Federated login to native applications

The right way

Author: S. Veeke

Version: 1.0

Date: 22-04-2016

Table of Contents

1 Terminology

NREN
National Research and Education Network.

Native application
An application program that has been developed for a specific platform or device. Other used terms are nonweb applications and rich clients.

SURFconext
SURFconext offers SAML-based single sign-on-access to a large number of online services. More than 120 Dutch organisations in the area of higher education and research are using SURFconext to securely log in to connected cloud services. In this proof of concept SURFconext was used to authenticate users.

SAML
Security Assertion Markup Language is an open-standard data format for exchanging authentication and authorisation data between Identity Providers and Service Providers.

OAuth
OAuth is an open standard for authorisation. The demo Service Provider uses OAuth 2.0.

URL-scheme
Custom URL schemes allow users to open applications from within other applications. For example, an e-mail application could be opened from a web browser using applicationname://token-info.

2 Introduction

SURFnet and many others NREN’s are using web based protocols such as SAML 2.0 for their authentication and authorisation platforms. Although this makes for a great and secure user experience on the web, it is difficult to apply this method to native non-web applications. These native applications are typically designed for an operating system and often do not support federated login.

Because there is no generic cross-operating system solution available, some developers implement various workarounds such as embedded browsers and application specific passwords. These methods are undesirable from a security standpoint. With most workarounds the user is unable to verify the server address and the SSL-certificate. In addition, application specific passwords are a hindrance to users and provide an additional access mechanism which can be exploited. The user’s credentials should never leave the Identity Provider so third parties won’t be able to intercept these.

With increasing demand for SURFconext to support federated login within native applications and the lack of an obvious and secure solution, SURFnet[1] and Egeniq[2] have examined federated login using the system browser. The results are described in this report.

[1] https://www.surf.nl
[2] https://www.egeniq.nl

3 Project

3.1 Problem definition

SAML heavily depends on common browser capabilities (such as HTTP-redirects), which works very well in a web browser. Native applications can communicate with web resources, but to fully support SAML and for example all the different login pages and technologies used by Identity Providers, an application developer must actually recreate a web browser inside the native application. This is undesirable, as building web browsers is often not the core business of most application developers. Therefore, several methods exist for developers to integrate federated login using SAML into native applications. This report identifies and compares the available methods (the results are described in chapter 5).

All methods use OAuth to facilitate authorization between a web browser component (used for authentication) and the application server.

3.2 Scope

This project consists of two objectives:

1. Finding the best method to do federated login in native applications using the (system) web browser. Demo applications will be built as a proof of concept.
2. Developing an SDK with documentation for every best method on all investigated operating systems. These SDK's will be reference implementations from SURFnet.

This report is the final product of the first project objective. The second objective will be completed in November 2015.

The scope of this project offers no solution to broader non-web challenges such as web authentication for SSH-access or native e-mail clients[3].

3.3 Example use cases

Below are some possible use cases for the OAuth-scenario.

- SURFnet's own SURFdrive uses an embedded browser for authenticating through SURFconext. This is an undesirable situation since users cannot see the location they're connecting to and cannot verify the SSL-certificate.
- Some application developers that want to connect their application to SURFconext struggle with the federated login scenario. SURFnet can help them a lot by providing them with a guide with best practices and ready to use SDK's.
- Some application developers use methods for federated login that are undesirable from a security standpoint. It helps if SURFnet has the aforementioned guide, SDK's and documentation to convince the application developers to change their applications with a more robust and secure solution.

3.4 Criteria

Methods for web-based authentication will be held against the following requirements (both from the user's and method's perspective):

- The user can see the web address the client is connecting to.
- The user is able to validate the SSL-certificate on the server.
- The user's credentials cannot be hijacked by the application.

[3] Such as Microsoft Outlook and Mozilla Thunderbird.

- The user’s credentials remain at the Identity Provider.
- The user can choose his or her favourite browser to use.
- The method must be user friendly.
- The method works with all common[4] browsers.
- The method works on a widely used version of the platform.
- The method is supported by the operating system and company behind it.
- The method doesn’t store user data (credentials, personal data etc.)

[4] Common browsers are Mozilla Firefox, Google Chrome/Chromium, Microsoft Internet Explorer/Edge and Apple Safari.

4 Proof of concept setup

4.1 Server

The proof of concept demo applications consists of platform specific demonstrator applications in combination with an authentication server. The server component uses a CentOS 7 webserver with OAuth 2.0 and a SAML 2.0 Service Provider. This setup was connected to SURFconext. The Identity Provider part was provided by the SURFconext DIY IdP. All used software:

- CentOS Linux 7
- Apache2 (httpd)
- mod_auth_mellon (https://github.com/UNINETT/mod_auth_mellon)
- php-oauth-as (https://github.com/fkooman/php-oauth-as)

4.2 Clients

For the clients a selection of commonly used, future proof and popular versions of desktop and mobile operating systems has been made. This means the following platforms have been examined:

- Microsoft Windows 8(.1) and 10
- Microsoft Windows Phone
- Windows RT
- Apple OS X 10
- Apple iOS 8 and 9
- Android

Windows 8(.1), 10, RT and Phone

Windows Desktop 8, 8.1, 10, Windows RT and Windows Phone are quite similar from a developer's viewpoint so it is relatively easy to investigate all of them. Moreover, a lot of people will be using Windows 8 and 10 in the near future so those platforms are most interesting.

Apple OS X 10

Apple OS X 10 is the most widely used Apple desktop operating system. The current version is 10.11 (El Capitan) but it is likely many people still use 10.10 or 10.9.

Apple iOS 8 and 9

When this project started, Apple iOS 9 was not released yet. However, Apple already announced a new method for authentication through the web browser in iOS 9. This method is not backwards compatible so that is why iOS 8 and iOS 9 are both included.

Android

Android is by far the most used smartphone operating system. Since there are a lot of different versions of Android around, the platform has been generally investigated.

The code and documentation of the SDK's are published on GitHub.

- https://github.com/SURFnet/nonweb-sso-android
- https://github.com/SURFnet/nonweb-sso-ios
- https://github.com/SURFnet/nonweb-sso-osx
- https://github.com/SURFnet/nonweb-sso-windows

5 Results

This chapter describes the results of the investigation into all available methods for integrating federated login with native applications. The first paragraph consists of ‘generic methods’, which are available on all investigated operating systems. The second part consists of ‘specific methods’; methods that are only available for a specific operating system.

5.1 Generic methods

There are two methods which can be used in all of the investigated operating systems, referred to as ‘Webview’ and ‘Browser redirect’.

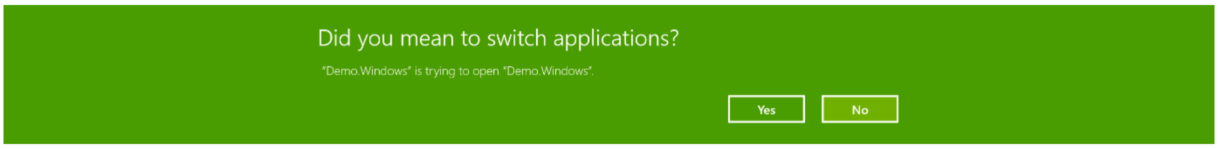
5.1.1 Webview

Webview (also known as ‘Embedded browser’) is an element in the operating system that acts like a mini-browser within the application it is used by. Every operating system has its own name for it (Apple: WebView, Android: WebView, Microsoft: x-ms-webview).

When the native application wants to authenticate, a browser window will appear inside a predefined area of the native application. This window is a stripped down version of the web browser that is included by default in the operating system. No address bar is shown so the user is not able to verify the website he is visiting (e.g. if the correct address is used and if the certificate is valid). In terms of user friendliness, this is a relatively good method: the user does not leave the application and the flow is very simple and natural. For some operating systems, user interaction is required due to dialogs like ‘Did you mean to switch applications?’.

In our setup the flow is as follows (example screenshot is from Windows 8.1):

1. The user wants to login, the Webview opens within the application.
2. The SURFconext discovery page is shown and the user selects his Identity Provider.
3. The user enters his or her credentials and logs in at his Identity Provider.
4. Some operating systems (mostly Desktop) will ask the user whether the native application should be opened. For example, Windows will ask the user if he or she meant to switch applications (see figure 1).
5. After pressing ‘yes’ the user is brought back to the application (with an OAuth-token).


Did you mean to switch applications?

"Demo.Windows" is trying to open "Demo.Windows".

Figure 1 - Did you mean to switch applications?

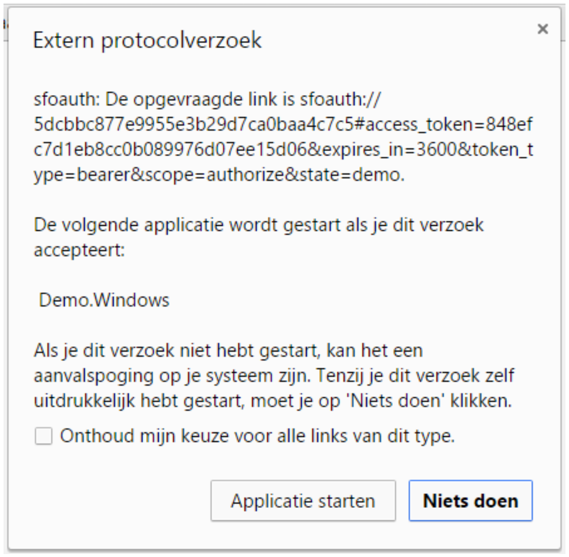
5.1.2 Browser redirect

Another well-known and commonly used method is to allow the user to use the default system browser for the login process. The user authenticates using the browser and the browser then redirects the user back with an OAuth-token to the application using a call-back URL-scheme. An example of this would be: ‘applicationname://token-details’.

When the application wants to authenticate, it will open the default web browser. Because it is a full browser, the address bar will be shown such that the user can verify whether the website is authentic. In terms of user friendliness, the major disadvantage is that the user must switch applications and a dialog like the one shown in figure 2 could be shown. On Windows, this dialog is system-triggered and cannot be overridden. This might trigger the user to click ‘do nothing’ and thus he will not return to the native application, which will not receive the token. On other platforms the dialog is not system-triggered but can be triggered by the browser itself (for instance Chrome will always show a dialog). Another advantage of this method is that most web browsers offer ways to save the user's credentials. Finally,the browser is able to access the user's credentials in case certificates are used for authentication,

In our setup the flow is as follows (example screenshot is from Windows in combination with Google Chrome):

1. The user wants to login, the application automatically opens the default system browser.
2. The SURFconext discovery page is shown and the user selects his Identity Provider.
3. The user enters his credentials at his Identity Provider (or uses the browser's autofill password feature) and logs in.
4. The browser asks the user if another application should be opened (figure 2).
5. After pressing 'Start application' the user is brought back to the application (with an OAuth-token).



Extern protocolverzoek

sfoauth: De opgevraagde link is sfoauth://5dcbbc877e9955e3b29d7ca0baa4c7c5#access_token=848efc7d1eb8cc0b089976d07ee15d06&expires_in=3600&token_type=bearer&scope=authorize&state=demo.

De volgende applicatie wordt gestart als je dit verzoek accepteert:

Demo.Windows

Als je dit verzoek niet hebt gestart, kan het een aanvalspoging op je systeem zijn. Tenzij je dit verzoek zelf uitdrukkelijk hebt gestart, moet je op 'Niets doen' klikken.

☐ Onthoud mijn keuze voor alle links van dit type.

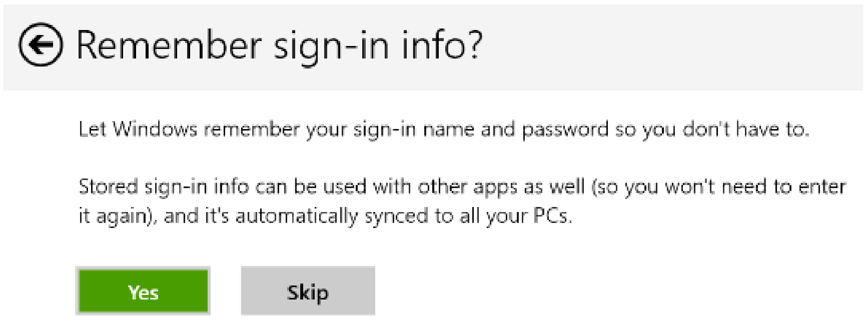
Figure 2 – Open external application?

5.2 Specific methods

5.2.1 Microsoft Windows – WebAuthenticationBroker

The Microsoft WebAuthenticationBroker is a class specifically designed to handle the OAuth-scenario. When supplied with the URL and corresponding method, the system will open a dialog which is the same dialog Windows uses to log into its own services like Microsoft Outlook and Xbox Live. Once the user has finished logging on (or has cancelled the procedure), a callback URL is called which is unique for the application. This means that, unlike with Webview and Browser Redirection, the token cannot be intercepted.

Figure 3 - Remember sign-in info?



Remember sign-in info?

Let Windows remember your sign-in name and password so you don't have to.

Stored sign-in info can be used with other apps as well (so you won't need to enter it again), and it's automatically synced to all your PCs.

Because the opened dialog is a system dialog, the application becomes modal and gets put in a 'suspended' state. An application developer has no control over de contents and is not able to read or manipulate the data entered by the user. Also, the developer has no control whatsoever about the look and feel of the user interface of the dialog. This means that it is not possible to show an address bar or control dialogs like the 'Would you like to save this password?'-dialog. The passwords are stored in the 'Credential Manager' under 'Web Credentials' and will be synced with the Microsoft Cloud so they can be synced across devices which use the same Microsoft account.

5.2.2 Apple iOS – Safari View Controller

When using OAuth-authorization on iOS, Apple does not give developers much choice for handling it. In the past there had been two ways to handle it: launching the built-in Safari web browser (browser redirect) or using a WebView component within the application itself. Somewhere in June 2015 Apple started rejecting applications[5][6][7][8] which used the browser redirection method, because Apple finds the user experience insufficient. Around the same time Apple introduced the Safari View Controller for iOS 9. This caused problems for developers who were reliant on the browser redirection method, since iOS 9 would be released mid-September. Despite the fact that browser redirection is no viable option anymore, it is still included in the outcomes of this report by means of a fall-back option for older iOS versions.

Safari View Controller is a miniature web browser window that is capable of being launched from the application, but without leaving the application. It is more or less a hybrid between Webview and Browser Redirect. It shares cookies and saved passwords with the default browser on iOS (Safari). Passwords are saved locally and encrypted, with the option of syncing them with other devices through the cloud.

This method will only be available in iOS 9 and above and although Apple iOS has relatively good adoption rates[9], this will be a problem for a widespread deployment in 2015 and 2016.

The flow will feel very responsive and easy: Safari View Controller is faster than Webview and has the same look in all applications it is used in. Furthermore, the launched window has a read-only address bar so users can verify whether the website is authentic.

Because the Safari View Controller is launched mostly for login purposes and can save passwords, there are a lot of times when it is launched and closed immediately if the user was already logged in. In this case it is possible to start the Safari View Controller as hidden[10]. If the user was already logged in, then no prompt is shown.

5.2.3 Android – Chrome Customs Tabs

Chrome Custom Tabs are the Chrome implementation of the native-to-web content transition. It is part of Google Chrome, a separate application which is not included on Android smartphones by default. Other browsers and applications can decide to offer an identical API and developers are free to choose any custom tabs running on the device.

The mechanism of discovering custom tabs applications is based on the Android intent resolver. The application requests a list of all applications on the device that can handle the URL and have the custom tabs API so the developer can differentiate the applications by name and if available, select Chrome.

The content is loaded from a provided URL. It is not possible to create, read or alter the content because the custom tabs are a complete black box to the application, the developer can start Chrome custom tabs but cannot stop it. Once the custom tabs have been started, the application has no control, it only receives callbacks with the GET URLs of the Chrome navigation. Any active sessions or cookies are shared with the custom tabs mode and any logins done in the custom tabs mode remain active in the Chrome browser. For an OAuth-

[5] http://kickingbear.com/blog/archives/492
[6] https://discussion.evernote.com/topic/27026-my-app-is-rejected-by-apple-for-popup-the-oauth-login-page-in-safari-please-help/
[7] http://sixcolors.com/link/2014/09/when-apple-forces-an-app-to-be-less-secure/
[8] https://groups.google.com/forum/m/#!topic/oauth/xo9V5-qWBjY
[9] Apple iOS 8 had an adoption rate of 90% in 12 months.
[10] https://library.launchkit.io/how-ios-9-s-safari-view-controller-could-completely-change-your-app-s-onboarding-experience-2bcf2305137f

flow this means there is no way to automatically return to the application once authentication has been successful.

Normally when the OAuth URL is opened in a web browser, the application is paused and sent to the background. However, when the URL is opened in custom tabs, Chrome prevents the application from being evicted by the system. As long as the custom tab is on top of the application, it raises its importance to the 'foreground level'. This puts the application in a special state, outside the regular foreground and background.

By setting the colour of the toolbar and the 'back'-icon, the application developer can blend Chrome's custom tabs in the overall design and make transitions between native and web content more seamless without having to resort to a WebView. The Chrome browser in the custom tabs mode has a minimalist look, so no navigation bar, no address bar and no tabs. Therefore, users won't be able to verify if the website is authentic. Just like with browser redirection, the URL-scheme can be copied. This causes the same potential security risks: if another (malicious) application deliberately copies the scheme in order to be chosen by the user, it can intercept the token.

5.3 Intercepting tokens

Most of the methods described above use URL-schemes in order to function. URL-schemes make it possible to open applications from other applications. This can also be used to perform specific actions, like pass through tokens from a browser to an application.

A disadvantage of using URL-schemes is that the registered URLs are not protected. This means that any other application can register the same URL-scheme and potentially intercept the OAuth-token. Most of the methods are susceptible, see the table at 5.4.

Hackers can exploit this by building an application with the same redirect scheme and getting it installed on the user's phone. If these conditions are met (if multiple applications on the user's device claim the same URL-scheme), the user will be presented a selector window, where he can select between the applications which should handle the URL (in our case, which one receives the token). A hacker can easily mimic the 'real' application by using the same name and icon as well. If the user selects the malicious application, it still needs to swap the authorization token for an access token. This can be done by using the client secret, which can be reverse-engineered from the application.

On most operating systems (except iOS 9) there is no system to determine which application is linked to the registered URL-scheme. One possible solution is that the application uses an ID only known to itself and the server, with which the server can encode the token and the application can use this encoded token. This also requires changes on the server. Even this approach would not be 100% secure since this ID (even if you encrypt it) could be reverse-engineered in native applications.

For iOS 9 though there is a way to securely redirect the user back to the application with Universal Links[11]. The developer can place a .json-file which contains the unique AppID on the root of their website. The associated domain where iOS can retrieve that json-file are defined in the application's .plist-file. The file has a unique ID that can never be used by another application and a link which will open your application when the browser tries to open it. For iOS 9 applications, this is the recommended way.

Even without additional measures, the hacker needs to invest a lot of time and effort to intercept tokens. The URL-scheme and swapping of the authorization token for an access

[11] https://developer.apple.com/library/ios/documentation/General/Conceptual/AppSearch/UniversalLinks.html

token needs to be figured out and an application has to be built. Moreover, the hacker has to convince the user to install his malicious application (alongside the official application) and click it when the choice is presented. The SDK's do not provide additional counter measures since the interception of tokens is a rather adverse side effect of using oauth2.

5.4 Overview methods

All the strong and weak points of different methods for authenticating through a web browser are included in the table below. The following definitions are based on the criteria formulated in chapter 3.3, some of which have been merged. When a method does not score 'Yes', the reason is listed in the table. A more elaborated explanation is given in chapter 5, Findings.

Security	The method is secure and robust	Yes	Yes, but…	No
User data	The method doesn't store user data (credentials, personal data etc.)	Yes	Locally encrypted	Cloud/unencrypted
Address bar	The user can verify the server address and SSL-certificate	Yes		No
User experience	The method provides a good user experience and is user friendly	Yes	Average	No
Specific browser	The method works with all browsers	Yes	No, system browser	No, third party browser
Widely used	The method is available on a widely used version of the platform	Yes	Yes, but…	No
Supported	The method is supported by the operating system	Yes	Yes but…	No

Method	Security	User data	Address bar	User experience	Specific browser	Widely used	Supported
Windows x-ms-webview	🟥 Insecure URL-scheme	🟥 Insecure	🟥 Shows no address bar	🟨 Switch applications dialog	🟩	🟩	🟩
Windows browser redirect	🟨 Token interception	🟨 Can save credentials in web browser	🟩	🟨 Another application in flow + protocol request dialog	🟩	🟩	🟩
Windows WebAuthenticationBroker	🟩	🟥 Can save credentials in cloud	🟥 Shows no address bar	🟨 Save password dialog	🟩	🟨 No Windows 7 support	🟩
Apple OS X WebView	🟨 Token interception	🟥 App has control over inputted credentials	🟥 Shows no address bar	🟩	🟩	🟩	🟩
Apple OS X browser redirect	🟨 Token interception	🟨 Can save credentials in web browser	🟩	🟨 Another application in flow + dialog	🟩	🟩	🟩
Apple iOS WebView	🟨 Bad security reputation	🟥 App has control over inputted credentials	🟥 Shows no address bar	🟩	🟩	🟩	🟩
Apple iOS Safari browser login	🟨 Token interception	🟨 Can save credentials in web browser	🟩	🟨 Another application in flow + dialog	🟨 System browser	🟩	🟥 Apple rejects apps
Apple iOS Safari View Controller	🟨 Token interception	🟨 Can save credentials	🟩	🟨 Save password dialog	🟩	🟥 Only in iOS 9	🟩
Android WebView	🟩	🟥 App has control over inputted credentials	🟥 Shows no address bar	🟩	🟩	🟩	🟩
Android Chrome Custom Tabs	🟨 Token interception	🟨 Can save credentials in browser	🟥 Shows no address bar	🟥 No automatic return to app	🟥 Third party browser	🟨 Only devices with Chrome	🟩
Android browser redirect	🟨 Token interception	🟨 Can save credentials in web browser	🟩	🟨 Another application in flow	🟩	🟩	🟩

Table 1 - Overview methods

6 Conclusion

Summary

There are several methods for federated login to native applications. Each operating system provides similar methods. For example, Webview and Browser Redirect are commonly used methods on all the operating systems. In addition, Microsoft, Apple and Google provide specialised methods such as Microsoft's WebAuthenticationBroker class, Apple's Safari View Controller (iOS 9) and Google's Chrome Custom Tabs for Android.

Webview has proven to be insecure and inadequate because of security issues and/or the absence of an address bar. It is important that users can verify the authenticity and security of the website they are visiting, this is not possible with the embedded Webview components provided by Microsoft, Apple and Google.

Microsoft's WebAuthenticationBroker class is also inadequate because it automatically saves user credentials in the Microsoft Cloud and doesn't show an address bar.

Google's Chrome Custom Tabs is only available on Android smartphones with (a recent version of) the Chrome web browser installed on it. This means that most users won't be able to use it. In addition, no address bar is shown to the user and there is no way to automatically return to the application once authentication has been successful.

Apple's Safari View Controller has similar features as Webview (the user doesn't leave the native application) but shares many things with the default browser (Safari) like saved passwords However, unlike Webview, Safari Web Controller does show the address bar so users can verify whether the website is secure. The main drawback of Safari View Controller is that it is not backwards compatible with previous versions of iOS. It is expected though that iOS 9 will have had substantial adoption among users by September 2016.

Browser redirection is available on all operating systems (except iOS 9) and proved to be the best method. Because the default web browser is used, the user has full access to the address bar and all of the browser functionality. Despite this there are also disadvantages. The user flow is less smooth than Webview because the user must temporarily switch to another application (the web browser). Instead of 'application -> in-app webview -> application' the user is presented a more complicated 'application -> system web browser -> application'. Also, mostly on desktop platforms, the OS or the browser will ask the user if he really wants to switch applications when the browser wants to return to the application using a callback URL. Another disadvantage is that other (malicious) applications can register the same URL-scheme inside the operating system. Those applications can then potentially intercept the response token.

Conclusion

Because of the above, Browser redirect is the best method for Windows, Apple OS X and Android. Unfortunately, Apple rejects iOS applications which use Browser redirect and more’s the pity that Apple provides no secure alternative for earlier iOS versions. Around September 2016 this will likely be no problem anymore since most of the iOS users will run iOS 9 by then. Until then, there are a two possible scenarios (with options 2 and 3 being insecure but pragmatic):

1) Only support Apple iOS 9 (iOS 8 or earlier users won’t be able to use the application).
2) Use Safari View Controller on iOS 9 with Browser redirect as fall-back on earlier versions.
3) Make two versions of the application (iOS 9 will be secure; iOS 8 with Webview will not be).

Based on the findings above, SURFnet and Egeniq have developed SDK’s and documentation for the following methods:

- Microsoft Windows: Browser redirect (desktop and mobile)
- Apple OS X: Browser redirect
- Android: Browser redirect
- Apple iOS 9 Safari View Controller, with Browser redirect as fall-back

The code and documentation of the SDK’s are published on GitHub.

- https://github.com/SURFnet/nonweb-sso-android
- https://github.com/SURFnet/nonweb-sso-ios
- https://github.com/SURFnet/nonweb-sso-osx
- https://github.com/SURFnet/nonweb-sso-windows