Authenticated Session

(a.k.a. Server-Side Cookies, Single Sign-On)

Abstract

An authenticated session allows a Web user to access multiple access-restricted pages on a Web site without having to re-authenticate on every page request. Most Web application development environments provide basic session mechanisms. This pattern incorporates user authentication into the basic session model.

Problem

HTTP is a stateless, transaction-oriented protocol. Every page request is a separate atomic transaction with the Web server. But most interesting Web applications require some sort of session model, in which multiple user page requests are combined into an interactive experience. As a result, most Web application environments offer basic session semantics built atop the HTTP protocol. And the protocol itself has evolved to provide mechanisms—such as basic authentication and cookies—that allow session models to operate correctly across different Web browsers.

An obvious use for session semantics is to allow users to authenticate themselves once instead of every time they access a restricted page. However, great care must be taken when using session semantics in a trusted fashion. Most session mechanisms are perfectly adequate for tracking non-critical data and implementing innocuous transactions. In such cases, if an end user circumvents the session mechanism, no harm is caused. But it is easy to make mistakes when applying session mechanisms to situations where accountability, integrity, and privacy are critical.

Solution

An authenticated session keeps track of a user's authenticated identity through the duration of a Web session. It allows a Web user to access multiple protected pages on the Web site without having to re-authenticate him-/herself on every page request. It keeps track of the last page access time and causes the session to expire after a predetermined period of inactivity.

The server maintains the authenticated user identity and the time of the last requests as part of the client session data. Every protected page contains a standard header (executed on the server) that checks the authentication information associated with the session.

The first time the user requests a protected page, the server executes the authentication check and notes that no authenticated identity is present in the session information. At that point, the server records the original request and redirects the user to a login page.

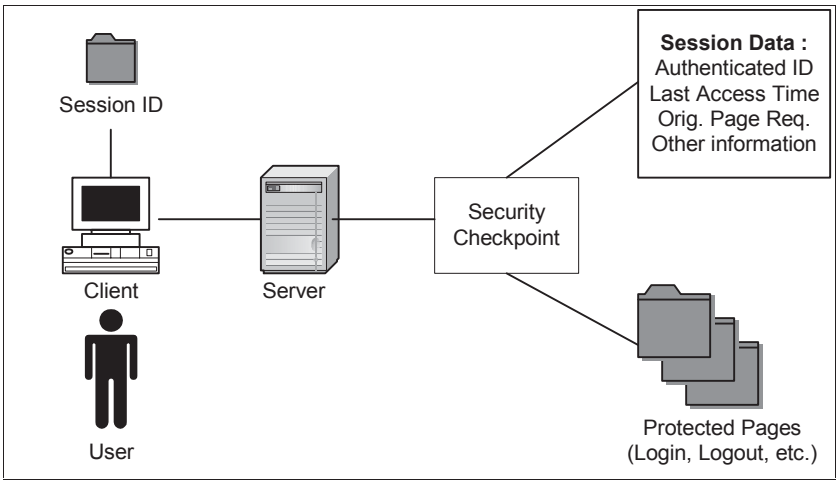
When the server receives and verifies a user's login credentials (typically a username and password), it redirects the user back to the originally requested page. On all subsequent page

requests, the server checks the authenticated identity without requiring that the user re-authenticate him or herself. If the authenticated user is allowed access to the page, the page is served. Dynamically created pages can use the authenticated identity in order to incorporate data that is sensitive to that user.

There are two ways in which an authenticated session can end. The user can explicitly invoke a logout page, which causes the cached credentials to be flushed. Alternately, if the user session remains inactive for some predetermined period of time, the session will time out and the first subsequent request will again require authentication. In order to effect this, each page request should check the current time against the recorded time of last page request, and update the time of last page request appropriately.

The *Authenticated Session* pattern caches the user's authenticated identity on the server. Because it is stored on the server, the application can be more confident that the user has not tampered with it. The only way that the authenticated identity session attribute can be set is if the user successfully authenticated to the authentication module. The *Authenticated Session* pattern relies on existing session mechanisms to associate the client with a particular session.

This pattern compartmentalizes the session security policy within a single component, so that changes to the policy (affecting usability, accountability, and performance) do not impact the client or other parts of the application.



Page request without valid authentication

The following interactions occur on a page request without valid authentication:

- The client requests a protected page from the server, passing the session identifier.
- The underlying session mechanism retrieves the session data and invokes the protected page object.
- The protected page invokes the authentication checkpoint.
- The authentication checkpoint determines that the authenticated identity field is empty or that

the last access time exceeds the session timeout window.

- The requested page URL and submitted data is stored in the session object as the *page originally requested*.
- The authentication checkpoint redirects the client to the login screen.

Login

The following interactions occur in the login procedure:

- The client requests the login page, submitting a username and password. The session identifier is passed as part of the request.
- The underlying session mechanism retrieves the session data and invokes the login page object.
- The login object validates the username and password.
- If unsuccessful, the unsuccessful login page ("try again") is returned to the user.
- If successful, the identity provided is stored in the authenticated identity field, and the current time in the last access time.
- The user is redirected to the page originally requested (stored in the session data).

Page request with valid authentication

The following interactions occur on a page request with valid authentication:

- The client requests a protected page from the server, passing the session identifier.
- The underlying session mechanism retrieves the session data and invokes the protected page object.
- The protected page invokes the authentication checkpoint.
- The authentication checkpoint validates the authenticated identity field in the session data and ensures that the last access time does not exceed the timeout period.
- The authentication checkpoint stores the current time as the last access time.
- The authentication checkpoint returns to the protected page object, which composes and delivers the requested page to the client.

Logout

The following interactions occur in the logout procedure:

- The client requests the logout page from the server, passing the session identifier.
- The underlying session mechanism retrieves the session data and invokes the logout object.
- The logout object clears the authenticated user id field in the session.
- The logout object returns a "logged out" notification page or redirects the browser to the home page.

Issues

The *Authenticated Session* pattern cannot be implemented using HTTP Basic Authentication, because Basic Authentication caches the user's password at the client and delivers it over the network on each page request. Basic Authentication also lacks the flexibility needed to implement session timeout and the *Account Lockout* pattern.

Some Web application environments permit session data to be stored on the client (as opposed to storing only a session identifier on the client). For example, iPlanet Enterprise Server's Server-Side JavaScript offers this feature. In general, client-side storage should not be used with this pattern. If the two absolutely must be combined, the *Client Data Storage* pattern must be used. One simply cannot trust the client to accurately report the authenticated user identity.

Policy Considerations

Sessions and authentication need not always be combined. For example, many e-commerce sites use sessions to track browsing habits and storing of items in a shopping cart. It is only the checkout and purchasing procedures that need to be authenticated.

It might be possible to make sessions optional. For example, e-Bay appears to store all state in the current URL and retain no session data on the server. When the user places a bid he/she is given the option of creating a session in order to avoid having to re-enter his/her password with each successive bid.

More dangerous transactions should not depend on session authentication and should require that the current password be explicitly provided as part of the transaction. (See the *Password Authentication* pattern.) If a session is somehow hijacked or a user walks away from the computer, this ensures that critical transactions cannot be triggered by somebody else. For example, authorization of an electronic funds transfer should always require authentication of the transaction itself. Password change requests also should always require that the old password be provided as part of the change request transaction.

Any page that contains sensitive data should include a header that instructs the browser not to cache the page. Depending on the level of sensitivity and perceived risk, at the end of a session involving sensitive data, the "logged out" page should either directly request the user's permission to exit the browser or warn the user that they must exit the browser soon to ensure that no one can see their sensitive data.

Session Protection

This pattern maintains session data (including the authenticated user identity) on the application server as part of a session object. In order to associate each client with a session object, the application server assigns each session object a unique, random identifier. This session identifier is then given to the client, and the client presents it on each subsequent page request.

The session identifier must be suitably random and hard to guess. Session identifiers are generally provided by the application server and cannot be modified by custom applications. Nevertheless, if the session identifier mechanism contains vulnerabilities, it could result in attackers being able to predict or guess the session identifiers assigned to active users. This would allow an attacker to hijack another user's session and gain access to sensitive data.

Developers and administrators should research the method used to generate session identifiers and check any known product vulnerabilities (at a site such as securityfocus.com). Examining the contents of cookies (or URLs) will give a basic indication of how long each session identifier is. Anything shorter than 20 hexadecimal characters ("hexits") is probably too short.

Administrators must also be aware of any session-guessing attacks underway. When a user submits an invalid session identifier, the application server should notify the application or the system administrator of that event. Large numbers of invalid session identifiers are a clear indicator that something is amiss (either the site is under attack or the server has just restarted). If a single network address is the source of many invalid session identifiers, it should be added to the network address blacklist (see the *Network Address Blacklist* pattern).

Session Encryption

If a site encrypts passwords, it should also encrypt any page that delivers an authenticated session identifier to the server. If the data on the site or the services offered on the site are sufficiently important, it should use encryption to protect passwords in transit (see the *Password Authentication* pattern for more details). If the session identifier can be used to gain access to password-protected pages, those pages should also be encrypted in order to protect the session identifier in transit.

Once the user has been authenticated, the session identifier will be communicated along with every page request. If those pages are not encrypted, the session identifier will be delivered in the clear. An attacker who can monitor network communications could use the session identifier to hijack the session. The attacker would not be able to see the user's sensitive data (such as a password or credit card number), but they would be able to use the session until the user logs out (longer if the user walks away from the keyboard). *Note that this is one reason why a Web site should never echo the most sensitive information back to the user*.

When using encrypted pages to protect session identifiers, it is important to understand how to prevent session identifiers from being inadvertently delivered to an unprotected page:

- When using cookies to store session identifiers, the cookie option that requires the cookie to only be delivered over an SSL-protected connection should be enabled. If the application server does not allow this option to be configured, the application should be divided into two

different servers: one for encrypted pages, one for other pages. The cookie identifier should explicitly name only the encrypted server.

- When using URL's to store session identifiers, the URL rewriting function that adds the session identifier to any links in the page should only be invoked when those links target other encrypted pages.

If all sessions on a server are authenticated, it might be possible to depend on the session mechanism to provide the timeout. Applications constructed using servlets and Java Server Pages even have access to per-application session timeout parameters. However, many session mechanisms are geared at general site usage and have fixed timeouts that are often too long for authenticated sessions.

Usability

The system should be tested with two sessions using the same user identity. Testing the system with two open windows in the same instance of the Web browser should also occur to ensure an appropriate user experience.

When an authenticated session times out, it should be possible to restore state to the page that was last accessed. The login page should not blindly overwrite all session data.

The length of the authenticated session timeout is a trade-off between usability and security. It should not be too short, because users often run multiple programs and work slowly. However, to protect users from unauthorized access if they walk away from their keyboard without logging out, the session timeout should not be more than 10-15 minutes.

The system must provide an explicit logout mechanism that clears the authenticated user identifier from the session. The logout button should be available from every page when the user's session contains an authenticated identity.

Possible Attacks

There are a number of possible attacks that can be perpetrated on authenticated sessions:

- Session continuation – if session data is not properly cleared from the client, it is possible for an attacker to revisit pages cached by the browser or even continue the authenticated session.

- Session hijacking – if the session identifier is observed traveling over the network, it is possible to jump into the session by using the identifier as a part of requests from another browser.

- Direct page access – if every page does not explicitly check authentication, it is possible to guess URLs (or use URLs stored in another user's browser) and gain direct access to the page.

- Manipulation of client data – if a session depends on data provided by the client (either hidden fields, encoded URLs, cookies, or referring pages), it is possible for an attacker to

manipulate that data to circumvent the server authentication model.

It is safe to use the referring page to ensure correct sequence within the application is observed, but one should not depend on the referring page to indicate whether a request has been authenticated.

Examples

Many significant Web banking and e-commerce applications rely on this pattern. Any site that enforces user authentication and does not store that information on the client uses something similar.

Trade-Offs

Accountability	This pattern increases accountability by providing a straightforward, secure approach to repeated authentication.
Availability	No effect.
Confidentiality	See Accountability.
Integrity	See Accountability.
Manageability	No effect.
Usability	The pattern increases usability by not requiring repeated logins. However, if the session timeout is too short or overwrites the session state, it will adversely affect usability.
Performance	There is little performance impact from storing authentication data on the server. However, storing session data on the server can increase server load and make load balancing difficult. If authentication data is stored on the client, significant overhead will be incurred in cryptographically validating the data against tampering.
Cost	Use of a standard authentication model for all protected pages can lower cost by reducing the quality assurance burden.

Related Patterns

- *Network Address Blacklist* – a related pattern that demonstrates a procedure for blocking a network address from further access attempts if a session identifier guessing attack is conducted.

- *Password Authentication* – a related pattern that presents the secure management of passwords, which are almost always used as the authentication mechanism for this pattern.

References

[1] Coggeshall, J. "Session Authentication". http://www.zend.com/zend/spotlight/sessionauth7may.php, May 2001.

[2] Cunningham, C. "Session Management and Authentication with PHPLIB". http://www.phpbuilder.com/columns/chad19990414.php3?page=2, April 1999.

[3] Kärkkäinen, S. "Session Management". *Unix Web Application Architectures*. http://webapparch.sourceforge.net/#23, October 2000.