DevAuth: Device Based Federated Login

Ravinder Singh
Stony Brook University
ravindsingh@cs.stonybrook.edu

Prateek Narendra
Stony Brook University
pnarendra@cs.stonybrook.edu

Kai-Chieh Huang
Stony Brook University
kaichhuang@cs.stonybrook.edu

Nishant Shankar
Stony Brook University
nshankar@cs.stonybrook.edu

Abstract

Federated login using protocols like OpenID Connect is convenient as it solves the problem of password reuse by allowing the use of the same account across different websites, but that creates another issue. The main problem in this infrastructure is reliance on these third-party identity providers to relay information to client applications. We seek to address the problem of our reliance on corporate ownership of sensitive user information. In this paper we propose DevAuth, a device based federated login mechanism that selectively allows client applications access to user information without them going through third parties. We design a flow for DevAuth that resembles the RFC-6819 implicit flow, and assess the security implications of such a protocol. We show that although future work needs to be done to holistically assess the protocol, it is a viable alternative to current federated login techniques.

1 Introduction

Password systems have been used as a way to authenticate users, as it is easy for user to use and simple for servers to implement. However, as time goes by, a lot of security issues start to appear not only due to poor protection of the server, but in how users manage their passwords. Thus, alternatives for user authentication have been proposed in recent years. OpenID Connect has seen increasing adoption across a wide variety of websites. It is primarily used to authenticate a user while signing up or logging in to client application. Most of the times authentication is done via social networking platforms or big tech providers, like Google and Facebook. Currently, the most popular authorization mechanisms have a single point of failure, that is, if the authorization server is compromised, then every user who has their data stored on it is leaked.

In this paper, we focus on designing a new mechanism that allows users to control their personal information. By ensuring that "identity providers" are not the actors that authenticate people, we are adding a certain flexibility to the federated login philosophy. Our design flow is based on OpenID connect flow, while changing the validation third-party to be our mobile device. The most different part from OpenID is that when user is signing in using third-party account, there are always a server handling the user request, but as we change our authentication mechanism to mobile device, there is no longer a server listening for the request. The flow here is replaced by notification service, by pushing the notifications, we are able to actively connect and wake up our phone to handle the requests, thus, with our small modification to the flow, we are introducing the new mechanism DevAuth.

Our paper is structured as follows. Section 2 briefly discusses prior work in the area of authentication mechanisms, with respect to passwords and federated login mechanisms. Section 3 delves into the design of DevAuth. As per this design, the threat model is assessed in Section 4. Section 5 details the implementation level concerns that we faced while developing a proof of concept. A list of extensions and improvements to be developed as part of future research are discussed in Section 6.

2 Related Work

2.1 Passwords

Plain text passwords have been used as a method to authenticate for a long period. Plain text password carries some benefits, as evaluated in previous work [2]. Take easy to use for example- everybody knows how to set or input a password, and it is also easy to recover or reset your account once you lose your password. Although benefits are apparent at a first glance, a deeper look shows that these systems are laden with problems.

From the perspective of an implementer, user passwords stored on a system need to be protected from illegal access which involves sophisticated methods of protection. Any error in these methods would result in the password of users being leaked or stolen [4]. From the perspective of the user, as we have access to multiple accounts, the number of credentials

to remember increases. This makes the user reuse passwords which allows attackers to use simple dictionary attacks to com-promise a system. A study has shown that 51% of the users reuse their passwords across multiple platforms and some make simple edits to their passwords, which are easy to identify [4].

2.2 SAML

Security Assertion Markup Language (SAML) is an XML based standard used for authentication across multiple web domains and is used as a way to implement the single-sign-on model for identity management. The SAML suite [3] provides a variety of profiles to achieve SSO, but all of them rely on XML and implicitly assume the use of web domains. The architecture follows a request/response flow wherein the Service Provider (SP) requests for identity information about a user from an Identity Provider (IP). Trust between domains is established using certificates/keys and provider-specific metadata. The flow of SAML is ideally what we want in authentication systems, but in practice has poor interoperability with RESTful APIs and mobile applications. Due to its mature ecosystem and web support, it is mainly implemented in enterprise level software to allow employee access to different web domains through an SSO. Although there exist some solutions that extend SAML to work on native apps, it is relatively cumbersome to deploy, and therefore consumer directed SSO options tend to avoid SAML

2.3 Federated Login

Modern authorization mechanisms rely on the OAuth 2.0 protocol. While OAuth was designed for authorization to data access, it was not designed for authentication purpose. However, many applications did use OAuth for authentication by writing some wrappers over it. The reason that OAuth shouldn't be used for authentication is because there is no standard way for getting user information and it doesn't have a common set of scopes for access. To resolve these issues with minimal overhead, researchers came up with OpenID Connect. It is not a separate standard or a protocol. It is a small extension of OAuth 2.0 for authentication. The additions for OpenID Connect are 1) Basic information of the user (ID Token), 2) 'userInfo' endpoint for getting more user information, 3) Standard set of scopes, 4) Standardized implementation.

The flow of OpenID Connect is very similar to that of OAuth. Except that in the initial call to the Authorization server, it adds 'openid' to the scope. And ID token is returned by the authorization server which the application can use to get basic user details [1]. With an OpenID system the user only needs to remember only a single master credential, hence limiting the number of credentials required. It is also easy for users to adapt, as the interface resembles the simple login/password systems. OpenID systems are mature and have been adopted by multiple platforms, with a strong developer community. Though OpenID seems like a holy grail solution, it is susceptible to information leak from within i.e. the owners of the credentials have complete access to other systems the user has granted access.

2.4 Self-Sovereign Identities

Self-Sovereign Identities(SSI) is a system that allows users to create and own their identities. This system enables the user to be the sole owner of his/her identity and can share his personal information at his/her discretion. These systems are being portrayed as alternatives to Federated SSO systems as these systems remove the need for a centralized authority [8]. The proponents of SSI cite problems like data breaches [6], non-ownership of the users identity and the mass analysis of the user's personal information for marketing [10] as the main evils of federated systems. To overcome the drawbacks, SSI employs decentralization using a blockchain system. The user's personal information is disseminated over a set of peers in an encrypted format. On request by the user, the information is retrieved from the system and presented to any willing entity that needs authentication [7].Though a blockchain-based SSI system can be implemented for authentication (uport, ShoCard and BlockAuth) [7], their intent is mostly for authorization of resources and applications involving Know Your Customer (KYC). These systems involve extensive use of cryptography keys that requires special storage mechanism, that may involve a hardware token or special software. These systems are are also infrastructure-heavy they require a substantial cost to set up and operate. A SSI based system has a steep learning curve and cannot be easily taught to the an average user. The current implementation of these systems cater to specific use cases of authentication and will need to develop further to be made mainstream [5].

3 Design

The major points we considered while designing our new system are -

1. It must be easy for developers to integrate with their pre-existing applications using OpenID and OAuth for authentication and authorization.
2. Data in transit must be secure
3. Store personal information of the user on their personal device and nowhere else.
4. Meet the same security guarantees offered by OAuth

To make it easy for developers to integrate the new authorization flow into their client applications, we felt it was necessary to closely model it after the most commonly used Authorization method - OAuth.

We conducted a comprehensive study of OAuth - their design goals and implementation. Modeling our flow closely to that of OAuth also helps us to meet the security guarantees offered by it. One of the points we discovered early on is their efficient use of Implicit and Explicit flows, which keeps data in transit secure from eavesdroppers.

If we were to store personal information on a users personal device, then the users personal device becomes analogous to resource server in OAuth. However, the major limitation of a personal device is that it does not have the resources to run a server instance. To overcome this, we decide to develop a repository where users register their devices and provide a unique ID that helps this repository forward requests from the client applications to this device using their Mobile Operating Systems Notification Service. The personal device can then respond to the third party app, providing all the data it requires if the user chooses to do so - hence behaving like the resource server in OAuth.

We needed to build a portal where client applications can register themselves so that they can contact our server when trying to authenticate a user trying to login or get access to user data. Figure 1 describes the process of registration. During registration, the developers of the client application have to specify the scope of data and also the endpoint where they can receive this data. This is done to as to notify the user about the information the client application is requesting for when it asks for it. If registered successfully, an application token is provided, which is to be sent on future requests to our system for authenticating an user or requesting data from existing users.

For the system to forward client application requests to their personal device, we need to maintain an identifier which can be used to contact the users' personal device for access to their personal data which resides on it. Figure 2 describes how a new user can register their personal device onto this service. Since this can only be done from the users personal device, we need a mobile application where the user can register their device and receive the push notifications from our system from here on.

After the client has registered their application on our service and the user has registered their device on our service, we are now ready to help client applications receive access to data that they desire. Figure 3 describes this flow. The user who desires to login into the client application is first redirected to our system where they log in. This redirection has a randomized request ID associated with it to identify it later on when the user grants permission for data to be accessed. If the user successfully logs onto our system, then the system forwards a notification to the phone containing the randomized request ID, the scope of data that the client application requires and the endpoint where data can be sent to. If the user grants the permission, then the users' personal device sends a HTTP request with the personal data in the payload along with the randomized request ID to help identlfy

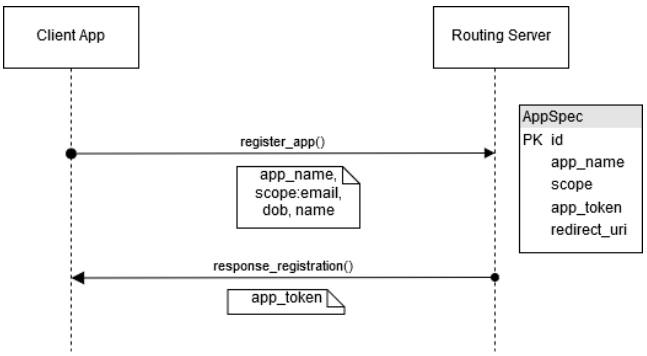

Figure 1: Client App Registration

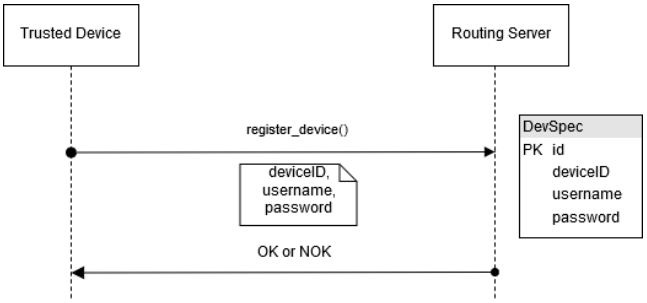

Figure 2: Device Registration

the request that it is responding to. If a user chooses not to grant permission, they can do so.

4 Threat Model

Threat Modelling signals the vulnerabilities in the structure and outlines necessary safeguards required to remove potential threats from a system. Being a novel design, we define the threat model for DevAuth and show the measures taken to remove potential threats. Since, we intended DevAuth to be identical to OpenID we refer to the RFC-6819 [9], published

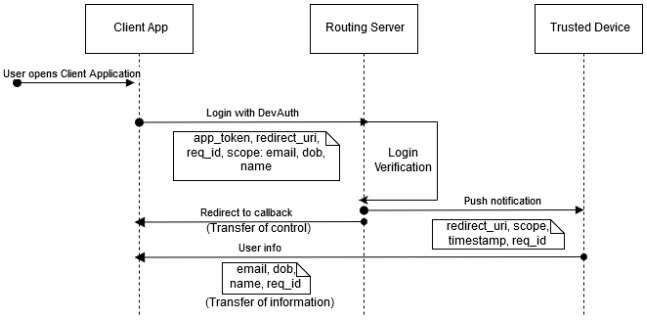

Figure 3: Login Flow

by the Internet Engineering Task Force, that details the threat model for OAuth2. Since OpenID piggybacks on OAuth2 it is logical to take inspiration from the threat model designed for OAuth2.

The threat model document identifies and details around 100 vulnerabilities and provides guidelines on how to prevent known vulnerabilities. The vulnerabilities target both the front-channel and back-channel design. DevAuth makes use of protocols that are identical to the front-channel, without the use of TokenIDs and exchange of TokenIDs for authorization. Hence, we focus on the front-channel vulnerabilities and omit vulnerabilities that target the back-channel design. We also focus on the vulnerabilities that target the message flow. The assessment follows STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege).

1. **Malicious Client Obtains Authorization**: A scenario where a malicious entity may obtain information by fraud and pretend to be the authorized client. Such scenarios can be prevented by authenticating the client at the routing server. We provide this mechanism by issuing a ClientID token that authenticates the client. Along, with the authentication, we need to perform a TLS digital certificate signature validation at the routing server to ensure the requests are coming from a legitimate source.

2. **Open Redirection on Clients**: The information from the end-user needs to be sent to the client, hence the client provides a redirect URL for the information to be forwarded to. These URLs can be tampered with and this information can be passed to a malicious end-point that will leak information. To prevent such information leaks, while registering a client we will also require the client to specify the full redirect URL. So when a request for Authentication is received from the client, we verify the redirect URL based on the TLS digital certificate signature and source of the redirect URL.

3. **Obtaining User Secrets on Transport**: Any adversary eavesdropping on the communication must not retrieve secret information. This is ensured by the use of TLS for communication between the client, routing server and the users' trusted device.

4. **Replay of Authorized Resource Server Requests**: An adversary eavesdropping on the communication between the client and routing server sees public information like the request ID that is part of the URL to identify a unique request from the client. An adversary could reuse this request to launch a replay attack. We prevent this by making the request ID as a one-time use token. We store the requestID and the ClientID to identify previously witnessed requests and discard them. We ensure that such replay attacks are prevented by employing sufficient entropy in the request ID to avoid duplicate requestID from being generated.

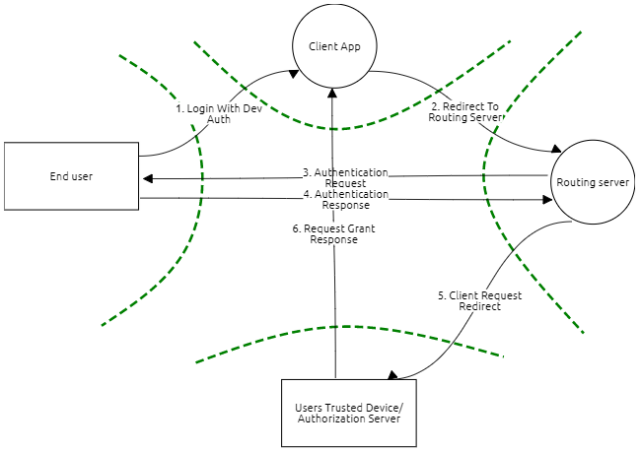

Figure 4: Threat model

5. **DDoS Attacks for the exhaustion of Resources**: A malicious entity could flood the server with many requests and try to exhaust the number of requestIDs. This is avoided by the use of UUID4 that has sufficient entropy and would take the attackers 264 messages to exhaust the requestID space.

6. **Change in Scope Values**: The client can be malicious trying to elevate the scope requested from the end-user. This way the client can obtain a greater privilege and would lead to an information leak. This is avoided by making the client register a scope. Any requests sent by the client must be a subset of the scope specified during registration. Additionally, the mobile app also notifies the end-user of the scope information requested by the client. Based on this information the end-user may accept or reject the authentication request.

5 Proof of Concept

To validate the design we build a prototype of DevAuth. The design involved the development of the routing server, the mobile application, and a sample client application. The routing server is developed using Django that houses a web application and web services. The web services have end-points to register a client application and the end-user. The web-services are accessed using POST calls. The web application is intended to be a login terminal that allows end-users to authenticate themselves on the routing server. Once authenticated, the authorization request from the client application is forwarded to the mobile application. To forward the requests

Figure 5: Mobile application

from the client to the end-users mobile device we used OneSignal. This service provided us push-notification capabilities to any mobile device, whose device ID we possess.

The client app is also developed in Django. The client has a web-interface that allows clients to log in with DevAuth, which is similar to many login services that have interfaces for federated login. The client application has two endpoints one for login and the other for once the login act is completed. This emulates the login transactions that occur in many web-applications.

The mobile application is using React Native as a hybrid solution. This application is able to let user sign up an account and register with OneSignal to receive push notifications from the routing server. The client houses a user interface that displays the authentication requests from a client application[5]. The interface displays the client application name and the OpenID scope information requested by the client for authorization. On response from the end-user, the mobile application generates a POST call to the client application with the required information.

6 Future Work

In the future work section, as we discuss a lot about what we have already implemented and designed, we can confirm that such architecture is workable. However, to truly put into practice in the real world, we still need some modification and farther design.

1. **Implementing Session Management for Client App**: Current implementation of DevAuth does not have support for Session Management. This problem involves a design decision to resolve. Should the session management be done at the client end after authentication or DevAuth handle the session management. If DevAuth takes care of session management, then we need to add in more protocols that will support the exchange of a refresh token.

2. **Revocation of the Access Right**: In a scenario when the end-user wants to revoke the authentication given to a client. This requirement is in line with the right to forget. Currently, OpenID has no support to implement such a policy. But this is an intriguing problem as there is a demand to implement such measures.

3. **Performance Assessment of Mobile Device**: A measurement assessment needs to be conducted to elicit the performance metrics of the device on multiple requests. This will be more crucial when session management will require the end-users mobile device. Additionally, users' usability assessment also needs to conducted to see if the app is easy and familiar to the users for authentication.

4. **Changing Auth Mobile Devices**: As we secure all the profile data of a user on mobile device, a scenario when the end-user changes his mobile device needs to be assessed. This will involve the adaptation of the routing server to the new change in a secure manner. The main consideration to be looked into is how do we safely migrate the device credentials to a new device.

References

[1] Openid implicit flow. .

[2] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. *2012 IEEE Symposium on Security and Privacy*, pages 553–567, 2012.

[3] Conor Cahill, John Aol, Atos Hughes, Hal Origin, Bea Lockhart, Michael Systems, Beach, Rebekah Boeing, Booz Metz, Rick Hamilton, Booz Randall, Hamilton Allen, Irving Wisniewski, Hewlett-Packard Reid, Paula Austel, Maryann Ibm, Hondo, Michael Ibm, Mcintosh, and Trustgenix. Profiles for the oasis security assertion markup language (saml) v2. 0. 01 2004.

[4] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and Xiaofeng Wang. The tangled web of password reuse. In *NDSS*, 2014.

[5] Uwe Der, Stefan Jähnichen, and Jan Sürmeli. Self-sovereign identity − opportunities and challenges for the digital revolution. 2017.

[6] Jim Isaak and Mina Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51:56–59, 08 2018.

[7] Shu Yun Lim, Pascal Tankam Fotsing, Abdullah Almasri, Omar Musa, Miss Laiha Mat Kiah, Tan Fong Ang, and Reza Ismail. Blockchain technology the identity management and authentication service disruptor: A survey. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2):1735–1745, 2018.

[8] Quinten Stokkink and J.A. Pouwelse. Deployment of a blockchain-based self-sovereign identity, 06 2018.

[9] M. McGloin T. Lodderstedt, Ed. and P. Hunt. Oauth 2.0 threat model and security considerations. RFC 6819, 1 2013.

[10] Julia Woolley, Anthony Limperos, and Mary Beth Oliver. The 2008 presidential election, 2.0: A content analysis of user-generated political facebook groups.