認証サバイバルガイド

João Angelo著

Auth0

認証サバイバルガイド

João Angelo、Auth0 Inc.

Version1.0、2017

目次

謝辞

順不同：**Kunal Batra**（本書執筆のきっかけとなる話し合いを始めていただいたことに感謝いたします）、**Amaan Cheval**（本書のレビュー）、**Bárbara Mercedes Muñoz Cruzado**（全体的な見栄えの向上）、**Alejo Fernández**と**Víctor Fernández**（本コンテンツ配布のための準備および作業）

はじめに

認証は複雑なトピックです。そう思わないと、おそらくやっていられないでしょう。テクノロジー、プロトコル、コンセプト、すべきこととしてはいけないことが数え切れないほどたくさんあります。その上、発展し続けるベストプラクティス、時代遅れになった専門用語や増え過ぎた専門用語をすべて把握しておくのは困難です。もしあなたがこの分野を学習し始めたばかりなら、異なるものが同じ名前で呼ばれていたり、そうでなかったり、とても混乱していると思います。

本ガイドの目的は、認証に関わる議論において、読者が冷静にきちんと対応できるように考察の材料を十分に提供することです。そのために、認証関連の作業を行う際に遭遇する可能性のあるすべての主要なプロトコル、フォーマット、コンセプト、および用語について簡潔に説明します。

認証システムの作成や再設計を目指して最前線で実装作業を行いながら、あらゆる可能性に常に晒され悩まれている開発者の方々にとっては特に有用な情報です。

そうは言っても、本書が認証全般に関する決定的なリファレンスになるわけではありません。本書の中で取り上げていないトピックや、簡単に触れるだけで十分に説明していないトピックもあります。手短に言えば、最も一般的なシナリオからスタートして、注意を要するものについて説明していきます。ただし、この旅をさらに続けるかどうかは読者の皆さん次第です。本書は

さらに理解を深める必要があるもの、さほど問題でないもの、そういったことを判断するための材料としてご活用いただければと思います。

コンセプト

認証

アプリケーションがやり取りをしている人物のアイデンティティを検証するプロセス。通常、その結果はアプリケーションにアクセスするユーザーとの関連付けが証明される属性のセットである。

認可

アプリケーションにアクセスするユーザーがリソースにアクセスしたり、特定の操作を実行したりするのに必要な権限を持っているかどうかをアプリケーションが判断するプロセス。通常、このプロセスの前に行われる、ユーザーのアイデンティティを確かめる認証ステップに依存する。

アイデンティティプロバイダ

アイデンティティ情報を作成、維持、および管理するエンティティ。ユーザーのアイデンティティの信頼できるソースと関連付けられる属性を提供する。

デジタル署名

特定のメッセージがよく知られているエンティティによって作成され、そのメッセージが作成時からまったく変更されていないことを保証するために使用される技術。

OAuth 2.0

OAuth 2.0（OAuth2）は、アプリケーションがHTTPベースのリソースへのアクセス権を取得できるようにする認可フレームワークです。通常、リソースはこのアプリケーションと関連のないアプリケーションのドメイン内にあり、そのドメイン内で特定のユーザーに関連付けられている場合もあります。OAuth 2.0は、クレデンシャルの共有を要求することなく、委任されたアクセスの問題を解決します。あるFacebookユーザーのためにFacebookのリソースにアクセスするカスタムアプリケーションを例にとって考えてみましょう。このアプリケーションは、ユーザーのFacebookのパスワードにまったくアクセスせずにこれを実行します。

OAuth2に関して言うと、主なプレイヤーは以下のとおりです。

- **認可サーバー：**リソースへのアクセス許可を与えることができる権限。
- **クライアントアプリケーション：**リソースへのアクセスを要求するアプリケーション。
- **リソースオーナー：**通常の場合はエンドユーザー。認可サーバーは、リソースへのアクセス許可をクライアントアプリケーションに与えてよいかどうかリソースオーナーに確認する。
- **リソースサーバー：**実際のリソースを提供するアプリケーション。

ここで重要なのは、このプロトコルは元々ユーザーの認証を目的としたものではないという点です。しかし、ユーザーリソースへのアクセス許可を与えるのに必要なステップの1つがユーザーの身元を確認することであったことから、このフレームワーク/プロトコルがユーザー認証を実現する方法としても使用されるようになりました。

再びFacebookの例を用いて、このプロトコルが認証に使用できるようになった1つの方法について以下に説明します。

1. アプリケーションXが匿名のユーザーにアクセスされる
2. そのユーザーのためにアプリケーションXがFacebookリソースへのアクセスを要求する
3. ユーザーが自分自身を認証する必要があるFacebookに導かれる
4. 認証が成功すると、Facebookはユーザーに対してアプリケーションXに自身のリソースへのアクセスを許可するかどうか尋ねる
5. ユーザーはアクセスを許可して、Facebookはユーザーをアプリケーションに導く
6. アプリケーションXは何か（はい、これはトークンです）を受け取ります。これは、属性（クレーム）の特定のセットを使ってユーザーが確かにFacebookにアカウントを持っていることを証明するのに使用できます。

ご覧のとおり、アプリケーションXはリソースへのアクセスを要求しただけで、認証の部分は単に必要な中間のステップです。ただし、受け取ったトークンは、ユーザーが誰であるかを直接識別したデータを含んでいるか、あるいはFacebookに追加のクエリを実行することによってこのデータを取得するために使用することができます。多くのアプリケーションはこのフローを完全に認証の目的で使用します。つまり、アプリケーションXはFacebook上のユーザーリソースにアクセスすることには関心すらなく、ユーザーが誰であるかを一意に識別することだけに関心がありました。Facebookの規約にしたがってトークンが有効であることを証明するには、少なくとも1つのリクエストが実行されなくてはなりません。

次に、匿名のユーザーがアプリケーションXにアクセスして、一連のステップが繰り返されます。既知のFacebookのユーザー識別子がアプリケーションに返された場合は、前と同じユーザーによってアクセスされていたことが分かります。

OAuth2は認証に使用可能で、現在でも使用されていますが、元来認証を目的として設計されたプロトコルではないということを知っておくことは重要です。OAuth2ソリューションの最も一般的で適用可能なシナリオは、サードパーティ製のアプリケーションにHTTP（REST）ベースのAPIへのアクセス許可を与えることでした。

保護されたリソースが各ユーザーに結び付いているHTTP APIを保持しているか、または保持する予定で、そのデータをサードパーティ製のアプリケーションに対してアクセス可能にする場合は、OAuth2がそのタスクに適したツールである可能性が高いです。少なくともより多く使用されるツールでしょう。IBMを選択して今までにクビになった人はいないと言われていますが、この特定のシナリオに関しては、OAuth2を使用することでかなりの安心感が得られるでしょう。

そうは言っても、OAuth2はこのタスクに使用できる唯一のツールというわけではありません。もっと単純なシナリオでは、よりシンプルな選択肢がほぼ確実に存在します。たとえば、誰がAPIを呼び出しているかを知りたいだけならば、APIキーを使う方がはるかにシンプルで目的にかなった方法です。これで、請求書の送り先が分かります。常に可能性のあるオプションを調査して、特定のシナリオに対するそれらのオプションのメリットとデメリットを比較検討しましょう。

OAuth2について知っておくべきことはもう1つあります。仕様書では認可フレームワークと記述されていて、プロトコルとは呼ばれていませんが、これは意図的なものです。仕様書は各実装に対する最小の要件を扱っていますが、多くの側面については各実装に特有のものとしています。

したがって、クライアントアプリケーションの開発者の立場からOAuth2にアプローチする場合には、以下に示す2つのことを行う必要があります。

- 仕様書（RFC 6749）[1]を確認して、特定のプロバイダとの統合の際に想定されるベースラインについて理解する。
- 特定のプロバイダのドキュメントを読んで、仕様書で規定されている事項以外に実行されたことを理解する。

独自の認可サーバーの実装を構築する観点からアプローチする場合は、既存のものを使用してみることをお勧めします。この方法を取ることができない場合は、少なくともOAuth 2.0 Threat Model and Security Considerations（RFC 6819）[2]を一読してください。そうすれば、考えが変わって、私の最初のアドバイスに従うようになるかもしれません。

---

[1] **https://tools.ietf.org/html/rfc6749**

[2] **https://tools.ietf.org/html/rfc6819**

OpenID Connect

OAuth2は大好評を博したため、人々は本来の設計意図以外のシナリオ、つまりユーザーの認証と識別にもこのプロトコルを使用するようになりました。ところが、このフレームワークを認証の問題を解決する目的にのみ適用した場合、仕様書を読んだ人の解釈に任される決定事項が多すぎることが明らかになりました。

これは、異なるソリューションを使って同じ根本的な問題にアプローチする実装につながり、あとで特定の実装で認証を行うために専用のクライアントライブラリに反映する必要がありました。さらに、決定的なガイドラインがないため、クライアントアプリケーションの開発者にとって適切なセキュリティを確保することがさらに難しくなりました。

OpenID ConnectはOAuth2を拡張したものであるとされ、機能的で安全な認証システムを実現する方法について明確なガイダンスを提供しています。

ユーザー認証という特定の問題の解決に重点を置くことによって、このプロトコルは種々異なる実装をもたらしていたOAuth2の曖昧な領域を形式化しました。たとえば、OAuth2では、使用するトークンについてフォーマットや特定の要件を課さないのに対し、OpenID　Connectでは、IDトークンはJWTフォーマットを使用し、使用可能な標準クレーム一式を常に持つと規定されています。

さらに、このプロトコルは、UserInfoエンドポイントを用いて、アイデンティティプロバイダがユーザーのアイデンティティに関する追加情報を利用できるようにする方法についても形式化しました。多くのプロバイダは机上の実装にこれを配置することによってすでに対応していましたが、これで相互運用を実現するためのベースラインが利用できるようになりました。

他の認証関連のプロトコルに比べると、登場してからまだ日が浅いため、ある程度納得ができることではありますが、特にコア仕様から実装は任意の関連する仕様に思い切って移行する場合、ライブラリとアイデンティティプロバイダの両方のレベルにおけるOpenID　Connectへのサポートは現在も発展途上となっています。

とは言っても、OpenID Connectはアプリケーションでの認証の問題を解決するために、習得して使用する必要のあるプロトコルです。

エンタープライズプロトコル

消費者向けウェブサイトが各アプリケーション/サービスごとに独立したユーザーのクレデンシャルを保持するという状況が急増し、問題化する以前に、ビジネスの世界はすでにそれを経験していました。各ビジネスアプリケーションがユーザーストアの独自コンセプトを実装して、カスタムの方法で認証を行うのは珍しいことではありませんでした。

これはIT部門のみならず、ユーザー自身にとっても問題でした。なぜなら、複数のクレデンシャルを管理しなければならず、あるアプリケーションから別のアプリケーションへの切り替えが必要になるたびにワークフローが常に妨害されるからです。こうした問題を解決するための最初の試みは、企業ネットワーク内に見られるような同種の環境で十分機能しました。Windows Active DirectoryやWindows統合認証のことを考えてみてください。この技術や類似の技術によって、シングルサインオン（SSO）と呼ばれるものが実現しました。ユーザーが中央サーバーで一度認証を行った後、すべてのアプリケーションはこの一元化されたサービスによって提供されるアイデンティティを信頼します。

ところが、アプリケーションが制限された企業ネットワークの境界の外にあると、すぐに振り出しに戻ってしまいました。そこで、2000年代初頭、セキュリティの境界全体にわたるSSOソリューションの必要性に取り組むという目標を掲げた新しいプロトコルが登場しました。この中で、SAMLとWS-Federationという現在でも広く使用されている2つのプロトコルについて見ていきましょう。

SAML

SAML（Security Assertion Markup Language）はアプリケーションセキュリティのシナリオにおいて幅広く使用されていますが、このプロトコルが最もよく使用される目的は、異なるドメインをまたがったウェブブラウザベースのSSOの実現です。

この標準プロトコルは、構造化されたメッセージのセットと、認証リクエストを実行する際に、これらのメッセージがどのようにアプリケーションによって使用されるのかを定義します。認証リクエストの結果、トークンがセキュリティドメイン全体にわたって送信可能になります。一方、リクエスト元のアプリケーションはそのトークンの真偽性を検証できるため、ユーザーの識別が可能になります。

SAMLの世界では、アプリケーションメッセージとトークン自体がXMLベースで定義されます。XMLはこのプロトコルが登場した時代に基づいているので、驚くことではありません。そのため、このプロトコルに必要なメッセージとトークンの2進表現に関して膨大なオーバーヘッドが発生します。理由はこれだけではありませんが、このプロトコルは現代のアプリケーションで使用するにはあまり理想的とは言えません。

このプロトコルに遭遇した場合は（おそらく既存のシステムと統合する場合）、まだ有効に使用されている2つの重要なプロトコルのバージョン（1.1および2.0バージョン）があることに注意してください。

SAMLには、開発者を戸惑わせる興味深い点がもう1つあります。SAMLを使用すると、プロトコルだけでなく、それに伴うトークンのフォーマットを参照することもできます。後ほど分かるように、トークンのフォーマット自体を基礎となるプロトコルから独立して使用できるため、実際この分離は重要です。

WS-Federation

ウェブブラウザベースのSSOに取り組んだもう1つの標準規格はWS-Federationです。WS-Federationは、サーバー間のやりとりからサーバー側のウェブサービスとやりとりをするリッチクライアントアプリケーションまで、アプリケーション統合の分野を広くカバーする大きな仕様グループの一部でした。このプロトコルのグループはWS-*プロトコルとして集合的に扱われました。正直なところ、これらのプロトコルはかなり複雑で、私にとってはまったく良い記憶がありません。なぜなら、元々実装が複雑であったため、特定の機能の豊富さが著しいコストの増加をもたらしたからです。

このガイドの目的上、幸いなことにWS-Federationはこのグループの中でまともなプロトコルであったと思われます。SAMLと同様、WS-FederationはSSOを実現するために、2つのシステム間で必要なウェブのやりとりを定義しました。また、SAMLよりもはるかにシンプルなやり方で定義したと言って間違いないと思います。おそらくその理由から、このプロトコルは今でもかなり使用されています。

ただし、SAMLプロトコルとは異なり、WS-Federationは独自のトークンフォーマットを定義せず、各実装に特有のものとしています。その結果、ほとんどのシナリオ

ではWS-Federationプロトコルと SAML、つまりトークンフォーマットを併用することになります。

過去のプロトコルについての考察（OpenIDとOAuth 1.0a）

OAuth2とOpenID Connectは、少なくとも名前から判断して多少関連があると思われる既存のプロトコルを置き換える目的で設計されたという事実を共有しています。ただし、どちらのプロトコルも最新版は旧版と互換性を持っていないため、この関連性はほとんど名前だけであり、私見では、完全に別々の標準規格と見なすべきであると思います。

OAuth2とOpenID Connectを比較したり、これらがそれぞれ一方より優れたバージョンであるといった主張をするつもりはありません。なぜなら、そうしたところで現実は変わらないからです。OpenID、OpenID 2.0、およびOAuth 1.0aは現在も使用されているにもかかわらず、アクティブな開発が行われていません。多くのプロバイダでは段階的に廃止する方向に向かっており、良くも悪くも他のプロトコルに置き換えられています。

これら標準プロトコルの初期の改善にかかわったり、開発を行う必要性はほとんどないと思われます。私からのアドバイスとしては、これらのプロトコルの存在については知っておく必要がありますが、他に選択肢がないシステムと統合する必要がない限り、これらに時間を費やすことは意味がありません。

トークンオーバーロード

特にJWTのようなトークンに関する誇大宣伝は比較的新しいものですが、認証にトークンを使用すること自体は決して目新しいことではありません。

このことについて、ユーザーを認証する1つの方法としてトークンを使用する良い例は、長年にわたってウェブアプリケーションで使用されてきた従来のCookieベースの認証システムです。ほとんどの実装は内部文字列に依存しています。文字列は少なくともランダムに生成され、おそらくはデジタル署名も行われます。そして、認証されたユーザーに関する追加情報を検索するためにサーバー側で使用されます。

このセッション識別子はトークンと変わりないですし、おなじみのCookieベースの認証システム全体はトークン認証システムと呼ぶ方が適切かもしれません。このシステムは自己完結型のトークンではなく、参照による（by-reference）トークンを使用し、クライアントアプリケーションからサーバーアプリケーション、つまりリソースサーバーにトークンを格納して送信するためのメカニズムとしてCookieに依存しています。

そうは言っても、今日トークンの使用について言及したり、誰かが言及しているのを見ると、自己完結型または値による（by-value）トークンのことを指している可能性が高いです。個人的には値によるトークンと呼ぶ方が好きです。なぜなら、参照によるトークンと明確な対照を成すことに加え、プログラミング言語で広く

使われていて、まったく同種の意味（具体的には値型と参照型のコンセプト）を含んでいる用語だからです。

フォーマット

確実に知っておくべきトークンフォーマットがあるとすれば、それはJWT（JSON Web Token）です。JWTは自己完結型のトークンです。自己完結型とは、トークンの値自体の中にエンティティ、トークンの対象に関する情報が含まれていることを意味します。認証されたセッションデータをサーバーからクライアントアプリケーションに移動させ、いわゆるステートレスな認証システムを構築するシナリオでは、理想的な選択となります。

また、JWTトークンフォーマットは、SAMLのような評判の良い以前のフォーマットに比べて比較的シンプルです。ただし、このシンプルさは相対的なものなので、JWTが非常にシンプルであると考えるべきではなく、処理には独自のロジックを使った方が良い場合もあります。

JWTに関連する仕様では、このトークンを使用する方法についてある程度の柔軟性が提供されています。たとえば、トークンを表す方法が複数あり、使用する署名および暗号化の方法を任意に選択できます。

JWTの最も一般的な表現としてはbase64urlエンコーディングが使用され、ヘッダー、ペイロード、および署名などの構築ブロックをHTTPで使いやすい方法で表します。JWTを使用する認証システムの一般的な実装で暗号化トークンが使用さ

れているのを見かけることは実際のところほとんどありません。主な理由は、OAuth2とOpenID Connectはネット上でトークンを送信する際にTLSの使用を義務付けていて、トークンの実際のコンテンツは機密情報を含まないためです。

JWTが登場する前は、たとえばシングルサインオンの実現のように、ドメイン境界を超えた認証問題の解決を目的とした認証システムに関わる作業を行う際に、最もよく見かけたのはSAMLトークンフォーマットでした。

このトークンフォーマットは署名と暗号化にも対応していますが、JWTに比べて実装が一層複雑になると言えるので、使用率の低下が見られます。主な理由は、JWTがJSONフォーマットに基づいているのに対し、SAMLはXMLに依存するためです。この違いだけで、トークンフォーマットが対応を判断する機能の全体的な複雑性が大幅に増加します。

さらに、SAMLはウェブサービスとSOAPが繁栄を極めた時代、つまりXMLがあらゆるところで使われていた時代に生まれたものです。今日では、物事をよりシンプルにすることに重点が置かれるため、SOAPよりもHTTP、XMLよりもJSON、そして認証に関してはSAMLよりもJWTということになります。

タイプ

トークンのフォーマットはその限られた一面に過ぎません。重要な違いは、各トークンが認証システムで果たす役割です。OAuth2とOpenID　Connectの分野でよく使用されるトークンのタイプは次の3つです。

- アクセストークン
- リフレッシュトークン
- IDトークン

アクセストークンは、特定のリソースサーバー上で使用可能な保護されたリソースにアクセスするためにクライアントアプリケーションによって使用されるクレデンシャルとして、OAuth2の仕様の中で定義されています。クライアントアプリケーションはこの認可リクエストを実行するためにトークンを使用しますが、アクセストークン自体はクライアントに対して不透明であると見なされるべきです。これは、認可が完了した後にクライアントアプリケーションにトークンが付与されることを単に意味しています。この場合、リソースオーナーはリソースへのアクセスを委任することに同意し、それをブラックボックスとして扱います。

特定のリソースサーバー上で認可リクエストを実行するために使用できることをクライアントアプリケーションが知っているのはアイテムですが、トークンの実際のコンテンツについての情報はまったく持っていません。アクセストークンには実装に特有の関連付けられた有効期限がありますが、通常数分から数時間までのかなり短いものです。有効期限が短い理由は、これらのアクセストークンは通常ベアラトークンとして使用されるため、寿命が短いと、起こり得るデータ漏えいの

影響が軽減されるためです。

アクセストークンに使用されるフォーマットに要件はありません。JWTのような自己完結型のトークンから参照によるトークンとして使用される、ランダムに生成された値まで様々なタイプがあります。最も一般的なアプローチは自己完結型のトークンを使うことです。その理由は、トークンの利用で、トークンに関連付けられている情報を取得するためにリソースサーバーが認可サーバーに追加の呼び出しをする必要がないためです。

そうは言っても、参照によるトークンがより優れた特性を発揮する可能性のあるシナリオは存在します。たとえば、同じエンティティが認可サーバーとリソースサーバーの役割を果たす場合、この種のトークンは情報を取得するために追加検索を行わないので大量のオーバーヘッドが発生せず、ネット上での動作が軽くなる可能性があります。

アクセストークンについて最後に一言申し上げると、理想的にはアクセストークンをユーザー認証する手段として使用するべきではありません。その理由は、アクセストークンをAPIで使用するために発行して、認証に使用すると、アプリケーションを脆弱にする場合があるからです。たとえば、アクセストークンがベンダーAのAPI Xに発行され、ベンダーBのアプリケーションYがそのアクセストークンをアプリケーション自体のユーザー認証にも使用すると決めた場合、そのアクセストークンがそれ自体によって開始された認可フローで発行されたのか、それともアクセストークンを使ってユーザーになりすまそうとしているハッカーの制御下

で全く関係のないアプリケーションに対して発行されたのかをアプリケーションは確認することができません。

アプリケーションが異なるオーディエンスを持つトークンを拒否できるように、ユーザーの認証に使用されるトークンはすべて、そのトークンが特定のアプリケーションでの認証のために発行されたことを確認する方法を備えている必要があります。

リフレッシュトークンはOAuth2で導入された別のタイプのトークンで、特定の目的を持っています。元のトークンの期限が切れると、リフレッシュトークンはクライアントアプリケーションが追加のアクセストークンを取得できるようにします。アクセストークンの基準とは異なり、リフレッシュトークンの寿命はかなり長くなっており、数年から、場合によっては有効期限がないこともあります。

このような寿命の違いは、クライアントアプリケーションの開発者にとって混乱の元となります。アプリケーションが寿命の短いアクセストークンと、追加のアクセストークンを取得するのに使用可能な寿命の長いリフレッシュトークンを発行した場合、単に寿命の長いアクセストークンを発行する方がよりシンプルではないでしょうか。

おそらくよりシンプルではありますが、より安全かと言うとそうではありません。アクセストークンは認可サーバーによって発行された後、リソースサーバーに対して使用されるということを思い出してください。このトークンが自己完結型であ

れば、認可サーバーが再び呼び出されることはありません。ただし、リフレッシュトークンは認可サーバーによって発行され、利用されます。つまり、クライアントアプリケーションがリフレッシュトークンを使ってそれをまた呼び出す必要があると確実に分かっているので、リフレッシュトークンのトークン失効ポリシーは認可サーバーによって実装が可能であることを意味します。

さらに、寿命の長いクレデンシャルがネット上で渡されるのに必要な回数も、2つの異なるトークンを持っている場合にはかなり少なくなります。寿命の長いアクセストークンだけの場合では、リソースサーバーへのそれぞれの呼び出しにこの寿命の長いクレデンシャルが含まれるため、最終的にデータ漏えいの影響が増大します。

リフレッシュトークンのもう1つの利点は、エンドユーザーの介入がなくてもトークンが機能し続けるように、ユーザーパスワードの保存に使用された既存のアプリケーションに対して即座にセキュリティを強化できることです。これにより、アプリケーションはパスワードの代わりにリフレッシュトークンを使って、トークンを保存できます。確かにパスワードもリフレッシュトークンも寿命の長いクレデンシャルです。しかし、特定の認可サーバーでのみ使用可能なトークンを保存することは、ユーザーが様々なサービスにわたって再利用する可能性のあるパスワードを保存するよりもはるかに優れた方法です。

パスワードを保存するより良いとは言っても、クライアントアプリケーションはトークンがストレージに存在する間、データ漏えいの可能性を軽減するために、

リフレッシュトークンを適切に保存できるようにする必要があります。混乱を招くもう1つの点として、どのタイプのクライアントアプリケーションも寿命の長いクレデンシャルを適切に保存できるわけではありません。このような状況では、リフレッシュトークンの使用を避ける必要があります。適切な保存手段を提供できないアプリケーションの最も一般的な例は、ブラウザベースのアプリケーションです。

最後に、IDトークンについて説明します。これは、ユーザー認証を語る上で最も重要なトークンのタイプです。このトークンはOpenID Connectの仕様と共に導入され、このプロトコルと同様、認証の問題を解決するために最適化されています。

前に説明した2種類のトークンでは各実装に特有の表現が許容されていますが、それらトークンとは異なり、OpenID Connectプロトコルに準拠する実装では、このトークンのタイプを表す手段としてJWTを使用するよう義務付けています。前に説明したように、ユーザー認証に使用されるトークンの要件の1つは、トークンが使用されるために発行されたことをクライアントアプリケーションに確実に知らせる手段を提供することなので、これはもっともなことです。

さらに、ユーザーを一意に識別するための情報も必要なため、自己完結型のトークンフォーマットを使用することはユーザー認証トークンに関して正しい選択と言えます。

IDトークンの検証が正しく行われなかったり、不完全であったりすると、ハッカーがアプリケーション内で大きな損害を引き起こす可能性があります。驚かれるか

もしれませんが、不幸なことにアプリケーションがトークンを正しく検証できない状況に遭遇することは珍しいことではありません。このような被害に遭わないようにするために、OpenID Connectの仕様書のセクション3.1.3.7に記載されているIDトークン検証のルールに厳格に従う必要があります。

トークンの保存

前述のように、トークンの適切な保存はトークンベースの認証システムにおいて必須の要件です。トークンをベアラトークンとして使用する場合は特に重要となります。また、役割やトークンの保存を必要とするアプリケーションのタイプに応じた取り組みも必要です。

ネイティブのモバイルクライアントアプリケーションでは、デバイス上の他のアプリケーションがトークンにアクセスできないようにする保存方法を利用する必要があります。たとえば、Androidでは、SharedPreferences機能がこの要件を満たすと考えられます。

ブラウザベースのアプリケーションの場合は少し複雑です。使用可能な保存方法として検討すべきオプションがいくつかありますが、いずれも特定の短所があります。ブラウザベースのアプリケーションに対して保存方法を選ぶのは「Pick Your Poison」ゲームのようなもので、最善の選択は使用するシナリオの正確な内容に大きく依存します。以下に選択の助けとなる使用可能なオプションと、各オプションに関連するメリットとデメリットをいくつか挙げて解説します。

ウェブストレージ（ローカルストレージまたはセッションストレージ）

メリット：

- ブラウザはウェブストレージからHTTPリクエストに何も自動的に含めないので、CSRFに対する脆弱性はない
- データを作成した全く同じドメインで実行されるJavascriptによってのみアクセス可能である
- HTTPでトークン認証用のクレデンシャルを渡す場合に、意味論的に最も適切な方法が使用可能である（ベアラスキームを利用する認可ヘッダー）
- どのリクエストに認証を含めるべきかを非常に簡単に注意深く選択できる

デメリット：

- データを作成したドメインのサブドメインで実行されるJavascriptによるアクセスができない（example.comによって書き込まれた値をsub.example.comによって読み取ることができない）
- XSSに対する脆弱性がある
- 認証リクエストを実行するために、リクエストのカスタマイズを可能にするブラウザ/ライブラリのAPIのみが使用可能である（認可ヘッダーでトークンを渡す）

HTTP限定のCookie

メリット：

- XSSに対する脆弱性がない
- ブラウザはCookieの仕様に対応するすべてのリクエストにトークンを自動的に含める（ドメイン、パス、および寿命）
- トップレベルドメインにCookieを作成でき、サブドメインによって実行されるリクエストで使用できる

デメリット：

- CSRFに対する脆弱性がある
- サブドメインでのCookieの使用可能性について把握し、常に検討する必要がある
- Cookieを含めるべきリクエストを注意深く選ぶことはできるが、面倒である
- ブラウザがCookieを処理する方法のわずかな違いによって、まだ問題が発生する場合がある
- 注意しないと、XSSに対して脆弱性のあるCSRF軽減戦略を実装する可能性がある
- サーバー側はより適切な認可ヘッダーではなく、認証のためにCookieを検証する必要がある

Javascriptによるアクセスが可能なCookie（サーバー側で無視される）

メリット：

- CSRFに対する脆弱性がない（サーバーによって無視されるため）
- トップレベルドメインにCookieを作成でき、サブドメインによって実行されるリクエストで使用できる
- HTTPでトークン認証用のクレデンシャルを渡す場合に、意味論的に最も適切な方法が使用可能である（ベアラスキームを利用する認可ヘッダー）
- どのリクエストに認証を含めるべきかを簡単に注意深く選択できる

デメリット：

- XSSに対する脆弱性がある
- Cookieを設定するパスに注意しないと、ブラウザが自動的にCookieをリクエストに含めてしまい、不要なオーバーヘッドが発生する
- 認証リクエストを実行するために、リクエストのカスタマイズを可能にするブラウザ/ライブラリのAPIのみが使用可能である（認可ヘッダーでトークンを渡す）

これは奇妙なオプションと思われるかもしれませんが、トップレベルドメインとすべてのサブドメインに対して使用できる保存方法があるというのは、ウェブストレージにはない素晴らしいメリットです。ただし、実装はより複雑なものになります。

最も一般的なシナリオでは、ウェブストレージのオプションを選択することをお勧めします。主な理由は以下のとおりです。

- ウェブアプリケーションを作成する場合は、トークンを保存する場所とは常に別にXSSを取り扱う必要があります。
- Cookieベースの認証を使用しない場合は、CSRFはレーダーに現れることすらないので心配には及びません。

Cookieベースのオプションはいずれもそのことに触れていませんが、HTTPSの使用はもちろん必須条件です。つまり、それを念頭に置いてCookieを適切に作成する必要があります。

保存に関して最後に申し上げることは、クライアントアプリケーションに特有のことではありませんが、認可サーバーを実装または維持する場合は、後でその有効性を確認できるように、生成した値によるトークンを保存しておくとよいかもしれません。このような場合、サードパーティから提供されたトークンが以前作成したものと一致することを確認する必要があります。そのためにトークンをテキスト形式で保存しなくても大丈夫なので、生成したトークンのハッシュのみを保存しておくことを強くお勧めします。

でもCookieが好き

現在、トークン、特にJWTのような自己完結型のフォーマットはとても人気がありますが、Cookieが時代遅れというわけではありません。Cookieは単にHTTP内のステート管理のためのメカニズムであるという観点から考えると、これは驚くことではありません。これまで見てきたように、Cookieはトークンのクライアントサイド

のストレージとしても使用できます。

ただし、Cookieを認証のメカニズムとして見る場合でも、Cookieには有効なユースケースがまだ存在します。たとえば、ウェブアプリケーションでIDトークンがユーザーのクレデンシャルとして提示されるユーザーとの最初のやりとりの際に、Cookieベースのセッションを確立できます。

Cookieがユースケースの簡易化に役立つと考えるならば、Cookieが利用可能な新しいものでない、またはステートレスの議論を投げかけられるというだけの理由で、Cookieを拒否しないでください。また、Cookieを使ってセッション状態をクライアントにプッシュすることもできます。ただし、それを正しく実行するために、Cookieの署名や暗号化を忘れないようにしてください。Cookieの大きな欠点は、ユーザーエージェントが発行したリクエストに自動的に含まれてしまうことです。そのため、CSRF攻撃を軽減するための適切な対策を講じる必要があります。

トラブルシューティング

ソフトウェア開発で物事がうまく行かない場合に、アプリケーションの制御以外の原因を考えるのはほとんど避けられないことです。それに加えて、開発者がコードを記述したときに、実行すべきであると考えていたことが実行されるのではなく、記述されたコードに書かれていることがその通りに実行されるという、オペレーティングシステムの頑固さに伴う人為的なミスもあります。この問題を解決する人はとてもリッチになるでしょう。

現代の認証システムの大きなメリットの1つは、HTTP通信に大きく依存するアプリケーションシステムに関わる認証の問題を解決するのに適用されることです。HTTPは長年にわたって膨大なツールによるサポートを提供していることに加え、見やすいため、トラブルシューティングを行う上で非常にありがたいです。

現代の認証スキームの実装とデバッグを成功させるにはHTTPを確実に理解している必要があり、これがマイナス面であるという議論もあるでしょう。それは正しい意見ですが、他の選択肢に比べるとはるかに優れているのです。私はSOAPよりWS-*のトラブルシューティングの方が好きだという人はいないと確信しています。

それはただのHTTPSに過ぎない

これまで見てきたように、認証プロセスを完了するために必要なやりとりはほ

ぼすべてHTTP上で起こり、HTTPリダイレクトに大きく依存するでしょう。これにより、ユーザーは認証プロセスに参加しているエンティティ全体にわたってナビゲートされます。少々時間を割いて、以下の分野について詳しく学ぶことをお勧めします。

- HTTPリクエスト/レスポンスの意味論
- HTTPリダイレクト
- HTTP認証（認可ヘッダー）
- HTTPステート管理（Cookie）

この知識とツールベルトに入れたいくつかのツールを身に着ければ、現代の認証フローの分析とトラブルシューティングはとても簡単だということが分かるでしょう。では、一般的な認証フローの例を見ていきましょう。これにより、どのようなことが起こるかが分かります。

- ユーザーがアプリケーションYの保護されたリソースにアクセスする（このアプリケーションは別の外部エンティティに認証を委任する）
- アプリケーションYが保護されたリソースへの匿名のユーザーリクエストを検知すると、HTTPリダイレクトを外部のアイデンティティプロバイダの認証エンドポイントに発行する
- また、アイデンティティプロバイダはユーザーが認証されていないことを検知して、クレデンシャルの入力を要求する（良く知られているSSOによって、ユーザーが前に認証され、アイデンティティプロバイダでセッションを開始した場合はこのステップと次のステップを省略できる）

- ユーザーは必要なクレデンシャルを提供して、アイデンティティプロバイダがそれらを検証する
- アイデンティティプロバイダはユーザーのアイデンティティの証明に使用できる十分な情報を使って（おそらくエンティティプロバイダによってデジタル署名されたトークン）、HTTPリダイレクトをアプリケーションYに発行する
- アプリケーションYは後続のリクエストすべてに対してアイデンティティプロバイダとのやりとりを要求しないように、ユーザーのアイデンティティ情報を処理してユーザーに関する認証されたセッションを確立する

このフローは複数の認証プロトコルによって使用される最も一般的な一連のステップを示しています。もちろんバリエーションはありますが、一般的な場合、これらのフローのデバッグは、各ステップで発行されたHTTPリクエストとレスポンスの確認と、含まれているペイロード（クエリーパラメータ、ヘッダー、Cookie、およびリクエスト/レスポンスの本文）が特定のプロトコルの要件に従って想定されるものと一致するかどうかの検証から構成されます。

もう1つの重要なポイントは、HTTPアーカイブファイルフォーマット（HAR）に依存することによって、事後に、または別のマシンで問題を簡単に解決できます。これは、特定の環境で発生する可能性があり、再現が難しい問題にも理想的です。このフォーマットはHTTPネットワークトレースをエクスポートできる主要なブラウザによってサポートされており、後でHARビューアを使って調べることができます。

認証トランザクションに関連付けられたHTTPリクエストをキャプチャすると、ユーザーパスワードを含む機密情報が含められます。そのため、この種のトラブルシューティング情報をユーザーに提供するように依頼する場合は、このことを必ず念頭に置く必要があります。最善の方法は、明確な説明をして、パスワードが含まれていないことを少なくとも確認するようにユーザーに要求することです。HARファイルはJSONベースのテキストファイルなので、任意のエディターを使用してこの種の機密情報を除外できます。

ツール

ネイティブのプラットフォームだけが開発対象でない限り、ブラウザに多くの時間を費やしていると言っても過言ではありません。幸いなことに、すべての主要なブラウザには素晴らしい開発ツールが備わっており、ネット上で何が起こっているかを調べたい場合には不可欠なものとなっています。ブラウザツールを使って直接作業することの主なメリットの1つは、追加の構成が不要で、暗号化されたHTTPSではなく、昔ながらの単純なHTTPを扱うことにあります。さらに、ほとんどの認証フローはリダイレクトを含むため、まず最初にすべきことは、ブラウザがナビゲーションの際のネットワークトレースを保存するように構成されていることを確認することです。多くの場合、問題の根本的な原因は、ブラウザが最終的にリダイレクトされる場所に関連する行の少し下のリクエストにあるからです。

HTTPのデバッグに関して言うと、もう1つの必須のツールはFiddler（現在はTelerik Fiddler）です。これはプロキシとして機能するスタンドアロン型のツールで、それを通過するHTTPトラフィックの分析ができます。選択するブラウザによって

異なりますが、Fiddlerプロキシを使って作業するためにはそれを構成する必要があるかもしれません。また、明示的にその機能を有効にしない限り、デフォルトでは解読されたHTTPSトラフィックを見ることはできないでしょう。

FiddlerはHARファイルのインポートにも対応しているので、このツールが提供する拡張機能の一部を使用したい場合は、ブラウザでトラフィックをキャプチャして、Fiddlerでそれを分析します。

別のツールをインストールせずに、HARファイルを確認できるようにするもう1つの方法は、Googleが提供する次のオンラインビューアを使用することです。
https://toolbox.googleapps.com/apps/har_analyzer/

最後に、トークンそのものを使って作業する際にツールベルトに追加すると便利なオンラインツールが2つあります。

- **http://samltool.io/** - SAMLトークンをデコードして、最も重要な属性をすばやく確認できる
- **https://jwt.io/** - HS256とRS256JWTの署名アルゴリズムを使って、JWTの署名をすばやくデコードして確認できる。前者の場合は、ペイロードを編集することもできます。

隠れた危険性について

この危険性についてはすでに広く認知されていて、OAuth2とOpenID　Connectの仕様では広範囲にわたって注意を呼び掛けていますが、非常に重要なことなので何度も繰り返しますが、これらのベースとなるHTTPプロトコルの使用は、関連するパーティ間の通信がTLS（HTTPS）を通じて発生することを意味します。これを怠るとシステムのセキュリティが大幅に低下する恐れがあるため、必須です。

アプリケーションユーザーの認証を外部のシステムに委任することは、アプリケーションがパスワードリセット、ブルートフォース攻撃検知、パスワードポリシーのような一定の機能について注意する必要がないということを意味します。ところが、もっと複雑になる面があります。

委任された認証が複雑化するケースの1つは、ユーザーをログアウトさせるための機能です。クライアントアプリケーションのタイプによっては、以下のどちらかを実行したくなるかもしれません。

- アプリケーション特有のセッションを作成するウェブアプリケーションで、このセッションを削除し、適用可能なCookieを消去して終わりにする。これは、ユーザーがアプリケーションで認証を行ったのと全く同じことである。
- ウェブAPIを使って通信するクライアントアプリケーションで、アクセストークンをクライアント側で削除する。サーバーはセッションデータに関してステートレスなので、他に何かをする必要はない。

これらの方法は不完全です。ユーザーの視点から見ると、認証がシームレスに行われたのでそれについては考えない傾向にありますが、委任された認証ではユーザーは実際に2つの独立したシステムで認証を行ったことになります。

もしあなたが自分のローカルアイテムだけを消去した場合、ユーザーはアイデンティティプロバイダのレベルでまだアクティブなセッションを保持している可能性があります。これは、再びログインしようとしたときにクレデンシャルの再入力が要求されないことを意味します。これは重要なことです。なぜなら、もしユーザーがアプリケーションを使用する目的のためだけにGoogleで認証を行い、パブリックコンピュータまたはマルチユーザーワークステーションでそのアプリケーションを使用した後、アプリケーションがユーザーにローカルセッションだけを消去するログアウト機能を提供する場合、基本的にユーザーのGoogleセッションはアクティブな状態のままとなるからです。これは多くの非技術系ユーザーが理解しないことでしょう。

用語集

アイデンティティプロバイダ

アイデンティティ情報を作成、維持、および管理するエンティティ。ユーザーのアイデンティティの信頼できるソースと関連付けられる属性を提供する。

アクセストークン

保護されたリソースにアクセスするためのクレデンシャルとして許可された文字列。アクセストークンが使用されるフォーマット、構造、またはプロセスについて課される要件はない。

クライアントアプリケーション

リソースオーナーのために保護されたリソースにアクセスするためのリクエストを発行する各種アプリケーション。

サービスプロバイダ

認証されたエンティティにサービスを提供するエンティティ。外部のアイデンティティプロバイダに対してサービスを使用しようとするエンティティの実際の認証を委任する。この用語はSAMLプロトコル内で使用され、WS-Federationのリライングパーティに相当する。

ベアラトークン

アクセストークンの特定のタイプで、保護されたリソースへのアクセスの検証に使用される唯一のメカニズムはトークン自体であることを意味する。トークンを保持しているユーザー、つまりベアラはそのトークンが有効である限り保護されたリソースにアクセスできるという事実にこの名称は由来している。

リソースオーナー

保護されたリソースへのアクセスを許可する権限を持ったエンティティ。このエンティティが人間の場合は、エンドユーザーと呼ばれることがある。

リソースサーバー

保護されたリソースをホストするサーバー。OAuth2との関連では、リソースサーバーは保護されたリソースへのアクセスを許可する1つの方法として、アクセストークンを許可する追加機能を備えている。

リフレッシュトークン

有効期限切れのアクセストークンを取得またはリフレッシュするために使用される特殊なトークン。アクセストークンは認可サーバーによって発行され、リソースサーバーで利用されるのに対し、リフレッシュトークンは認可サーバーでのみ発行されて利用される。リフレッシュトークンは、リソースオーナーがアクセスを許可するすべてのリソースを表し、これらのリソースの全部またはサブセットのみを対象とするアクセストークンの取得に使用できる。

リライングパーティ

認証されたエンティティにリソースまたはサービスを提供するエンティティ。ただし、呼び出すエンティティの認証を保証するために、アイデンティティプロバイダによって発行されたセキュリティトークンに依存する。この用語は主にWS-Federation内で使用され、SAMLのサービスプロバイダに相当する。

認可

アプリケーションにアクセスするユーザーがリソースにアクセスしたり、特定の操作を実行したりするのに必要な権限を持っているかどうかをアプリケーションが判断するプロセス。通常、このプロセスの前に行われる、ユーザーのアイデンティティを確かめる認証ステップに依存する。

認可サーバー

関連付けられたリソースサーバーで使用するアクセストークンを発行できるサーバー。アクセストークンを発行するプロセスで、リソースオーナーを認証してその同意を取得する必要がある。

認証

アプリケーションがやり取りをしている人物のアイデンティティを検証するプロセス。通常、その結果はアプリケーションにアクセスするユーザーとの関連付けが証明される属性のセットである。

デジタル署名

特定のメッセージがよく知られているエンティティによって作成され、そのメッセージが作成時からまったく変更されていないことを保証するために使用される技術。

その他のアイデンティティおよびセキュリティ関連の用語については、Auth0アイデンティティ用語集（**https://auth0.com/identity-glossary**）を参照してください。