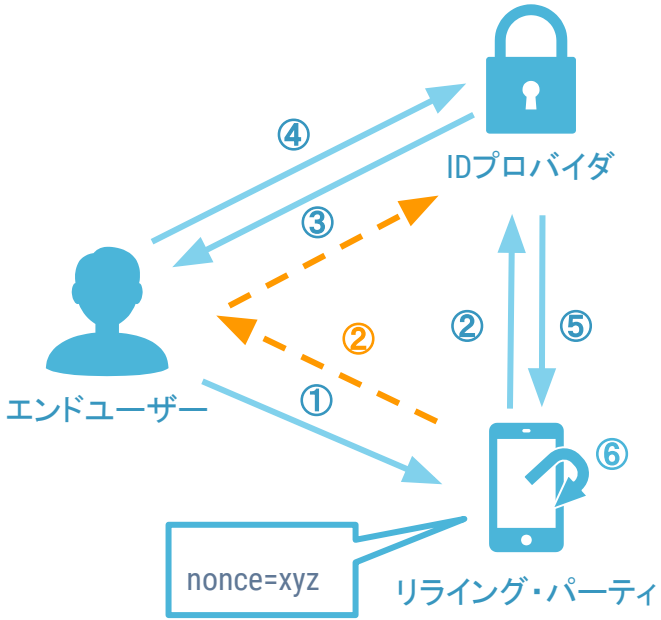
nonceの使い方

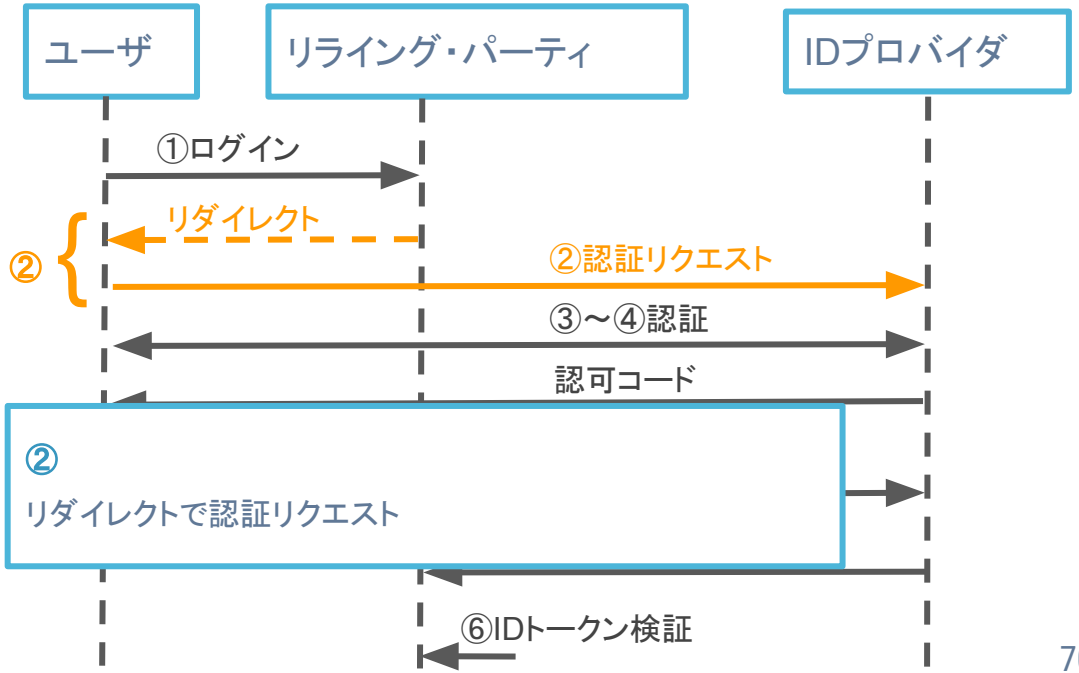
OpenID Connectのシーケンス図



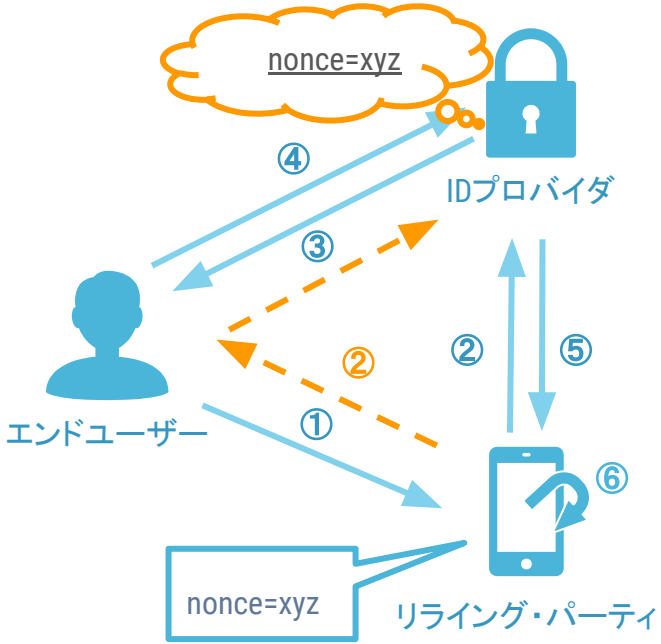


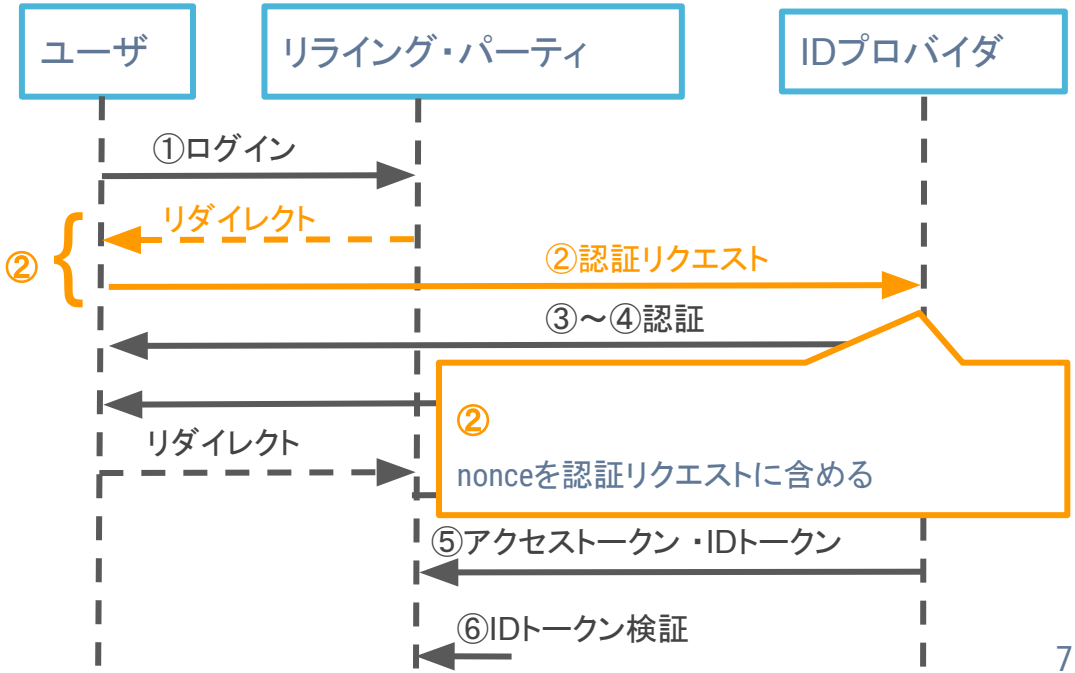
OpenID Connectのシーケンス図



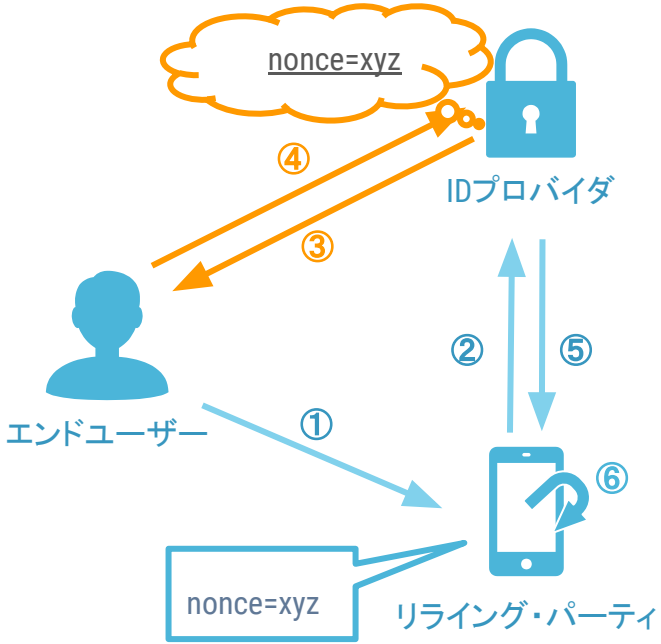


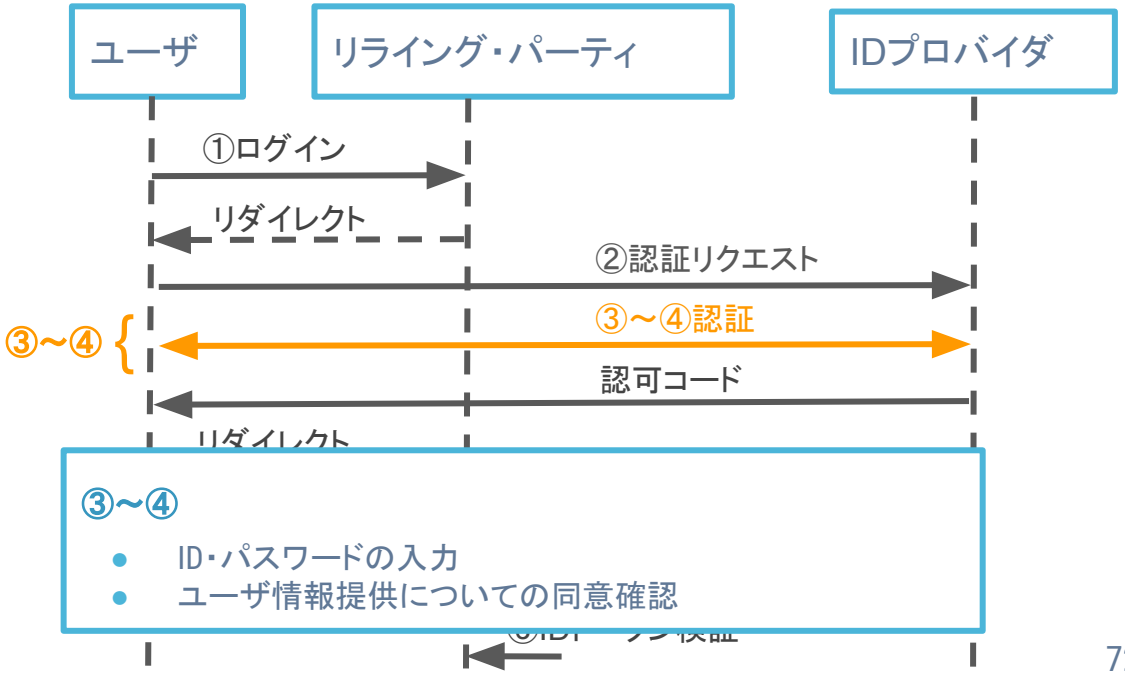
OpenID Connectのシーケンス図



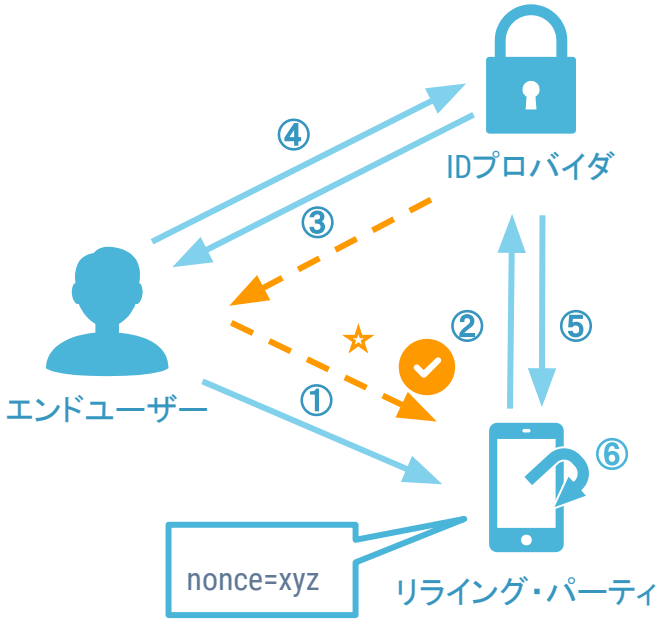


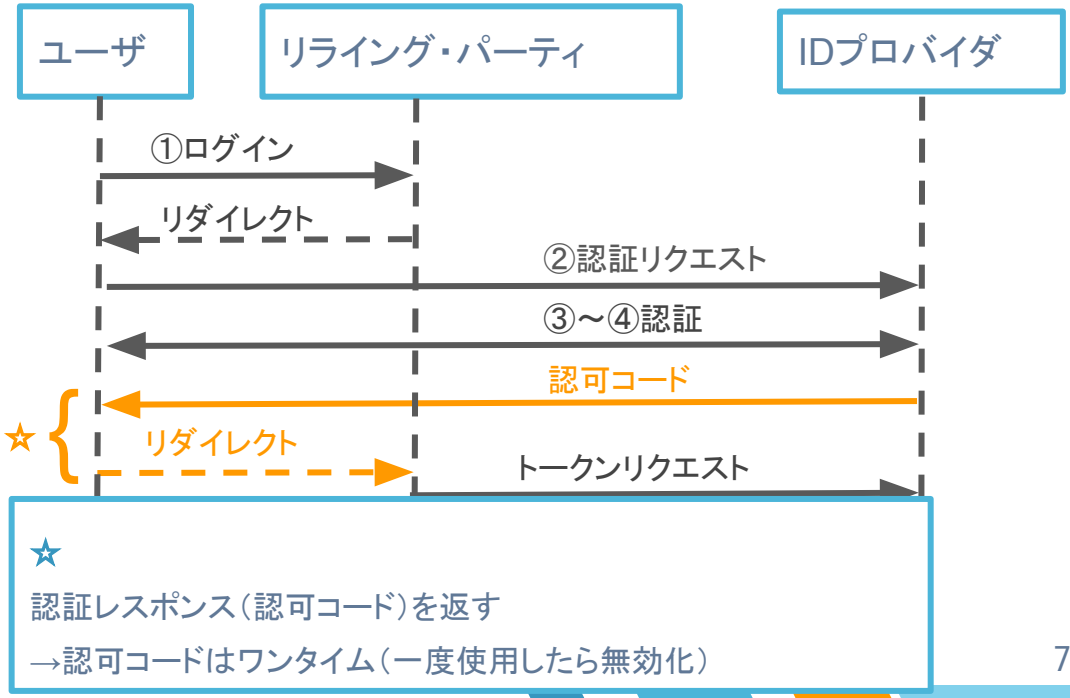
OpenID Connectのシーケンス図



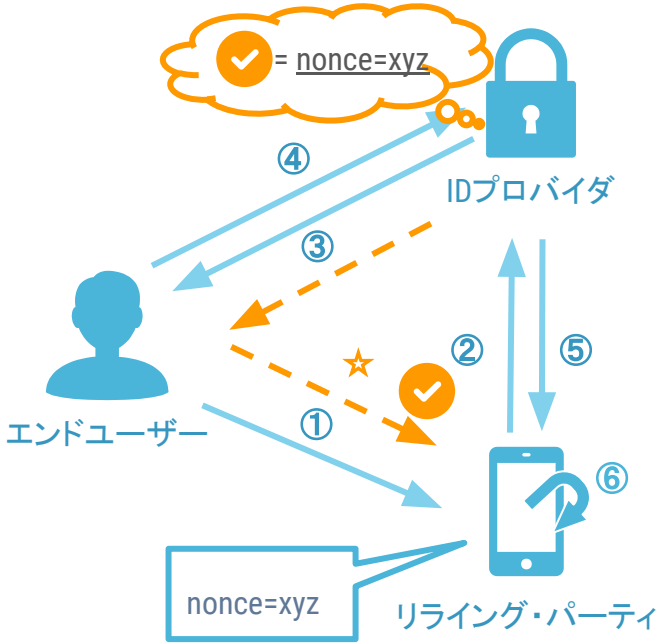


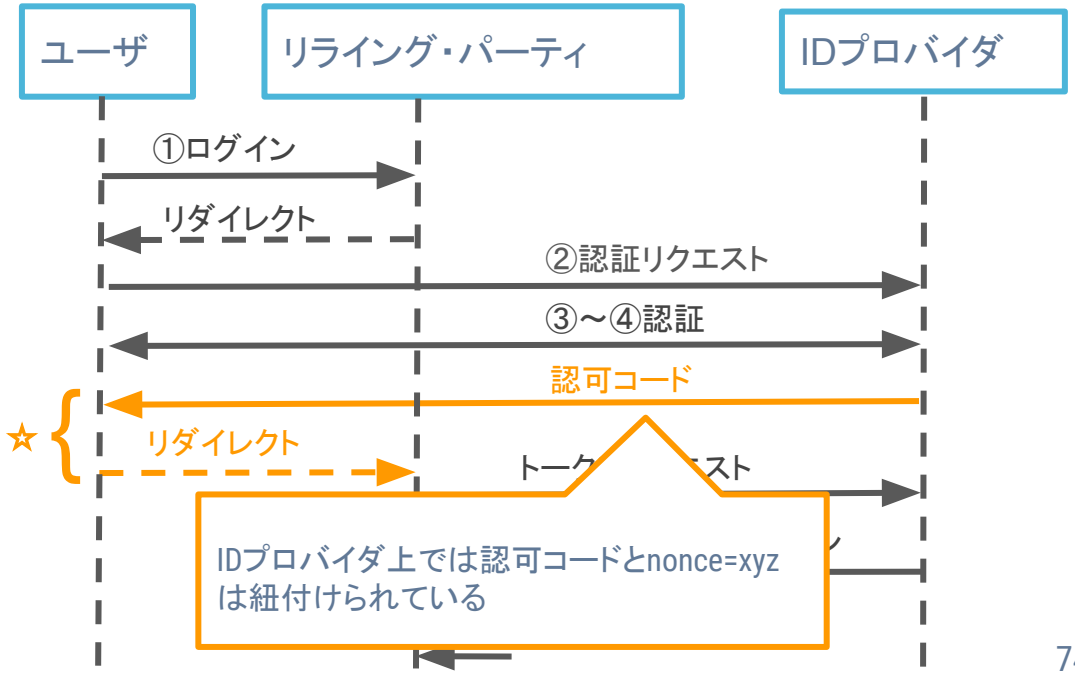
OpenID Connectのシーケンス図



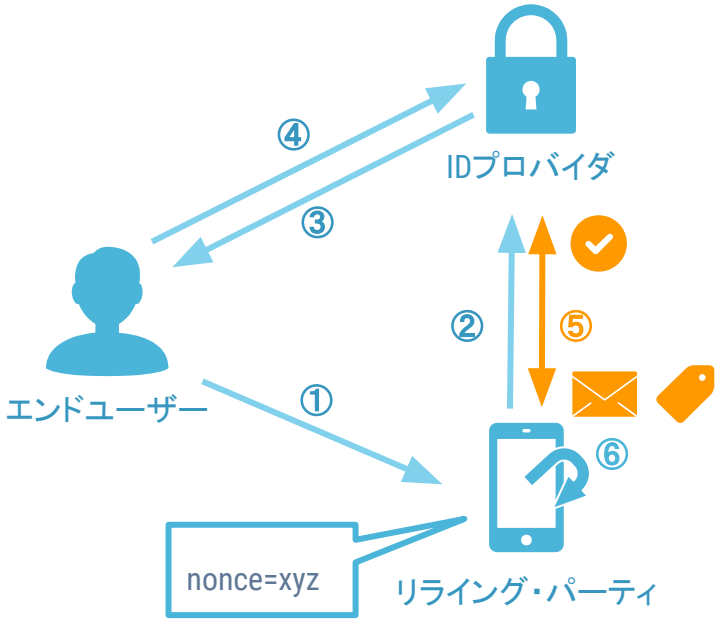


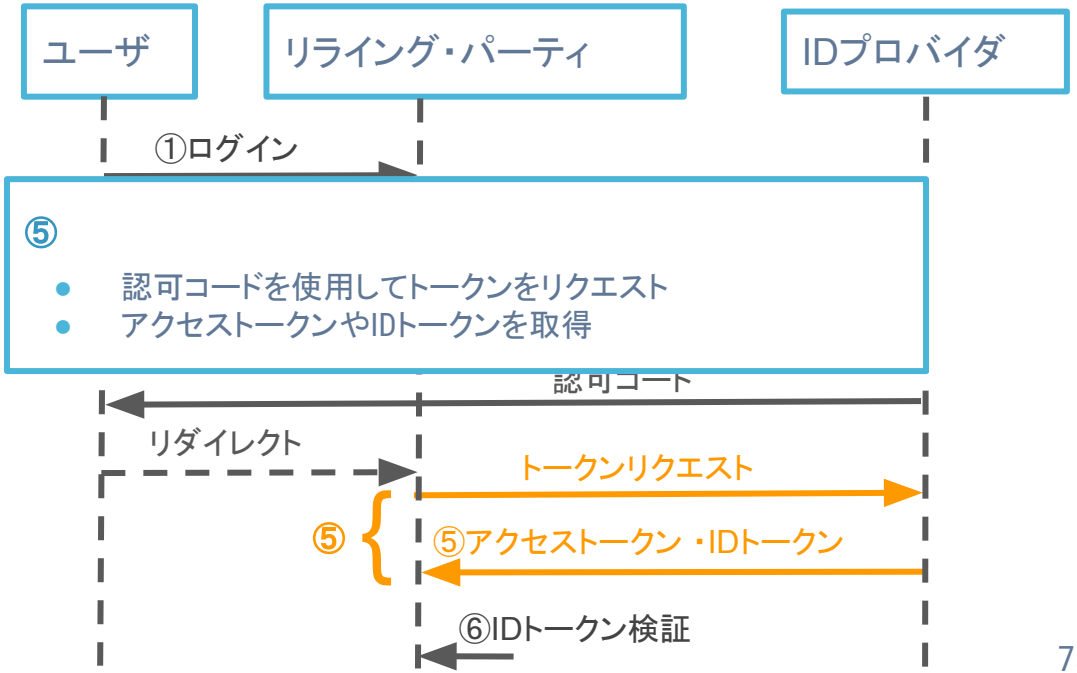
OpenID Connectのシーケンス図



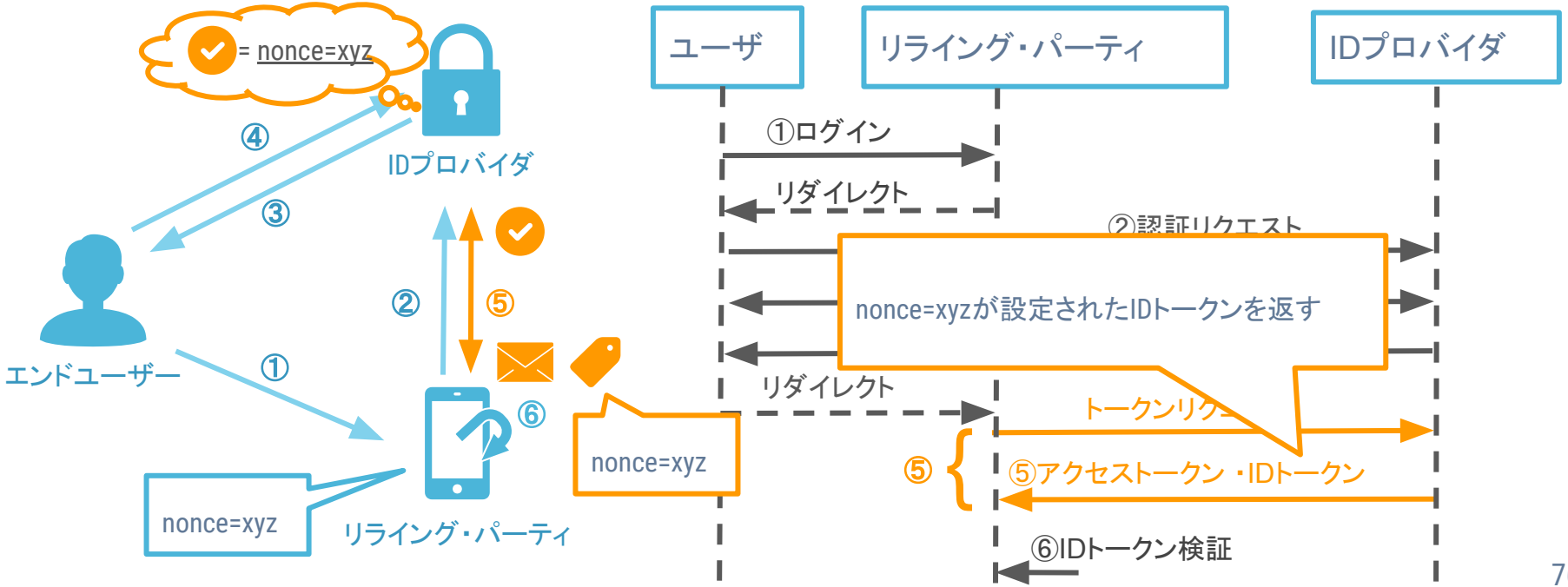


OpenID Connectのシーケンス図

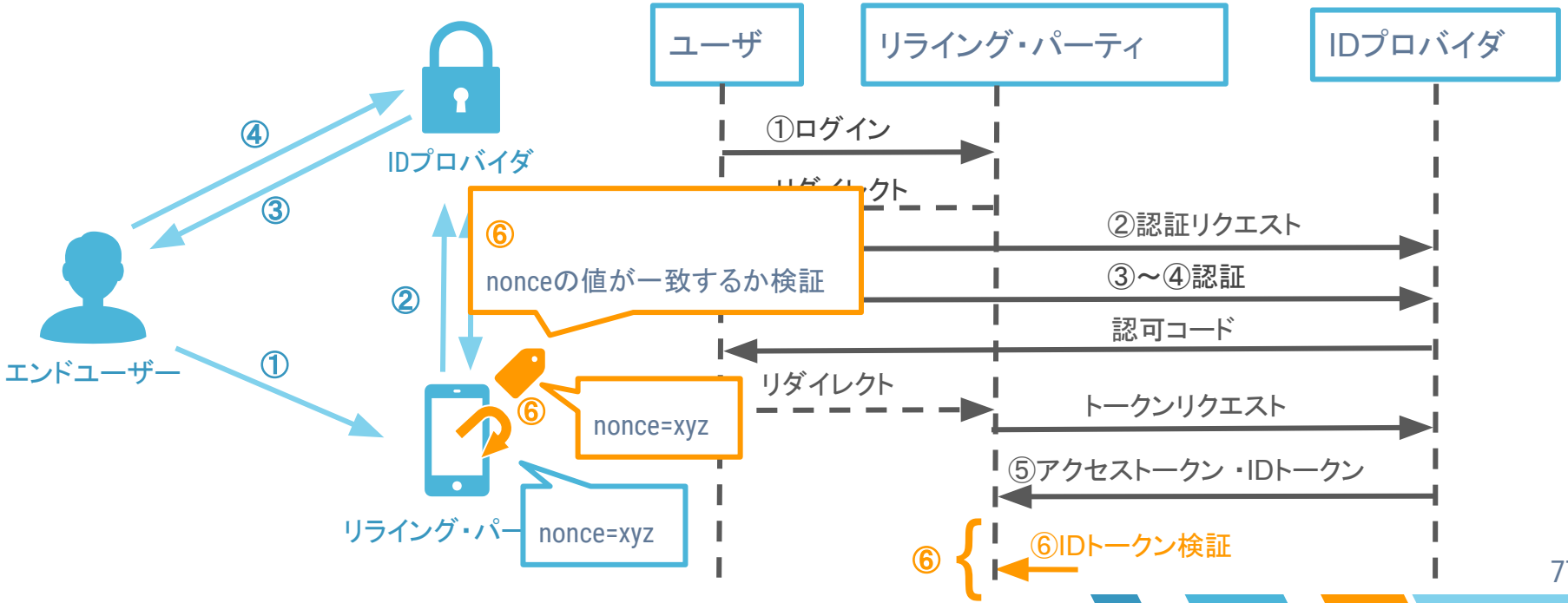




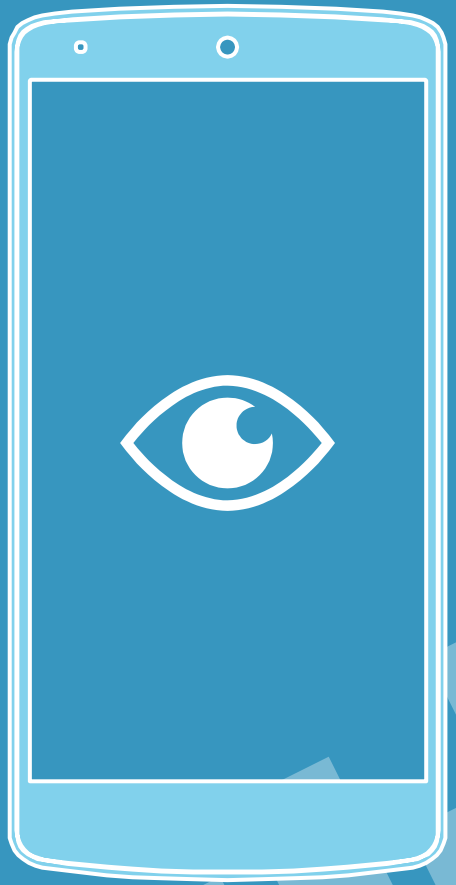
OpenID Connectのシーケンス図



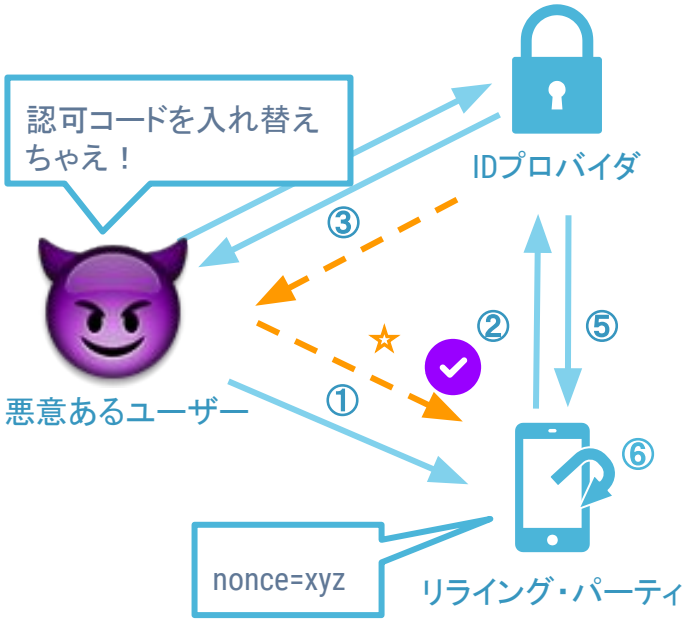
OpenID Connectのシーケンス図

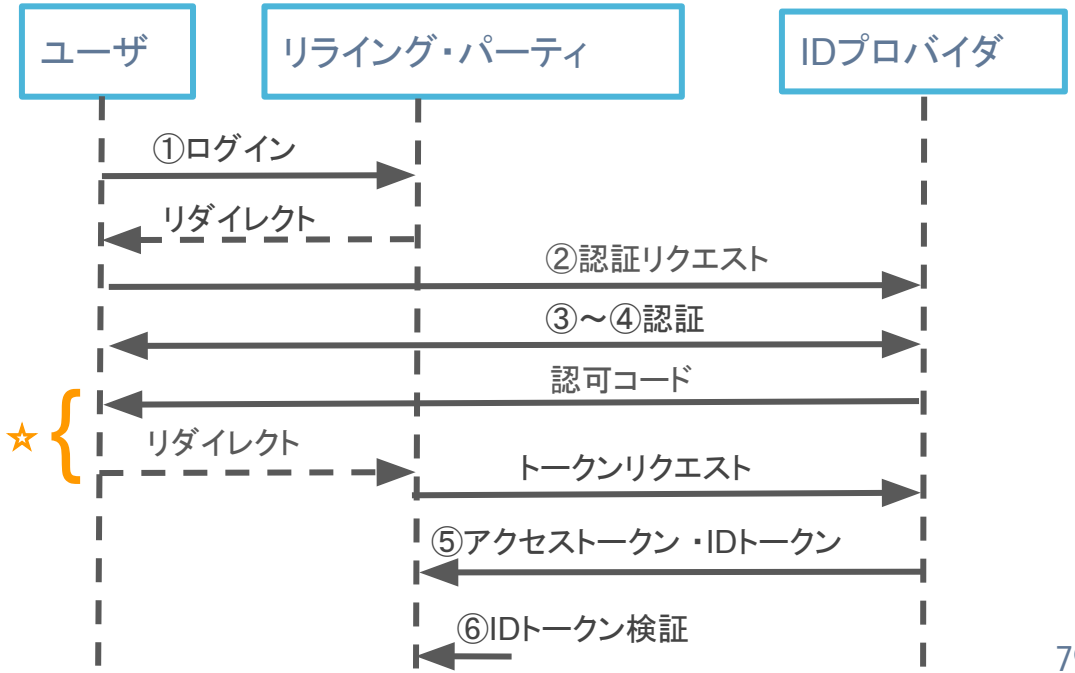


OIDCで
入れ替えを
受けた場合

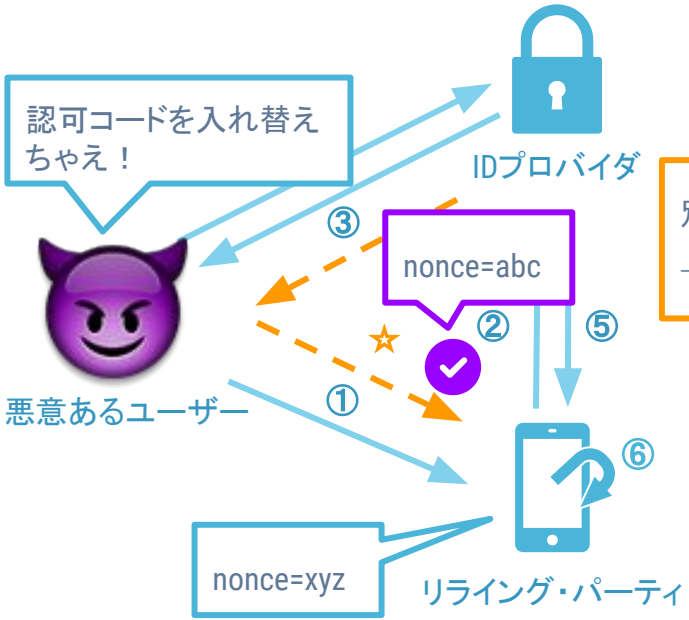


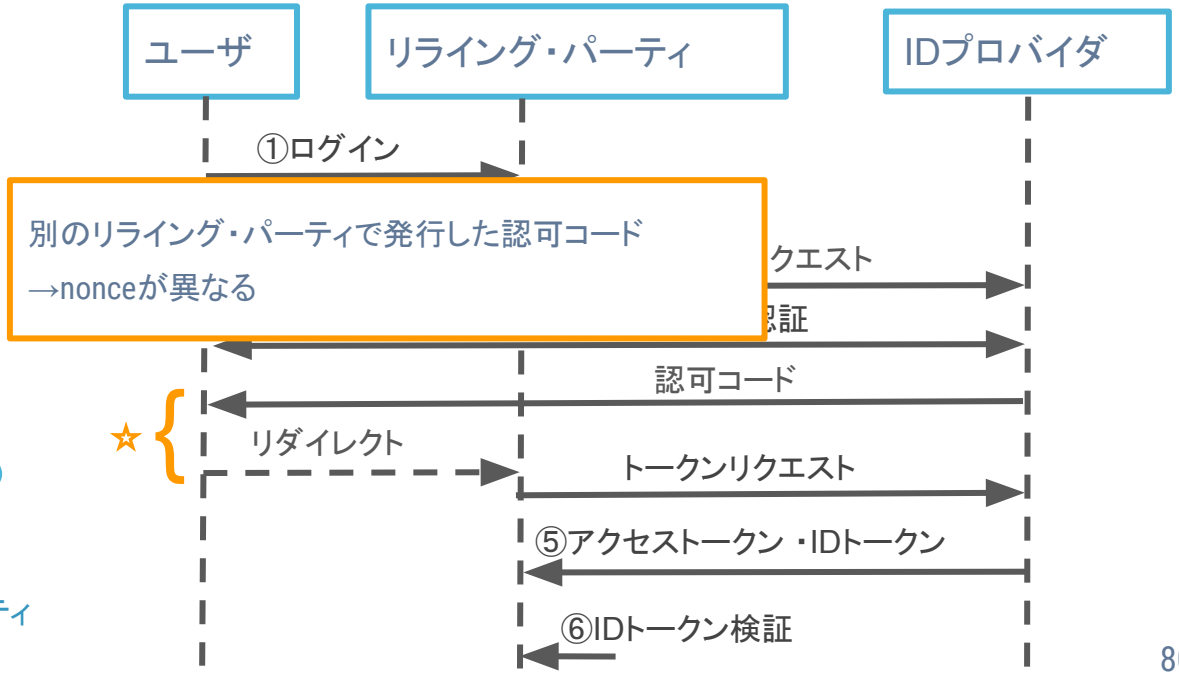
認可コードを入れ替える場合



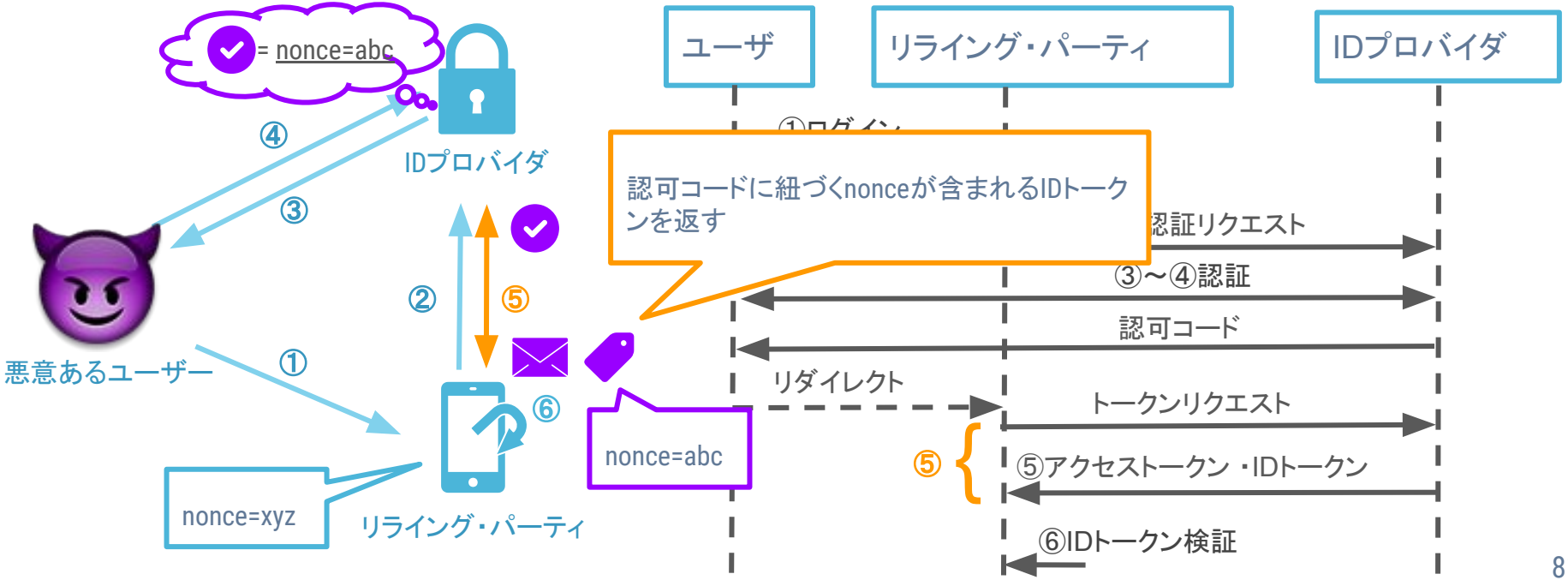


認可コードを入れ替える場合

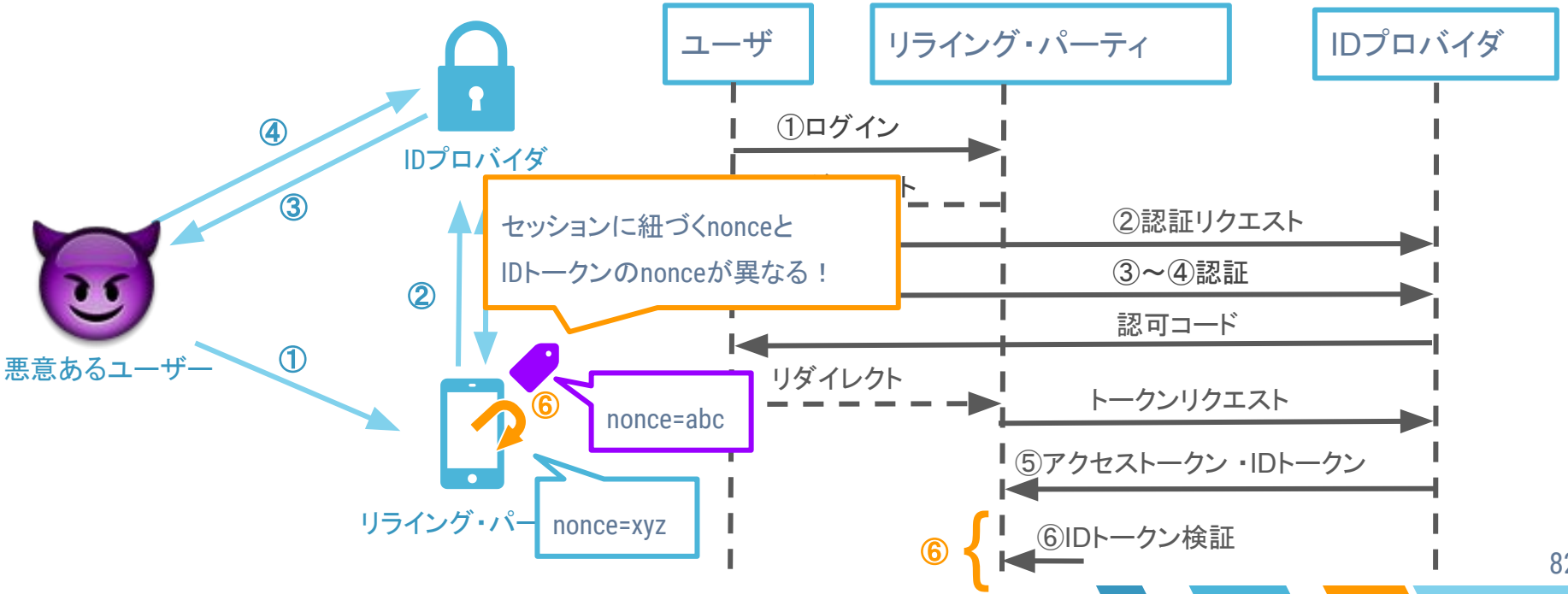




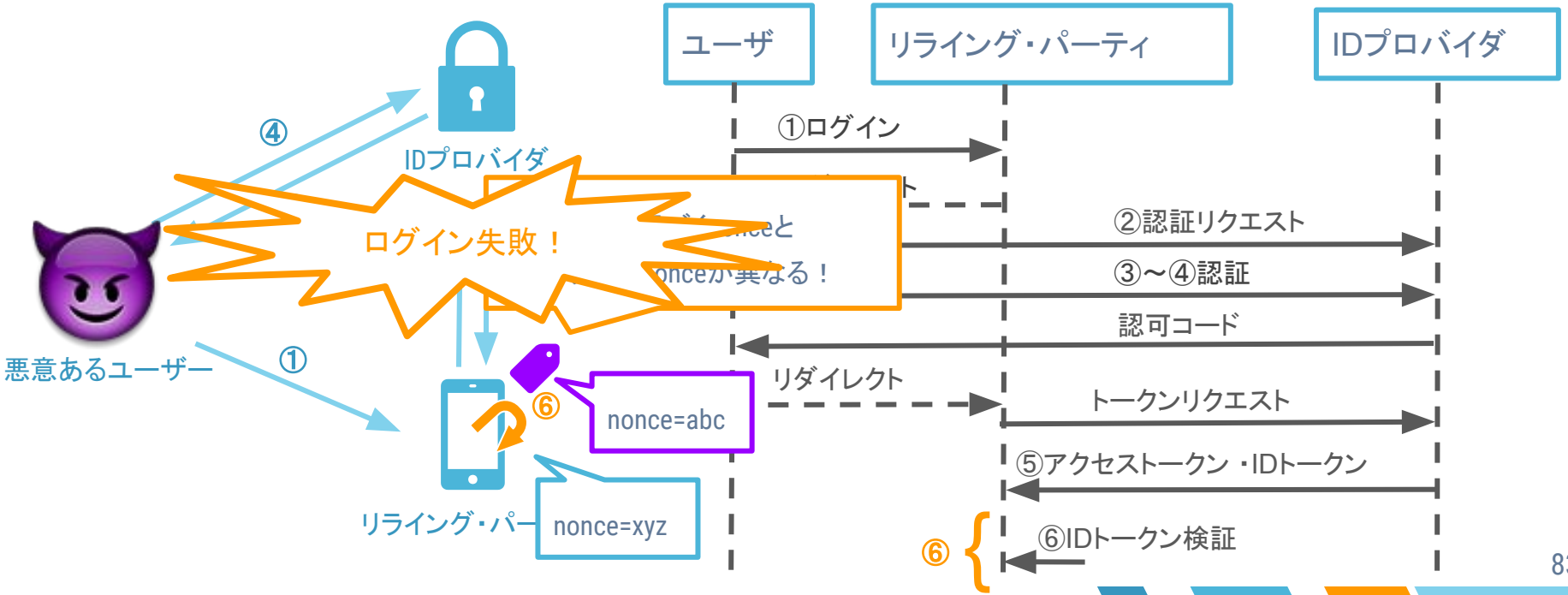
認可コードを入れ替える場合



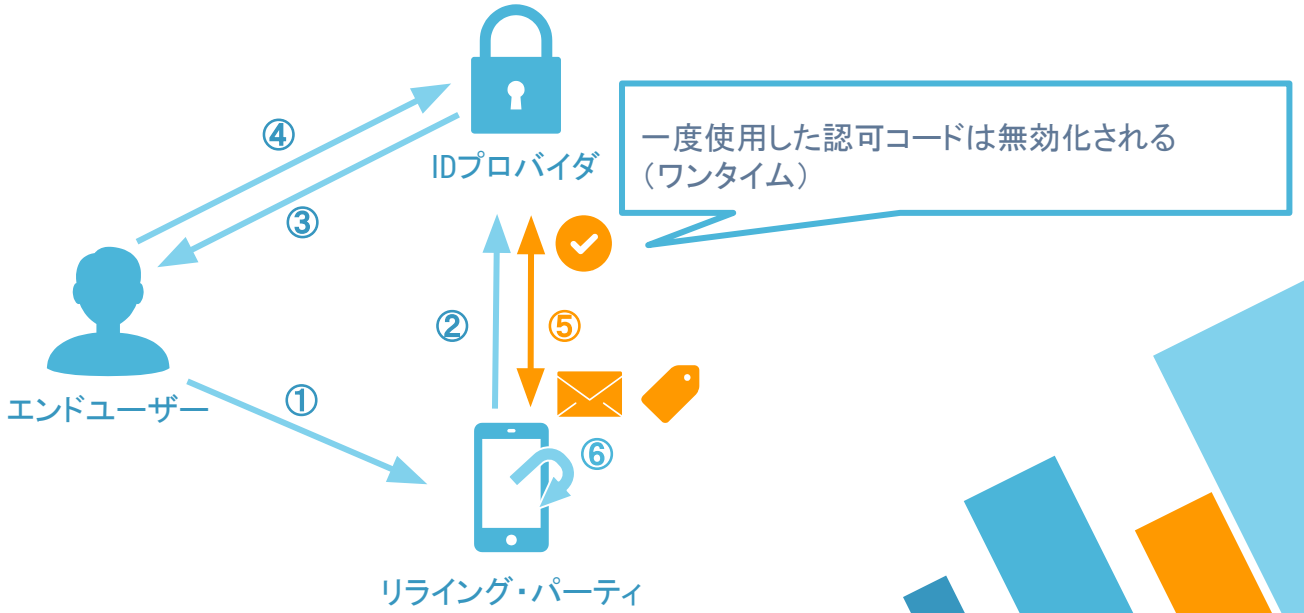
認可コードを入れ替える場合



認可コードを入れ替える場合

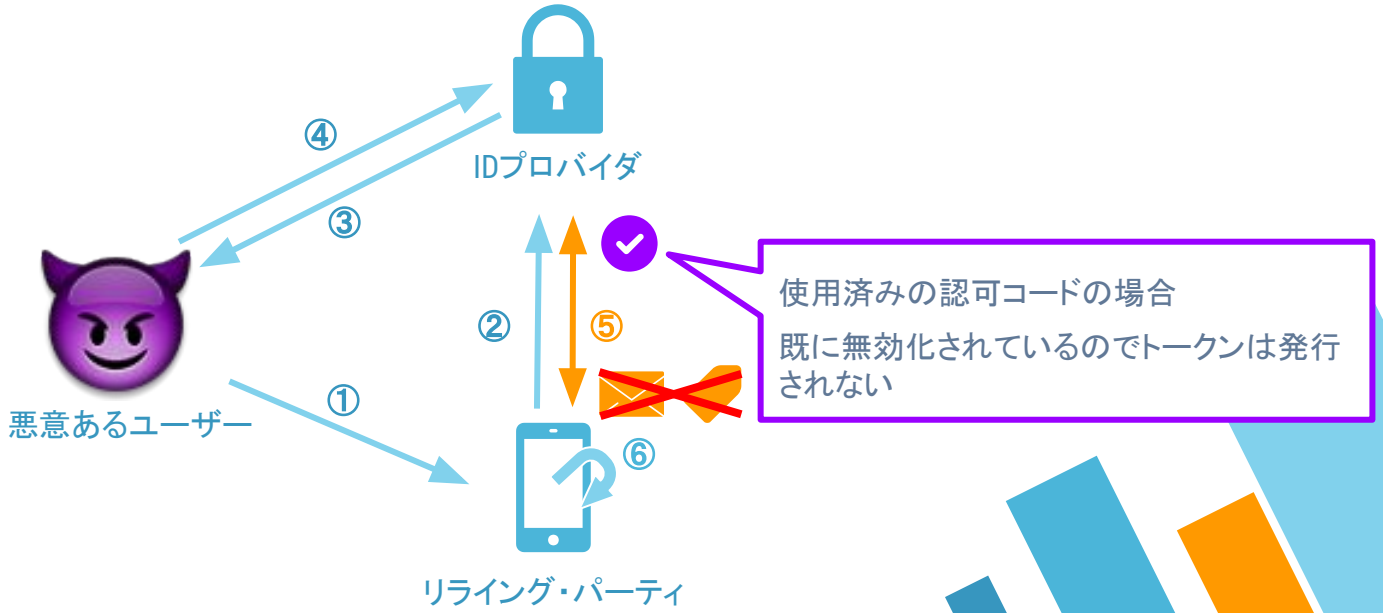


補足：リプレイアタック対策



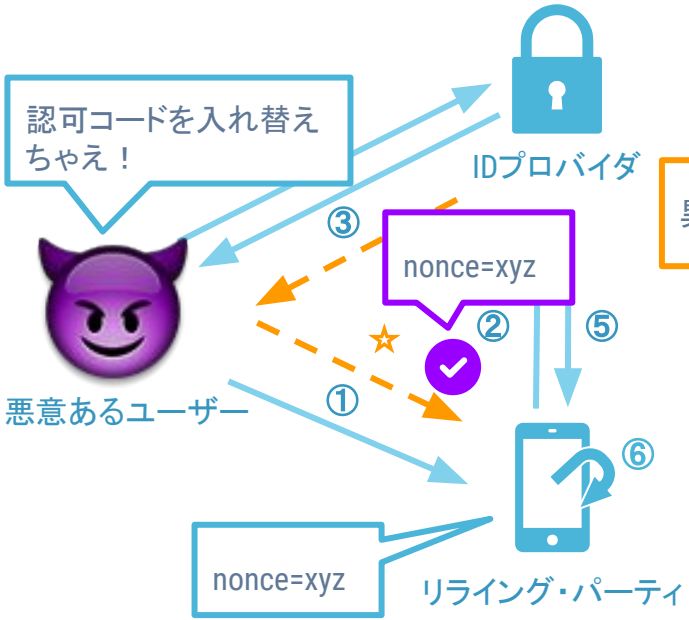
補足：リプレイアタック対策

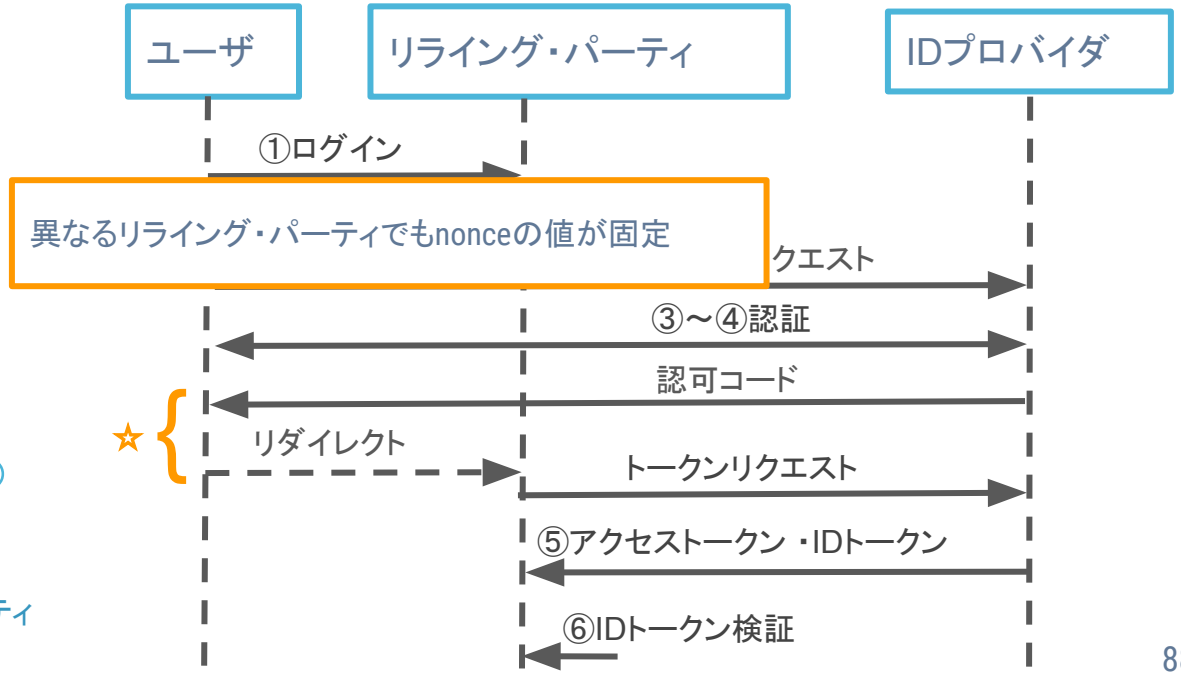
補足：リプレイアタック対策



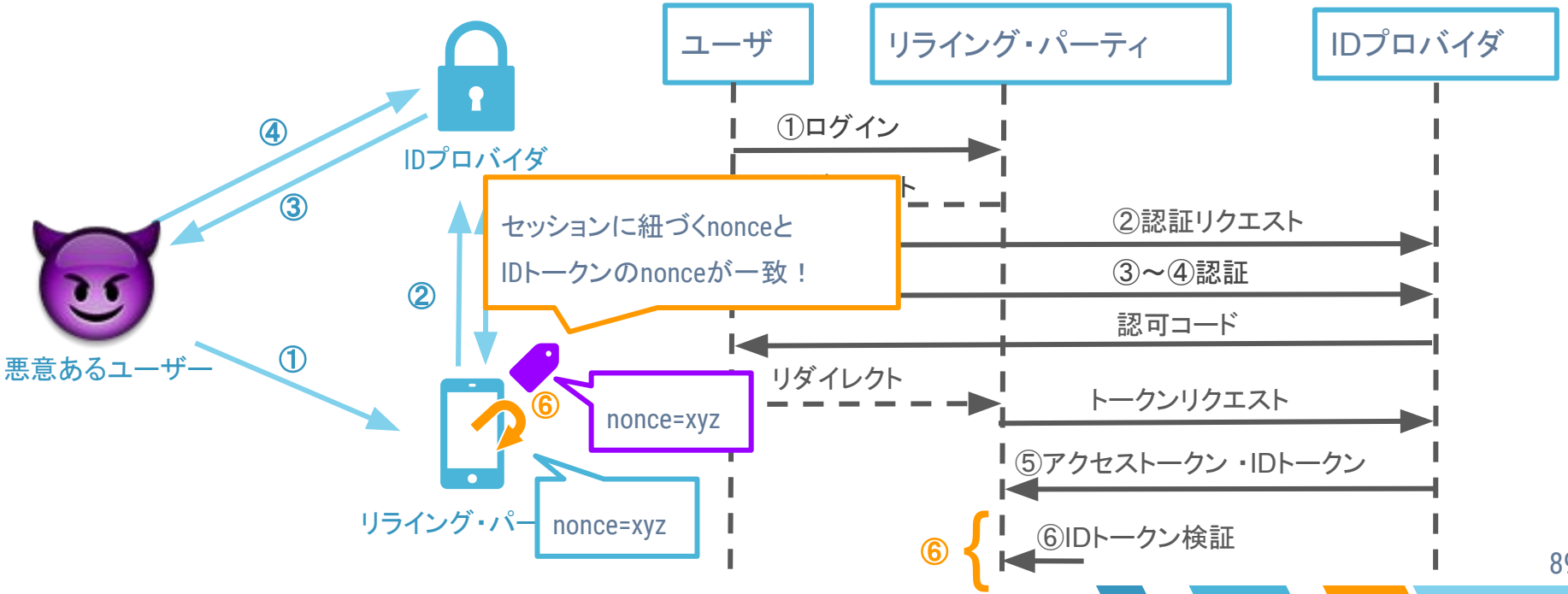
“アンチパターン”
nonceを固定値にした場合

認可コードを入れ替える場合

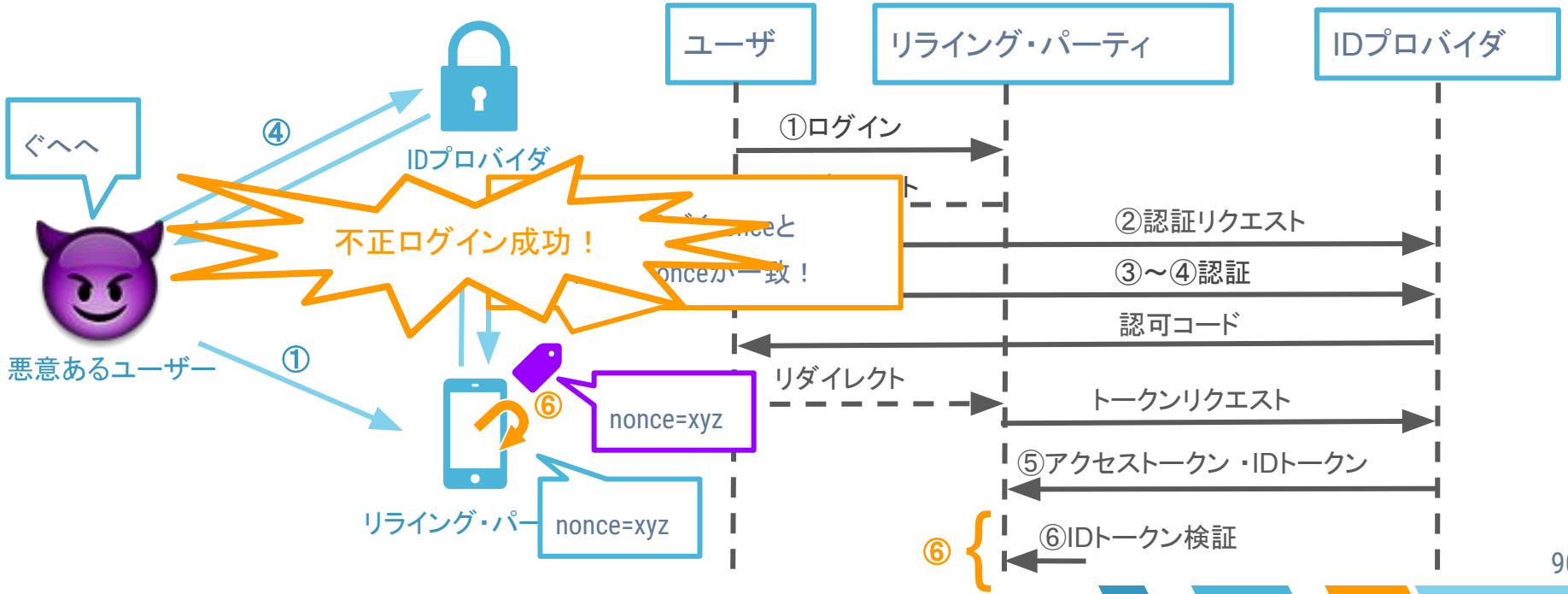




認可コードを入れ替える場合



認可コードを入れ替える場合



OpenID Connectまとめ

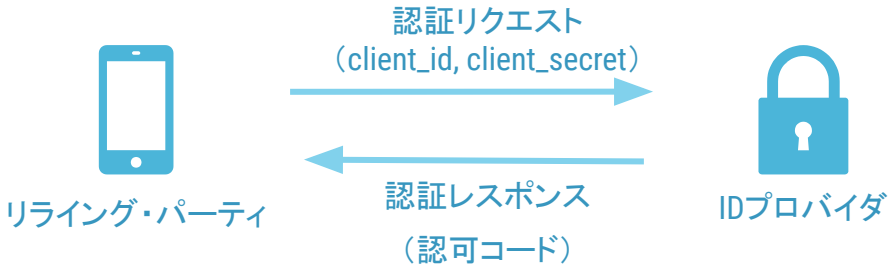
- **クライアント**が認証できるプロトコル
- IDトークンを使用して安全に認証する
- Nonceはランダムな文字列にし、必ず検証する
- 認可コードをワンタイムにすることでリプレイアタックを防ぐ

4.

OIDCとネイティブアプリ

ネイティブアプリでのOIDC

- client secretを秘匿に保てない
  - 認可コードフローだとトークン取得時に漏洩する
  - 自由に認可コードやトークンが発行可能に！
- 長期間有効なAPIアクセスを必要とする

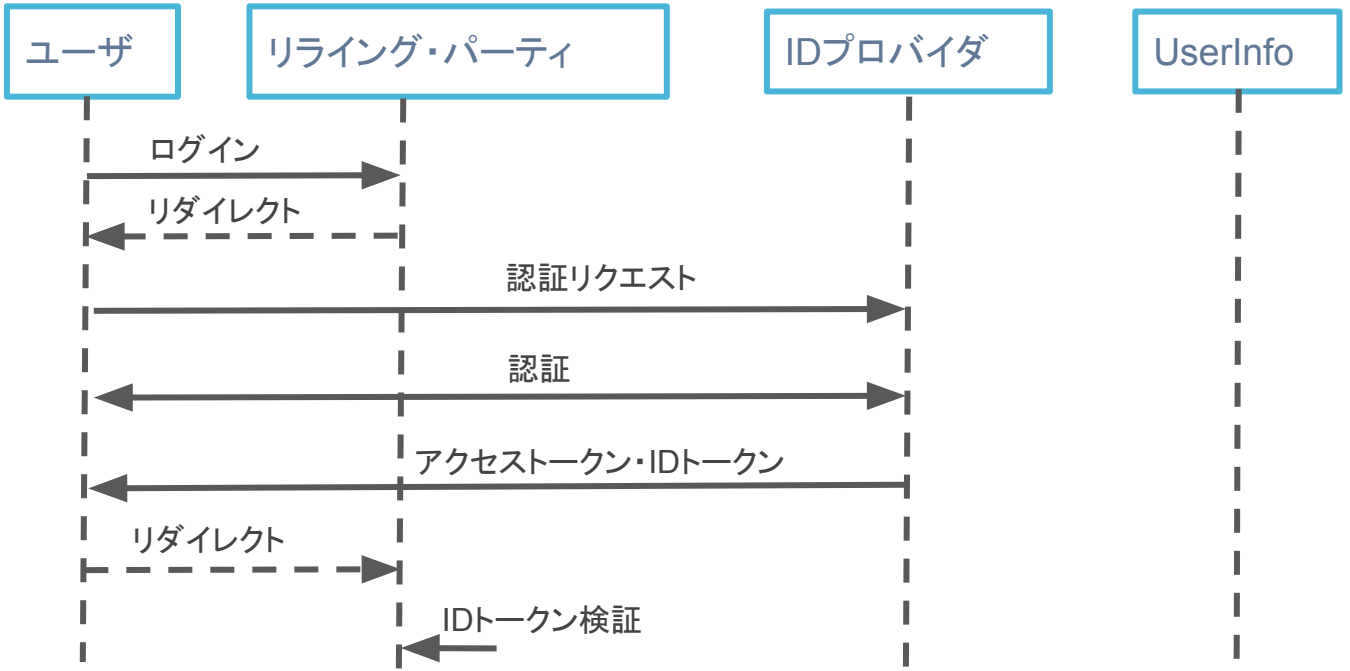


インプリシットフロー

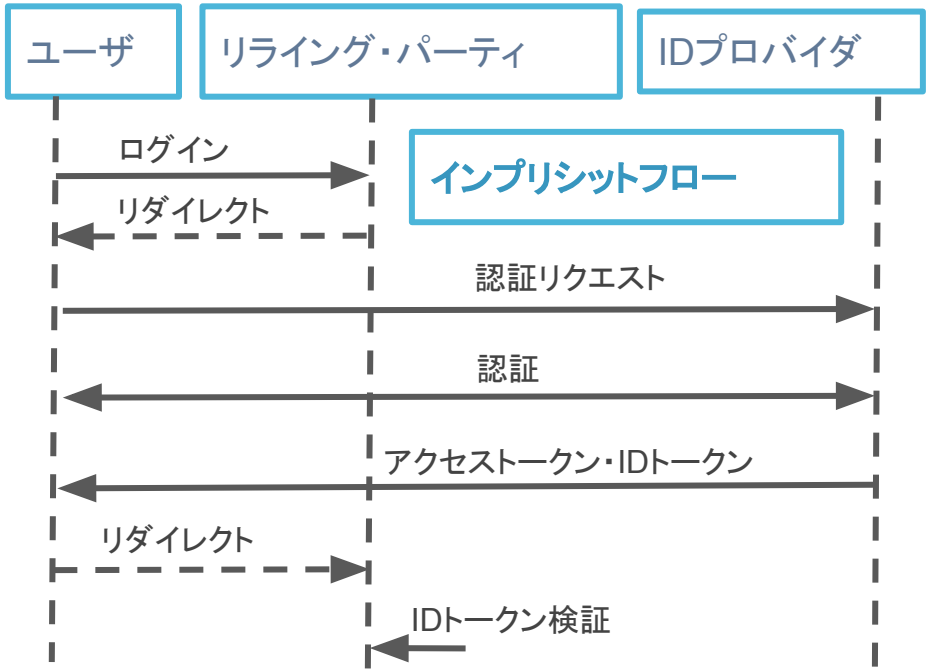
ClientSecretを安全に保存できないため、それを使用せず認可コードフローを簡略化したフロー

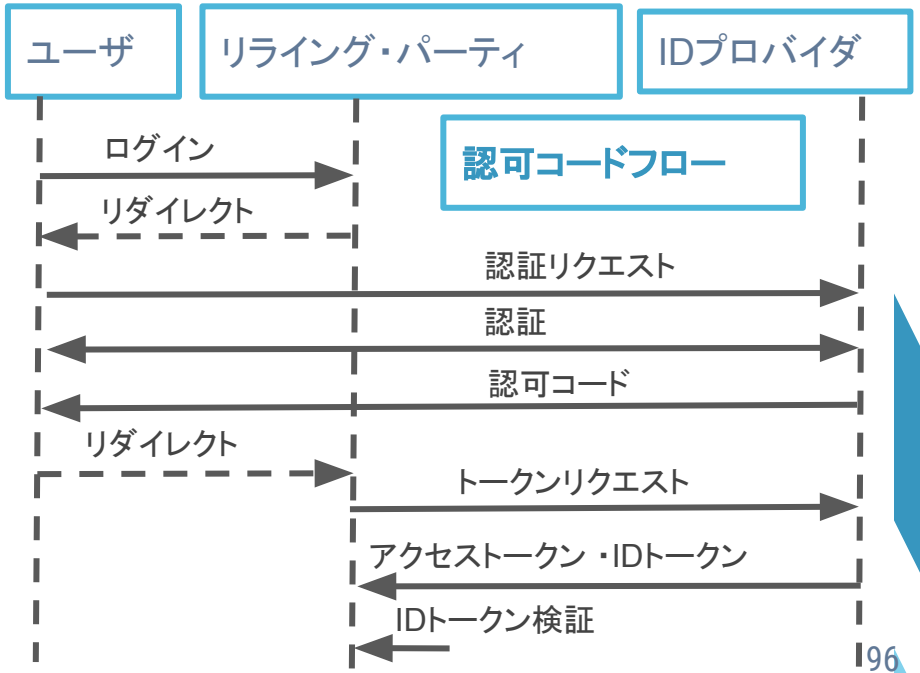
- クライアントの認証がIDプロバイダで行われない
  - client secretを使用せずにトークンの発行が可能
- リフレッシュトークンの発行が禁止
  - 再取得時にはインプリシットフローを再度行う必要がある

インプリシットフローのシーケンス図

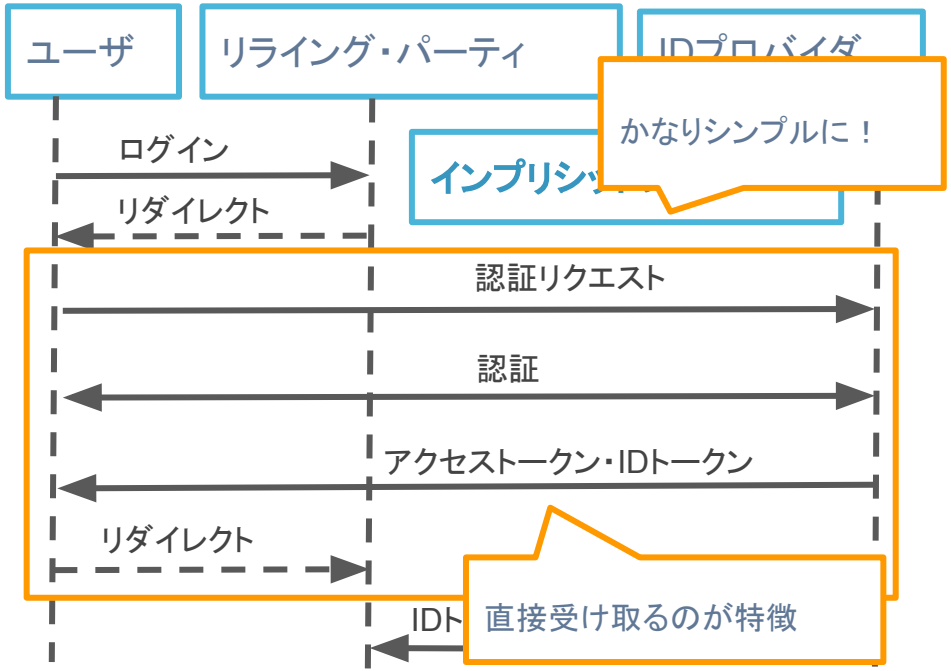


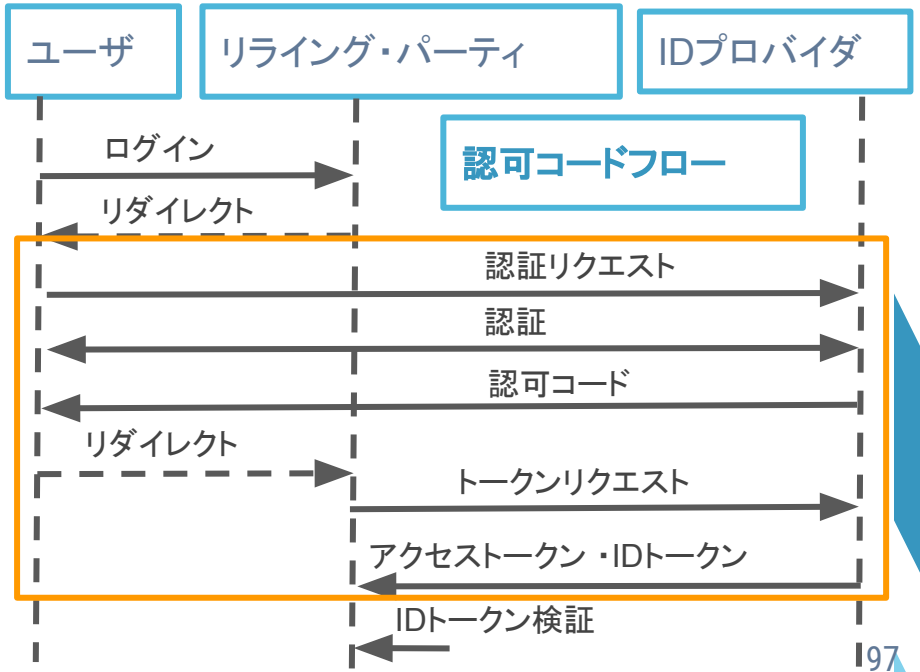
シーケンス図の比較


ユーザ
リライング・パーティ
IDプロバイダ
インプリシットフロー
ログイン
リダイレクト
認証リクエスト
認証
アクセストークン・IDトークン
リダイレクト
IDトークン検証


ユーザ
リライング・パーティ
IDプロバイダ
認可コードフロー
ログイン
リダイレクト
認証リクエスト
認証
認可コード
リダイレクト
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証

シーケンス図の比較


ユーザ
リライング・パーティ
IDプロバイダ
かなりシンプルに！
ログイン
リダイレクト
認証リクエスト
認証
アクセストークン・IDトークン
リダイレクト
直接受け取るのが特徴


ユーザ
リライング・パーティ
IDプロバイダ
認可コードフロー
ログイン
リダイレクト
認証リクエスト
認証
認可コード
リダイレクト
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証

インプリシットフローの問題点

リソースアクセスと併用する場合

- IdPによってはアクセストークンの有効期限が切れるたびに再認証が必要
  - 数分～数時間に一度ログイン画面が表示されてしまう

→どうしてもユーザー体験が下がってしまう

インプリシットフローの問題点

リソースアクセスと併用する場合

- IdPによってはアクセストークンの有効期限が切れるたびに再認証が必要
  - 数分～数時間に一度ログイン画面が表示されてしまう

→どうしてもユーザー体験が下がってしまう

安全にリフレッシュトークンが発行できればなー・・・

PKCE（ピクシー）

- OAuthの拡張仕様
- パブリッククライアントで認可コードフローを安全に行う仕組み
  - ClientSecretを安全に保存できないことへの対策
  - 認可コードの横取り攻撃を防ぐ
  - 認可コードフローなのでRefresh Tokenが取得可能

安全にリフレッシュトークンが取得できる！

パブリックとコンフィデンシャル

- パブリッククライアント
  - クライアントの認証情報（ClientID, ClientSecret）をセキュアに保存できないクライアント
  - ネイティブアプリやブラウザベースのアプリ
- コンフィデンシャルクライアント
  - クライアントの認証情報をセキュアに保存できるクライアント
  - サーバーなど

認可コード横取り攻撃

- 以下のような場合、開発者が入れ替わりを防ぐのは困難
  - 悪意のあるアプリが同時にインストールされている
  - 悪意あるアプリで同じカスタムスキームが設定されている
  - 悪意あるアプリがクライアント IDを知っている

認可コード横取り攻撃

- 以下のような場合、開発者が入れ替わりを防ぐのは困難
  - 悪意のあるアプリが同時にインストールされている
  - 悪意あるアプリで同じカスタムスキームが設定されている
  - 悪意あるアプリがクライアント IDを知っている

PKCEでの対策

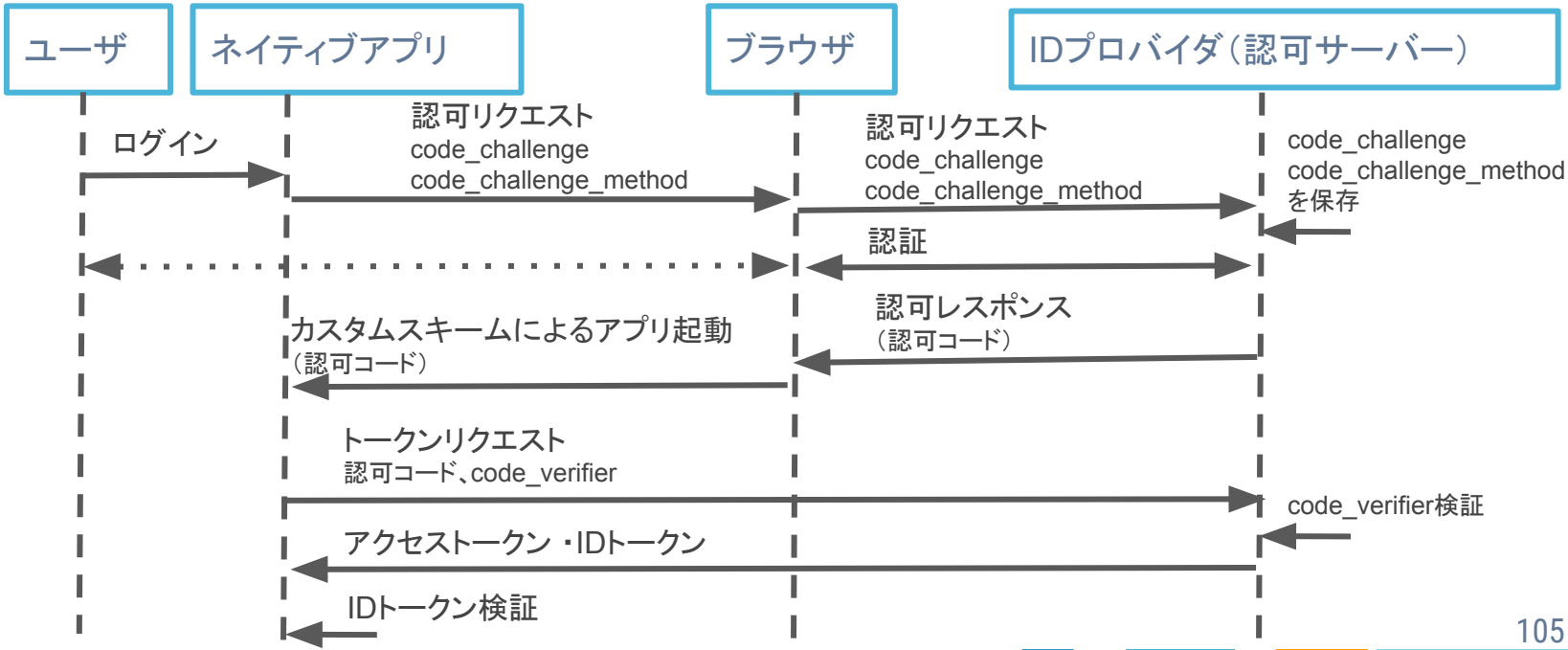
入れ替わりが発生した場合、アクセストークンを発行させない

PKCE（ピクシー）での対策

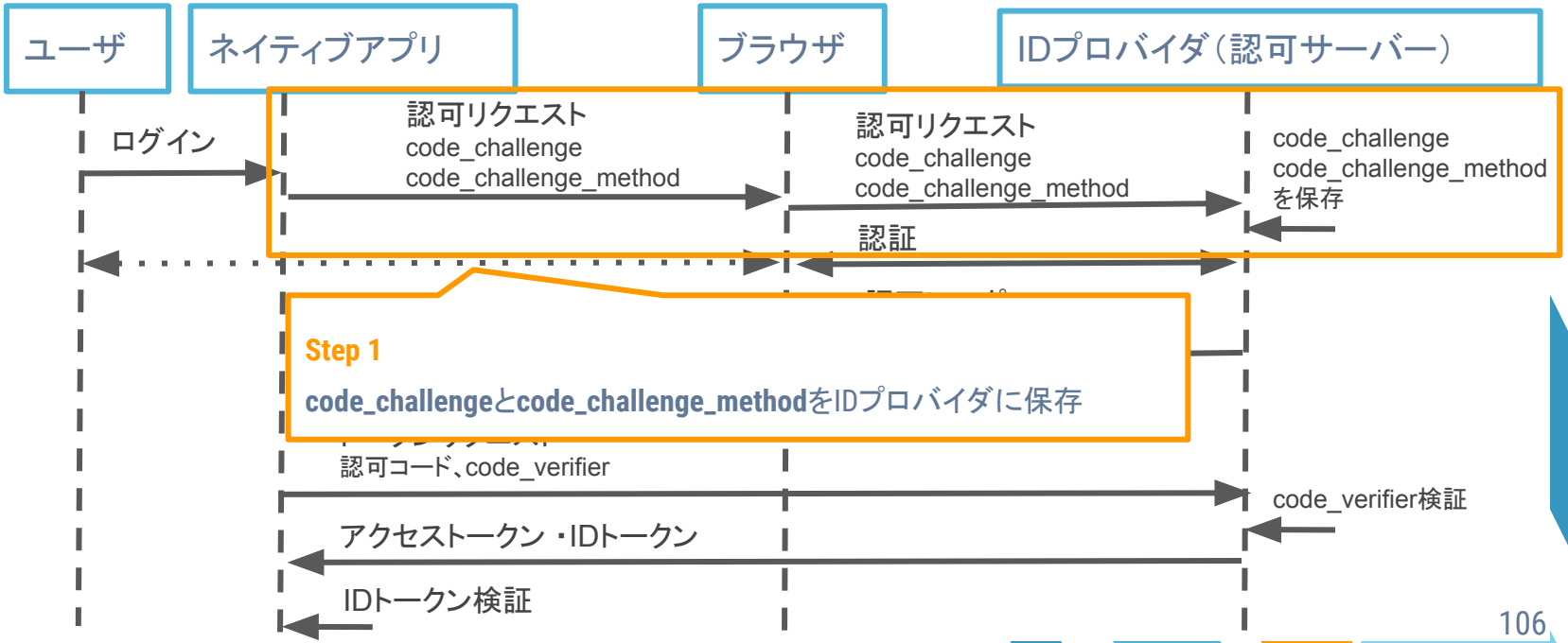
以下の三つの値を使用して対策

- code_verifier
  - ランダムな文字列
- code_challenge
  - code_verifierにcode_challenge_methodを適応して算出した値
- code_challenge_method
  - plainまたはS256

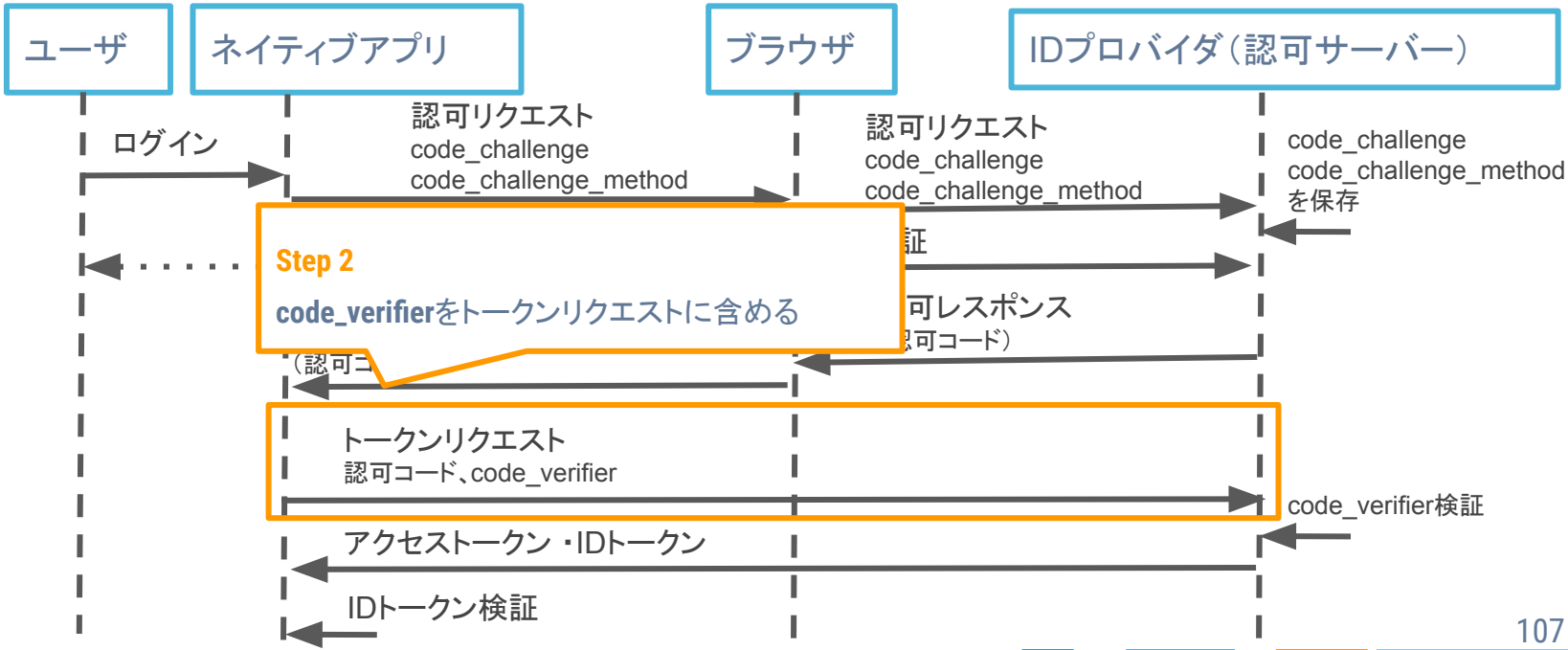
認可コードフロー+PKCEのシーケンス図



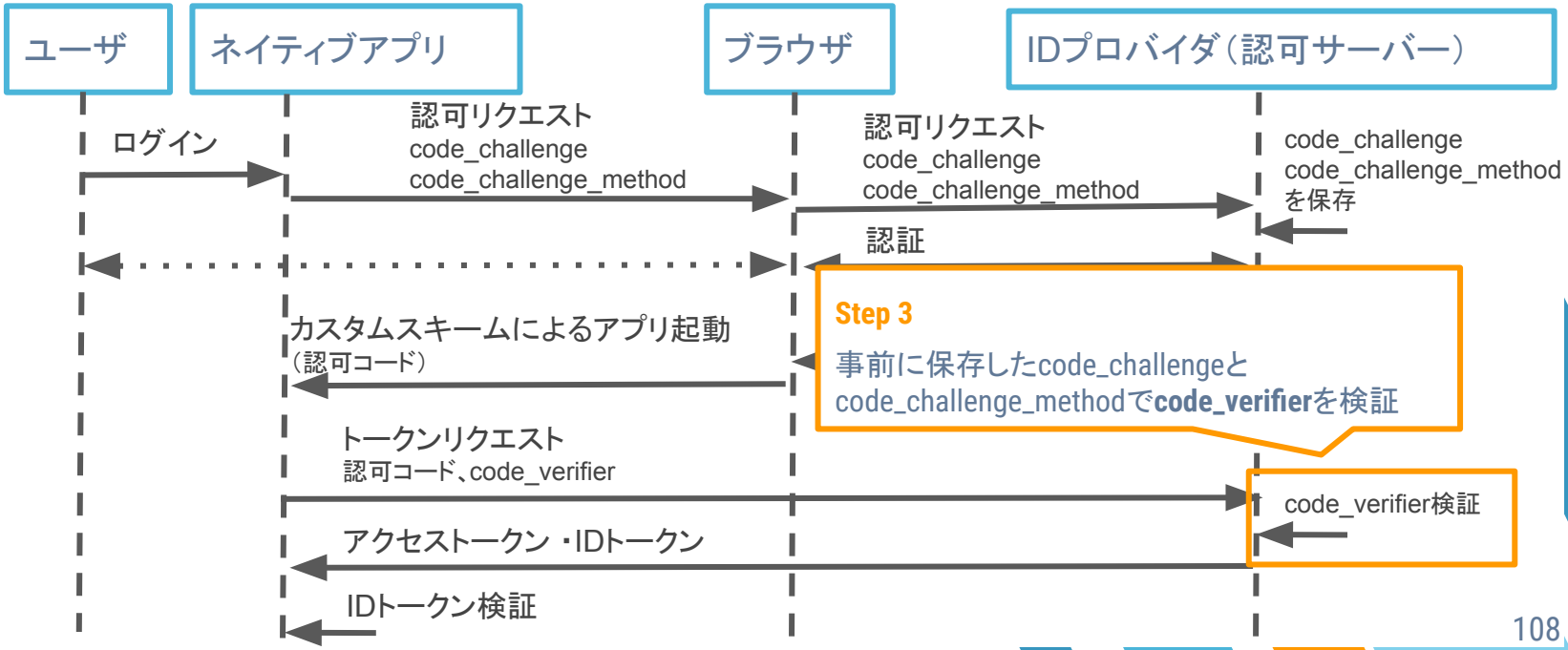
認可コードフロー+PKCEのシーケンス図



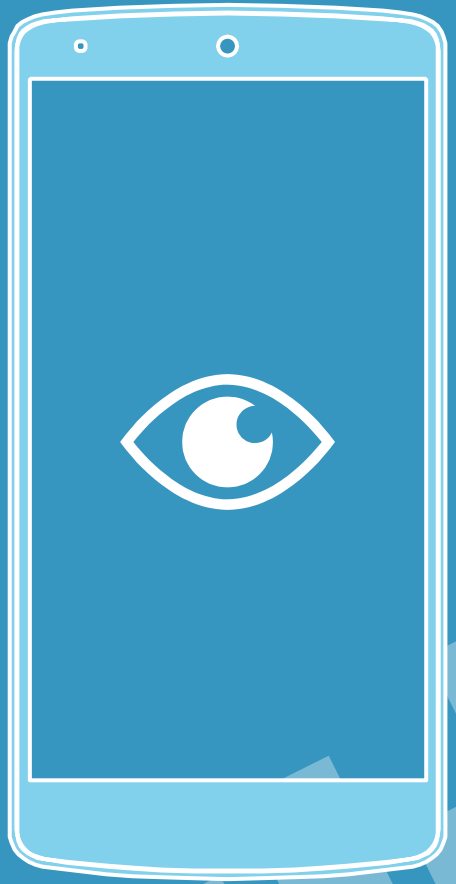
認可コードフロー+PKCEのシーケンス図



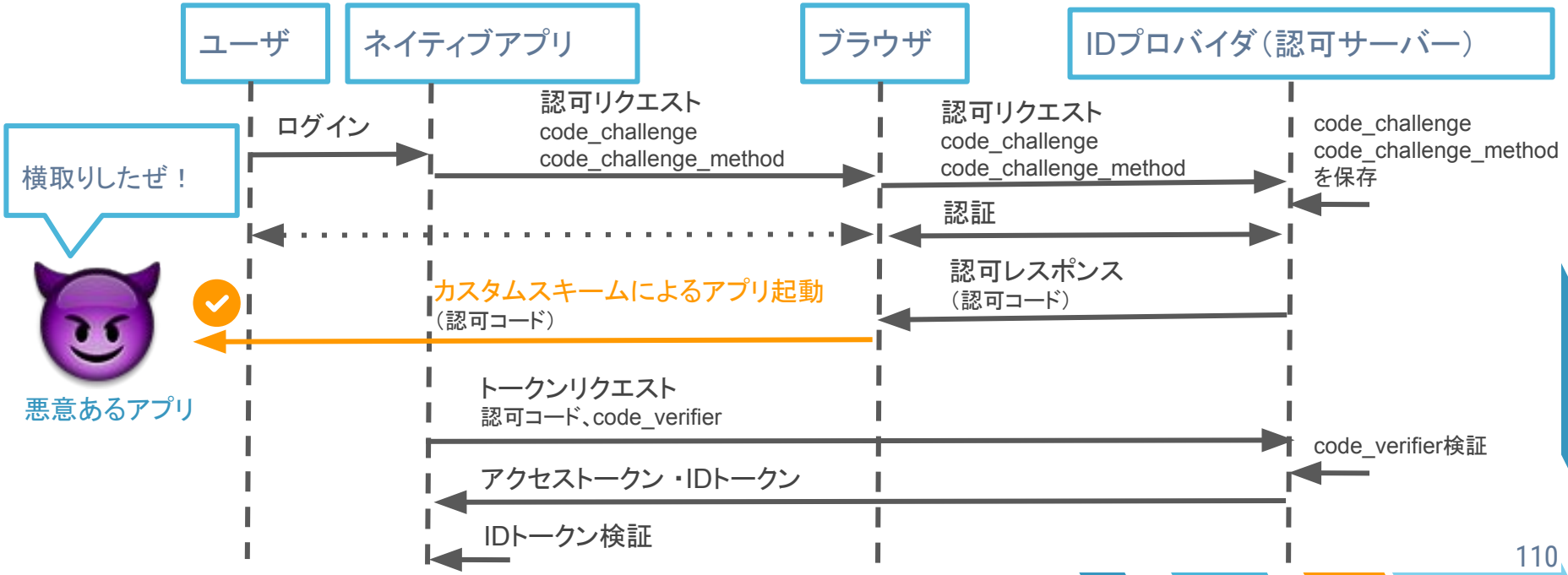
認可コードフロー+PKCEのシーケンス図



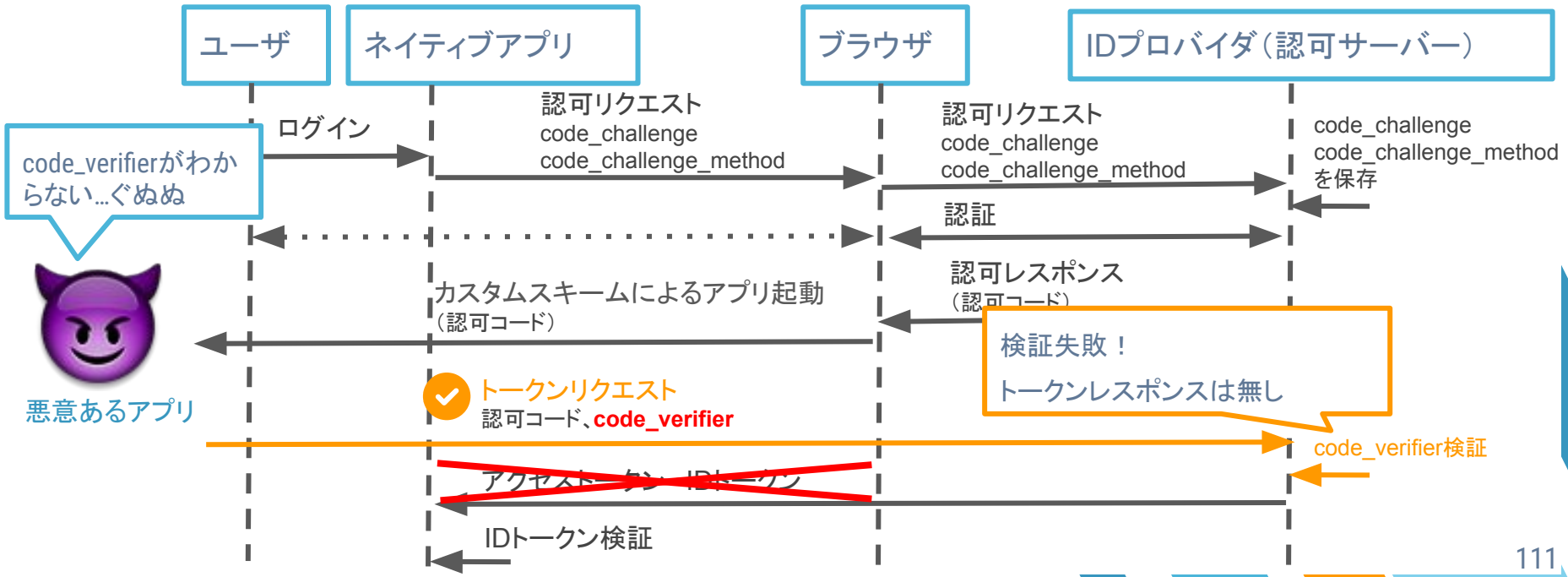
認可コード
横取り攻撃を
受けた場合



認可コード横取り攻撃を受けた場合



認可コード横取り攻撃を受けた場合



OIDCとネイティブアプリのまとめ

- client secretを保存するのは厳禁
- 認証だけでいいならインプリシットフロー（非推奨）
- UXを考えるなら**認証コードフロー+PKCE**（推奨）

Tips

OAuth2.1ではインプリシットフローはなくなるので注意

5.

OIDCとSPA

SPAでのOIDC

SPA（ブラウザ）では安全にアクセストークンやIDトークンが保存できない

- Web Storage(localStorageやsessionStorage)に保存
  - 自分のアプリでXSS対策をしても他のJSが悪さをすると台無し
- iFrameでIdPとのセッションをメモリ上に格納
  - サードパーティCookieを保持するのはNGな流れになってきた

→今のところベストプラクティスはない

SPAでのOIDC

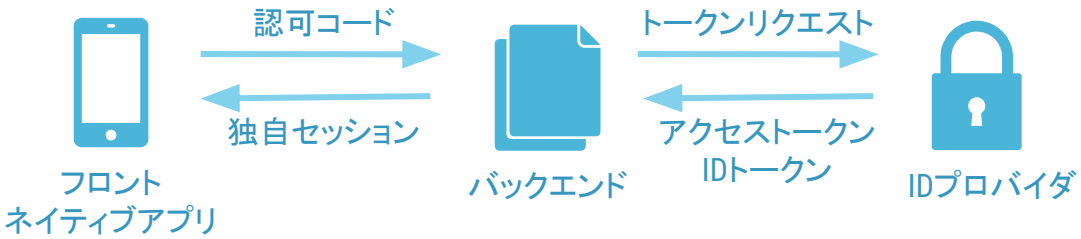
二つの対策

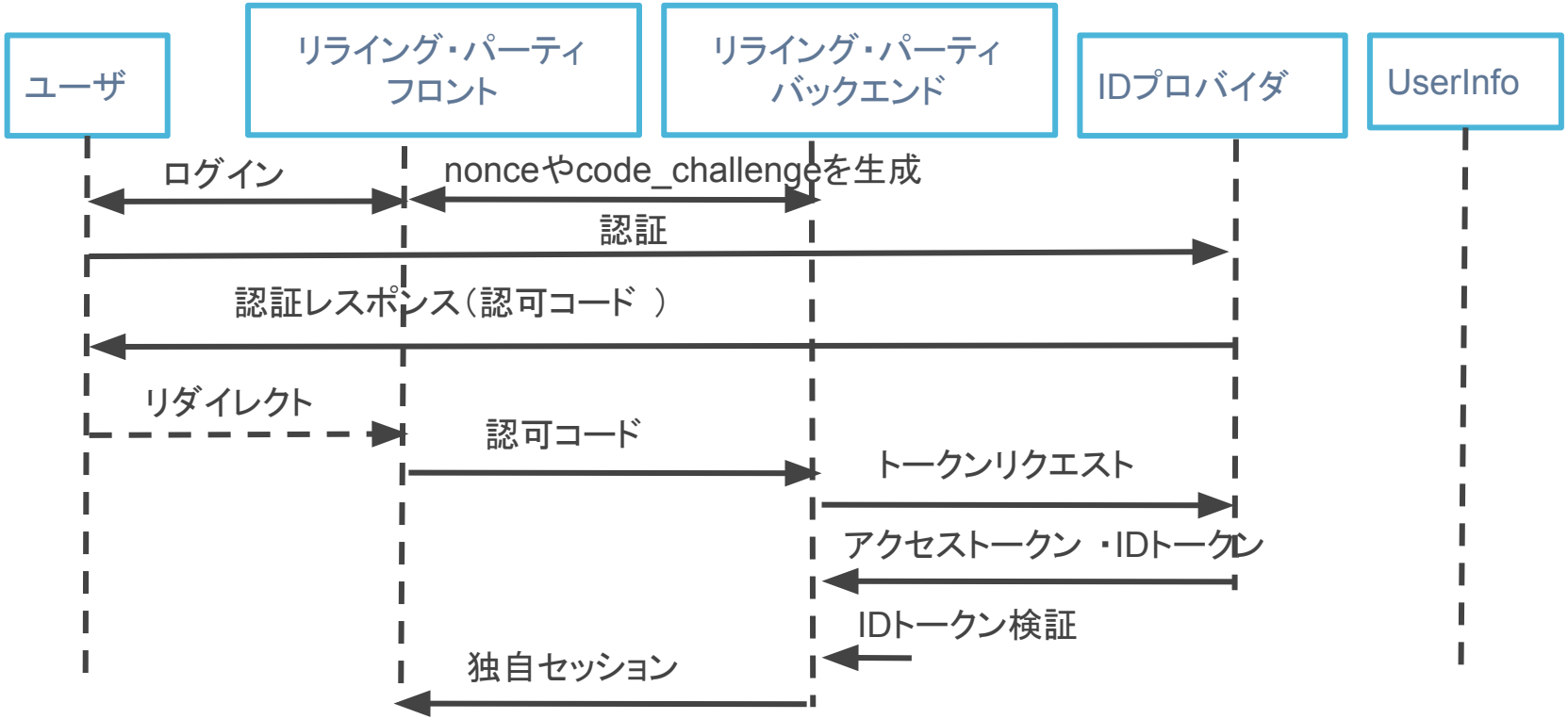
- 認可コードフロー＋独自のセッション
- RefreshTokenをローテーションする（Auth0など）

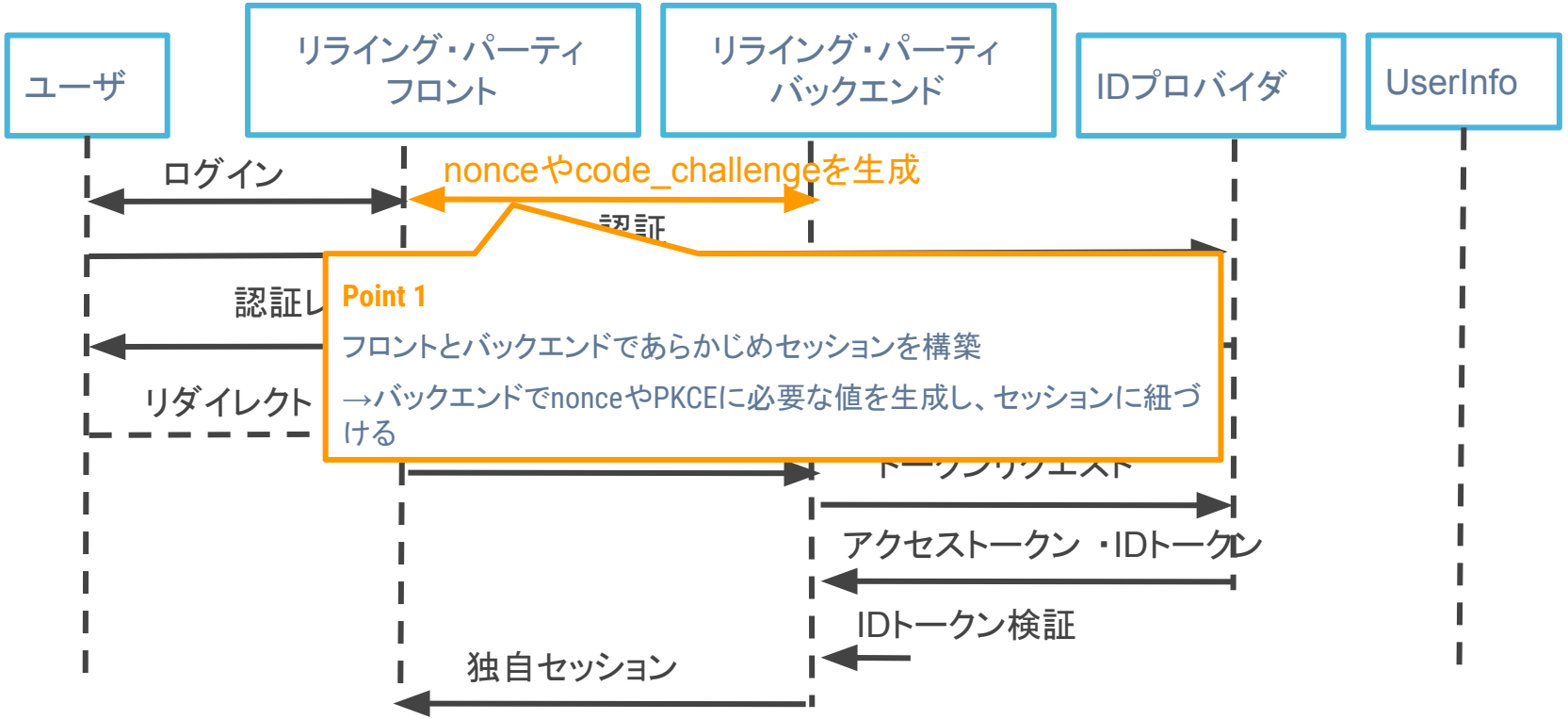

認可コードフロー+独自セッション

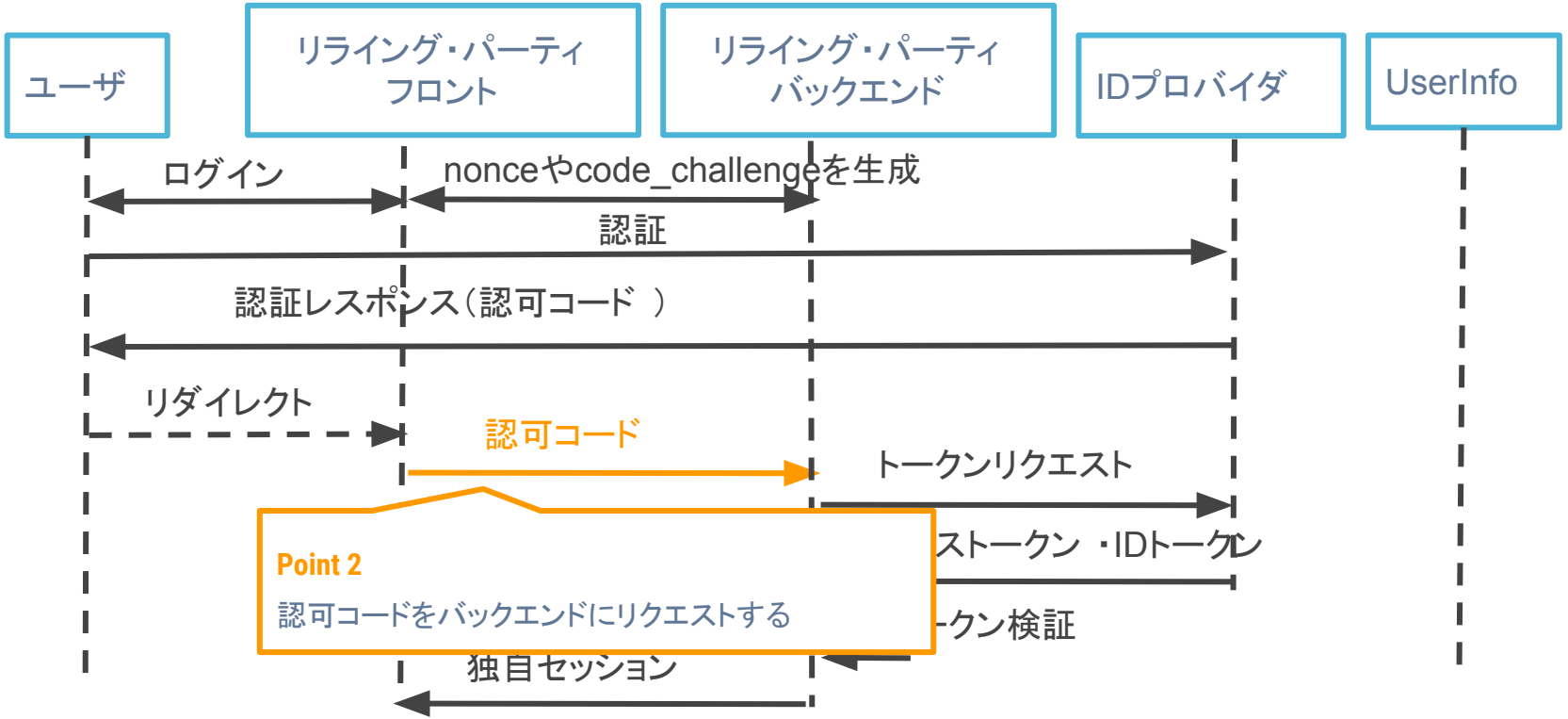
認可コードフロー+独自セッション

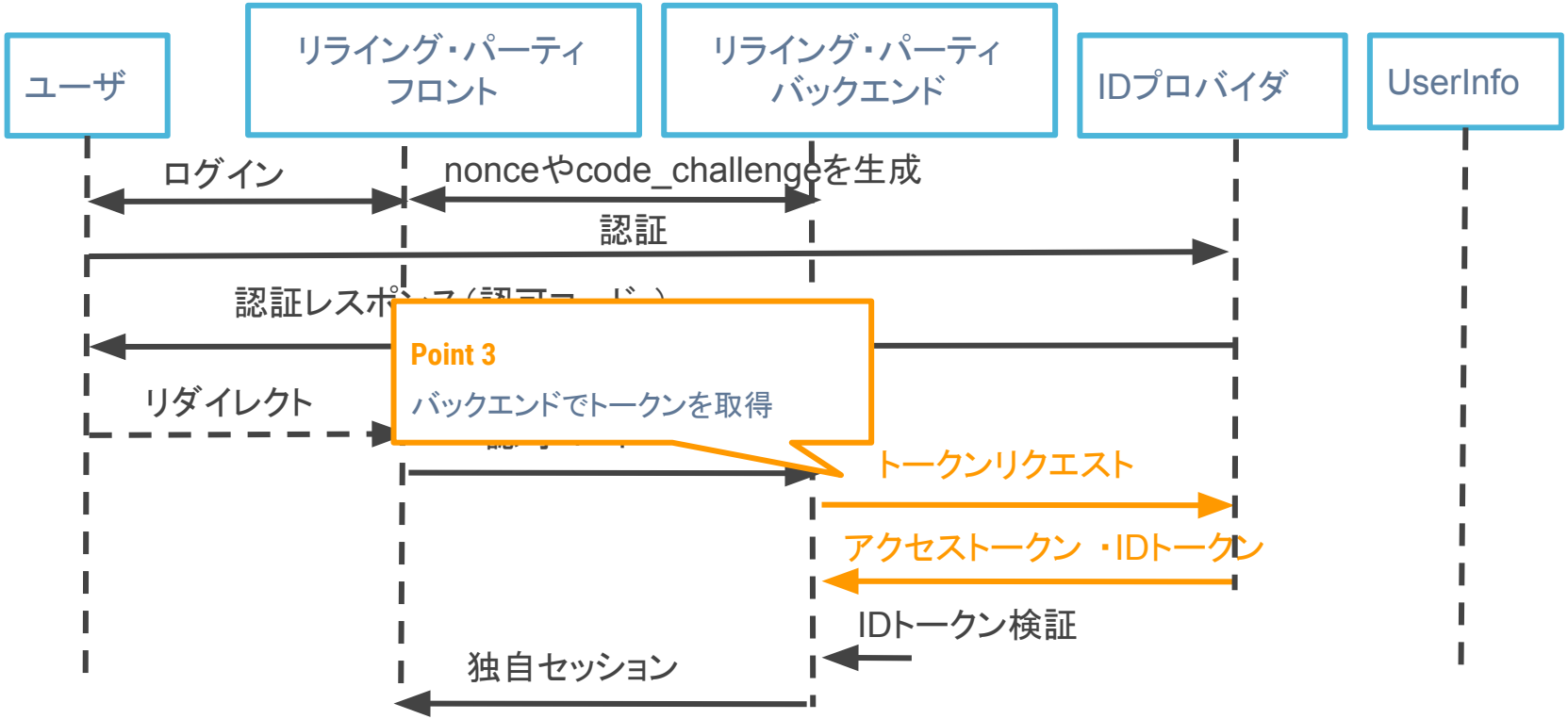
- **OIDCをログイン管理に使用しない**
  - 認証のみに使用する
  - 独自のセッション（cookieなど）でログイン管理を行う

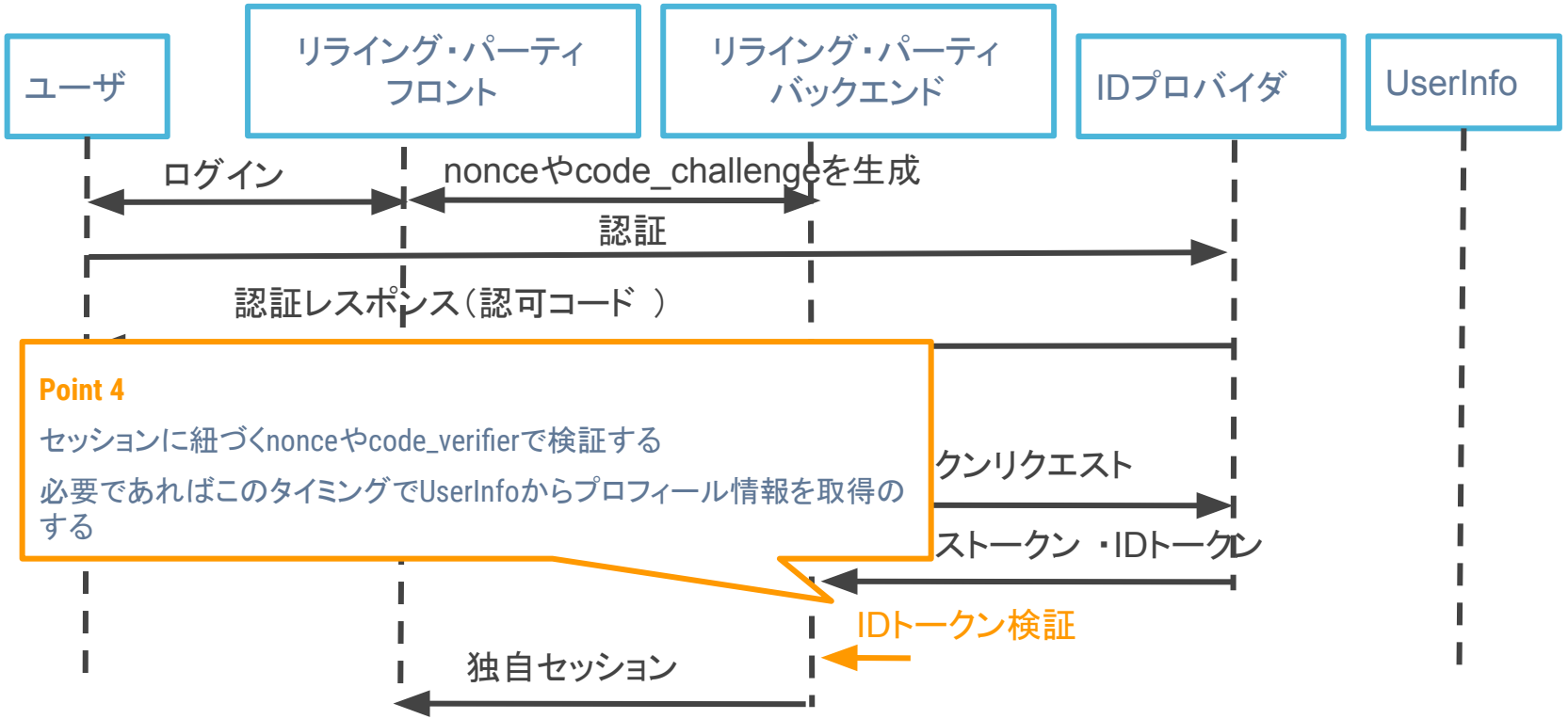


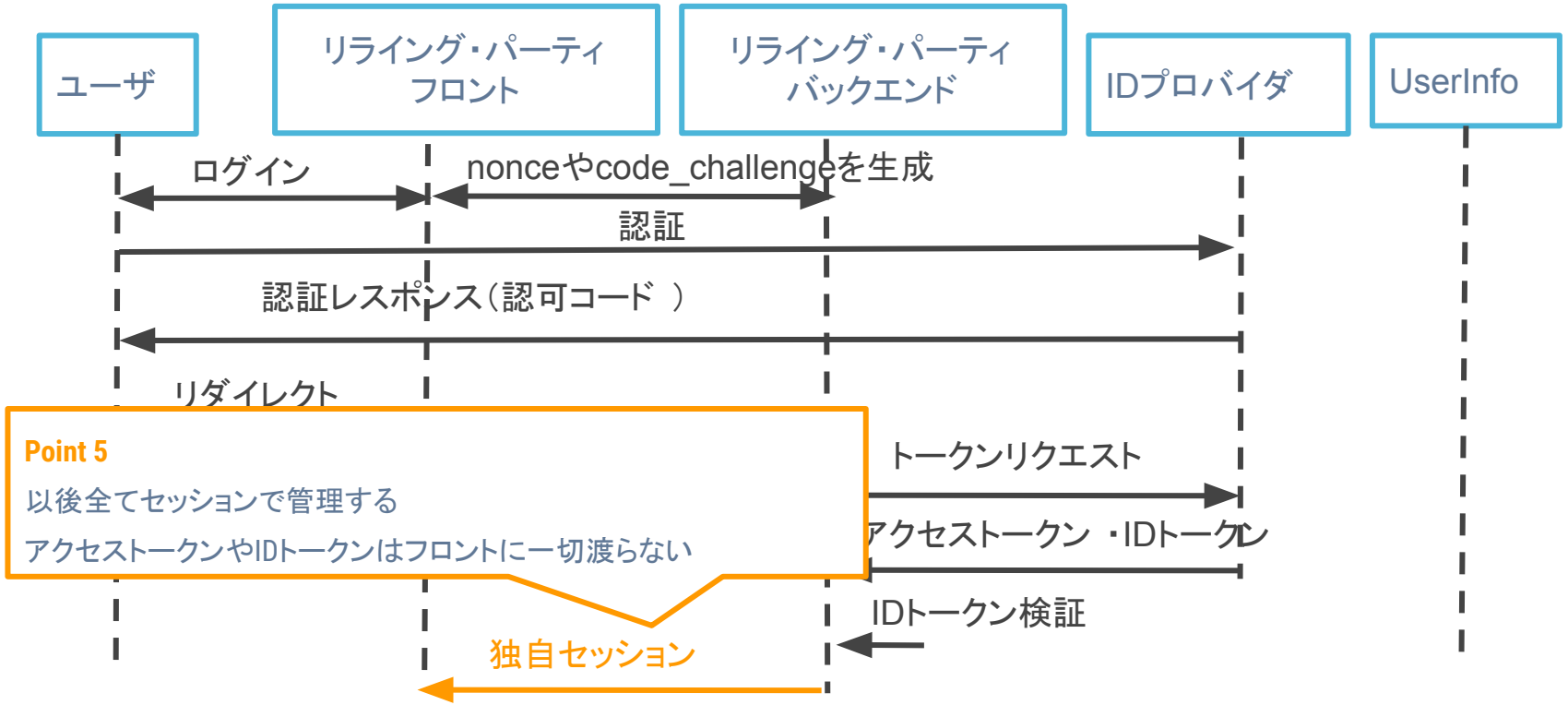

ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レスポンス（認可コード ）
リダイレクト
認可コード
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
独自セッション


ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レ
Point 1
フロントとバックエンドであらかじめセッションを構築
→バックエンドでnonceやPKCEに必要な値を生成し、セッションに紐づける
リダイレクト
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
独自セッション


ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レスポンス（認可コード ）
リダイレクト
認可コード
トークンリクエスト
ストークン ・IDトークン
ークン検証
独自セッション
Point 2
認可コードをバックエンドにリクエストする


ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レスポンス（認可コード）
Point 3
バックエンドでトークンを取得
リダイレクト
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
独自セッション


ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レスポンス（認可コード ）
Point 4
セッションに紐づくnonceやcode_verifierで検証する
必要であればこのタイミングでUserInfoからプロフィール情報を取得のする
クンリクエスト
ストークン ・IDトークン
IDトークン検証
独自セッション


ユーザ
リライング・パーティ
フロント
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
nonceやcode_challengeを生成
認証
認証レスポンス（認可コード ）
リダイレクト
Point 5
以後全てセッションで管理する
アクセストークンやIDトークンはフロントに一切渡らない
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
独自セッション

認可コードフロー+独自セッション

まとめ

- フロントにトークン類を保持しないのでセキュア
  - セッションにユーザーIDなどの識別子を含めないように注意
  - CSRF対策も忘れずに
- SPAとネイティブアプリを同時に使用する場合はバックエンドの実装を分ける必要がある
- 実装が割と手間

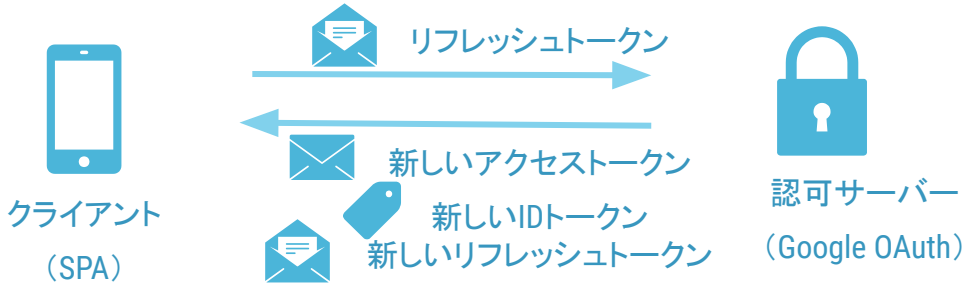
RefreshTokenのローテーション

RefreshTokenをローテーションする

- RefreshTokenを使用して新しいトークンを発行する時
  - →古いRefreshTokenは無効化&新しいRefreshTokenを発行
    - OAuthの仕様だと自動では無効化されない
- 古いRefreshTokenでアクセスされた場合、全てのRefreshTokenを無効化
- IdP側に仕組みが必要（Auth0などがサポート）

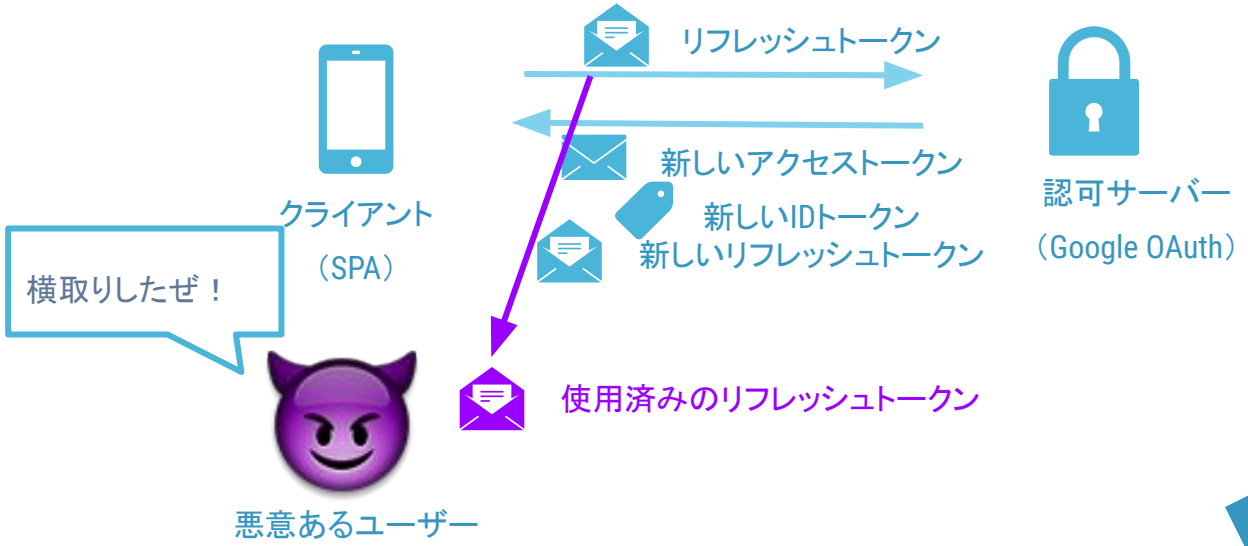
RefreshTokenのローテーション

通常のトークンリフレッシュ



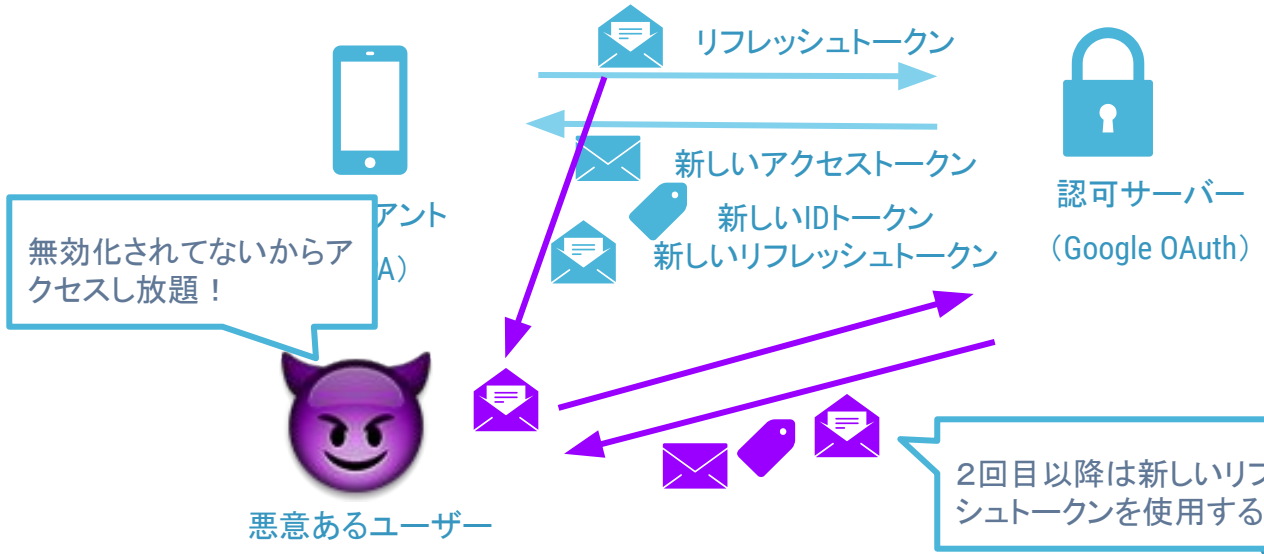
RefreshTokenのローテーション

通常のトークンリフレッシュ



RefreshTokenのローテーション

通常のトークンリフレッシュ

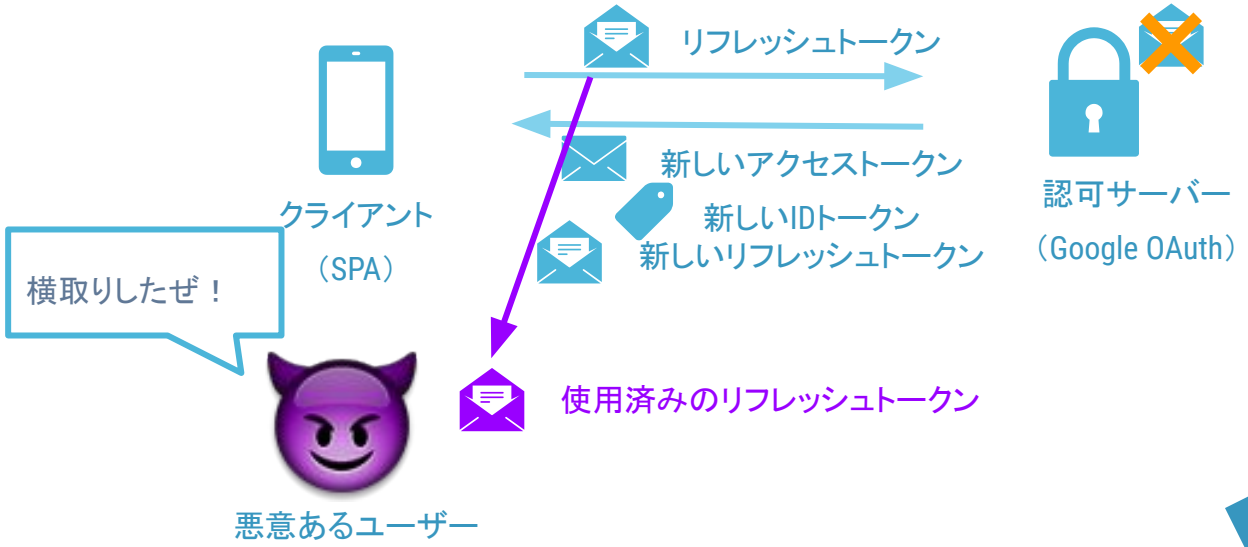


RefreshTokenのローテーション

Auth0のトークンリフレッシュ

RefreshTokenのローテーション

Auth0のトークンリフレッシュ



RefreshTokenのローテーション

Auth0のトークンリフレッシュ

RefreshTokenのローテーション

Auth0のトークンリフレッシュ

RefreshTokenのローテーション

まとめ

- リフレッシュトークンを盗難されても被害を最小限にとどめる
- IdP側での実装が必要
- **トークンを盗まれるの自体は防げない**
  - トークンの有効期限を短くすることで対策

OIDCとSPAのまとめ

- SPAでトークンを安全に保管する方法は今のところ存在しない
- OIDCを認証として使用し、独自のセッションでログイン管理を行う（推奨）
- RefreshTokenをローテーションする（Auth0など）

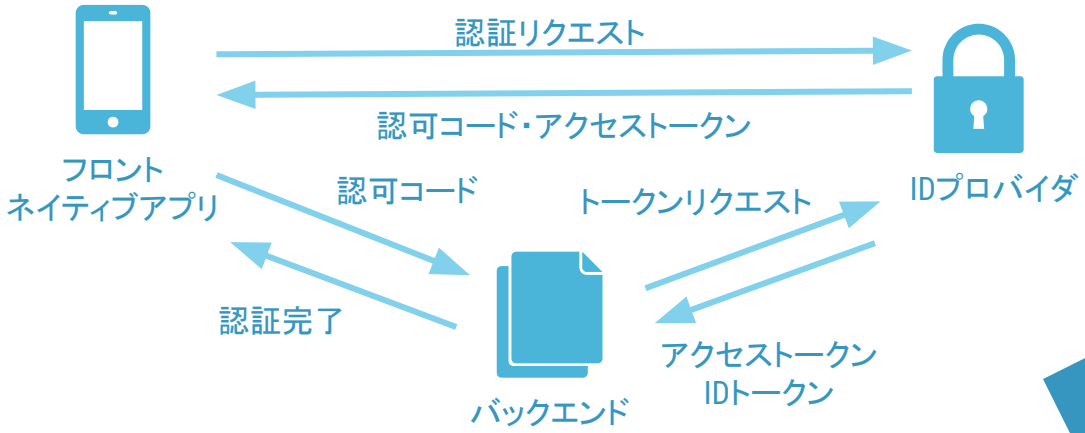
6.

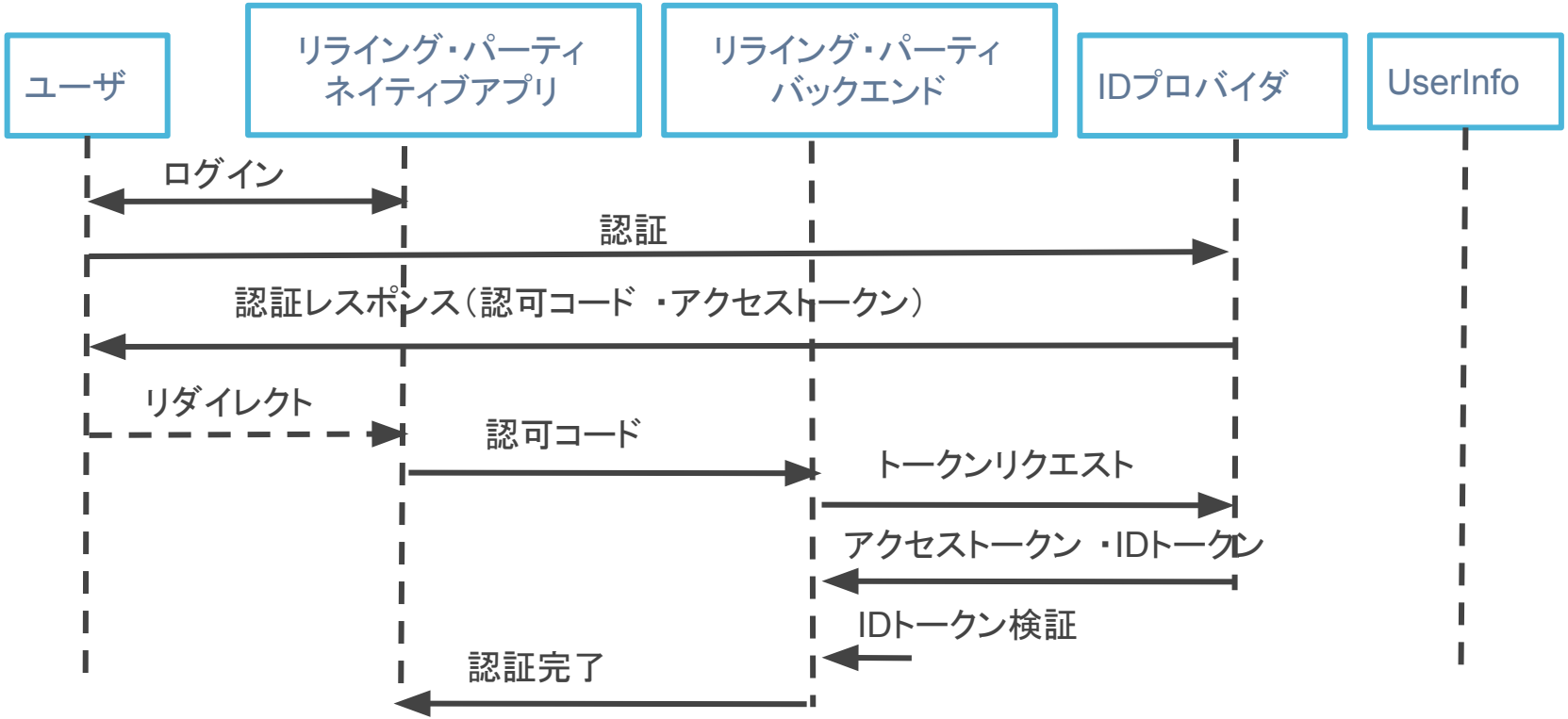
おまけ

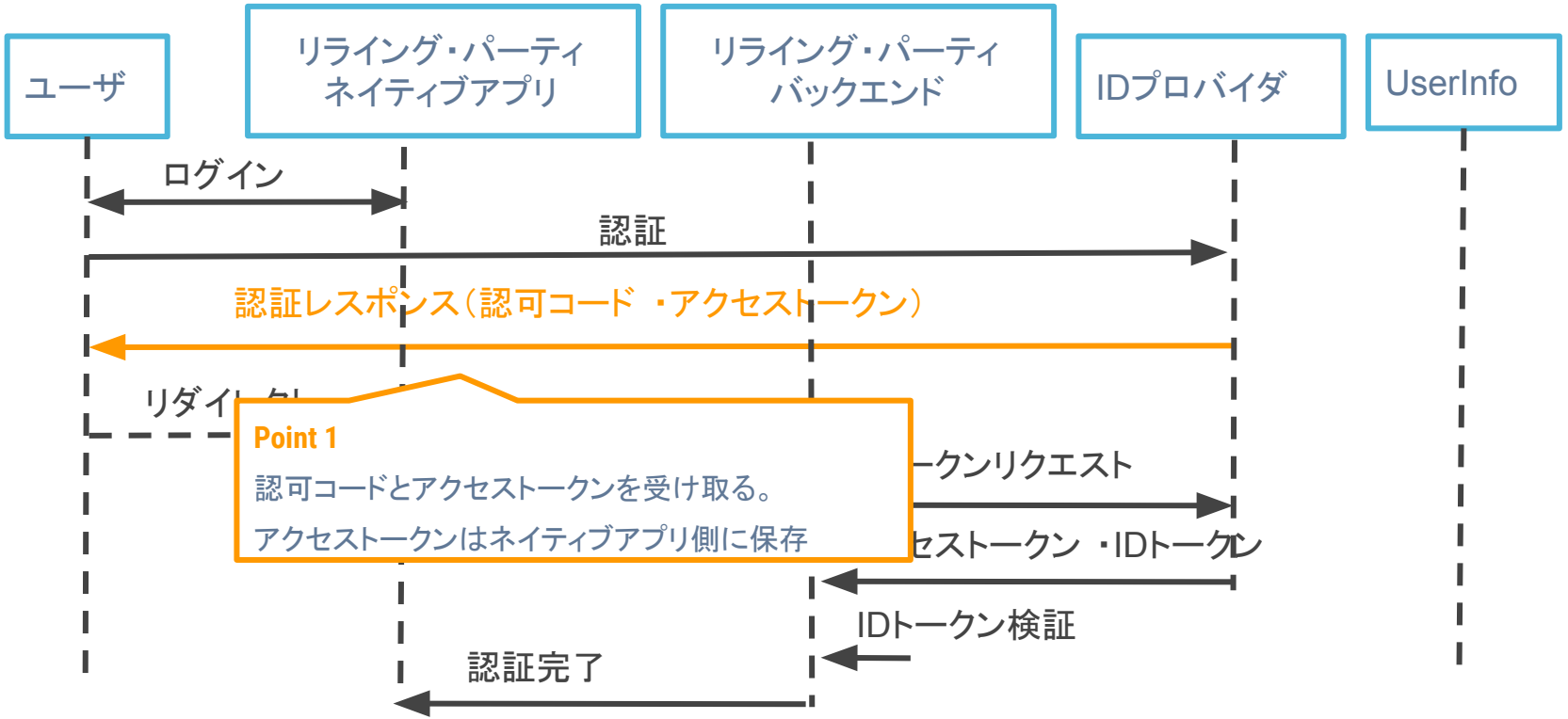
ハイブリッドフロー

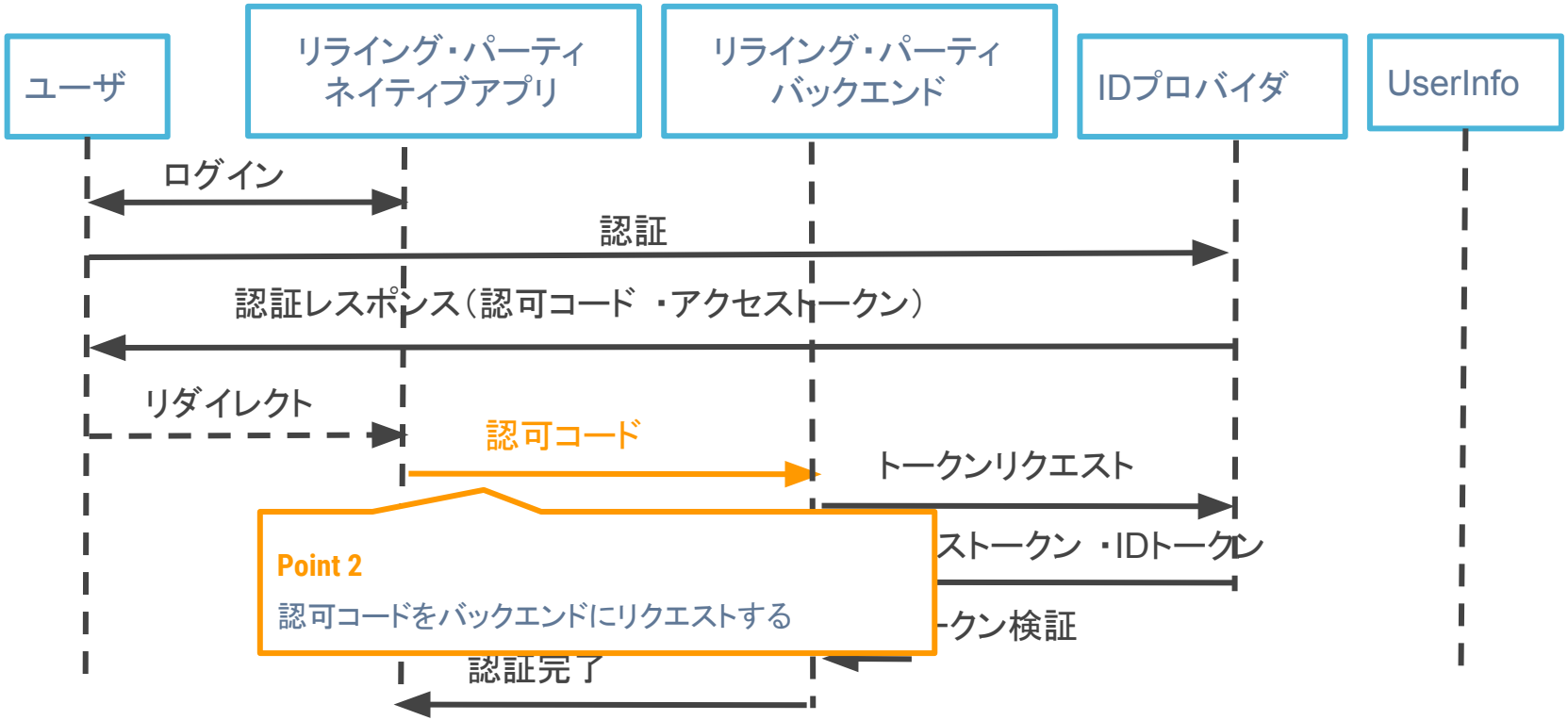
ハイブリッドフロー

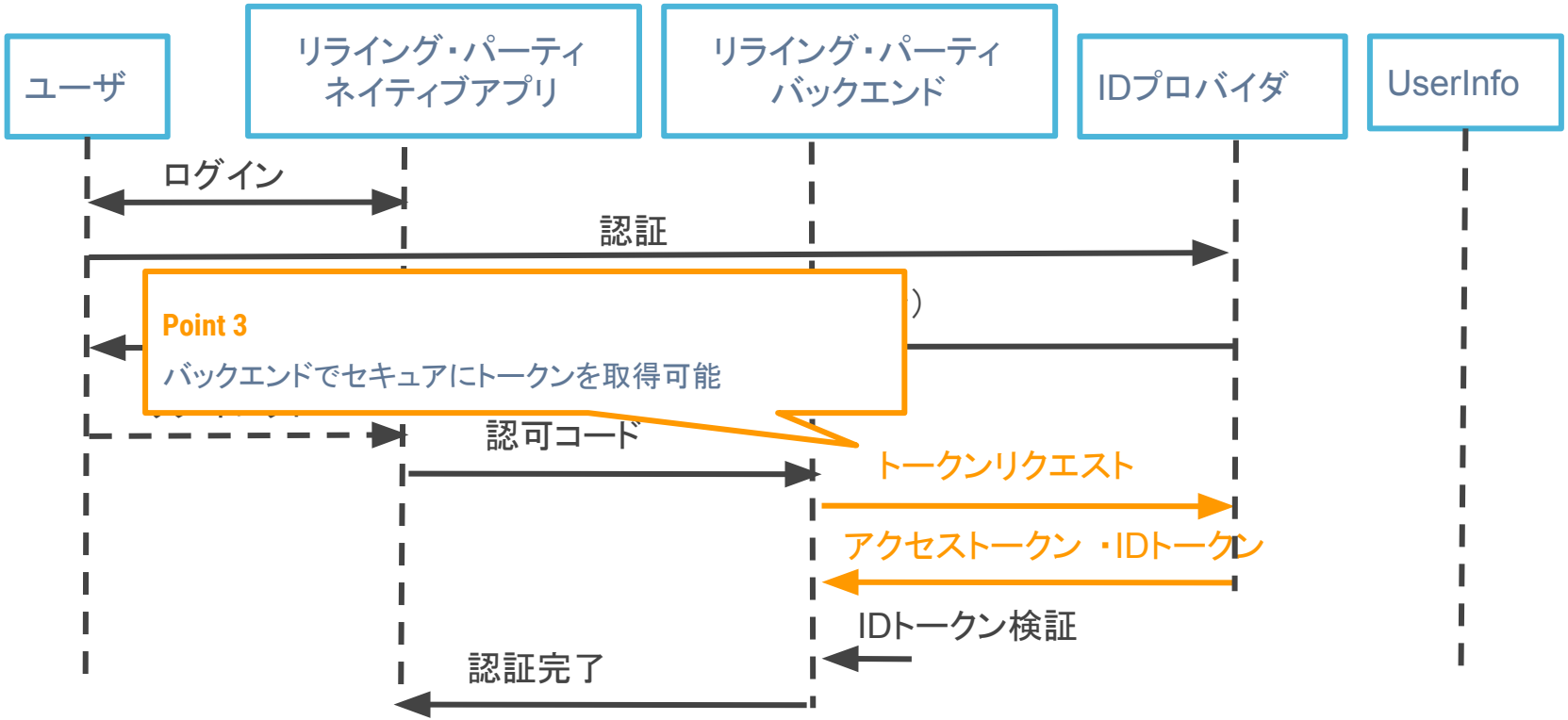
- ハイブリッドフロー
  - 認可コードフローとインプリシットフローを合わせたもの
  - 認可レスポンスとトークンレスポンスの両方があるのが特徴

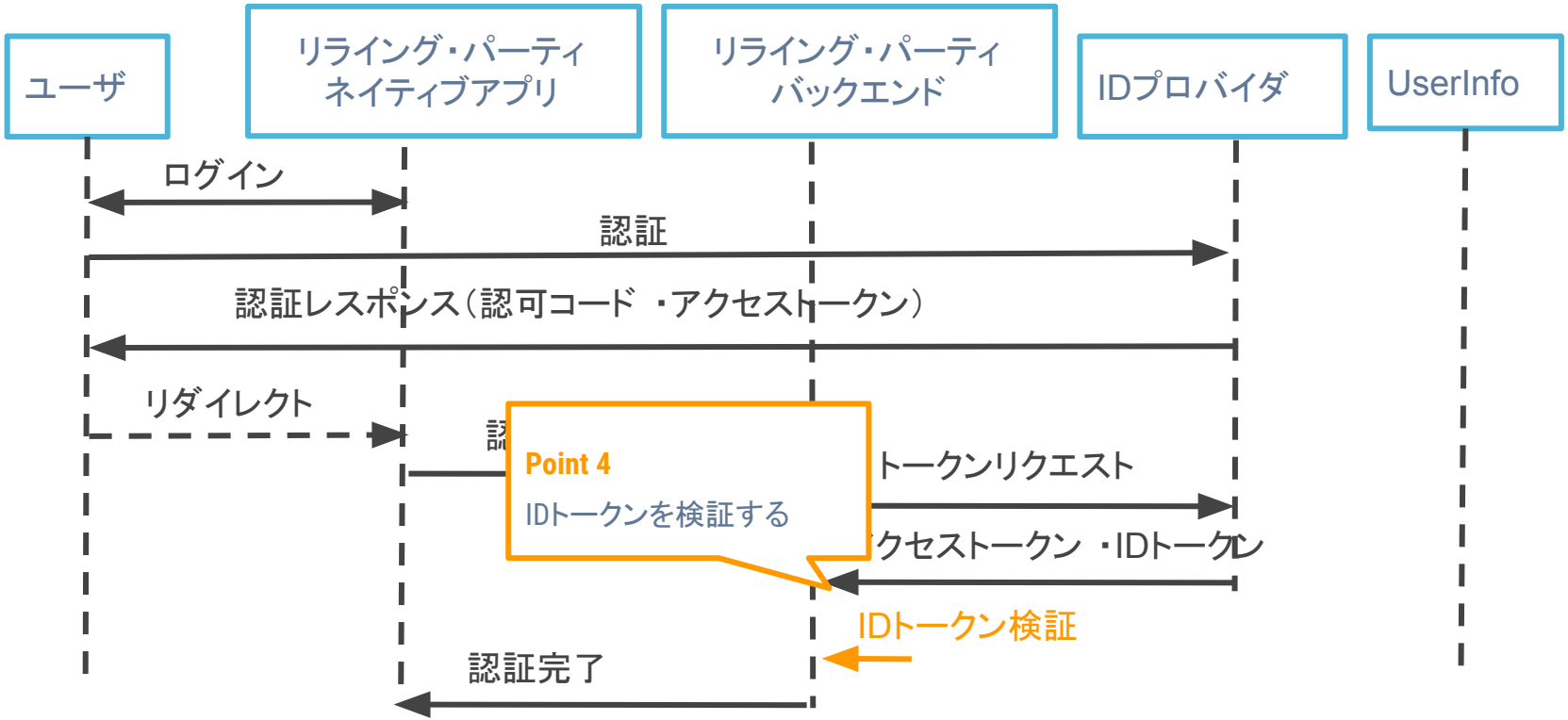


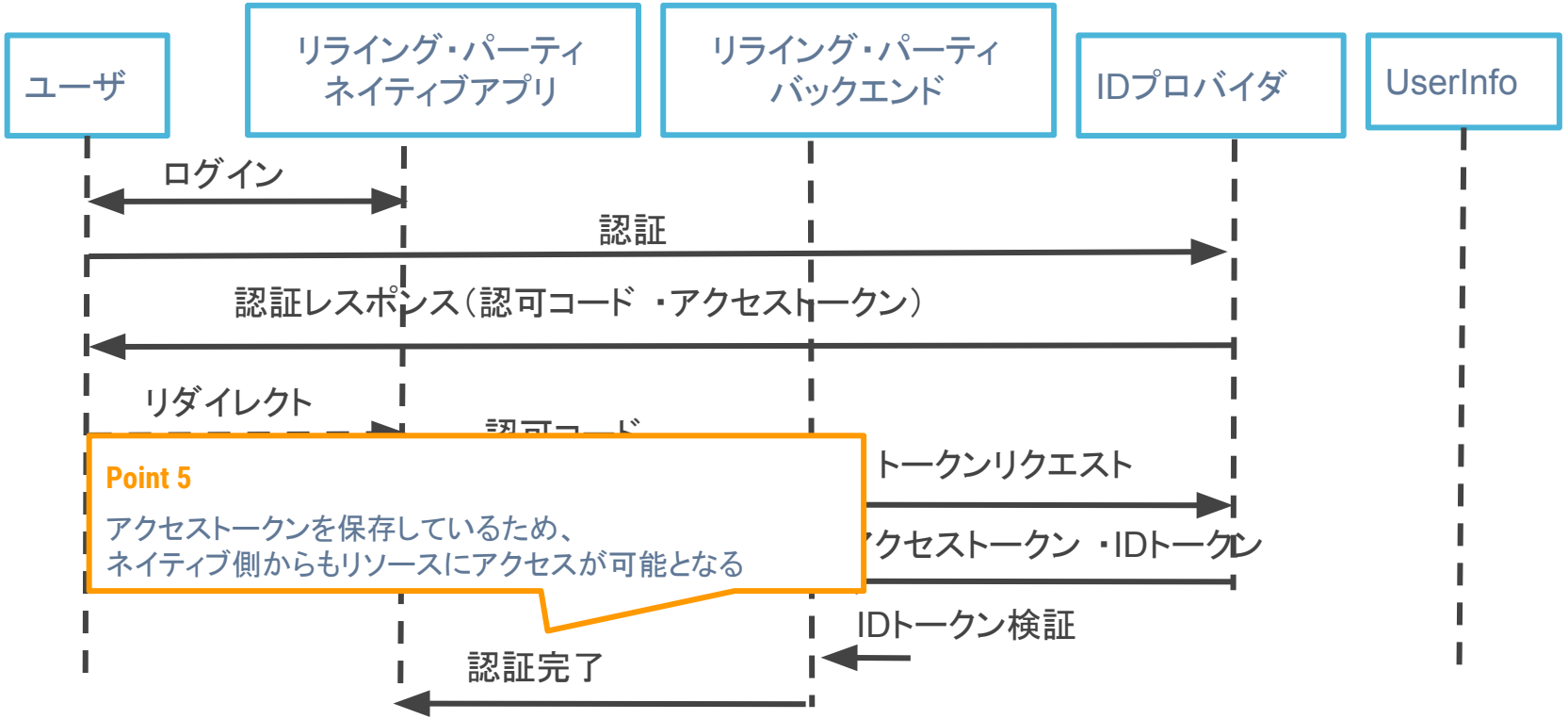

ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
認証レスポンス（認可コード ・アクセストークン）
リダイレクト
認可コード
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
認証完了


ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
認証レスポンス（認可コード ・アクセストークン）
Point 1
認可コードとアクセストークンを受け取る。
アクセストークンはネイティブアプリ側に保存
ークンリクエスト
セストークン ・IDトークン
IDトークン検証
認証完了


ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
認証レスポンス（認可コード ・アクセストークン）
リダイレクト
認可コード
トークンリクエスト
ストークン ・IDトークン
ークン検証
認証完了
Point 2
認可コードをバックエンドにリクエストする


ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
Point 3
バックエンドでセキュアにトークンを取得可能
認可コード
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
認証完了


ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
認証レスポンス（認可コード ・アクセストークン）
リダイレクト
Point 4
IDトークンを検証する
トークンリクエスト
クセストークン ・IDトークン
IDトークン検証
認証完了


ユーザ
リライング・パーティ
ネイティブアプリ
リライング・パーティ
バックエンド
IDプロバイダ
UserInfo
ログイン
認証
認証レスポンス（認可コード ・アクセストークン）
リダイレクト
認可コード
トークンリクエスト
アクセストークン ・IDトークン
IDトークン検証
認証完了
Point 5
アクセストークンを保存しているため、
ネイティブ側からもリソースにアクセスが可能となる

ハイブリッドフローのまとめ

- 認可コードフローとインプリシットフローを合わせたもの
  - 認可レスポンスとトークンレスポンスの両方があるのが特徴
- バックエンド側でセキュアに認証が可能
- パブリック（ネイティブアプリ）側からリソースにアクセスすることが可能

Last.

まとめ

今日の話

まとめ

- OAuthの概要
- OAuth認証はNG
- OpenID Connectの検証はしっかりと行う
- ネイティブアプリとSPAでのOIDCの注意点

“IDトークンを検証したからといって
安全になるわけではありません。”

“OpenID Connectを正しく理解して使用しましょう。”

Special Thanks!

Auth屋

- 書籍:雰囲気でOAuth2.0を使っているエンジニアがOAuth2.0を整理して、手を動かしながら学べる本
- 書籍:OAuth、OAuth認証、OpenID Connectの違いを整理して理解できる本

ritou

- ブログ:r-weblife

Thank you watching!

遠藤 大輔

Twitter: @DddEndow