# 1 Introduction

The Self-Issued OpenID Providers Specification Draft[1] describes two versions of the Self-Issued OpenID Provider Protocol, one of which is the Same-Device Flow. A core goal of the Self-Issued OpenID Provider Same-Device Flow is to provide authentication.

To achieve this, it is vital that no 'fresh' nonce or ID token associated with an honest End-User is leaked to an attacker. Otherwise, the attacker might be able to impersonate the End-User, or inject their own identity in a session of the End-User.

As of now (Jan 14, 2022), the specification's security considerations advise to take care when invoking the Self-Issued OP, but give little guidance for the Self-Issued OP invoking the browser in the response. However, if the Self-Issued OP procedes without caution here, an attacker controlling an application on the device where the Self-Issued OP resides, might be able to obtain the Authentication Response by the Self-Issued OP, under certain assumptions.

# 2 Attacker model

We assume that the attacker

- controls an application on the same device as the Self-Issued OP of the End-User.
- can call the Self-Issued OP of the End-User.
- has no control over the operating system.
- can obtain responses from the Self-Issued OP - either by the Self-Issued OP giving responses to the caller without further checks, or by the attacker application registering as a handler for URLs[2].

We also consider the End-User that is careless in certain concerns and will authorize requests, if they seem sufficiently legitimate.

# 3 Same-Device Token Leak

The Same-Device Flow might allow an attack similar to the request replay attack on Cross-Device Self-Issued OP from the Security Considerations of the Specification under certain assumptions.

In short an attacker can bypass the use of a legitimate browser, call a Self-Issued OP and - if the user authorizes the request and the attacker applications gets access to the response - the attacker can impersonate the End-User at an honest RP.

[1] `https://bitbucket.org/openid/connect/src/master/openid-connect-self-issued-v2/openid-connect-self-issued-v2-1_0.md`

[2] Outside of the theoretical possibility of this, see Section 3.4.1 for a possible setup in Android.

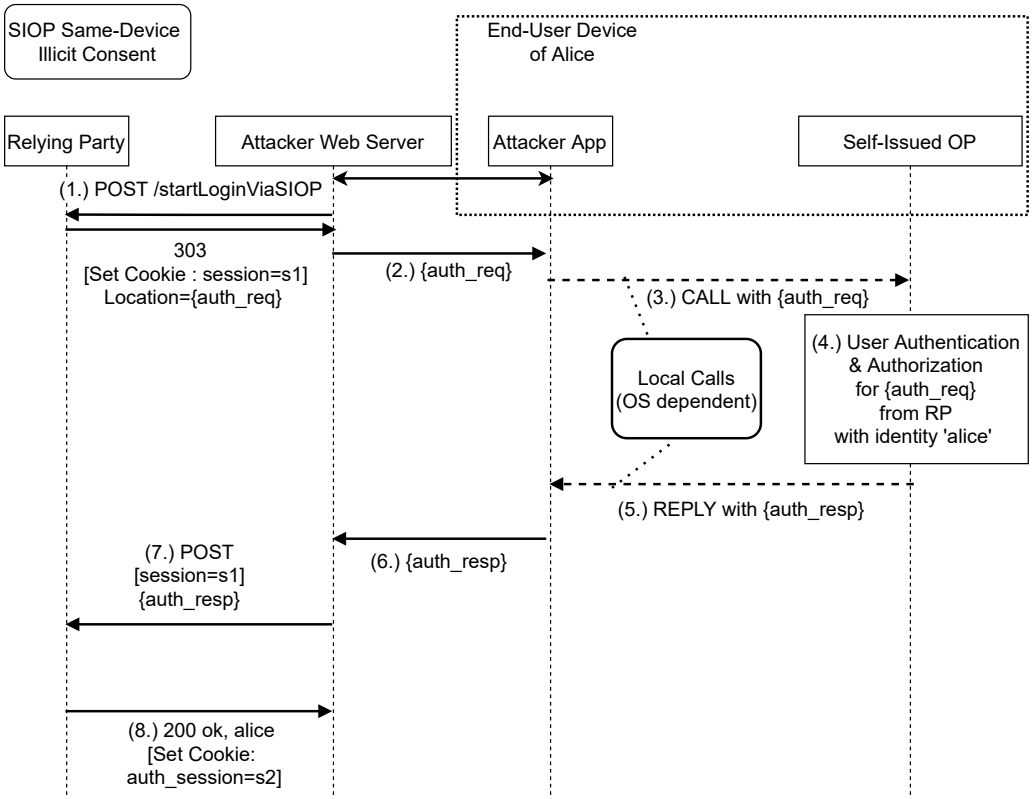


Figure 1: Sketch of the ID Token Leak - {auth_resp} contains an ID token and possibly other parameters

## 3.1 Result of the Attack

The attacker obtains an ID token from the End-User Self-Issued OP for a nonce and audience of their choosing. With this they can impersonate the End-User, breaking authentication.

As the attacker with the ID token obtain a fresh nonce this can also allow the attacker to log the user in under an identity of the attacker using a CSRF attack - given the user starts the login flow in a local browser, and the Self-Issued OP answers to the attackers application instead of the browser.

## 3.2 Attack Flow

The attack proceeds as follows:

1. The attacker starts a login flow at a Relying Party using their web server. Towards the RP they behave like a legitimate browser that requests a login via a Self-Issued OP.

2. The attacker passes the Authentication Request they receive to their application on the End-Users Device.

3. The attacker's application calls the Self-Issued OP with the Authentication Request, like a local browser would.

4. The (careless) End-User authorizes the Authentication Request for the Relying Party (aud) in the Self-Issued OP.

5. The Self-Issued OP gives the Authentication Response to the calling application - the application controlled by the attacker- , or gives the End-User the option to reply to the attacker's application and the End-User chooses this option.

6. The attacker's application posts the Authentication Response to the attacker's web server.

7. The attacker's web server passes the Authentication Response to the Relying Party.

8. The Relying Party confirms that the attacker is logged in under an identity of the End-User.

Figure 1 illustrates the information flow of the attack.

## 3.3 Suggested Mitigation

We see that an honest Self-Issued OP should not trust the requests it receives, and should be wary of the context they originate from. In particular, a Self-Issued OP needs to act in the awareness that it emits sensitive data and needs to take care to where it passes the response data to - the response must remain known only by the Self-Issued OP, the RP and the browser as a trusted intermediary. Valid recipients include only trustworthy browsers (and potentially the Relying Party directly if it is a local app).

We suggest to add to the Security Considerations a note, for the Self-Issued OP implementers to take (OS dependent) measures to ensure the response of the Self-Issued OP is only given to trustworthy recipients and thus remains confidential.

The remainder of this section sketches some possible measures to achieve this.

**Local RP Application** If the Self-Issued OP wants to support RPs that are applications on the same device as the Self-Issued OP, the Self-Issued OP needs to check if the a legitimate App is installed for handling the redirect_uri of the RP and ensure that only this app receives the response. For this the Self-Issued OP needs to take considerations for secure App2App Communication[3] into account.

If the Self-Issued OP cannot verify that a legitimate RP app is installed, it needs to take OS dependent measures such that only legitimate browsers are called.

**Default Browser** A very restrictive possibility to ensure that the response only is passed to a good browser is to limit the Self-Issued OP to only pass its responses to the default browser of the system[4]. While this is promising to prevent leakage of data, it obstructs cases where the End-User uses different browsers.

[3] `https://danielfett.de/2020/11/27/improving-app2app/`

[4] We assume that this browser is a good one.

**List of trusted Browsers** The Self-Issued OP might be configured to give responses back only to a fixed list of trusted browsers (e.g. ones that the implementers of the Self-Issued OP have verified to be legitimate and compatible with the Self-Issued OP) when they cannot verify a local application as a legitimate recipient. If the Self-Issued OP can identify the calling application, they can then filter out requests from malicious callers and give back responses only to good ones. If the Self-Issued OP cannot verify the identity of the calling application, then they can present the list of trusted browsers to the user and let the user choose one of these.

Depending on the required security level a Self-Issued OP could also only warn the End-User if the origin of a request[5] cannot be verified to be a trusted browser, or simply drop the request.

## 3.4 In Practice

The process of applications (including browsers) calling applications looks different depending on the OS and there seems to be no unifying standard. As such, general statements about the behavior of applications are difficult. In this section we take a look at the plausibility of this attack in Android.

### 3.4.1 Android

For the attack, the attacker needs to control an app on the End-Users device with an Intent Filter like in Figure 2 or, alternatively, with the specific RP redirect_uri instead of the wildcard * in their manifest.

> Note: From Android 12 (API 31) onwards these kind of deep links without 'proof' for authorization to handle the URL are to be prevented.[a] It will however take time until a majority of Android devices run this OS version. Currently, Android 11 and 10 are the most prevalent versions[b].
>
> [a] `https://developer.android.com/training/app-links/deep-linking` (first note)
> [b] `https://gs.statcounter.com/android-version-market-share/mobile/worldwide/#monthly-202102-202201`

If the End-User now authorizes a request, and the Self-Issued OP replies with code like

```
Intent httpsIntent = new Intent(Intent.ACTION_VIEW, response_uri));
startActivity(httpsIntent);
```

the End-User is presented with the choice of the malicious application and possible legitimate browsers for the response. An implementer also might be tempted to obtain the calling application by using the `getReferrer`[6] method of the `Activity` class (even if the documentation states its result should not be trusted).

```
Uri referrer = getReferrer();
Intent httpsIntent = new Intent(Intent.ACTION_VIEW, response_uri));
httpsIntent.setPackage(referrer.getHost());
```

[5] It might suffice to verify where the response is passed to. Even cases where the attacker can spoof the origin to look legitimate, or no 'origin' is available, it might be sufficient to check that the response is passed to a legitimate recipient and not an arbitrary, potentially malicious app.

[6] `https://developer.android.com/reference/android/app/Activity#getReferrer()`

```
<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="https"
        android:host="*" />
</intent-filter>
```

Figure 2: Intent Filter for URLs

```
startActivity(httpsIntent);
```

In this case, if the attacker application invokes the Self-Issued OP (for an honest RP), and the user authorizes the response, the Self-Issued OP directly responds to the attackers app - leaking a valid ID token for a nonce and an RP of the attackers choice.

# 4 Conclusion

Even if the attack requires strong assumptions to be viable in practice, the openness (with respect to possible OS / environments for the protocol to run in) of the specification does not explicitly disallow the scenario.

As such, we advise to clarify in the security considerations of the specification the need for confidentiality of the authentication response sent by the Self-Issued OP when calling the browser (or potentially a native RP application). Then, implementers can evaluate options in their respective OS / environment to secure the response.