Securing Session Management

The defensive measures web applications must take to prevent attacks on their session management mechanisms correspond to the two broad categories of vulnerability that affect those mechanisms. First, to perform session management securely, an application must robustly generate its tokens and protect them throughout their lifecycle from creation to disposal.

Generate Strong Tokens

The tokens used to re-identify a user between successive requests should be generated in a manner that does not provide any scope for an attacker who obtains a large sample of tokens from the application in the usual way to predict or extrapolate the tokens issued to other users.

The most effective token generation mechanisms are those that:
(a) use an extensive set of possible values, and
(b) contain a strong source of pseudo-randomness, ensuring an even and unpredictable spread of tokens across the range of possible values.
In principle, any item of arbitrary length and complexity may be guessed using brute force given sufficient time and resources. The objective of designing a mechanism for generating strong tokens is that it should be extremely unlikely that a determined attacker with large amounts of bandwidth and processing resources can successfully guess a single valid token within the lifespan of its validity.

Tokens should consist of only an identifier the server uses to locate the relevant session object for processing the user's request. The token should contain no meaning or structure, either overtly or wrapped in layers of encoding or obfuscation. All data about the session's owner and status should be stored on the server in the session object to which the session token corresponds.

Care should be taken when selecting a source of randomness. Developers should be aware that the various sources available will likely significantly differ in strength. Some, as with java.util.Random , are perfectly useful for many purposes where a source of changing input is required, but can be extrapolated in both forward and reverse directions with perfect certainty based on a single output item. Developers should investigate the mathematical properties of the algorithms used within different available sources of randomness and read relevant documentation about the recommended uses of different APIs. In general, if an algorithm is not explicitly described as cryptographically secure, it should be assumed to be predictable.

In addition to selecting the most robust source of randomness that is feasible, a good practice is to introduce as a source of entropy some information about the individual request for which the token is being generated. This information may not be unique to that request, but it can effectively mitigate any weaknesses in the core pseudo-random number generator being used. Examples of information that may be incorporated include:
- The source IP address and port number from which the request was received.
- The User-Agent header in the request.
- The time of the request in milliseconds.

A highly effective formula for incorporating this entropy is to construct a string that concatenates a pseudo-random number, a variety of request-specific data as listed, and a secret string is known only to the server and generated afresh on each reboot. A suitable hash

is then taken of this string (for example, SHA-256 at the time of this writing) to produce a manageable fixed-length string that can be used as a token. (Placing the most variable items towards the start of the hash's input maximizes the "avalanche" effect within the hashing algorithm.)

Protect Tokens throughout Their Lifecycle

Having created a robust token whose value cannot be predicted, this token needs to be protected throughout its lifecycle from creation to disposal to ensure that it is not disclosed to anyone other than the user to whom it is issued:

■ The token should only ever be transmitted over HTTPS. Any token transmitted in clear text should be regarded as tainted, as not assuring the user's identity. If HTTP cookies are being used to transmit tokens, these should be flagged as secure to prevent the user's browser from ever transmitting them over HTTP. If feasible, HTTPS should be used for every application page, including static content such as help pages, images, etc. If this is not desired and an HTTP service is still implemented, the application should redirect any requests for sensitive content (including the login page) back to the HTTPS service. Static resources such as help pages are not usually sensitive and may be accessed without any authenticated session; hence, secure cookies can be backed up using cookie scope instructions to prevent tokens from being submitted in requests for these resources.

■ Session tokens should never be transmitted in the URL, as this provides a trivial vehicle for session fixation attacks and results in tokens appearing in numerous logging mechanisms. Sometimes, developers use this technique to implement sessions in browsers that have cookies disabled. However, a better means of achieving this is to use POST requests for all navigation and store tokens in a hidden field of an HTML form.

■ Logout functionality should be implemented. This should dispose of all session resources held on the server and invalidate the session token.

■ Session expiration should be implemented after a suitable period of inactivity (e.g., 10 minutes). This should result in the same behavior as if the user had explicitly logged out.

■ Concurrent logins should be prevented. Each time a user logs in, a different session token should be issued, and any existing session belonging to the user should be disposed of as if she had logged out. When this occurs, the old token may be stored for a period. Any subsequent requests received using the token should return a security alert to the user stating that the session has been terminated because she has logged in from a different location.

■ If the application contains any administrative or diagnostic functionality that enables session tokens to be viewed, this functionality should be robustly defended against unauthorized access. In most cases, there is no necessity for this functionality to display the actual session token at all. Instead, it should contain sufficient details about the session owner for any support and diagnostic tasks to be performed without divulging the session token being submitted by the user to identify her session.

■ The domain and path scope of an application's session cookies should be set as restrictively as possible. Cookies with the overly liberal scope are often generated by poorly configured web application platforms or servers rather than by the application developers themselves. There should be no other web applications or untrusted functionality accessible via domain names or URL paths included within the scope of the application's cookies. Particular attention should be paid to any existing subdomains to the domain name used to access the

application. In some cases, to ensure that this vulnerability does not arise, it may be necessary to modify the domain- and path-naming scheme employed by the various applications within the organization.

Specific measures should be taken to defend the session management mechanism against the variety of attacks with which the application's users may find themselves targeted:
■ The application's codebase should be rigorously audited to identify and remove cross-site scripting vulnerabilities. Most such vulnerabilities can be exploited to attack session management mechanisms. In particular, stored (or second-order) XSS attacks can usually be exploited to defeat every potential defense against session misuse and hijacking.

■ Arbitrary tokens submitted by users that the server does not recognize should not be accepted. Instead, the token should be immediately canceled within the browser, and the user should be returned to the application's start page.

■ Cross-site request forgery and other session attacks can be made more difficult by requiring two-step confirmation and/or reauthentication before critical actions such as funds transfers are carried out.

■ Cross-site request forgery attacks can be defended against by not relying solely upon HTTP cookies for transmitting session tokens. Using the cookie mechanism introduces the vulnerability because cookies are automatically submitted by the browser regardless of what caused the request to take place. For example, suppose tokens are always transmitted in a hidden field of an HTML form. In that case, an attacker cannot create a form whose submission will cause an unauthorized action unless he already knows the token's value, in which case he can simply perform a trivial hijacking attack. Per-page tokens can also help prevent these attacks.

■ A new session should always be created after successful authentication to mitigate the effects of session fixation attacks. Where an application does not use authentication but does allow sensitive data to be submitted, the threat posed by fixation attacks is harder to address.

Per-Page Tokens
Finer-grained control over sessions can be achieved, and many kinds of session attacks can be made more difficult or impossible by using per-page tokens in addition to session tokens. Here, a new page token is created every time a user requests an application page and is passed to the client in a cookie or a hidden field of an HTML form. Each time the user requests, the page token is validated against the last value issued, in addition to the normal validation of the primary session token. In the case of a non-match, the entire session is terminated. Many of the most security-critical web applications on the Internet, such as online banks, employ per-page tokens to provide increased protection for their session management mechanism, as shown in Figure -1.

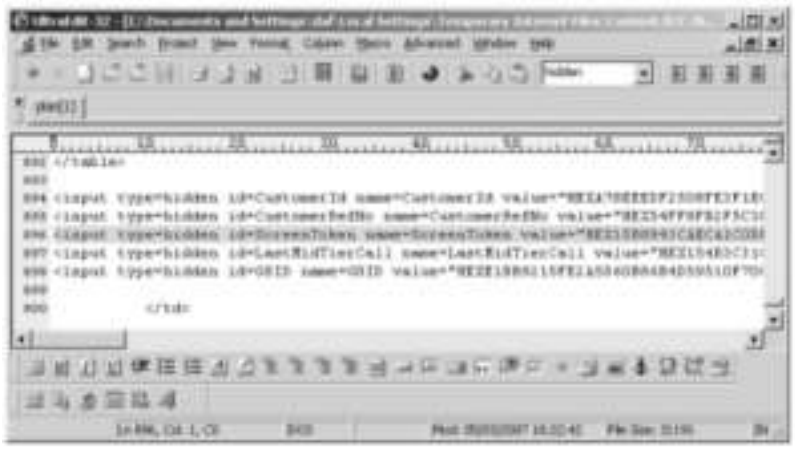

Figure -1: Per-page tokens used in a banking application