Editors: Brian Chess, bchess@vantuyl.com | Brad Arkin, barkin@adobe.com

Integrating User Customization and Authentication: The Identity Crisis

Željko Obrenović and Bart den Haak | Backbase

Web applications, such as iGoogle, My Yahoo, or services based on the emerging user-experience-management platforms,[1] are increasingly more personalized, giving users the power to tailor Web applications to their needs. User customization is often tightly coupled with security. It usually requires authentication because personalized applications must be able to uniquely identify users. It also requires securely storing users' personal data.[2] Even if customization data don't contain sensitive information (for example, customization might be limited to data about color, layout, and page element positions), connecting user identities obtained from the authentication provider to these settings poses security challenges.

On the basis of our experiences in integrating advanced user customization mechanisms with existing security infrastructures, we've identified four integration patterns (see Figure 1):

- the *local-user pattern*,
- the *external-user pattern*,
- the *local- + external-user pattern*, and
- the *masked-external-user pattern*.

Personalization is also possible without authentication and without server-side persistence of user preferences, using anonymous-user customization approaches. For example, many websites encode user preferences in a cookie sent back to the browser. The next time the user accesses a page, the server receives the cookie with the preferences and can personalize the page according to these preferences. In this article, however, we don't address such forms of personalization but focus on situations requiring long-term and cross-device persistence of application-specific user customization on the server side.

The Local-User Pattern

This pattern is the simplest way to combine customization and authentication. The application maintains an internal database with data about users and their settings. The database is used for both customization and authentication (see Figure 1a).

In the Web's early days, this was the de facto standard for Web applications. Also, this pattern is relatively simple and convenient for development and testing because there's no need to connect to external systems. However, this pattern's popularity is declining. As users interact with an increasing number of applications, many of them find managing credentials for all the sites difficult and are reluctant to create new accounts. Some users adopt flawed strategies to deal with this information overload, such as using the same password for all sites, using easily memorized passwords, or writing down their passwords and storing them in easily discovered places.[3]

Another drawback is that many implementations of this pattern store user credentials in the same database as user public information and customization preferences. This complicates maintenance and backup because it requires extra care to protect user credentials in the database and the backup storage. It also increases the risk of accidentally revealing usernames and passwords.[4]

The External-User Pattern

This pattern delegates user authentication to external applications. It defines and manages users apart from the application that employs user customization (see Figure 1b). External authentication providers, such as LDAP (Lightweight

1540-7993/12/$31.00 © 2012 IEEE

Directory Access Protocol) or pre-authentication servers, authenticate users. The application uses returned user identifiers (such as usernames) as keys for storing and retrieving customized data.

This pattern's main advantage is that it stores user credentials in a secure external environment, not with the user customization data. Also, users can use the same authentication provider to authenticate with different applications. This pattern makes sense only in situations in which authentication providers are trusted or under your control, such as on your company's LDAP server.

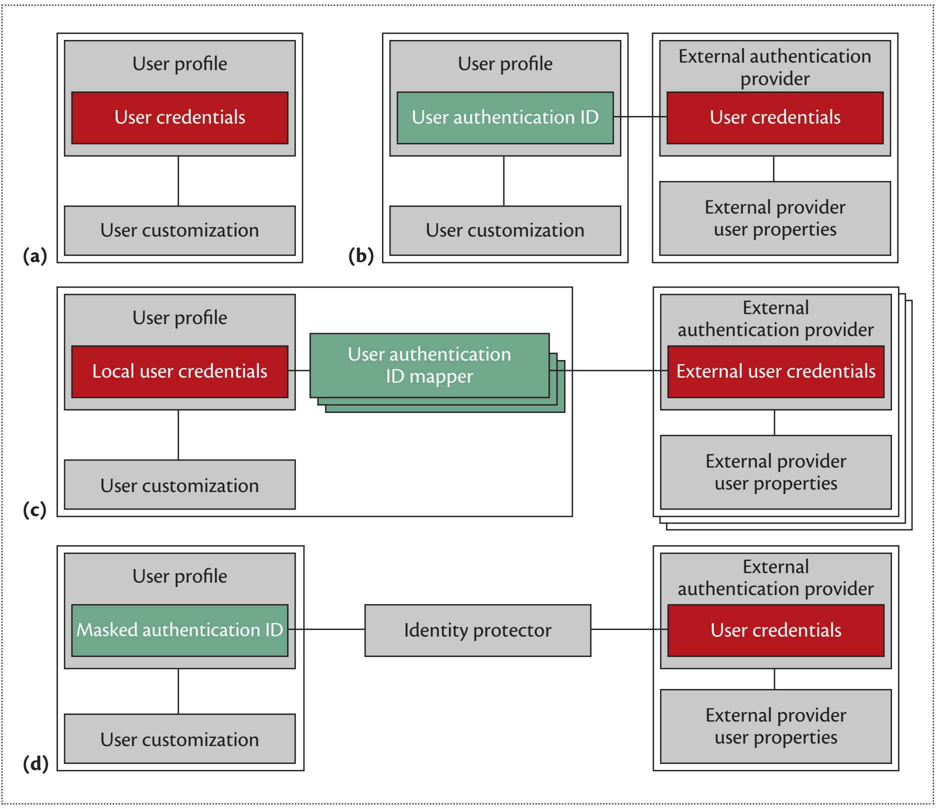


Figure 1. Four patterns for integrating user customization with user authentication. (a) The local-user pattern stores user customization profiles and user credentials in the personalization database. (b) The external-user pattern stores user preferences in a preferences database and stores user credentials in the external authentication provider's database. (c) The local- + external-user pattern always associates user profiles with local user accounts, but profiles can be optionally connected to one or more external authentication providers. (d) The masked-external-user pattern stores user identifiers with customized user profiles in a protected form.

The Local- + External-User Pattern

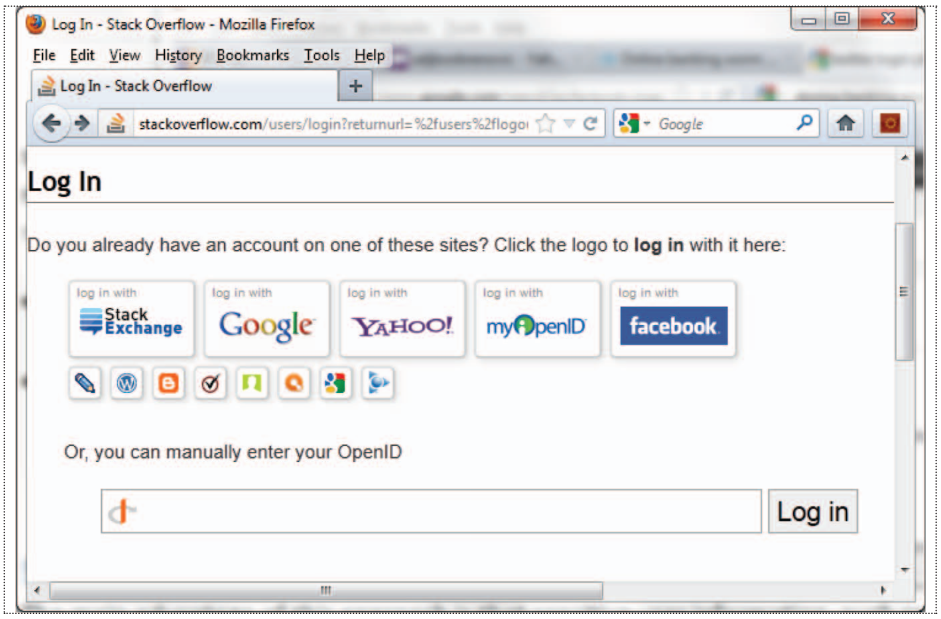
The local-user and external-user patterns are often combined. In the resulting pattern, users have a local account but can connect multiple external authentication providers to that account (see Figure 1c). The local account's user identifier stores the user customization data. After the connection with the external authentication providers is established, users can authenticate using both the local account and those authentication providers, in all cases accessing the same customization data.

This pattern has become common with the increased popularity of OpenID (http://openid.net) and OAuth (http://oauth.net). OpenID is a distributed-identity system that lets people use a single username and password to log in and authenticate themselves at any OpenID-compliant website. Google, Yahoo, and many other providers use it. OAuth is an authorization protocol based on a token-based mechanism that allows third-party websites or applications to access users' data without the users needing to share login credentials. Facebook and Twitter use it. Although OpenID and OAuth differ, they both can serve as external authentication providers.[5]

Users don't have to explicitly create the local account. The system might implicitly create such an account when users first sign in with external providers. Authentication through local accounts is optional and can be disabled, and local-user credentials can be limited to only a unique user identifier. This approach can solve the local-user pattern's main problem—that it stores user credentials in the same place as user public data and customization settings.

Various user interface design approaches can help simplify management of local accounts and external authentication providers' links. For example, local accounts' creation can be hidden from users. Users simply "sign up" for new applications using a Facebook account or an OpenID provider. The applications then create local accounts in the background and automatically connect them to the original sign-up providers, often populating the local accounts with data from those providers.

This approach is flexible and user friendly. Users can choose how to authenticate, reusing their existing accounts. Users can be given a choice of provider they'd like to authenticate with, in all cases accessing the same customization data. Users also can choose not to reveal their external identities; that is, they can use only the local account. This pattern also gives more control to website administrators because they can control user


Log In - Stack Overflow - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Log In - Stack Overflow
stackoverflow.com/users/login?returnurl=%2fusers%2flogo
Google
log in with Stack Exchange
log in with Google
log in with YAHOO!
log in with myOpenID
log in with facebook
Log in

Log In

Do you already have an account on one of these sites? Click the logo to **log in** with it here:

Or, you can manually enter your OpenID

Figure 2. The StackOverflow login page. StackOverflow offers 14 ways to log in via external authentication providers.

account creation and can deploy an additional approval step.

Relying on external providers poses several risks. For instance, some external authorities might not work for periods of time; such problems will prevent users from accessing sites.

Another challenge is the increasing use of social networking sites as authentication providers.[5] One potential problem is phishing attacks. For example, the Ramnit worm recently received attention in the news for stealing thousands of login and password details for Facebook users.[6] With stolen Facebook credentials, attackers can log on to not only Facebook but also sites where users have enabled Facebook authentication.

Another problem is that many social networking sites use OAuth, through which applications also gain access to the OAuth site's API. Potential application vulnerabilities might give malicious code access to additional information about users or even perform actions on users' behalf, such as sending messages. For example, when someone authenticates as a Twitter user, he or she can do almost everything the Twitter API provides, including updating the user profile and posting messages. Facebook is potentially more restrictive, and authenticating applications can be granted access to a subset of API functionality. However, even if sites give more granularity, developers should ask for only minimal access rights. But many developers end up asking for more rights than their applications actually need.

As users log on to more and more websites using their social network credentials, they're becoming less critical and are routinely approving all access requests from applications. Google provides an interesting solution for this problem; it combines OpenID and OAuth. Developers can use OpenID without OAuth for authentication and combine it with OAuth if they want access to Google APIs. Generally, we recommend using OpenID over OAuth because OpenID is a pure authentication provider, whereas OAuth always comes with some authorization. However, if you want to get more users to your site, you might not have such a choice because many popular social authentication providers are using OAuth only. When using OAuth, if your primary goal is authentication, you must study the site's API carefully and ask as little as possible of the permission.

Another potential problem with this pattern is that, to simplify login and attract more users, some sites offer dozens of authentication providers but hide that those providers use different protocols and standards. For example, StackOverflow offers 14 external login options (see Figure 2). Although you might use these options in the same way, they use different standards: OpenID, OAuth 1.0 Rev. A (http://tools.ietf.org/html/rfc5849), and OAuth 2.0 (http://oauth.net/2). Each standard and its implementation have their own characteristics and risks.[7] Having multiple authentication providers increases the risks because more (potentially compromised) access routes to the application exist.

Although adding multiple authentication providers might improve the user experience, it could present usability problems. For example, users who have accounts on multiple social sites might accidentally create several unconnected accounts. This can happen when users come back to a site that they've used before but that no longer recognizes them (the cookie expired or was deleted). The site will present users the same choices it did before. However, if they can't remember which authority they used and they choose a different one, it won't recognize them as returning users and will create new accounts.

The Masked-External-User Pattern

One problem with the previous patterns is that the unique user identifiers must be stored in the customization database to enable you to retrieve and store customized data based on authenticated users. In some situations, such as banking applications, the identifiers might themselves be sensitive information, such as bank account numbers, Social Security numbers, or private email addresses.[8]

What further complicates secure implementation of user customization in these situations is that many applications add personalization as a separate layer atop the existing system. This layer often has its own storage and might run on a different server with different security settings than other parts of the system. So, managing and storing sensitive information in this layer requires care. The key challenge, thus, is how to uniquely identify users and store their customization without storing sensitive information in the customization database.

Some external authentication providers can themselves mask users' identities. Some OpenID providers have this ability in the form of *claimed identifiers* that might be opaque.[9] Different OpenID providers employ different implementations of claimed identifiers. Some providers return different identifiers for each relying party so that user accounts can't be correlated across sites. Other providers, such as Yahoo, enable users to have several identifiers for the same account and to choose which to use when logging on to a website. However, the latter approach puts the responsibility on users to create and use opaque identifiers rather than identifiers that are easier to remember but might contain sensitive information.

If authentication providers can't mask users' identity, a mechanism that masks sensitive user identifiers with new, safer user identifiers is necessary (see Figure 1d). Two approaches to such masking exist:

- Instead of storing user identifiers directly, use encrypted user identifiers as safe identifiers for the customization system.
- Use a separate service to map sensitive user identifiers to secure user identifiers. The application then can exchange the sensitive user identifiers for the secure user identifiers, and use only the latter. This is similar to token exchange in OAuth or Central Authentication Service systems, but this exchange shouldn't be limited to a current session.

Masking's main advantage is that it doesn't store sensitive user information with less sensitive customization settings. This minimizes the possibility of revealing user identities and other sensitive information if the customization database is compromised.

Masking adds a layer of complexity and might require infrastructure changes. The masked-external-user pattern isn't common, and few practical resources outline its potential problems. Encrypting user identifiers requires an encryption module, and maintaining and configuring that module require additional care. If you use a service to exchange sensitive user identifiers for secure ones, this service must have its own database that stores the mapping between the identities. Such a database introduces additional overhead and risks.

Many people have come to expect user customization, but creating good designs isn't trivial. Even simple things, such as connecting user identities to noncritical preferences, pose challenges. The patterns we presented are relevant not only for personalization but also for any situation that requires mapping user identities to application-specific data. ■

References

1. G. Phifer, *The Emerging User Experience Platform*, tech. report G00211625, Gartner, 2011; www.gartner.com/id=1610217.
2. A. Rezgui, A. Bouguettaya, and M.Y. Eltoweissy, "Privacy on the Web: Facts, Challenges, and Solutions," *IEEE Security & Privacy*, Nov./Dec. 2003, pp. 40–49.
3. D.A. Norman, "When Security Gets in the Way," *ACM Interactions*, vol. 16, no. 6, 2009, pp. 60–63.
4. M. Dembowski, "How to NOT Store User Credentials in a Database," blog, 1 Oct. 2012; http://blog.goyello.com/2012/01/10/not-store-user-credentials-database.
5. M.N. Ko et al., "Social-Networks Connect Services," *Computer*, Aug. 2010, pp. 37–43.
6. J. Kirk, "Ramnit Worm Goes after Facebook Credentials," *Computerworld*, 5 Jan. 2012; www.computerworld.com/s/article/9223173/Ramnit_worm_goes_after_Facebook_credentials.
7. R. Wang, S. Chen, and X. Wang, "Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services," *Proc. 2012 IEEE Symp. Security and Privacy*, IEEE, 2012, pp. 365–379.
8. V. Moen and T. Tjøstheim, "Case Study: Online Banking Security," *IEEE Security & Privacy*, Mar./Apr. 2006, pp. 14–20.
9. "Authentication Best Practices—Claimed Identifiers vs. Email Addresses," Google; https://developers.google.com/google-apps/marketplace/best_practices#claimed.

Željko Obrenović is a researcher and best-practices evangelist at Backbase. Contact him at obren@acm.org; http://obren.info.

Bart den Haak is an IT architect at Backbase. Contact him at bart.denhaak@softsens.com.