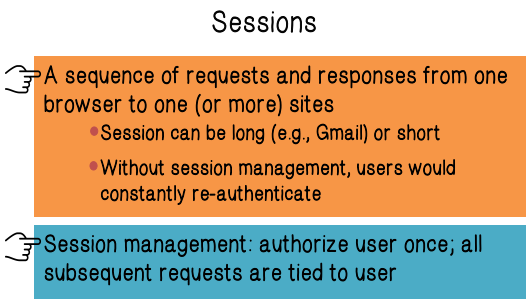
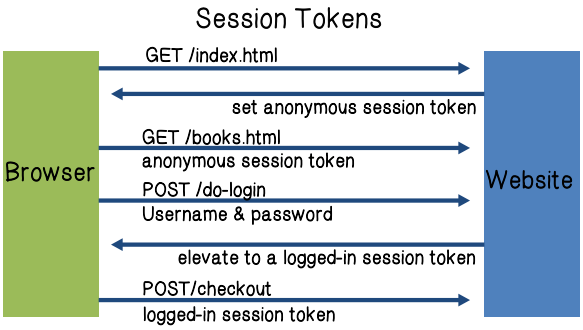
Recommended Readings for this lesson are:

Secure Session Management With Cookies for Web Applications
Origin Cookies: Session Integrity for Web Applications

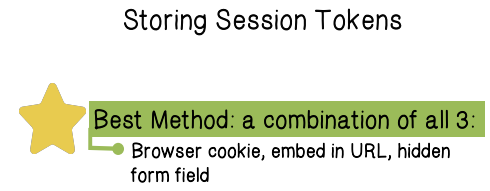


Now, let us discuss session management on the web. What is a session? A session is a sequence of requests and responses from a browser to a server. A session can be long. Without session management, a user can be asked to reauthenticate himself again and again. So, the goal of session management is to authenticate user only once so that all subsequent requests are tied to the authenticated user.

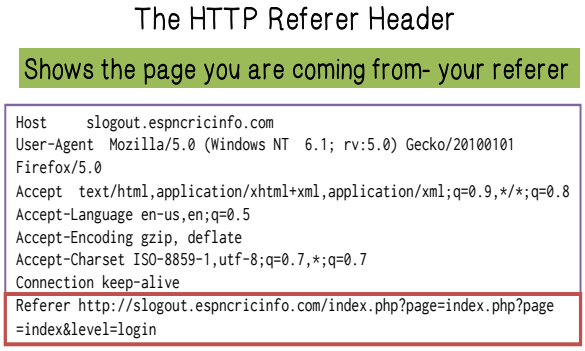


So, the general idea behind session management is to use session tokens. So, for example, there is the initial handshake that is in the browser and the web server. And then, as the user wants to access some more secure content, he may be asked to authenticate himself. And once the user has been authenticated, the server can elevate the token from anonymous browsing token to an authenticated token. And when the user logs out or checks out, this login session token should be cleared.

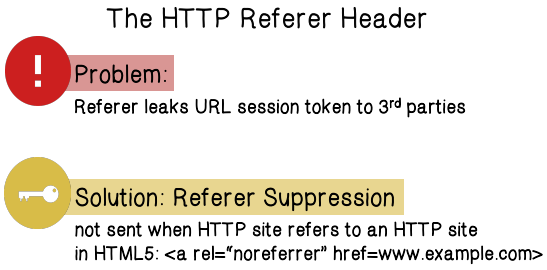
There are many ways to store the session tokens. Obviously, we can use browser cookie. For example, we can create a session token cookie or session cookie. The problem with browser cookie is that a browser can send a cookie with every request, even when it does not, this gives rise to the cross-site request forgery attack. A session token can be embedded in a URL, which means that every request will have the session token. This means that if the application is not returned securely, there can be token leaks via http referrer header, or if the user posts URL in a public forum. Another option is to store that session token in a hidden field in a forum. The downside to this method is that every user action must result in a submission of a form, or you lose the session token.



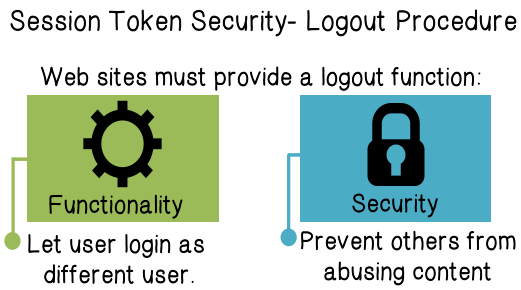
So, none of these methods are perfect. The best solution is, depending on the application, is you choose a combination of these three options.



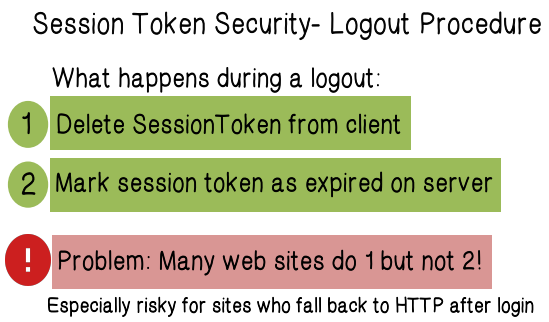
Now, let us discuss the HTTP referrer header. When a browser sends a URL request to a server, if the request contains a HTTP referrer header, it tells the server the page that you are coming from, meaning your referrer. Here is an example. It shows that the user was here. Again, by checking the referrer, the web server can see where the request originated. In the most common situation, this means that when the user clicks a hyperlink in the web browser, the browser sends the request to the server. The request includes the referrer field, which indicates the last page the user was on, that is, the one where the user clicks the link.



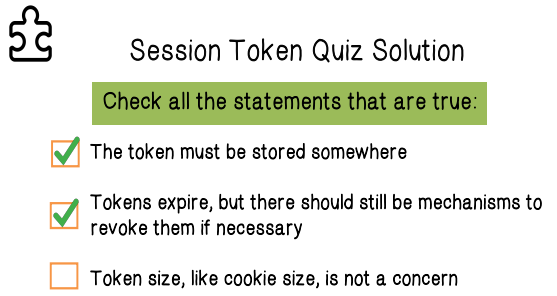
The problem with referrer is that it can leak the session token to the previous server. The solution is that he can suppress the referrer, which means that don't send referrer when you refer to a site.



For example, after the user logs out, he should be allowed to log in with a different account. And a website should prevent a user from accessing content left behind by a previous user.



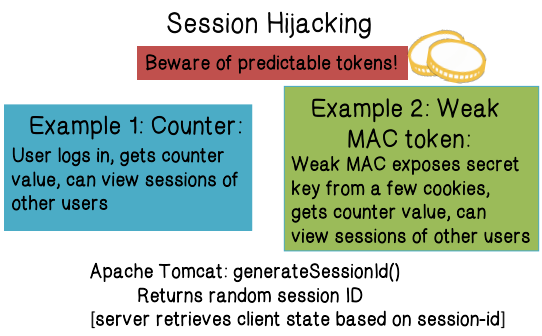
So, what should happen during a log out? First, the session token on a browser should be deleted. Second, on a server side, the session token should be marked as expired. The problem is that many web sites do 1, but not 2, this is especially dangerous for sites that use HTTPS for login, but then fall back to the clear text HTTP after login. This is because an active network attacker can intercept the cleartext HTTP traffic and steal a copy of the session token. Then even after the user logs out, because the server does not expire the session token, the attacker can continue to use that session token.



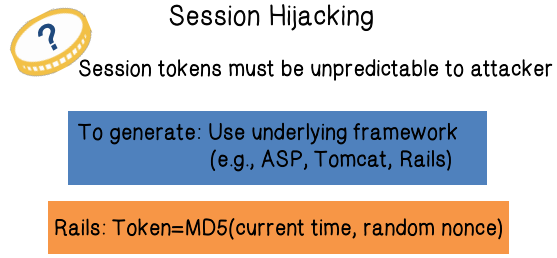
Now let us do a quiz on session token. Check all the statements that are true. First, the token must be stored somewhere. Second, tokens expire, but there should be mechanisms to revoke them if necessary. Third, token size, like cookie size, is not a concern.

The first two statements are obviously true. The third statement is false, because depending on how much information you store in it, tokens can become quite large. Cookies, on the other hand, are quite small.

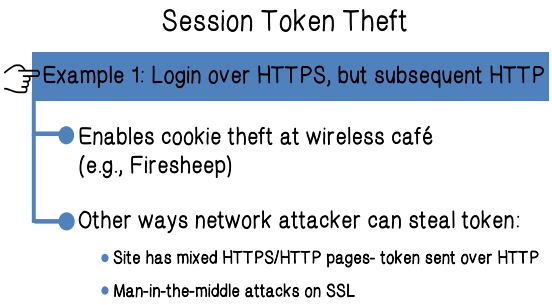
A major threat in web session management is session hijacking. Here, the attacker waits for user to log in, and then the attacker can steal the user session token and hijacks the session.



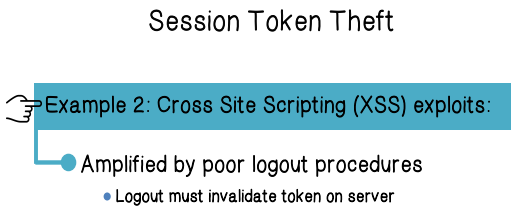
And session hijacking is not limited to active network attacker that intercept traffic. For example, if counter is used as a session token, then when a user logs into a website it can get a counter value, then he can view sessions of other users because he would know other counter values. Similarly, even if the token is protected using cryptography, if the cryptographic algorithm or the key is weak then a user can still break the protection, get the counter value, and then view sessions of other users. So, the point here is that we should use tokens that are not predictable, and there are APIs that allow us to generate random session IDs.



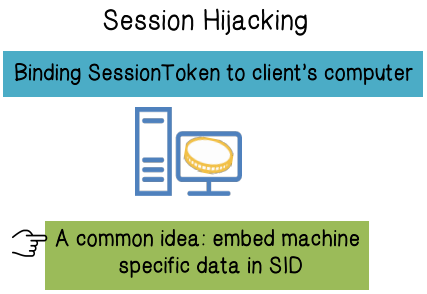
Again, to make session tokens unpredictable to attacker, we can use the underlying framework. For example, rails, for example, by combining the current time stamp and random nouns and compute this values over MD5, that should give you a very unpredictable token.



Even when a session token is random, there is still a security threat of session token theft. For example, if a web site uses HTTPS for log in, but subsequently use HTTP for the rest of the session, then an active network attacker, for example, can sit at a wireless cafe and use a tool, for example, Firesheep to intercept the cleartext HTTP traffic and steal the session token. Another way for the attacker to steal the session token is to play man-in-the-middle at the beginning of the SSL connection.



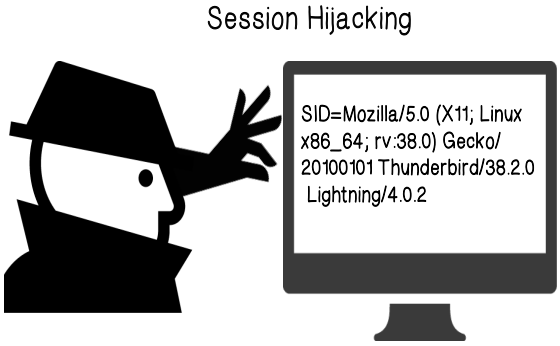
Another approach to steal session token is to use cross-site-scripting attacks, and if the server does not invalidate a session token after the user has logged out, then the stolen token can still be used by the attacker even after the user has logged out.



One idea to mitigate session hijacking is to bind a session token to the user's computer. For example, we can embed some machine specific data in the session ID.

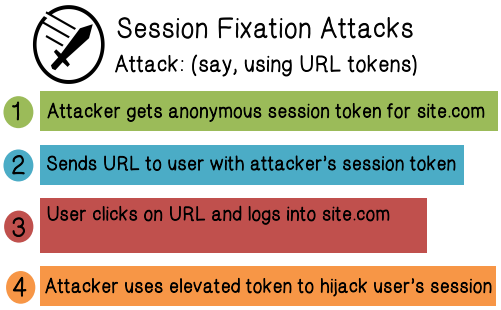
So, what machine specific data of a user can be used? We begin by binding the session token to the user's computer. Now we must decide specifically what information we should use as the session token. We want it to be unguessable and unique to the machine, but still quick to generate. So, is using the IP address a good idea?

Probably not and the reason is that the user's computer changes its IP address. For example, due to DHCP (dynamic host configuration protocol), then the user will be locked out of his own session.

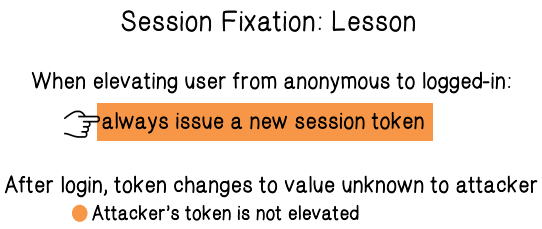


What if we used the browser user information instead of the IP address as a session token? The problem with this approach is that such information is easily stolen or guessable by the attacker. So, the conclusion is that, while it is appealing to use some kind of site information in a session token, there is not a good solution when we consider both security and convenience. Therefore, the best approach is still an unpredictable session token generated by the sever.

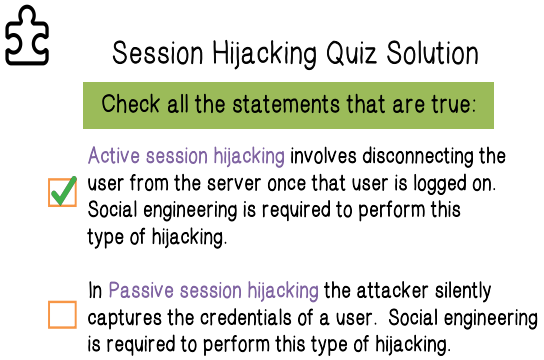
In addition to stealing tokens, an attacker can also fake session tokens. For example, the attacker can trick the user into clicking a URL that sets a session token, or it can use cross-site-scripting attacks to set token values.



Here is an example of how an attacker can use session fixation attack to elevate his anonymous token to a user log-in token. First, the attacker gets anonymous browsing session token from site.com. He then sends a URL to the user with the attacker's session token. The user clicks on the URL and logs in www.site.com. Now the attacker can use the elevated token to hijack user's session.

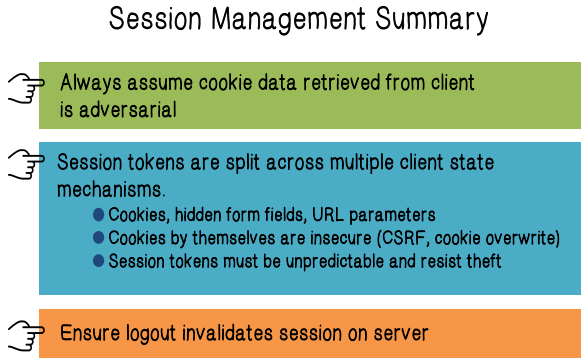


To mitigate such attacks when elevating a user from anonymous to logged in, a website should always issue a new session token. So, with this, after the user logs in, the token will change to a different value unknown to the attacker. That is, the anonymous token that the attacker had originally obtained is not elevated.



Now let us do a quiz on session hijacking. Check all the statements that are true. First, active session hijacking involves disconnecting the user from the server once that user is logged in. Social engineering is required to perform this type of hijacking. Second, in passive session hijacking, the attacker silently captures the credentials of a user. Social engineering is required to perform this type of hijacking.

The first statement is true. The second is false. Of these two methods, passive hijacking is less likely to raise suspicions.



To summarize what we have learned about the security of session management, we should always assume cookie data retrieved from a client is adversarial, or not trusted. There are multiple ways to store session tokens. Cookies, by themselves, are not secure. For example, they can be overwritten. Session tokens should be unpredictable. And finally, when a user logs out, the server should invalidate the session token.