SECURING APACHE : ATTACKS ON SESSION MANAGEMENT

In this part of the series, we are going to concentrate on attacks on session management. Application-level attacks on the session is about obtaining or manipulating the session ID without any prior information to the client and the Web server. The sole aim of the attacker here is to somehow gain access to a valid session between the victim and the Web server.

Before explaining the principles behind attacks on sessions, let's have a look at exactly what a session is, and why we need it.

HTTP is a stateless protocol. Every time the client asks for a page, or for a graphic within a page, a new connection is set up between the client's browser and the Web server. There is no relationship at all between one connection and another, because there is no state. The second connection does not know anything about what took place during the first connection.

Though this statelessness of HTTP is perfectly acceptable in some cases (for example, when using the Web to search for information), it is not appropriate for a Web application where context needs to be maintained from page to page. For example, on an e-commerce site, on the page where users enter their credit card number to complete the purchase transaction, the code needs to know what items they have chosen on the previous pages, in order to compute the total amount. The e-commerce site needs a mechanism that identifies a "session", a virtual "established connection" between a browser and a Web server, to pass a context from page to page.

Technically, this is done as shown in Figure 1. When the browser connects to the Web server for the first time, the user has not been authenticated yet. The Web server asks for credentials, and generates a unique identifier (the session ID). The server can associate a context with each session ID, storing any kind of information in that context. The generated session ID is sent back to the browser. For every subsequent call to the server, the browser sends the session ID to the server. The server can then use the context associated with the transmitted session ID, and “remember” data from page to page.

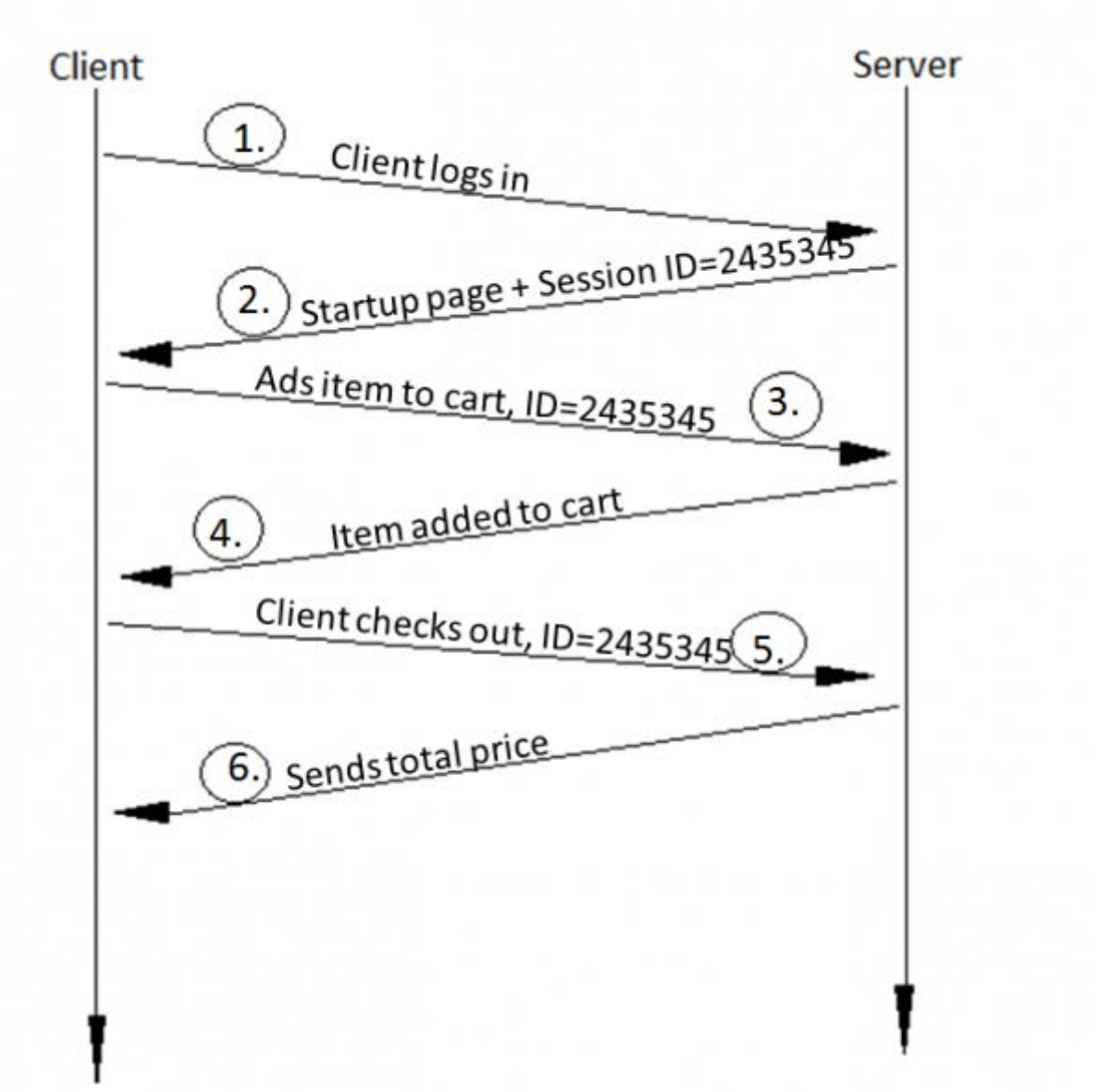

Figure 1: Normal client-server communication using session ID

It is important to understand here that to identify a session, the session is given a session ID. This ID is sent between client and server for those HTTP requests that belong to that session. The Web application sessions are usually implemented in two ways:

1. **Client-side session management:** In client-side session implementation, the bulk of authorisation and identity information for the user is stored client-side, in a cookie. The client sends the information found in its cookie (including the session ID) to let the server know who is sending these requests.
2. **Server-side session management:** In contrast, the server-side implementation stores the bulk of authorisation and identity information in a back-end database on the server. The session ID is used to index the user's information in that database, so the server can access the appropriate information upon receiving requests from the client.

Session identifier mechanisms

To identify and maintain valid sessions, the following three mechanisms are used:

1. Unique identifier embedded in URL.
2. Unique identifier in hidden form field.
3. Unique identifier in cookies.

We will now discuss each one of them, and their advantages and disadvantages.

Embedded in the URL

Such identifiers are received by the Web application through HTTP GET requests when the client clicks on a link embedded in a page. Say, for example, this could be like:http://www.bank.com/account.php?sessionid=IE60012219. However, Web-based invitation services typically use unique session IDs embedded in URLs.

Advantages

- ♣ This method is immune to CSRF attacks that are leveraged through external websites.

- This method is also independent of browser settings. Passing parameters along with URLs is a feature that is always supported by all browsers. This is one of the reasons why URL parameters are a fallback solution to cookies.

Disadvantages

- The session ID in the URL is visible. It appears in the HTTP referrer header sent to other websites when the client follows an external link from within the application. Moreover, it also appears in the log files of proxy and Web servers, as well as in the browser history and bookmarks.
- There is also a risk of clients copying the URL (including the identifier), and mailing it to others, while they are still logged in.
- All the client's links within the Web application must include the session ID and either the application must take care of it or it must be handled by the application's framework.

In hidden form fields

Typically, session ID information is embedded within the form as a hidden field, and submitted with the HTTP POST command. For example:

```
<FORM METHOD=POST ACTION="/account.php">
<INPUT TYPE="hidden" NAME="sessionid" VALUE="IE60012219">
<INPUT TYPE="hidden" NAME="allowed" VALUE="true">
<INPUT TYPE="submit" NAME="Read News Article">
```

Advantages

- In contrast to GET request parameters, hidden form fields are transmitted in the request body, and do not appear in proxy logs or referrer headers. Moreover, users cannot accidentally copy them into mails.

- This method is also immune to CSRF attacks via external websites, and is also independent of browser settings.

Disadvantages

- As the session identifier appears in the HTML page, the mechanism is vulnerable to session identifier theft via XSS.
- The most important functional disadvantage is that all embedded objects like images, FRAMEs, and IFRAMEs cannot be included with POST requests. Resources that are referenced in HTML documents within (for example) the tags IMG, IFRAME, etc., are always retrieved by the browser via HTTP GET requests. The only alternatives are to include these objects without any session information, which also means that these objects are accessible without authentication — or to resort to mechanisms like session identifier in URL parameters, for those requests.
- So using hidden form fields for transmitting the session identifier is limited to those application fields where no session state information is required for performing requests for embedded objects. This includes, in most cases, that no authentication checks are possible for such resources.

Within cookies

For most situations, the best way for sending session identifiers along with each page request is using cookies. A typical cookie used to store a session ID looks much like:

www.bank.com FALSE / FALSE 1293840000 sessionID IE60012219

The fields above illustrate the six parameters that are stored in a cookie. Here is what each field represents:

- domain: The website domain that created the cookie and can read the variable (here,www.bank.com).

- flag: A TRUE/FALSE value indicating whether all hosts within a given domain can access the variable. (here, FALSE)
- path: Pathname of the URL(s) capable of accessing the cookie from the domain.
- secure: A TRUE/FALSE value indicating if an SSL connection with the domain is needed to access the variable.
- expiration: The UNIX time that the variable will expire on.
- name: The name of the variable (here, sessionID).
- value: The value of the variable (here, IE60012219).

The Web application sets the cookie by sending a special HTTP header like this:

Set-Cookie: sessionID="IE60012219"; path="/"; domain="www.bank.com"; expires="2011-06-
00:00:00GMT"; version=0

Advantages

The browser automatically sends cookie values with each request to the Web application. The application does not need to include the identifier in all links or forms, as is necessary for URL or form-based mechanisms.

Disadvantages

- The fact that the cookie with the session identifier is sent automatically with each request to the application makes this mechanism vulnerable to CSRF attacks from external sites.
- The cookie is vulnerable to session identifier theft via XSS, as they can be accessed via JavaScript.
- Another important disadvantage of session identifiers in cookies is that users might have cookies turned off in their browsers. This can pose hindrance in authentication.

There are two types of session-management attacks: Session hijacking and Session fixation. We will examine them one by one.

Session hijacking (a.k.a. sidejacking)

Session hijacking is an attack on a user session over a protected network. It involves employing various techniques to tamper with, or take over, TCP and Web application user sessions. If the session hijacker successfully impersonates the user/client, he gains access to the sensitive information found in the session.

Session hijacking occurs on two levels: the network level and application level. Network-level session hijacking involves the interception of and tampering with of packets transmitted between client and server during a TCP or UDP session. Application-level session hijacking involves obtaining session IDs to gain control of the HTTP user session as defined by the Web application.

Now, because we are dealing with Web application security, we will concentrate here only on HTTP session hijacking. So, for now, please assume that the word “session” means HTTP/application-level sessions.

Past articles in this series tells us that the session credentials can be attacked and compromised through XSS (Part 2), CSRF (Part 3), XST and XSHM (Part 4), and also by HTTP message attacks (Part 5). We have also looked at many attack scenarios dealing with stealing session information and impersonating the client via the above attacks. However, there are still other, and more effective, ways to hijack a client’s session. These are as follows:

Credentials/session prediction

This is an effective and easy way to guess someone's session IDs. Depending upon the randomness and the length of the session ID, this process can take as little time as a few seconds. Let's have a look at the attack scenario to get a more grasp on this…

Consider a Web application using incrementing session IDs, or using proprietary algorithms, which have essentially zero entropy — that is, they are completely predictable. The application gives out session IDs starting at some number, and simply increments that number for each new session. However, attackers don't yet know this about the behaviour of the application, but they can still discover it and abuse it. They will go through the following steps:

1. The attacker registers for an account on the target Web application and logs in.
2. He/she then notes his/her session ID, stored anywhere in the above three locations. Say it turns out to be IE60012219.
3. The attacker then logs out and logs in again, checking the new session ID: say, IE60012350.
4. From this, the attacker can guess another issued ID to be, say, IE60012329. He/she can then change their session ID to IE60012329 in the cookie, and will then have good chances of being authenticated as some other client.

Cracking Apache IDs

Some Web applications use the built-in cookie session ID generation algorithms that ship with the Apache Web server. While these cookies are meant to be used for Web log tracking only, Web applications also use it for authentication purposes. It has been found that the ID generated via the algorithm in mod_usertrack.c can be guessed using automated scripts.

Sniffing session IDs

Sniffing session IDs is one of the most loved methods for attackers. If the HTTP traffic is sent unencrypted, the attacker can examine the interpreted data and access the session IDs. Unencrypted sessions can also contain usernames and passwords. To make such an attack happen, attackers use sniffers to capture the HTTP traffic.

One such sniffer is the Effetech HTTP sniffer. Such a sniffer can capture IP packets containing HTTP protocol, rebuild HTTP sessions, and can also tamper with files sent through the HTTP protocol. Figure 2 depicts, in simple steps, how sessions are sniffed.

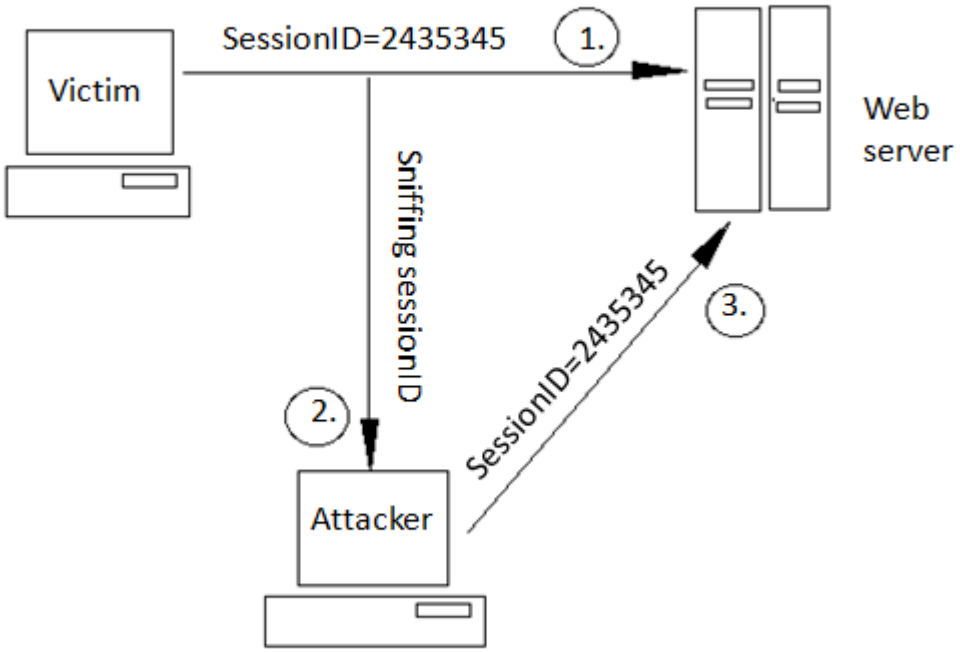

Figure 2: Sniffing session IDs

Session fixation

The other type of session attack is session fixation. Here, instead of stealing/hijacking the victim's session, the attacker fixes the user's session ID before the user even logs into the target server (that is, before authentication), thereby eliminating the need to obtain the

user's session ID afterwards. Before going into detail of session fixation attacks, we must classify two types of sessions managed on Web servers:

1. **Permissive sessions** allow the client's browser to propose any session ID, and create a new session with that ID if one does not exist. After that, the server continues to authenticate the client with the given ID.
2. **Strict sessions** allow only server-side-generated session ID values.

A successful session fixation attack is generally carried out in three phases:

1. Phase I or session set-up: In this phase, the attackers set up a legitimate session with the Web application, and obtain their session ID. However, in some cases the established trap session needs to be maintained (kept alive) by repeatedly sending requests referencing it, to avoid idle session time-out.
2. Phase II or fixation phase: Here, attackers need to introduce their session ID to the victim's browser, thereby fixing the session.
3. Phase III or entrance phase: Finally, the attacker waits until the victim logs into the Web server, using the previous session ID.

To make the above more clear, let's understand it with the help of the following attack scenario…

Consider an online bank, www.bank.com. Session IDs are transported from the client's browser to the server within a URL argument named sessionID. Now, to perform a session fixation attack:

1. The attacker logs into the bank's website as a legitimate user.
2. The attacker is then issued a session ID — say, 2435345 — by www.bank.com.
3. The attacker then sends a link say, www.bank.com/login.php?sessionID=2435345 to the victim, and somehow tricks the victim into clicking on it.

4. As soon as the victim clicks on the link, it sends a request to the bank's server forlogin.php?sessionID=2435345.
5. The Web application here notes that session ID 2435345 already exists in an active state, and hence, a new one doesn't need to be created. Finally, the victim provides his credentials to the login script, and the bank grants access to the victim.
6. But, at this point, the attackers also knows the session ID to be 2435345 (after all it's their session), so they too can access the victim's account usinghttp://www.bank.com/account.php?sessionID=2435345.

We can summarise the above steps as that the session has already been fixed before the user logged in, or we can say that the victim is logged into the attacker's session. The attack scenario above is the simplest example of session fixation attacks, and it requires the attackers to be legitimate users of the website. Also, they have to trick the victim into clicking on the malicious link. Figure 3 can help in visualising this attack.

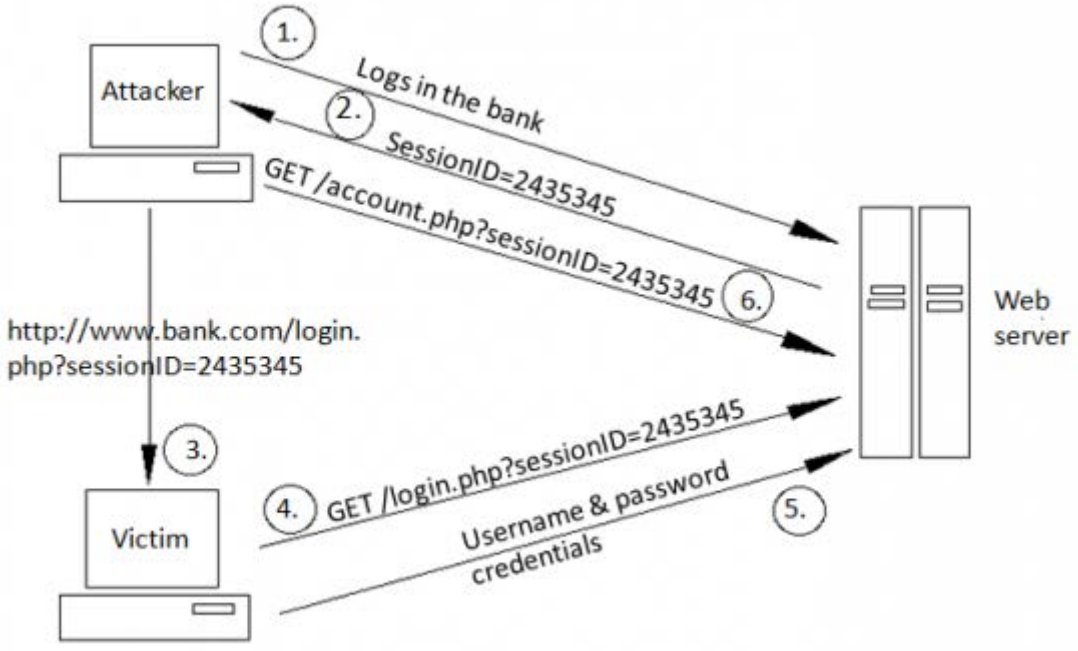

Figure 3: Session fixation attack

Methods of fixing a victim's session ID include the techniques listed below.

Fixing in URL argument

This method is used for those applications that send session identifiers using the HTTP GET method. The above attack scenario is an example of this method. Now, this method comes with quite a high risk of detection, hence attackers use techniques like URL shortening to decrease the chances of detection.

Fixing session ID in cookie

Cookies provide the most convenient and effective way to do session-fixation attacks. However, there are also two different ways to fix session IDs using cookies — using client-side scripts, and using the META tag.

In a client-side script attack, the attacker exploiting the XSS vulnerability on the bank.com server will send the following sample malicious URL to the victim:
http://www.bank.com/<script>document.cookie="sessionID=2435345;
%20path="/";%20domain=.bank.com";</script>
Note that the domain=.bank.com attribute in the above URL will instruct the victim's browser to send the cookie back not only to the issuing server, but also to any other server in the specified domain. However, in case of permissive sessions, a long-term session-fixation attack can be achieved by issuing a persistent cookie (for example, expiring in 10 years), which will keep the session fixed even after the user restarts the computer. This can be done by crafting the following malicious URL and sending it to the victim:
http://www.bank.com/<script>document.cookie="sessionID=2435345;%20path="/";%20domain
01-2021%2000:00:00%20GMT";</script>

The META tag-based attack is used when cookies are issued to victims by including an appropriate <META> tag in the returned HTML document, such as:

<meta http-equiv=Set-Cookie content="sessionID=2435345">

Here, the attacker might post the following crafted URL to the victim:

http://www.bank.com/<meta%20http-equiv=Set Cookie%20content="sessionid=2435345;%20domain=.bank.com">

Note that although <META> tags are usually found between <HEAD> and </HEAD>, they are still processed by the browser if found anywhere within an HTML document.

Other vulnerabilities in session management

This section focuses on a lot of other dangerous mistakes in managing sessions.

Insufficient session expiration

Insufficient session expiration occurs when a Web application permits an attacker to reuse old session credentials, or session IDs, for authorisation. It increases the application's exposure to attacks that re-use the user's session identifiers.

Session expiration comprises two timeout types: inactivity and absolute. An absolute timeout is defined by the total amount of time a session can be valid without re-authentication, and an inactivity timeout is the amount of idle time allowed before the session is invalidated.

The lack of proper session expiration may increase the likelihood of success of certain attacks. A long expiration time increases an attacker's chance of successfully guessing a valid session ID. The longer the expiration time, the more concurrent open sessions will exist at any given time. The larger the pool of sessions, the more likely it will be for an attacker to guess one at random. Although a short session inactivity timeout does not help

if a token is immediately used, the short timeout helps to ensure that the token is harder to capture while it is still valid.

A good example of this can be seen at public cyber-cafes, where the victim forgets to log out from the bank's site, and goes away. The next user on that computer looks in the browser history to return to the last URL. In this case, it is the victim's bank account information. Now, since the victim's session is still active, the second user can perform transactions masquerading as the victim. However, if the banking application had enforced an inactivity timeout set for 2 minutes, for example, the victim's failure to sign out would not give the second user a chance to use the victim's session to make fraudulent transactions. Short session expiration would drastically reduce the risk of such an occurrence.

Weak session cryptographic algorithms

Weak session cryptographic algorithms account for one of the major weaknesses in managing sessions, leading to brute-force or prediction attacks. It's a common practice to use the MD5 hash algorithm to encrypt session IDs, but it has been found that session IDs based on MD5 can be easily decrypted with free tools such as Cain & Abel, THC Hydra, John the Ripper, etc.
Apart from this, many websites also use algorithms based on easily predictable variables, such as time or IP address. In such cases, it becomes relatively easy for attackers to reduce the search space necessary to produce a valid token.

Insufficient session ID length

Even the strongest cryptographic algorithm still allows an active session ID to be easily determined by the attacker if the session ID is not long enough.

Transmission in clear

Assuming that the website is not using SSL (the `secure` field in the cookie is set to `false`), SSL is not used to protect the information. In such a case, if attackers sniff the HTTP traffic, they can easily interpret the session IDs.

Proxies and cache—revealing it all

In most cases, clients will access Web applications through corporate, ISP, or other proxies, or protocol-aware gateways (like firewalls). In such cases, whenever a session ID is passed, it is cached/stored at the intermediaries, and even in local caches. These intermediaries can be a good target for attackers, to search for active session credentials.

Insecure server-side session ID storage

Some frameworks use shared areas of the Web server's disk to store session data. In particular, PHP uses `/tmp` on UNIX, and `c:\windows\temp` on Windows, by default. These areas provide no protection for session data, and may lead to compromise of the application if the Web server is shared or compromised.

Time for security

1. The main countermeasure that will make your packets harder to interpret is to implement SSL. This will make the session hijacker's job considerably harder, because it adds the extra steps of figuring out how to decrypt the communication, and how to encrypt the attacker's own malicious packets, so that they make sense to the host.
2. Increase the length and character range used in the session ID to such an extent that the attacker should not be able guess valid ones before the session expires. It is usually recommended to make the session ID as large as 50 characters. Also, don't forget to increase its randomness.
3. Provide re-authentication when accessing critical parts of the Web application. The result of this will be that even if attackers gain control over a user's session, they will

not be able perform critical actions, such as transferring money or changing passwords and security questions.

4. Verify the domain before accepting cookie-based session IDs.

Security from session fixation

1. As far the nature of session fixation, that is, the victim logging into a session with the attacker's chosen session ID, there should be forceful prevention of logging into an already chosen session. To do this, Web applications must reject any session ID provided by the victim at login, and *must* generate a new session instead, if a user is successfully authenticated.
2. Using the browser's network address and a time-stamp in session IDs, and raising a red flag if the network address changes, is also a good security step.
3. Ensure that cookies transmitted over an encrypted connection have the secure attribute set.
4. Don't forget to eliminate XSS vulnerabilities in your Web apps — it will help curb both session hijacking and fixation a lot. For more on XSS, refer to part 2 of this article series.
5. Moreover, if possible, make the application or system log attempts to connect with invalid or expired session tokens, along with the IP address of the client.

Tools of the secure trade

1. Achilles acts as an HTTP/HTTPS proxy that allows a user to intercept, log, and modify Web traffic on the fly.
2. Paros are man-in-the-middle (MITM) proxy tools. These tools will intercept an HTTP session's data in either direction, and give the user the ability to alter the data before transmission. You can use these to determine how susceptible you are to session attacks by setting up MITM situations.

3. Burp suite is an integrated platform for performing security testing of Web applications. Beside being an MITM, proxy it is also an efficient Web app scanner.
4. Firesheep is a Firefox extension that demonstrates HTTP session-hijacking attacks. It works in a Windows/Mac OS X background, by simply installing it in Firefox. It simplifies the capturing of login sessions. Once a login session is captured, the attacker can use the captured information to access accounts belonging to other users. It does this by intercepting unencrypted cookies from those websites that only encrypt the login process, and not the cookie created during the login process. For this to happen, the attacker has to be in a shared, unsecured network (such as open Wi-Fi), and can then access the accounts of everyone using this shared network. It has been found that Facebook, Twitter, Yahoo Mail, etc., accounts can be accessed easily using Firesheep, without any notification to the victims, because these use HTTPS/HTTP in their login forms, and the rest after that goes as plain HTTP. This extension is quite dangerous, because it requires no tech background to use it; even a newbie can use it. To save yourself from falling victim to Firesheep, use the HTTPS-Everywhere extension in Firefox, which converts every URL you visit (if the domain is in its checklist), from HTTP to HTTPS. Another such extension is Force-TLS. Another Firefox extension, known as BlackSheep, has been created as a counter for Firesheep. This extension prompts users if someone is using Firesheep in their network, along with its IP. It works by sending fake session IDs for Firesheep to detect. A program, FireShepherd, can also be used, because it aims to destroy any running instances of Firesheep on the same network, by sending out overwhelming data packets.

All the examples and attack scenarios explained above are just for educational purposes. I once again stress that neither I nor LFY aim to teach readers how to conduct malicious attacks. Rather, these attack techniques are meant to give you knowledge that you need to protect your own infrastructure

Source : http://www.opensourceforu.com/2011/02/securing-apache-part-6-attacks-session-management/