Java aktuell

iJUG Verbund
www.ijug.eu

Mobile Entwicklung
Kotlin Multiplatform, Jetpack Compose, Modularisierung u. v. m.!

Verteilte Datensammlung
Zeitserien-Daten mit Apache TsFile und Apache IoTDB

Das Gehirn als Datensystem
Reverse Engineering des Zentralen Nervensystems

Ausgabe 02/2024
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 02

OpenID Connect für native Mobile-Apps

Sven-Torben Janus, Conciso GmbH

Im Zeitalter mobiler Anwendungen gewinnt die Sicherheit der Benutzerauthentifizierung in nativen Apps zunehmend an Bedeutung. In diesem Artikel zeige ich Best Practices für die Mobile Authentication unter Verwendung des OpenID-Connect-Protokolls (OIDC) auf. Besonderer Fokus liegt dabei auf nativen Apps. Ich gebe Einblicke in den Autorisierungsablauf, die Inter-App-Kommunikation und grundlegende Sicherheitsaspekte. Erfahren Sie, wie die Verwendung externer User Agents und standardisierte Kommunikation mittels URIs die Sicherheit erhöht, die Benutzerfreundlichkeit steigert und Implementierungskomplexität minimiert.


23:56
YOUR CODE
56698
USER NAME
****
ENTER CODE

Einleitung

Die rasante Verbreitung von mobilen Apps hat die Anforderungen an die Sicherheit und Effizienz von Authentifizierungssystemen stark beeinflusst. Insbesondere bei nativen Mobile-Apps steht die Gewährleistung eines reibungslosen und gleichzeitig sicheren Authentifizierungsprozesses im Mittelpunkt. In diesem Kontext hat sich das OIDC-Protokoll als Standard etabliert und spielt dabei eine entscheidende Rolle.

In diesem Artikel gehe ich auf bewährte Methoden und Best Practices für die Mobile Authentication unter Verwendung von OIDC ein, wobei der Schwerpunkt auf der Authentifizierung für native Mobile-Apps liegt. Ich stelle praxisnahe Beispiele vor, um Lesern eine fundierte Grundlage für die Implementierung in ihren Projekten zu bieten.

Im weiteren Verlauf des Artikels behandle ich die wichtigsten Aspekte, angefangen bei einer Beschreibung des Autorisierungsablaufs für native Apps über den Browser bis hin zur Implementierung der Inter-App-Kommunikation mittels *Uniform Resource Identifiers* (URI). Zusätzlich zeige ich bewährte Varianten für den Empfang von Autorisierungsantworten in nativen Apps auf. Grundlegende Best Practices in puncto Sicherheit, einschließlich *Proof Key for Code Exchange* (PKCE) zum Schutz des Autorisierungscodes, bleiben dabei nicht außen vor. Entwicklern, Architekten und Sicherheitsbeauftragten bietet dieser Artikel praxisrelevante Handlungsanleitungen.

Autorisierungsablauf für native Apps über den Browser

OpenID Connect unterstützt diverse Abläufe zur Authentifizierung von Nutzern, sogenannte *Flows*. Allen voran sind hier der *Authorization Code Flow*, *Direct Grant Flow* und *Implicit Flow* zu nennen.

Zunächst möchte ich erläutern, wie der *Authorization Code Flow* aussieht:

1. Der Flow beginnt mit dem sogenannten *Client-Initiated Authorization Request*. Dabei startet der Nutzer den Authentifizierungsablauf durch Interaktion mit der Client-Anwendung, in unserem Fall einer nativen App.
2. Die Anwendung leitet den Nutzer zwecks Autorisierungsanfrage an den *Authorization Server* weiter.
3. Anschließend erfolgt die Nutzerauthentifizierung. Der *Authorization Server* präsentiert dazu dem Nutzer in der Regel eine Anmeldeseite und bittet um Zustimmung zur Autorisierung der Client-Anwendung. Nach Zustimmung generiert der *Authorization Server* einen Autorisierungscode (*Authorization Code*).
4. Der Autorisierungscode wird an den in der Anwendungsregistrierung festgelegten Umleitungs-URI zurückgegeben. Die Client-Anwendung kann nun den Autorisierungscode als den des URI abrufen.
5. Es folgt eine Tokenanforderung durch den Client, indem die Client-Anwendung den Autorisierungscode verwendet, um beim Token-Endpunkt des *Authorization Servers* ein Zugriffstoken (*Access Token*) anzufordern.
6. Der *Authorization Server* validiert den Autorisierungscode. Bei erfolgreicher Validierung sendet der *Authorization Server* Zugriffs- und Auffrischungstoken (*Access und Refresh Token*) an die Client-Anwendung.
7. Die Client-Anwendung kann das erhaltene Zugriffstoken daraufhin verwenden, um auf geschützte Ressourcen beim *Resource Server* zuzugreifen.

Der *Authorization Code Flow* stellt den Quasi-Standard im Sinne einer Best Practice für die Authentifizierung mobiler, nativer Apps dar.

Die Verwendung von *Direct Grant Flows* (aka *Resource Owner Password Credentials*) und *Implicit Flows* für die Authentifizierung in nativen Apps ist mit erheblichen Sicherheitsrisiken verbunden und daher heutzutage nicht mehr empfehlenswert. Bei *Direct Grant Flows* werden der Benutzername und das Passwort direkt an den *Authorization Server* übermittelt, was potenziell unsicher ist und die Gefahr birgt, dass (potenziell weniger vertrauensvolle Apps) sensible

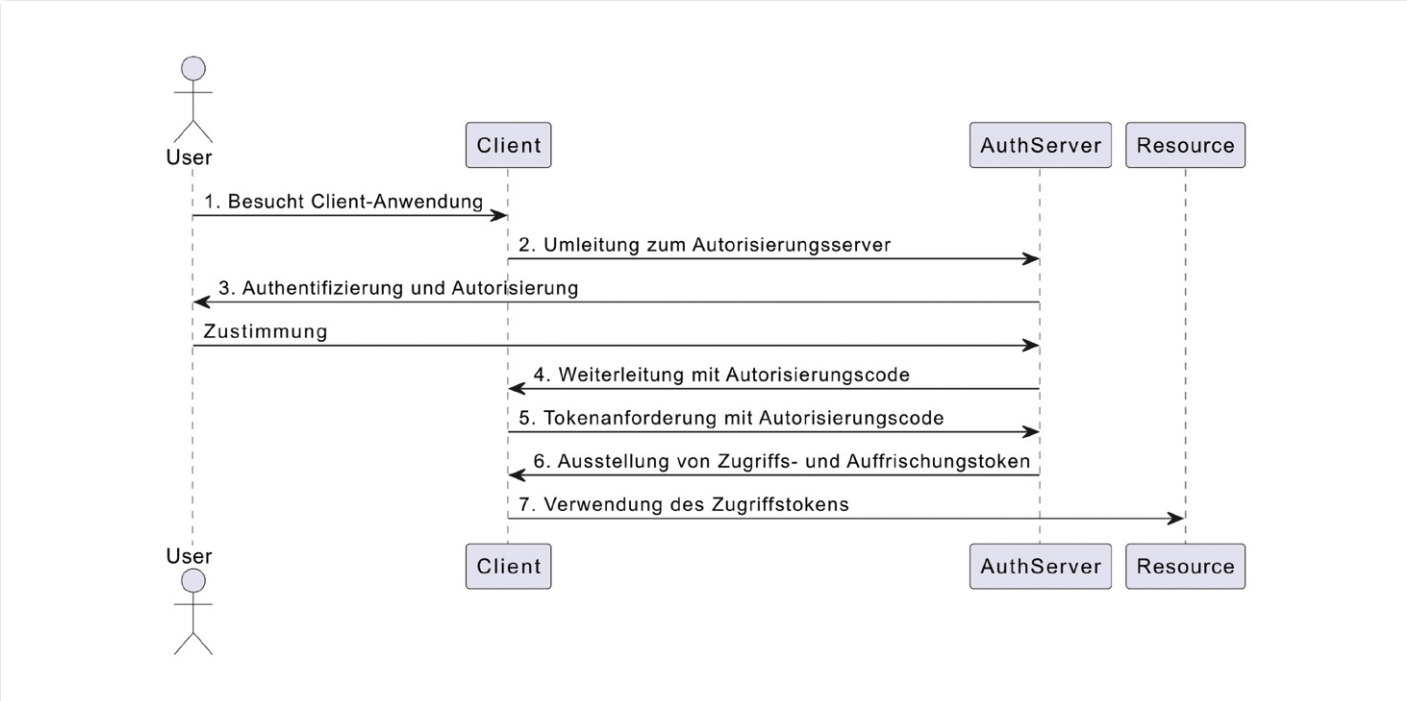

Abbildung 1: Schematische Darstellung des OIDC Authorization Code Flows

Anmeldeinformationen abfangen, beziehungsweise mitschneiden. Der *Implicit Flow* hingegen überträgt *Access Tokens* direkt über den Browser, ohne dass ein *Authorization Code* verwendet wird. Dies erhöht das Risiko von *Token Exposure*, da die Tokens möglicherweise in unsicheren Umgebungen wie Browser-Verlauf oder Cookies gespeichert werden.

In einem nativen App-Szenario, in dem der Schutz von sensiblen Benutzerdaten von höchster Priorität ist, sind diese Flow-Typen meiner Ansicht nach unangemessen.

Für die Sicherheit und Integrität der Authentifizierung von nativen Apps ist es daher ratsam, auf den *Authorization Code Flow* in Kombination mit *Proof Key for Code Exchange* umzusteigen. Dieser Flow minimiert das Risiko von *Token Exposure* und unerlaubten Zugriffen, indem er einen zusätzlichen Schutzschritt für den *Authorization Code* einführt, der speziell auf die Anforderungen nativer Apps zugeschnitten ist. Wie das genau aussieht, beschreibe ich später.

Inter-App-Kommunikation für OIDC

Wie der *Authorization Code Flow* zeigt, nutzt OIDC zur Inter-App-Kommunikation URIs, ähnlich wie im OAuth-2.0-Webkontext (Open Authorization), um die Autorisierungsanfrage zu initiieren und die Autorisierungsantwort an die anfordernde Website zurückzugeben. In nativen Apps können URIs verwendet werden, um die Autorisierungsanfrage im Browser des Geräts zu starten und die Antwort dann an die anfragende native App zurückzugeben.

Indem dieselben Methoden wie im Web für Open Authorization übernommen werden, ergeben sich auch im Kontext nativer Apps Vorteile. Die Benutzerfreundlichkeit einer Single-Sign-On-Sitzung und die Sicherheit eines separaten Authentifizierungskontexts werden so realisiert. Die Wiederverwendung dieses Ansatzes reduziert zudem die Implementierungskomplexität und steigert die Interoperabilität. Dabei wird auf standardbasierte Web-Abläufe gesetzt, die nicht auf eine bestimmte Plattform beschränkt sind.

Um bewährten Praktiken zu entsprechen, müssen native Apps einen externen Benutzer-Agenten, zumeist einen Browser, verwenden, um OAuth-Autorisierungsanfragen durchzuführen. Dies wird erreicht, indem die Autorisierungsanfrage im Browser geöffnet wird und ein Umleitungs-URI verwendet wird, der die Autorisierungsantwort zurück an die native App sendet. Diese Inter-App-Kommunikation ermöglicht eine sichere und effiziente Autorisierung unter Verwendung von OIDC.

Autorisierungsanfragen nativer Apps

Um die Autorisierung nativer Apps zu initiieren, erstellen diese einen Autorisierungsanfrage-URI mit dem sogenannten *Authorization Code Grant Type*. Hierbei wird ein Umleitungs-URI verwendet, der von der nativen App empfangen werden kann.

Die Funktion des Umleitungs-URIs für eine Autorisierungsanfrage nativer Apps ist vergleichbar mit der einer webbasierten Autorisierungsanfrage. Statt die Autorisierungsantwort an den Server des OAuth-Clients zurückzugeben, sendet der von einer nativen App verwendete Umleitungs-URI die Antwort an die App zurück. Verschiedene Optionen für einen Umleitungs-URI, der die Autorisierungsantwort an die native App in verschiedenen Plattformen zu-

rücksendet, erläutere ich später. Jeder Umleitungs-URI, der der App ermöglicht, den URI zu empfangen und seine Parameter zu überprüfen, ist grundsätzlich geeignet.

Nachdem der Autorisierungsanfrage-URI erstellt wurde, verwendet die App plattformspezifische APIs, um den URI in einem externen Benutzer-Agenten zu öffnen. Typischerweise handelt es sich dabei um den Standardbrowser, der für die Verarbeitung von *„http"-* und *„https"-URIs* auf dem System konfiguriert ist. Es können verschiedene Auswahlkriterien für die Browservariante und andere Kategorien von externen Benutzer-Agenten verwendet werden.

Für native Apps, die eine Benutzerautorisierung benötigen, ist die Erstellung einer Autorisierungsanfrage mittels OpenID Connect ein empfohlener Ansatz. Die Nutzung von OIDC ermöglicht es, neben der reinen Autorisierung auch Authentifizierungsinformationen zu erhalten, was die Sicherheit und Benutzerfreundlichkeit weiter verbessert.

Öffentliche native App-Clients sollten besonders die PKCE-Erweiterung für OAuth implementieren, um die Sicherheit des Autorisierungscodes zu stärken. Dieser Schutzschritt ist entscheidend, um mögliche Angriffe auf die Autorisierungsanfrage zu minimieren.

Die Verwendung eines externen Benutzer-Agenten, wie beispielsweise eines Browsers, für OAuth-Autorisierungsanfragen ist eine bewährte Methode, die die Sicherheit erhöht und die Interoperabilität zwischen verschiedenen Plattformen verbessert. Insbesondere wird die Verwendung von In-App-Browser-Tabs empfohlen, sofern von der Plattform unterstützt, um sowohl die Usability als auch die Sicherheitskontexte zu optimieren.

Varianten zum Empfang von Autorisierungsantworten

Um die Autorisierungsantwort von einem Browser zu empfangen, stehen nativen Apps verschiedene Umleitungs-URIs zur Verfügung, deren Verfügbarkeit und User Experience je nach Plattform variieren.

URI-Schema-Umleitung mittels Custom URL Schemes

Viele mobile und Desktop-Plattformen unterstützen die Inter-App-Kommunikation über URIs, indem Apps private URI-Schemata registrieren (umgangssprachlich als *Custom URL Schemes* bezeichnet), wie zum Beispiel `com.example.app.my`. Bei Verwendung eines solchen URI-Schemas wird die entsprechende App gestartet, um die Anfrage zu verarbeiten.

Um eine OAuth-2.0-Autorisierungsanfrage mit einer URI-Schema-Umleitung durchzuführen, startet die native App den Browser mit einer Standard-Autorisierungsanfrage, bei der der Umleitungs-URI ein zuvor registriertes privates URI-Schema verwendet.

Es ist wichtig, dass Apps ein URI-Schema basierend auf einer unter ihrer Kontrolle stehenden Domain verwenden. Wie in RFC 7595 empfohlen, wird dieses Schema rückwärts ausgedrückt. Zum Beispiel kann eine App, die die Domain `my.app.example.com` kontrolliert, `com.example.app.my` als ihr Schema verwenden.

Nach Abschluss der Autorisierungsanfrage leitet der Autorisierungsserver zum normalen Umleitungs-URI zurück, der die native App startet und den URI als Startparameter übergibt. Die App kann dann die Autorisierungsantwort normal verarbeiten.

Claimed „https"-Schema-URI-Umleitung

Einige Betriebssysteme ermöglichen es Apps, *„https"-Schema-URIs* in den von ihnen kontrollierten Domänen zu beanspruchen (*claim*). Bei Aufruf eines solchen URIs startet der Browser nicht die Seite, sondern die native App mit dem URI als Startparameter.

Solche URIs können von nativen Apps als Umleitungs-URIs verwendet werden und sind für den Autorisierungsserver nicht von regulären, webbasierten Client-Umleitungs-URIs zu unterscheiden. App-beanspruchte *„https"-Schema-Umleitungs-URIs* bieten Vorteile hinsichtlich der Identifizierung der Ziel-App gegenüber anderen nativen Umleitungsoptionen, weshalb native Apps sie nutzen sollten, wo immer dies möglich ist.

Best Practices in puncto Sicherheit

Nachdem ich zuvor beschrieben habe, wie OIDC für die Authentifizierung nativer Apps genutzt werden kann, gebe ich nachfolgend einige Hinweise zu Best Practices in puncto Sicherheit.

Schutz des Autorisierungscodes durch PKCE

Die genannten Umleitungs-URI-Optionen teilen den Vorteil, dass nur eine native App auf demselben Gerät oder die Website der App den Autorisierungscode empfangen kann, was die Angriffsfläche begrenzt. Dennoch besteht die Möglichkeit des Code-Abfangens durch eine andere native App auf demselben Gerät.

Eine Einschränkung bei der Verwendung von privaten URI-Schemata für Umleitungs-URIs besteht darin, dass mehrere Apps in der Regel dasselbe Schema registrieren können, was unklar macht, welche App den Autorisierungscode empfangen wird. Die RFC 7636 zum bereits erwähnten PKCE-Protokoll beschreibt, wie diese Einschränkung für einen Angriff verwendet werden kann, um den Autorisierungscode abzufangen.

Umleitungs-URIs basierend auf Loopback-IP-Adressen können auf einigen Betriebssystemen anfällig für Abfangen durch andere Apps sein, die auf dieselbe Loopback-Schnittstelle zugreifen. Von Apps beanspruchte *„https"-Schema-Umleitungs-URIs* sind weniger anfällig für das Abfangen aufgrund der Anwesenheit der URI-Autorität, jedoch handelt es sich weiterhin um öffentliche Clients. Darüber hinaus wird der URI mithilfe des URI-Dispatch-Handlers des Betriebssystems mit unbekannten Sicherheitseigenschaften übermittelt.

Das PKCE-Protokoll wurde speziell entwickelt, um diesen Angriffsvektor abzuschwächen. Im Kern handelt es sich um eine Proof-of-Possession-Erweiterung für OAuth 2.0, die den Autorisierungscode vor dem Abfangen schützt, bevor dieser verwendet wird. Um diesen Schutz zu bieten, generiert der Client einen geheimen *Verifier*. Er übermittelt einen Hash dieses *Verifiers* in der initialen Autorisierungsanfrage und muss den nicht gehashten Verifier beim Einlösen des Autorisierungscodes gegen ein Token vorlegen. Eine App, die den Autorisierungscode abgefangen hat, würde nicht im Besitz dieses Geheimnisses sein und der Code wäre nutzlos.

Die Verwendung von PKCE ist zur sicheren Authentifizierung sowohl für Clients als auch für Server bei öffentlichen nativen App-Clients erforderlich. Autorisierungsserver sollten im Sinne einer Best Practice Autorisierungsanfragen von nativen Apps ohne PKCE ablehnen und einen entsprechenden Fehler zurückgeben.

Cross-App Request Forgery Protections

Cross-App-Request-Forgery-Angriffe ähneln stark den aus dem Web bekannten Cross-Site-Request-Forgery-Angriffen (CSRF). Um diese zu unterbinden, ist eine bewährte Methode, Client-Anfragen und -Antworten zu verknüpfen. Die RFC 6819 empfiehlt hierzu den `state`-Parameter zu verwenden.

Um *Cross-App Request Forgery* über Inter-App-URI-Kommunikationskanäle zu verhindern, wird ebenfalls empfohlen, dass native Apps eine hochentropische, sichere Zufallszahl im `state`-Parameter der Autorisierungsanfrage einschließen. Die App sollte eingehende Autorisierungsantworten ohne einen übereinstimmenden `state`-Wert für eine ausstehende ausgehende Autorisierungsanfrage ablehnen.

Die Verwendung des `state`-Parameters in diesem Kontext ermöglicht es der nativen App, Anfragen und Antworten zu verknüpfen und sicherzustellen, dass die Autorisierungsantworten auf gültige, ausstehende Autorisierungsanfragen zurückzuführen sind. Dieser Mechanismus stärkt die Sicherheit der Inter-App-Kommunikation und schützt vor CSRF-ähnlichen Angriffen, bei denen ein böswilliger Dritter versucht, unbefugt auf autorisierte Ressourcen zuzugreifen, indem er die Autorisierungsantworten manipuliert.

Probleme mit Embedded User-Agents

OAuth 2.0, im speziellen RFC 6749, dokumentiert zwei Ansätze für native Apps, um mit dem Autorisierungsendpunkt zu interagieren. Im Sinne einer Best Practice sollten native Apps eingebettete User-Agents nicht verwenden, um Autorisierungsanfragen durchzuführen. Autorisierungsendpunkte können und sollten sogar Maßnahmen ergreifen, um Autorisierungsanfragen in eingebetteten User-Agents zu erkennen und zu blockieren.

Eingebettete User-Agents sind zwar eine Möglichkeit zur Autorisierung nativer Apps, diese eingebetteten User-Agents sind aber per Definition unsicher für die Verwendung durch Dritte gegenüber dem Autorisierungsserver. Die App, die den eingebetteten User-Agent hostet, hat vollen Zugriff auf die Authentifizierungsinformationen des Benutzers und nicht nur auf die für die App vorgesehene OAuth-Autorisierungsgenehmigung.

In typischen Implementierungen von eingebetteten User-Agents basierend auf Web-Views kann die Host-Anwendung jeden Tastenanschlag im Anmeldeformular aufzeichnen, Benutzernamen und Passwörter erfassen oder sogar Formulare automatisch senden. Hierdurch kann die Zustimmung des Benutzers umgangen oder Sitzungscookies kopieren werden, um authentifizierte Aktionen im Namen des Benutzers durchzuführen.

Auch wenn sie von vertrauenswürdigen Apps desselben Anbieters wie der Autorisierungsserver verwendet werden, verstoßen eingebettete User-Agents gegen das Prinzip des geringsten Privilegs, da sie Zugriff auf mächtigere Anmeldeinformationen haben, als sie benötigen. Dadurch wird potenziell die Angriffsfläche erhöht.

Benutzer dazu zu ermutigen, Anmeldeinformationen in einen eingebetteten User-Agent ohne die übliche Adressleiste und sichtbaren Zertifikatsvalidierungsfunktionen, die Browser haben, einzugeben, macht es für den Benutzer unmöglich zu wissen, ob sie sich bei einer legitimen Website anmelden. Selbst wenn dies der Fall ist, führt es dazu, dass sie ohne vorherige Validierung der Website ihre Anmeldeinformationen eingeben.

Fazit

Die Best Practices für die Sicherheit und Effizienz der Authentifizierung in nativen Mobile-Apps legen den Fokus auf die Vermeidung eingebetteter User-Agents, die Integration von PKCE und eine präzise Handhabung von Autorisierungsanfragen. Diese Maßnahmen dienen nicht nur der Erhöhung der Sicherheit, sondern garantieren auch eine optimale Benutzererfahrung. Die Implementierung von OpenID Connect stellt eine zuverlässige Lösung für Authentifizierungsszenarien in nativen Apps dar.

Sven-Torben Janus

sven-torben.janus@conciso.de

https://twitter.com/sventorben

Sven-Torben Janus ist Partner bei der Conciso GmbH, wo er den Bereich Softwarearchitektur mit verantwortet. Er arbeitet als praktizierender Principal Software Architect und befürwortet einen agilen und praktikablen Entwurf von Softwarearchitekturen. Ein Schwerpunkt seiner Berater-Tätigkeit liegt im Bereich Identity & Access Management (IAM) mit besonderem Fokus auf die Open-Source-Lösung Keycloak.