Designing user authentication systems

By Cal Henderson

If your web application has more than one class of users, the you're going to need some sort of authentication system. On the web, authentication systems have to constantly carry "tokens" around, submitting them to the server at every request. This is a downside to the stateless HTTP model - infact, with web based applications, you'll spend alot of time creating the illusion of a stateful environment through the use of authentication and location tokens.

When designing your own authentication system, there are a few things you'll need to decide upon. The first question you should be asking yourself is if you really need to write your own. An insecure authentication system can leave your application data and even your server copen to attackers. If you decide you really do need to write your own system, then read through the following checklist first.

Elements

There are a number of elements to an authentication system. You'll need login and logout functions. In some cases you'll want a logged in user to be able to log in as another user without first logging out. Once a user has logged in, you'll need to track their session until they logout. You might want to store cookies so they remain logged in on their next visit. You might want to give users this option too.

Users often forget their login credentials - a good authentication system allows users to retrieve or reset their password. The cleanest method for this is to store an email address for each user, and email them their password on request. For systems where passwords can't be retrieved, the resetting of a password should require email confirmation to stop malicious users restting each other's passwords. If you allow users to register then you'll need a registration system which checks for duplicate details (usernames, email address). A good registration system requires email confirmation to activate a user account. This ensures you have a working email address for all your users and also cuts down on trolls registering - the more hoops you give users to jump through, the more undedicated users will give up. This works in reverse too though - more people will register if there are less steps. A good registration system chould be as quick and simple to use as possible.

Storage

Account details should generally be stored in a database of some sort, though flat files are acceptable. As always, if working with flat files, ensure that you don't leave yourself open to filename based attacks. There are two real issues with storage: encryption and session storage.

The encryption question is simple: encrypt stored passwords or not? Encryption of stored passwords should be done with a one way cipher like unix crypt() (Built into most databases as the PASSWORD() function). If password are encrypted then they become safe from administrators accidentally seeing them, and saf(er) from being exposed anywhere by the application. The downside is that you can't retrieve a user's password - you have to reset it instead. It also means that admins can't quickly log in as another user to check/test anything. Unless you build in a feature to let them log in with the password.

Session storage is an interesting one: should you store the session as part of the user record or have a seperate table of sessions. The reason for the later is simple - to enable more than one session per user. Why might you want multiple sessions per user? Well, more than one person might use a single account for one thing. If this isn't the case, it can still sometimes be useful to allow multiples sessions. If one person is working on two different computers at the same time, or logs in once then logs in again by mistake, then it's nice for their first login not to expire. On the

other hand, single session per user allows for tighter access control.

HTTPS

You have three basic options here. You can use HTTPS, not use it, or use it for the important bits. Using HTTPS completely provides the most secure method for authentication - you can pass login details around in POST data and no one can spy on them. The downside is that it's very slow. Painfully slow over a modem and annoying slow on a fast connection.

You could opt for no HTTPS at all. The advantages are cost (no certificate needed) and speed. The downside is that at some point you'll be passing your login details around in plain text. If you don't use sessions then you might be passing them around all the time; very insecure.

The intermediate solution is to use HTTPS where it matters: perform the submission of login details over HTTPS, then create a session and pass it around over plain HTTP. This gives the advantage of security where you need it, and speed everywhere else. You still need a secure certificate, but if you're serious about security then the small cost is nothing.

Token Passing

Once you have authentication tokens, how do you pass them around? Cookies are a nice option - they remove the need for explicit passing code in each link or form in your system. On the downside, they aren't supported everywhere and using them can open up interesting security holes (See my article about that). Another option is to pass them around as query parameters in your GET urls. This is pretty insecure - they appear in log files and in the browser's titlebar. It's pretty disconcerting for users when their password appears on the screen in plain text. You could always use a simple encoding (like base64) to mask them, but that doesn't help from a security point of view - just in tricking your users. The third solution is to always pass the details over HTTP POST. This has the effect that the details don't appear on screen or in log files. The huge downside here is that youn have to use all forms instead of links, which can get very annoying very quickly and turn complex pages into huge monstrosities. Overall, the cookie option is cleanest - just make sure you add in security checks in all places that perform actual (data-changing) actions.

Sessions

To use sessions or not to? Sessions are nice for a number of reasons. They stop you passing around login details. If found out they are only useful for one time, often for only a short time. They reduce token passing to only a single token. The downside is that sessions can timeout (else they're no more secure than passing around login details) and add another layer of complexity to your authentication system.

People often get stuck with deciding on the format and length of a session token. A session variable with uppercase letters and digits withs 36 different characters and is easy to generate. With just 5 characters you have over 60 million (36^5) possible sessions. A 10 character session gives you three and a half thousand trillion possible sessions (36^10). A good way to think about the granularity of your sessions is thus: estimate the total amount of sessions you expect to ever be active at once. Multiply this number by a million. If you have that many possible sessions then an attacker would have to try a million combinations to find a valid session. And that turns into a DOS attack rather than a session attack ;)

Conclusion

This has only been a very brief introduction to creating user authentication systems, but should give you some idea of the decisions and issues involved.

(end of article)