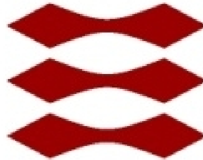
Secure Session Management

Fariha Nazmul



Kongens Lyngby 2011
IMM-M.Sc.-2011-45

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc.-2011-45

Preface

The stateless behavior of HTTP requires web application developers to use separate stateful or stateless mechanisms with HTTP for maintaining state and user specific session information. The task of maintaining user based state information in a logical connection between the server and the user device is known as session. Web session management is a method that allows the web server to exchange state information to recognize and track every user connection.

A critical issue in web security is the ability to bind user authentication and access control to unique sessions. Vulnerabilities in the session management process can cause serious damage since the sessions generally maintain important and sensitive data of the web based systems.

The aim of this Master thesis is to concentrate on the security of session management in a single server environment. The thesis focuses on analyzing the important aspects of a secure session management mechanism that are the ability to bind an incoming request to the session it belongs to, to determine where and how the session state can be stored and to find out measures to protect the session handling mechanisms from security attacks. In addition, this thesis shows the basic steps of implementing a session with PHP and discusses the implications of manipulating some of the session management configuration options on the security level of the application. Furthermore, the focus of this thesis is to study the best practices available for secure session management and to put forward a standard way of maintaining a secure session in single server system.

Acknowledgements

I would like to express my gratitude to my supervisors, Professor Tuomas Aura of School of Science at Aalto University and Professor Christian W. Probst at the Technical University of Denmark, for their continuous supervision, valuable suggestions and collaboration during the thesis process.

I also owe my gratitude to my instructor, Sanna Suoranta, for her constant help and guidance and for the time dedicated to read through my drafts every week and providing feedback on it.

I would also like to thank all the professors, lecturers and friends whom I have met in Aalto and DTU during my two years of study.

Finally, my acknowledgement to Almighty for His blessings and gratitude and love to my parents and my beloved husband, Wali for their support and inspiration in every aspect of my life.

Espoo, June 2011

Fariha Nazmul

Abbreviations and Acronyms

HTML	HyperText Markup Language
PHP	PHP:Hypertext Preprocessor
HTTP	Hyper-Text Transfer Protocol
HTTPS	Hyper-Text Transfer Protocol Secure
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TCP	Transmission Control Protocol
ID	Identifier
SID	Session Identifier
MD5	Message Digest 5
RFC	Request For Comments
URL	Uniform Resource Locator
XSS	Cross-Site Scripting
IETF	Internet Engineering Task Force
PKI	Public Key Infrastructure
SHA	Secure Hash Algorithm
SSO	Single Sign-On
MAC	Message Authentication Code
HMAC	Hash-based Message Authentication Code
PNG	Portable Network Graphics

Contents

Preface i

Acknowledgements iii

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Organization of the Thesis . . . 3

2 Session Management 5
- 2.1 Session Tracking Solutions . . . 7
- 2.2 Session Functionalities . . . 13
- 2.3 Secure Sessions . . . 17

3 Handling Session Data 21
- 3.1 Session Identifier (SID) . . . 22
- 3.2 Server-side Sessions . . . 23
- 3.3 Client-side Sessions . . . 25

4 Session Vulnerabilities 29
- 4.1 Session Hijacking . . . 30
- 4.2 Session Fixation . . . 36
- 4.3 Other Attack Possibilities . . . 41

5 Session Management in PHP 43
- 5.1 Handling Session . . . 44
- 5.2 Creating Session . . . 45
- 5.3 Session ID . . . 49
- 5.4 Session Cookie . . . 49
- 5.5 Storing Session Data . . . 51

5.6 Destroying Session 52
5.7 Controlling Session Lifetime 53
5.8 Session Storage Security 53

6 Best Practices 55
6.1 Authentication 56
6.2 Token Handling 57
6.3 Session Data 59
6.4 Session Lifetime 60

7 Conclusion 63
7.1 Future Work 65

Chapter 1

Introduction

All over the world Internet users are using web based systems that follows the client server paradigm. The web browser acts as the client for a web server that provides the service. These web based systems rely on the HTTP protocol for communication, but it is a stateless protocol. Therefore, the application developers have to use alternative methods to identify and authenticate the user and maintain the user's state. Both stateless and stateful mechanisms can be used with HTTP for session tracking and remembering user specific information. Sessions save the user specific variables and state through consecutive page requests. Sessions are commonly used to enforce security restrictions and to encapsulate run-time state information.A critical issue in web security is the ability to bind user authentication and access control to unique sessions.

Session management allows web based systems to create sessions and maintain user specific session data so that the user does not need to authenticate repeatedly while performing actions. An authentication process is carried out at the first stage to check if the user has the privilege to access the resource. Once authenticated, the user is granted a unique session ID. Thus session management is achieved by maintaining the unique session ID on the client and server side and the browser needs to submit the session ID with every new request.

Unfortunately, existing session management methods are designed originally for a trustworthy environment that the Internet is no longer and thus they cannot

provide flawless security. There are two main categories of attacks that can compromise the security. Session hijacking is a general term used to represent any kind of attack that tries to gain access to an existing session. Some common forms of session hijacking are session prediction, session sniffing, session exposure and cross-site scripting. Another kind of attack is known as session fixation. In session fixation, the user is made to use an explicit session which has been provided by the attacker. Session poisoning is an attack that can be connected to session exposure, when it is done by modifying or deleting the session data. It can also be the creation of new session and thus relating to session fixation.

Session management is critical to the security of web based system. Additional measures are needed in the existing session management mechanism to ensure a reliable and sufficient level of session security. One measure is to use strong encryption on all transmissions and to control the session lifetime in a more efficient way. Special care has to be taken also about generating the session ID to make it unique, unpredictable and hard to regenerate. Handling session data storage and session cookies is another area that needs to be modified to provide better security.

1.1 Problem Statement

Secure session management can be considered as a hurdle between the web based resources and the general users. Even developing a secure web based system is economically reasonable than dealing with the fallout later. But in real life, most applications often have weak session management. It is difficult to implement a secure session management mechanism because there is no common guideline for it and there exists many little-known flaws. Moreover, there is no single solution that suits best all and there is no perfect solution. Furthermore, all the session-management solutions have significant drawbacks in one way or another. Even then some common techniques can be applied to provide a secure and reliable session in a general way that can also defend against session-based attacks. Existing attacks can be mitigated with good web application design practices.

This thesis concentrates on the security of session management in a single server environment. The thesis emphasizes on studying how sessions can be handled in many different ways for web applications, and on analyzing existing open source PHP applications to find out the security measures taken to provide a strong and secure session management. Furthermore, the focus of this thesis is then to study the best practices available for secured session management and to put forward a standard and abstract way of maintaining a secured session in

a single server system.

1.2 Organization of the Thesis

The thesis is organized in such a way that it helps the reader follow the thesis in a graceful manner. This chapter introduces the research scope and goal of the thesis. After the introduction, Chapter 2 gives an overview of session management and explains different methods of session tracking with their advantages and disadvantages. Chapter 3 continues the session management but from the data perspective and takes a closer look at session data handling mechanisms and discusses the different aspects of session state management. Chapter 4 gives an overall picture of the typical attacks against sessions and their countermeasures. The session management implementations in a popular web application framework, PHP is discussed in Chapter 5. Chapter 6 offers a list of best practice recommendations for implementing safe and secure session management mechanisms. The thesis concludes with a brief summary of the current state of secured session management and a suggestion of possible future work in Chapter 7.

CHAPTER 2

Session Management

A session means a logical connection between a user device and a server. Sessions are commonly used in a client-server architecture. Services that need to keep some track of their users maintain sessions with their user's client program. The session is established in the beginning of a connection, usually involving the exchange of a series of requests and responses between the client and the server. Typically, some state information is associated with each session. Mostly sessions are used to identify a user, to manage user preferences, to impose security restrictions or to encapsulate other state information.

Hypertext Transfer Protocol (HTTP) is used for web page and service access and it is not session aware. HTTP uses TCP as its transport protocol. Initially, HTTP 1.0 [4] was designed to use a new TCP connection for each request. Therefore, each request was independent of any previous or upcoming requests even though they belonged to the same user. When there was a series of requests and responses, there was no way of identifying which requests belonged to which client. Usually web pages have embedded images and each image is retrieved through a separate HTTP request. The use of a new TCP connection for each image retrieval created network congestion and increased load on HTTP servers.

To resolve the problem of network congestion, HTTP 1.1 [16] allows to make persistent connections by default. In a persistent (also known as keep-alive) connection, the clients, servers and proxies assume that a TCP connection will

remain open after the transmission of a request and its response. Multiple requests can be transmitted over a single TCP connection until client or the server sends the `Connection:close` message to close the connection. The server or the client browser can also set a timeout value for the persistent connections.[23] Even though the HTTP requests in a persistent network connection are not independent, there is no built-in support for maintaining contextual information and thereby it is not possible to maintain a stateful session with all the HTTP requests and responses transmitted between the server and a specific client. HTTP servers respond to each client request without relating that request to previous or subsequent requests. The web session of a user refers to a logical stateful connection created from the HTTP requests and responses passed between that user and the server.[24]

Web session management is a method that allows the web server to exchange state information to recognize and track every user connection. It is basically user authentication and preference management e.g. storing a history of previously visited contents and status information.[15] Most common examples of session-oriented systems are the web based applications that need to maintain user specific information. The basic pattern of a web based session management is to authenticate a user to the web server with his login credentials once, and to formally set up a session. In this way, the user does not need to send the username and password to the web server with every request.

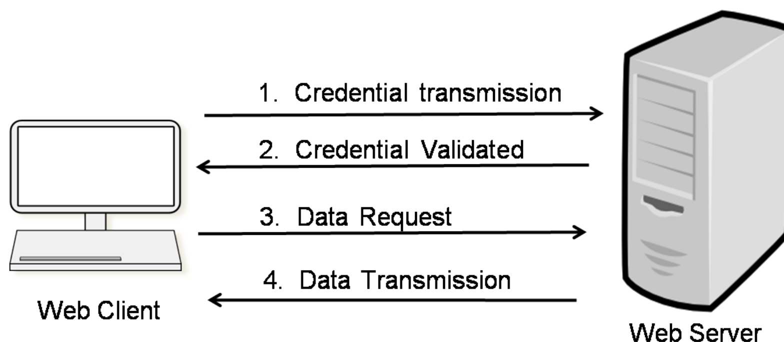


Figure 2.1: Typical Session

Figure 2.1 shows the steps involved in setting up a normal session between a client and a web server. At first, the user passes the credentials to the web server using a form. After processing the credentials, the web server uses some kind of session tracking mechanism to maintain the status of the authenticated user

in the session. The state of the session is then used to control the user access to the web application. When the user quits the application, the authenticated status of the user is destroyed and the session of the user is closed.

In general, a session control mechanism is dependent on a client-side token known as the session token or session identifier and a server side session storage. A session token is normally composed of a string of variable width. After a successful authentication of the user, the web server issues a session token that is associated with a level of access and transmits the token to the client. The level of access and other user specific data associated to a session token are stored on a server side session storage, typically implemented as a hash table. The session token is used to locate the user specific session data on the server side and thereby, identify the user to the web server for the rest of the session. Thus, once a client has successfully authenticated to the web server, the session token is used to track the client throughout the web session and also to uniquely identify the authentication status. The session token or identifier can be transmitted between the server and the client in different ways, like embedded into the URL, in the elements of a web form or in the header of the HTTP message as a cookie.

2.1 Session Tracking Solutions

Session tracking is the process of maintaining information about the users of a web based system across multiple request. There exists a number of different ways to simulate a stateful session on top of a standard, stateless web requests. Some of the most common methods are HTTP basic and digest authentication, URL encoding, web forms with hidden fields ,and cookies. Each of these methods have the same purpose, enabling the web client to pass some user specific information to the web server with each request. In case of HTTP basic and digest authentication, the user client transmits the user credentials, and in other cases, the client uses a session ID. The server is able to track and associate user actions with the help of this additional information.

2.1.1 HTTP Basic and Digest Authentication

First attempt to provide user authentication with HTTP is HTTP basic authentication allows a client application to send its credentials i.e. username and password in plaintext to the web server as a part of an HTTP request. Any web client or server can support basic authentication. When a user requests to ac-

cess a web based system that uses basic authentication, the server responds with a `WWW-Authenticate` header. This header includes the authentication scheme 'Basic' and the domain for which the credentials are required. After receiving this header, the client browser prompts the user for his username and password for the domain. Once entered, these credentials will be automatically included in the `Authorization` header of the successive HTTP requests from the user to the same server. The server will respond by sending the requested content if the credentials are valid. The client can continue to send the same credentials with other requests to the specified domain of the server.[18] These message sequence of the basic authentication mechanism is shown in Figure 2.2.

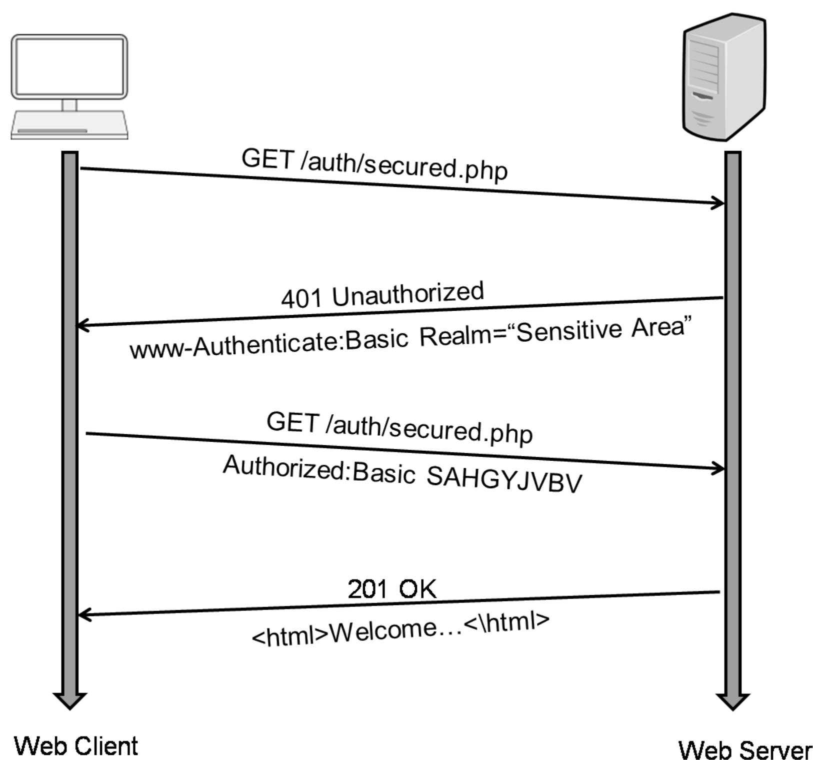


Figure 2.2: Messeage Sequence of Basic Authenticaton

However, the basic authentication is vulnerable to packet sniffing as the username and password is transmitted in base64-encoded plaintext. It is trivial for an attacker to intercept the credentials when transmitted over non-encrypted connections. Moreover, web servers and proxy servers configured to log informa-

tion provided as part of the HTTP headers may store the user credentials in a log file. As a result, the user credentials are at potential risk to be revealed.[18]

In HTTP digest authentication, the password is not transmitted in plaintext. This authentication mechanism uses the same headers that are used in the basic authentication but with a scheme of 'Digest'. Instead of sending plain password, a challenge-response mechanism is followed and a cryptographic hash of the username, password, client and server generated nonces, HTTP method of the request and the requested URI is sent to the server. This mechanism ensures that a network eavesdropper cannot extract the password and also the cryptographic hash is valid for a single resource and method. Therefore, the credentials transmitted with the request for a resource do not provide access to other resources.[23]

Nevertheless, this authentication scheme does not guarantee enough support on all client browsers.[15] To handle authentication on the server side, the HTTP stack must have access to a username-password database. One major problem with both these methods is that they can only provide authentication but they do not provide any mechanism to customize or integrate other session information. Moreover, there is no option for explicit session expiration in these methods.[18]

2.1.2 URL Encoding

URL encoding refers to a session tracking mechanism where the session token is embedded into the URL and transmitted to the web server through HTTP GET requests. The benefit of using this mechanism is that this does not depend on the browser setting or security restrictions of the client.[15] For example, in a scenario where cookies are not allowed, this method can act as a fallback method. If a user wants to propagate the right of accessing the information resource to other users, he can send a copy of the embedded URL to them. Moreover, in the systems where the HTTP Referer field includes the URL, it can be used to check if a client has followed a particular path within the system.

An example of an URL embedded with the session id is shown here.

```
http://www.example.com/start.php?page=1;sessionid=AB20394726
```

However, there are some critical security issues in this mechanism. The session information contained in the URL is clearly visible. The URL along with the session data can be sent in the HTTP Referer field to other web servers. Sometimes, this URL of the referring location is also stored in the log files of

web servers and proxy servers. As a result, the session identifier embedded in the URL will be included in the log files of the servers. Furthermore, users frequently bookmark or send URLs of web pages to others. If the user sends the URL containing the session ID and the ID has not been updated, any one using that link gets full access to the user's account. So this mechanism has the high risk of disclosing the session data to any unauthorized third party.

2.1.3 Web Forms with Hidden Fields

The session data can be included in the hidden fields of a web form and then the client application sends the form to the web server with the help of the HTTP POST command. When each page visited by a user contains the hidden fields with the session information, the web server can track the user and access the user's session information across multiple requests. For example, the HTML code for a web form with three hidden fields namely `customerno, productno, orderno` will look like following:

```
<FORM METHOD=POST ACTION="/cgi-bin/order">
<INPUT TYPE ="hidden" NAME="customerno" VALUE="1234">
<INPUT TYPE ="hidden" NAME="productno" VALUE="2345">
<INPUT TYPE ="hidden" NAME="orderno" VALUE="3456">
</FORM>
```

A server side script can be used to dynamically generate the HTML code containing these fields and also to read and compare the field values with the user specific information residing on the server. One of the main advantages of this method is that it is not dependent on client security restrictions.[15] This method can be used even if the client web browser has disabled the use of cookies. It also enables a user to safely store the URL of the web page without keeping their session information.

However, this form-based method does not provide any predefined security. The hidden fields are not secret and the client can view the value of the hidden fields by looking at the HTML source of the document. This way, a client is able to know the session data and the session identifier.[29] To use this mechanism of session tracking, every page of the web application needs to be dynamically generated by a form submission. The repeated transmissions of HTTP forms using the POST command also create a performance overhead. In addition, this mechanism does not work for HTTP GET requests whereas all the embedded objects such as, images, frames referenced in HTML documents are always retrieved using HTTP GET request. As a result, hidden form fields transmitted through POST method are only suitable for applications that do not need any

session information while performing the requests for embedded objects. [14]

2.1.4 HTTP Cookies

Another easy and flexible way of handling session tracking is the use of HTTP cookies. A cookie is a little piece of information that is transmitted from the server to the browser upon session creation. Each time a web client accesses the contents from a particular domain, the client browser will transmit the relevant cookie information along with the HTTP request. In this way, cookies can also be used to store user specific information that can offer the user a personalized experience over multiple requests, even multiple sessions. Cookies, containing the expiry information to specify when the browser shall delete it, may last beyond a single session. This type of cookies are stored on the client disk and are called 'persistent' cookies. When a cookie is created without any expiry information, it is only stored in the memory of the client browser and is erased when the browser is closed. This kind of cookies are known as 'session' cookies.[24]

A stateful HTTP transaction contains two headers for cookies: the `Set- Cookie` response header and the `Cookie` request header. When a client sends a request, the web server includes a `Set-Cookie` header in the response. This header requests the client browser to include the cookie with all upcoming requests to this web server. If cookies are enabled in the client browser, the browser will add the cookie to all subsequent requests using the header `Cookie` as long as the cookie is valid. This `Cookie` header provides information to the server to identify the web client.

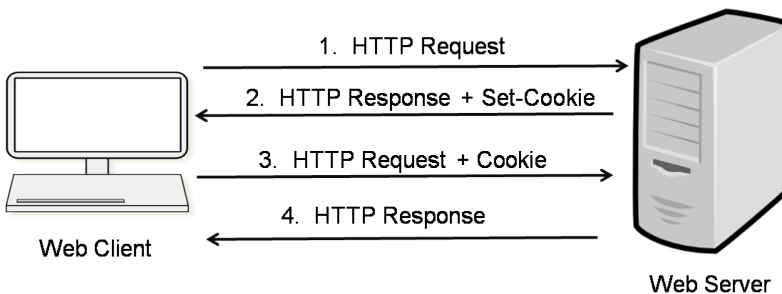


Figure 2.3: Cookie Exchange

A typical cookie-exchange scenario is shown in Figure 2.3. At first, the web client sends an HTTP request to the web server and the server sends an HTTP response including the `Set-Cookie` header. After receiving this response, the client includes the `Cookie` header in the next HTTP request and the server sends back an HTTP response with the requested resource.

A cookie can be set with a name-value pair and some additional attributes.[24] The name of the cookie is used to identify a cookie for a user and the value is the information, typically the session ID that the web server desires to store in the cookie. Additional cookie attributes are:

- **Domain:** The domain attribute specifies the domain of the web server that created the cookie. This is the same domain for which the cookie is valid.
- **Path:** This attribute is used to refer to the URL of the page on the origin web server.
- **Max-Age:** To specify the lifetime of the cookie, the Max-Age attribute is used by the web server. If this attribute has a value, then the user agent will store the cookie. When the cookie time is expired, the user agent will discard it. Otherwise, the cookie will be deleted once the user closes the user agent.
- **Secure:** This attribute is used to instruct the user agent that the cookie can be sent to the origin server only over a secured connection.
- **Port:** This attribute mentions the port through which the cookie can be returned. Basically, the port tells if the cookie is an HTTP cookie or not.

Session cookies are suitable for storing the session identifiers because the SIDs will be removed from the client when the browser is closed. This type of cookies remain safe even when the user does not log out explicitly and the server is unable to unset the cookie. An advantage of using HTTP cookies over other session tracking solutions is that the server does not need to perform any action to add the SIDs to all the links or forms. The web server generates and transmits a cookie containing the SID to the client and the client browser automatically sends the cookie to the server with each request. In addition, when a server regenerates the SID, the new SID is available to the client browser immediately and there exists no problem with the back or reload buttons. Cookies are also safe to use when the web server is configured not to log the HTTP headers, leaving no traces.[14]

Typically, a cookie is stored on the client browser in a file and the information in the cookie is easily readable and accessible by the clients, if not properly encrypted or hashed. Encrypting the cookie value prevents a malicious user from accessing the information. However, it may also create vulnerability by giving the attacker an option to find out the encryption key. If an attacker can figure out the encryption key, he can perform extreme damage to the system.[9] A major disadvantage of cookie mechanism is that a malicious attacker can make the client perform some application actions on behalf of him. A malicious site can include a reference to the target web application in a fake resource and the reference may contain a request to perform some harmful action. An authenticated, already logged-in user of the target web application will automatically send the cookie with the reference of the trap resource while visiting the malicious site and perform the specified action. Furthermore, an attacker can craft a URL impersonating the original web site and make the client send the cookies to the attacker. This type is attack is known as cookie-harvesting.[38]

The cookie is also vulnerable to cross site scripting attack where the attacker can access the cookie via malicious script. However, an additional cookie attribute `HttpOnly` is introduced to prevent this attack. When this attribute is present in a cookie, it is not accessible via Javascript.[14] When the cookie is transmitted over unsecure channel, it can be captured by a malicious user. Cookies with the `Secure` flag turned on will never be transmitted over an insecure connection. Another major problem of using cookies is they can be disabled through browser setting. A web application will not be able to track user sessions for users who have disabled cookies.

2.2 Session Functionalities

Web session management is a combination of many functionalities. First of all, a session is established between a user and a web server after successful authentication. Then the session is maintained by following one of the session tracking solutions. At some point, the session is terminated either by the client or the server. The following sections present the different functionalities that are associated with session management.

2.2.1 Session Initialization

A session has to be negotiated between the client and the web server before they start communicating. The basic concept of a web based session is to recognize

a user and maintain the authenticated state of the user. In order to maintain user specific information, web based applications need to provide mechanisms to identify and authenticate each user to the web server and associate a unique session identifier to each user after successful authentication. The authentication mechanism is the entry point to the application and provides control access. There exists many different methods to authenticate users on the web based systems. One way to provide authentication is to use HTTP basic and digest authentication mechanisms, which were discussed in section 2.1.1. The most common way of authentication in web based systems is form based authentication where the user credentials are transmitted to the server using an HTML form and the server validates the credentials against a database. Single sign-on (SSO) is another method that allow users to log into one server and get automatically authenticated for multiple other servers under the same SSO system.[32]

After the successful authentication of the user, the server issues a session identifier for the user and transmits the SID to the client. The session identifier is used as an index to the location where the user specific session data is stored on the server. The session identifier can be transmitted between the server and the client in many ways, depending on the session tracking mechanism being used by the server. By storing the user session data on the server and associating a SID to the user to identify his status, the server formally sets up a session with the user.

2.2.2 Session Termination

Session termination is an important aspect of secure session management. Sessions that are left open over time consume server resources and present a potential risk of session attacks. Unterminated sessions are vulnerable to session capture attacks and potential impersonation of the session user. Special care needs to be taken to shorten the exposure period of a session. A session can be terminated under many different situations. A good practice is to provide the users with a feature to logout of the system reliably. When a user chooses to logout of the system, he assumes that no one can access this session in future. Basically, by selecting the logout option, the user requests a logout script to be executed by the server that explicitly ends the session. From security point of view, it is important to remove the user session data from the server on performing the logout function, rather than relying on the client browser to remove the cookie or session ID. In addition, any occurrence of security error in the system needs to result in termination of the session.

Most often web applications allow users to create multiple sessions. Typically, a

session is bound to the client device by saving the session identifier on the device. When a user chooses to logout of the system, only the session tied to that device is destroyed. Often the applications do not provide an option to invalidate all the sessions related to a particular user. One way to implement this global logout option is to maintain a random number, known as salt, for each user on the server side and use this salt while generating the user's session identifier.[47] If a user selects to logout of the system locally, only the SID bound to that device can be invalidated. On the other hand, if the user chooses to logout globally, then the server can simply change the user's salt to a fresh one. In this way, the server will not recognize any SID of that user which was created using the previous salt and all the previous sessions of that user will be invalidated automatically. Moreover, the server can maintain a general salt and use it in the creation of all session identifiers. Changing the general salt will result in the termination of all the sessions of all users the server was maintaining.

2.2.3 Session Timeout

A session can be terminated by the user when he selects to logout of the system. However, a server can never be sure that a user will always logout of the system after finishing the use of the application. For this reason, the server needs to remove the sessions that have not been used for a period of time. This type of session termination, also known as relative timeout, can be accomplished by placing a certain time limit on session inactivity. Any session that has not been active over a reasonable time is removed from the session storage. Relative timeout is a useful way of cleaning up the stale sessions. It also prevents session exposure attacks where an attacker might get hold of a session whose user has not logged out explicitly.[14] To implement the inactive timeout, an application can keep track of the last request being received from a user and calculate the elapsed time at regular intervals. When the elapsed time reaches the time for relative timeout, the application will redirect the user to a page which destroys the session. The user's last request time is reset whenever there is an activity on the web browser.

Moreover, despite of being an active session, there is no way for the server to determine whether the session is being used by a legitimate user or it has been stolen by an attacker. Therefore, an application needs to implement an absolute timeout that restricts the overall duration of a session. Absolute timeout is fruitful in a case where an attacker, who has stolen the session ID, tries to keep it valid by making periodic requests. The time limit of relative and absolute timeouts can vary depending on the applications environment and security requirements. Typically, 15 to 30 minutes of session inactivity is reasonable enough to terminate the session. For absolute timeout, the time limit can vary

from 6 to 12 hours, depending on the required security level of the system.

It is recommended to use both kinds of timeout in a system for better security. Furthermore, a two-level timeout mechanism can be used for better usability.[14] In the first level, the application can temporarily lock the user session after a certain period of time and prompt the user for his credentials. The application will wait for the user credentials for another period of time. If the client fails to provide credentials in time, the application will permanently delete the session information from the server. In this way, the user session is locked after a period of time to prevent it from being used by an attacker but still provides an opportunity for the user to continue his session by presenting his credentials.

2.2.4 Context Binding

Often some context binding mechanisms are followed. They are intended to provide additional security measures against session hijacking. The idea is to complicate the task of session hijacking where the attacker has gained knowledge of the session identifier only. One common mechanism is to bind the session to the IP address of the client.[14] This can be effective only in the cases where the attacker is not in the same network as the victim and the IP address of the client is always same. In the cases where the attacker and the victim is in different networks, the attacker will not be able to establish a TCP connection with a forged IP address whereas, when he is in the same network, he can spoof the IP address of the victim. Moreover, the assumption that all requests from the same client will always have the same IP address is not true anymore in today's network environment. The user can be behind load-balancing proxies or the client may be allocated addresses dynamically. Therefore, this mechanism will also reject valid requests more often.

Another mechanism is to bind the session to some HTTP header value. For example, the session can be bound to the user agent string or content type accept header. This headers will always remain same when the requests will come from the same browser. However, the value of these headers are not unique as many users can use the same browser. They are also easy to be forged by an attacker. Therefore, using these context binding mechanisms make the system complex but they do not provide enough security.[14]

2.2.5 Session Mobility

An opposite aspect of context binding is known as mobility. Different types of mobility are defined by Bolla et al.[6] Mobility can be of people, services, session state and terminals. Service mobility gives users access to their personal configuration settings and services across the many devices they use. The situations where the mobile device maintains connection to the service even after the device switches from one network to another is referred to as terminal mobility. Personal mobility allows a user to use any device and switch devices during a task. In order to provide personal mobility and service mobility, session mobility is essential. Session mobility provides the option to move the user session with the service when the user changes devices. [51, 45]

Often web services establish sessions with the user device. To provide session mobility, the sessions need to be bound to the user, not the user device. Nowadays, most often the users need to be allowed to switch devices depending on their needs. Session mobility provides usability as the user does not have to start the session from beginning after changing the device. However, implementing session mobility is not problem-free. Session information needs to be captured and forwarded to the device the client is switching to. The target device can be different from the actual device and thereby the session information needs to be modified to fit the target device. [45]

2.3 Secure Sessions

The primary condition of securing the session is to secure all its components. The most basic rule is to secure the user credentials and the transmission of any other information that is sufficient for accessing the session. Following the authentication process, a session is initiated. This session is associated with the user through a session ID. The session identifier acts as a temporary password as long as the session is valid. Therefore, the session ID needs to be cryptographically strong, secure and protected against hijacking or tampering. The security of the network connection between the user and the server is another critical issue. From the security point of view, it is also important to ensure that the session state information is sufficiently protected against unauthorized access on the client and server sides. To ensure the protection of the network connection between the client and the server, the authentication information and session data can be transmitted over a secure connection. Transport Layer Security can provide such a secure, authenticated and encrypted connection.

2.3.1 TLS Sessions

Transport Layer Security (TLS) [12] is a standard security protocol that is used to create a secure transport layer connection between devices. In this protocol, cryptographic techniques are used to encrypt the network connections above the transport layer to provide a secure communication channel over an insecure network which is shown in Figure 2.4. TLS is the protocol that is usually used for securing the transmission of sensitive user credentials and session tokens. It is based on the previous secure sockets layer (SSL) protocol proprietary protocol developed by Netscape.

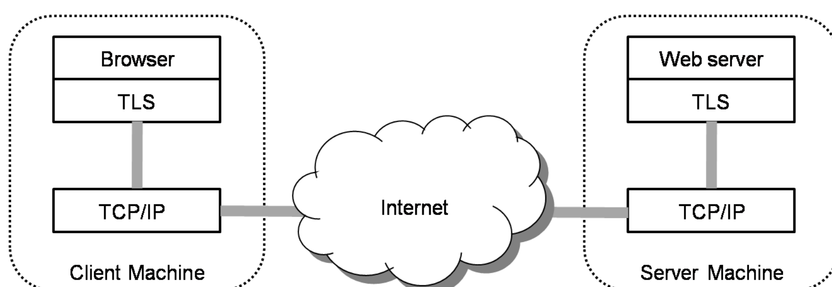


Figure 2.4: Connection between TLS and the Network Layer in the Client and Server

TLS uses both asymmetric and symmetric cryptography for security. TLS can use a collection of cryptographic signing and encryption algorithms, called cipher suites, for the transactions. In the beginning of a connection, TLS negotiates which suite will be used. When a user connects to a TLS enabled server, the server responds with its digital certificate. Digital certificate is a message stating the identity and the public key of an entity signed by a trusted authority. The public keys for official trusted authorities are usually embedded in the client browsers to check the authenticity of the digital certificates of web servers. The client browser validates the digital certificate sent by the server by checking if it is signed by a trusted authority and verifies the server's identity. The client browser then performs an encrypted exchange with the server to generate the shared key. The asymmetric public key cryptography basically protects the exchange of the shared secret, which is used for the symmetric cryptography. The shared secret is used to encrypt the data transfer between the client and the server. After the exchange of the shared secret, the session between the client and server continues encrypted.

TLS provides the feature of session resumption.[40] Through session resumption, devices can resume previously created TLS session by storing the session state

such as, cipher suite and master secret in a ticket in the client side. The ticket is encrypted by a key known only to the server. The client stores this ticket and the master secret. When the client wants to resume the session, it includes the ticket in the request to the server. An attacker with a stolen ticket cannot resume a connection as the ticket is encrypted and he does not know the secret key. The TLS session resumption can only resume the TLS connection but it cannot resume the transaction the TLS connection was protecting. It is handy because creating TLS connections can be heavy for light weight clients.

Another feature of TLS is the ability to provide client certificates. The client can prove his identity by presenting his certificate and responding to an encrypted message. In this way, the client can prove his identity without even revealing his secret key i.e. the private key. Use of a client certificate can mitigate the problem of credentials being captured by an attacker. However, the private key is a long series of random binary data which is not possible to remember. The private key needs to be stored somewhere, typically on the user computer. This restricts the client's mobility while using the certificate as an authentication credential.[14] Moreover, handling of millions of clients in the certificate authorities is not easy.

TLS provides secure data transmission by using an encrypted communication channel. It also authenticate the servers and optionally, the clients and allow the opportunity to prove the identities of the participating parties in a secure communication. TLS also provides the integrity of the transmitted data by using an integrity check. In addition to data protection, TLS can protect against man-in-the-middle or replay attacks.

Chapter 3

Handling Session Data

Authenticating a user enables a web application to determine the sequence of requests coming from the same user and a session is initiated. Typically when a user session is initiated and authenticated, the user session is assigned a unique session identifier. This session identifier is used to identify the user session in the subsequent requests and it is propagated between the client and the server using one of the session tracking solutions, explained in Chapter 2. Once a user session is authenticated, some services need to maintain state information for each user between requests. All data that is required by the web application across different requests within the same session is called session state information.

Maintaining user state information can be challenging depending on the amount of information to be stored. In general, the user session data is supposed to include any data that is not highly sensitive. Any data related to user authentication i.e. username or session identifier, user profile information for personalization, information about group membership, statistical information and other preferences can be stored as session data. However, it is not a good idea to include any information regarding site navigation. The navigation related information needs to be transferred through the URL. Otherwise it will disable the 'Back' button of the browser.

After each request is complete, the user session data needs to be stored somewhere in the system or in an external system from where the application can

access it in the next request. Depending on which part of the service architecture stores most of the session information, the session management techniques are categorized into two types.[42] In a server side session, the session states are mostly stored on the server and the client only stores an identifier of the session whereas in a client-side session the client maintains the entire user session information. These cases are elaborated in sections 3.2 and 3.3, but first we discuss the session identifier.

3.1 Session Identifier (SID)

Session identifiers are composed at application-level. It is a kind of unique identifier that is assigned to each user to differentiate them from each other. In session-oriented web based systems requiring users to authenticate themselves, the session IDs effectively become an alternative of the user credentials for the lifetime of the session. An important aspect of session management mechanism is the security and strength of the session ID in the system.

The session ID needs to fulfill certain requirements for any session tracking solution in order to prevent prediction or brute force attack. First of all, a session ID needs to be unique in order to differentiate between multiple sessions. The two most important characteristics of the session ID are its length and randomness and these properties are helpful to prevent against prediction of the session ID by an attacker.[35] For secure session management, a cryptographically strong algorithm is used to generate a unique random session ID for an authenticated user. It is also important that the session IDs should have a sufficient length to mitigate the brute force attacks. It is recommended to use a session ID whose length is at least 128 bits.

The most common vulnerability with session ID is predictability. The causes are lack of randomness or length or both. Some web applications also use sequential numbers as session IDs. This types of session IDs are extremely weak and prone to session prediction. Moreover, an attacker can hijack the session of a legitimate user by capturing the session ID.

In order to mitigate the effects of an attack where the attacker has stolen the session ID, the server can maintain a timeout mechanism and regenerate the session ID after the maximum lifetime of a session ID is reached. The session ID can be transmitted over a secured connection to avoid being sniffed by an eavesdropper. If session IDs are re-created whenever there is a change of privilege, the application can prevent session fixation attacks where a user logs into a session fixed by an attacker.

3.2 Server-side Sessions

Server-side sessions store most of the session state information on the server side and require only a small amount of content transmission between client and server. By storing the session data in the web server, an application is protected from situations where a user can perform any accidental or incidental changes to the session information. Typically a server-side session management system initiates a user session by assigning a session ID to it and then passes the SID to the client through a session tracking mechanism. All the session state data are stored on the server end in a file system or in a database. The SID transmitted through a client request is used by the server to identify the corresponding session states in the session storage.

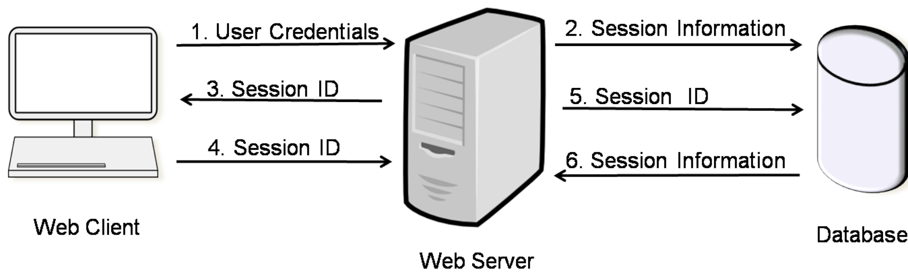


Figure 3.1: Server-side Session Handling

Figure 3.1 shows the overall structure of an application with server-based sessions. Here, the client sends a request to the server along with his credentials. The server validates the user credentials and initiates a session after successful authentication. To initiate a session, the server generates a unique session ID for the user and stores the session state information in a database. The session information stored on the server database is indexed by the user session ID. After initiating a session, the server transmits the session ID to the client. The client stores the session ID and passes it to the server with every subsequent request. When a server receives a request from a user including a session ID, it uses the ID to locate the session information of the user in the database and retrieves the session state. Once the user session information is retrieved, the server sends the response to the client.

Server-side session management is capable of storing large amounts of state information that is easily accessible to the application. Moreover, server-side sessions are easily expandable to include additional session information. In-

stead of storing all the session information, this mechanism stores only a session identifier on the client and thus solves the problem of increased request size. However, storing session information in a database makes it hard to distribute over multiple servers and often the database becomes the resource bottleneck in the system. Since sessions consume resources on the server, the server requires mechanisms to remove the inactive sessions from the database at regular intervals.

Ye et al.[55] have proposed a scheme for the web servers to efficiently store session information on the server and verify the cookie state information received from client. In this scheme, the web server is also able to record the expiration state information of the cookies and thus prevent the application from cookie replay attacks. The server maintains an access control entry for each user in a cookie state database. A secret key is used by the server to generate and verify the cookies. Two different scenarios have been presented for this scheme: the simple scheme and M/K scheme.

In the simple scheme, the most recent time when a user has requested a log out is stored on the access control entry. Initially, it is set to 0. When a user logs into the application, the server generates an authentication cookie. First of all, the server creates a `Msg` that contains the username, the IP address of the user device, and the current time. Then a keyed MAC of `Msg` is computed using the secret key of the server and the cookie is generated by concatenating the MAC code and the `Msg`. The server then sends this cookie to the client and the client is able to access the server resources using the cookie in the subsequent requests. When the server receives a request from the client including a cookie, the server first validates the cookie using the MAC code and then compares the timestamp of the cookie with the recent logout time stored in the access control entry. If the timestamp is more recent than the recorded time, the client is given access to the resource. When a user selects to log out of the system, the server again compares the cookie timestamp with the recorded time in the access control entry and, if the cookie timestamp is more recent, the server updates the access control entry with the cookie timestamp.

In an application where a user can have multiple parallel sessions, the simple scheme needs to be modified, Ye et al. [55] propose the second scheme for this and it is known as M/K scheme. This scheme allows a server to keep track of a maximum of m authentication cookies within k days. The structure of the access control entry and the cookie in this scheme are shown in Figure 3.2. The (`ctime`) is entry creation timestamp, (`mtime`) is entry update timestamp. The current cookie id represents the id to be used next, the session counter counts the number of cookies generated on each day over a k-day period and the cookie states vector (`cstates`) maintains the state of each cookie. When a user logs into the application, the server sets the `ctime, mtime` to current time, sets the

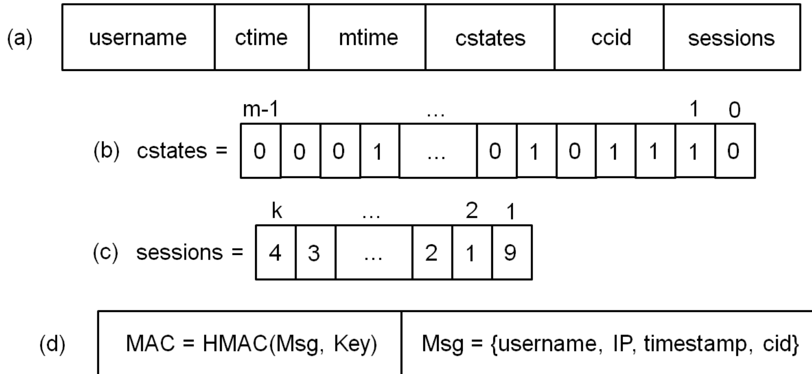


Figure 3.2: Access Control Entry and Cookie in M/K Scheme[55]

first bit of `cstates` to 1, and adds 1 to the first cell of `sessions`. The server then generates the cookie and sends it to the client in the same way as in the simple scheme. When the user signs in at another time, the server accepts the request only if the sum of the k entries in the cookie session counters has not exceeded the maximum cookies limit(`m`) and updates the access control entry for the new log in. When a client requests a resource, the server first validates the cookie and then compares the timestamp of the cookie with current time. If the cookie is generated in the last k days and the cookie state bit of this cookie is set to 1, then the client request will be accepted. The cookie session counter ensures that a maximum of m fresh sessions can be initiated within k days and these parameters m and k can be adjusted according to the security requirement.

In both schemes, the integrity of the cookie is protected through the MAC and the confidentiality of the cookie is maintained by using a server secret key. These schemes also have the ability to revoke the sessions of a client and it can prevent replay attacks.

3.3 Client-side Sessions

In client-side sessions, all or most of the user session information is maintained at the client side. For this reason, the session state data are transmitted between the client and the server on each request. If the amount of data being passed from client to server is large, then a client-side session mechanism may become heavyweight. This type of session management is good enough when the con-

tent being transferred is small. Moreover, any kind of back-end support is not necessary in client-side session. As a result, client-side sessions are suitable for distributed systems. A typical client-side session management process is shown in Figure 3.3.

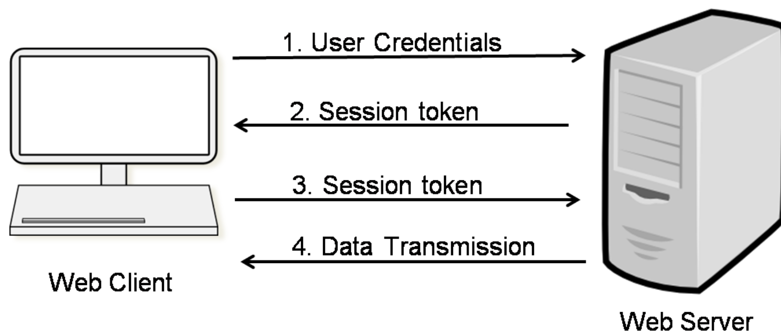


Figure 3.3: Client-side Session Handling

Cookies are most suitable for storing session information in client-side sessions. They are an attractive method for transmitting small amounts of session data. One of the most distinctive features of client-side sessions is that this technique has a low back-end overhead. All the session data are passed through the client requests. Therefore, the client-side sessions are scalable and easy to distribute over many servers. Moreover, this technique can scale to a large number of clients. The task of storing the session information is solely managed by the clients and there is no chance of creating a resource bottleneck at the session storage.[42]

Most web applications do not want their users to be able to modify their own session data and, in fact, the user's ability to modify session data can lead to security vulnerabilities. Therefore, the client-side sessions need to provide confidentiality and integrity of the session data and this is ensured when no party except the server is able to interpret or manipulate the session data. To accomplish this, an application can encrypt the user session data before passing it to the client. Alternatively, the server can sign the transmitted session data in order to prevent any tampering on the client side and verify the session data when received from the client within requests.

Typically, the cookies in a client-side session are vulnerable. It is recommended to avoid the use of persistent cookies. Session cookies are destroyed when the

client exits the user agent software, whereas persistent cookies are usually stored on the system for a longer time. In addition, it is a good idea to have some session revocation option on the server side, so that the server can invalidate the sessions that are already expired.[19]

A stateless session cookie bases session management mechanism is presented by Murdoch in Hardened Stateless Session Cookies.[31] In this session management mechanism, only a salted iterated hash of the user password is stored in the database and its pre-image is kept in a session cookie for each user. The session management uses the following the recursive definition:

$a_0(salt, password) = H(salt||password)$
$a_x(salt, passowrd) = H(a_{x-1}(salt, password)||password)$

Where the *salt* is a long cryptographically secure pseudorandom number maintained for each user and *password* is the user password. $H(.)$ is a cryptographically secure hash function. When a user logs into the application for the first time, he passes his credentials to the server and the server generates the random salt and calculates the authenticator, known as $v = H(a_n(salt, password))$, and stores both of them in the server database. The value n is the count for hash iteration and it is a public value. When a user presents his credentials to the server to log in, the server calculates $c = a_n(salt, password)$ using the received password and stored salt in the database. The server rejects the user request if $H(c) \neq v$ and accepts otherwise. If the user request is accepted, then the server generates a cookie as following:

$exp = t\&data = s\&auth = c\&digest = MAC_k(exp = t\&data = s\&auth = c)$

Here, t is the expiry time for the cookie, s is the state the web application needs to maintain, and the digest is a message authentication code under a key known only to the server. After receiving a cookie, the client can request other resources by using this cookie.

In this stateless session cookie based system, an attacker with read access to the server is not able to spoof an authenticated session. Even if the attacker gains knowledge about the MAC key of the server, the attacker needs to know the user password to create a valid cookie. Since the user password is not stored on the server, the attacker cannot generate any valid session cookie.

Chapter 4

Session Vulnerabilities

In a system, a vulnerability is a weakness that allows an attacker to cause damage to the service and its owner and users. An attacker uses methods, known as attacks to manipulate the vulnerabilities in the service.[36] In web based services, the system is divided between a client and a server that creates and maintains a session for the client. Most website attacks are carried out on sessions. Session handling is a critical part of web based services. Users assume that web based systems are designed securely but in reality most website developers do not think thoroughly about how they are setting up the site security. As a result these systems can only provide medium to low level of security and it is easier to exploit vulnerabilities with the session data in these systems.

Capturing or identifying the session identifier is the main target in a majority of attempted security breaches. To the web system, anyone presenting a valid SID is a genuine user. There is no way for a web service to determine whether it is presented by a valid user or by an attacker. Therefore, by retrieving a valid session ID, the attacker can impersonate the user effectively and obtain the permissions of the legitimate user. Session attacks can be categorized into two major types: session hijacking and session fixation. The following sections in this chapter discuss different types of session attacks and the methods to protect against these attacks for secure web services.

4.1 Session Hijacking

Session hijacking attack refers to any kind of attack where the sensitive session token or ID is leaked or compromised. The attacker obtains a valid session ID after it is associated with a legitimate user session. Using the stolen session token, the attacker can gain the permissions of the user and access to the web service. The session token can be compromised in many different ways. [36] Depending on the exploitation method, session hijacking can be named differently. Some common methods of session hijacking are session prediction, session sniffing, cross site scripting and session capture.

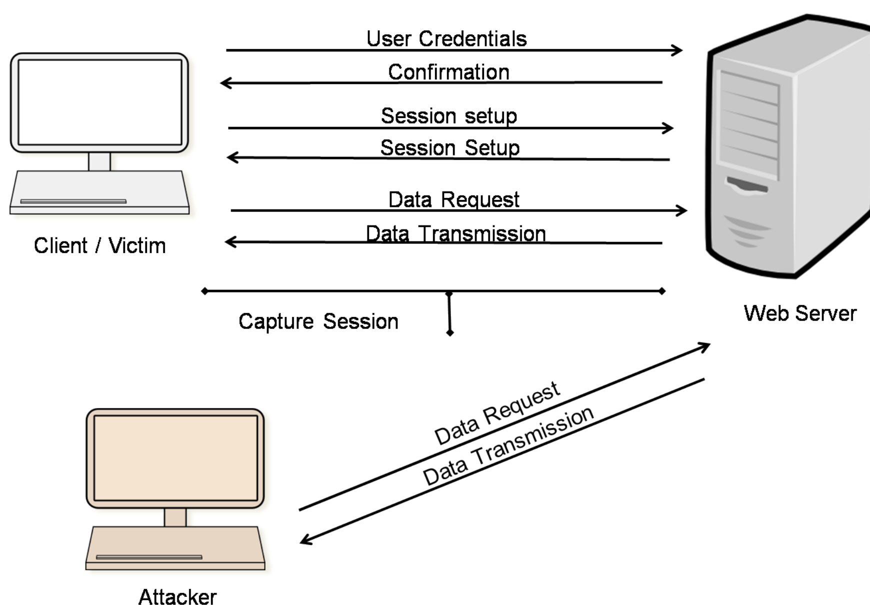


Figure 4.1: Session Hijacking

Figure 4.1 presents a general scenario of session hijacking attack. At first, a legitimate client passes his credentials to the web server and the server responses with a confirmation after validating the credentials. Once the client is authenticated, a session is established between them and they start communicating. An attacker can capture the session between the client and the server using one of the methods described later in this section and start impersonating as the client with his session information. By following these steps, an attacker can gain the access permission of the client.

In general, there are many ways to protect against session hijacking. One way is to transmit the session identifier for secure contents through a secured connection. Hence session hijacking can be prevented with the use of different session identifiers when shifting between secure and unsecure contents i.e. authenticated and unauthenticated sessions or unsecure http and secure https connections.[15] Another precaution for session hijacking is to re-authenticate the user prior to any sensitive actions. In this way, any attacker impersonating a valid user cannot proceed further and cannot perform restrictive actions. In addition, each session can have a maximum lifetime. After being active for a specific duration, the user can be asked to re-authenticate. Hence an attacker cannot keep on using a hijacked session for a very long period. For systems using the GET method of session propagation, the SID should be regenerated frequently to avoid the exposure of the SID for a longer time.

It is rather impossible to correctly detect a session hijack as very few attributes are passed from the client. Even then, some precautions can be adopted to make the session hijacking more difficult. Some of the client information that can be used to detect a session hijacking attempt are client user agent and the IP address. User agent is an HTTP response header sent from the browser, and it includes the name of the browser and its version and the operating system. Typically the user agent does not change at short intervals. However, user agent is easily predictable and the value may not be constant in a system with a number of proxy servers. In this kind of system, change of user agent may generate a false alarm. The other used information, the IP address of the client can change often nowadays and thus it is not wise to check the exact IP address. A better solution is to check only the subnet part of the IP address. Even though checking the IP address is not reliable, it can make the job of an attacker a bit harder. Any user who has failed this check should be automatically logged out of the session with the existing SID to prevent any session hijacking attempt.[33]

4.1.1 Session Prediction

In session prediction, an attacker predicts a session ID. There are several methods for this. Typically session prediction happens when an attacker can detect a pattern in the SIDs given by the service. This is possible when the web application has poor predictable session identifiers. For example, when the SID is assigned sequentially, by knowing any one session ID the attacker acquires the knowledge of the previous and next SID. Furthermore, if the services have weakly encrypted or too short SIDs, it is easier for the attacker to guess a valid session ID.[49]

Before predicting a session identifier, an attacker needs to gather a good number

of valid SIDs of legitimate users of the system. After having a collection of SIDs, the attacker is able to perform analysis. He may gain knowledge about the information that is used to create a SID and figure out the structure of SIDs. Furthermore, the session generation process might also include encryption or hashing. Once the attacker gets an idea of the pattern in SIDs, he may predict a valid SID to gain access to the system.[36]

For example, a security bulletin of Netcraft.com [53] has given a scenario where a session identifier of a system may include a session counter which increases with each new session, the current timestamp when the session id is being created, the IP address of the server generating the SID and a few bytes of random salt. The identifier generated from the mentioned information may also be weakly encrypted. Here the session counter and timestamp always increments, the IP address of the server will remain constant for a single server system and typically the random salt is either zero or only refreshed with server reboot. A malicious user may create many sessions with the target system and analyze the collection. He might gather the collection of SIDs by other means also. Once the attacker is able to decrypt the SIDs and know the information used in generating them, he can try generating many new SIDs following the same structure and become successful eventually.

However, the predictability level of SIDs of different types depends on their patterns. Some session tokens are harder to predict than others. Hashed or encrypted session IDs are less prone to prediction. Even session IDs containing the information of random salt can be considered more secure. When session IDs are generated, it should have sufficient entropy to prevent prediction. It is recommended that application generated session identifiers must have at least 128 bits of entropy. Strong large random numbers retrieved from a seeded random number generator can be used as session IDs to prevent session prediction.[33] Strong encryption on all transmission can also prevent an attacker to gather a collection of SIDs to perform analysis.[13]

4.1.2 Session Sniffing

Session sniffing is a kind of interception. Interception is possible when an attacker is able to gather data from which he can figure out the SID. The implementation of session interception is more difficult than session prediction. Session sniffing can be thought of as a man-in-the-middle attack where the attacker can capture the sensitive session token transmitted between the user and the server through sniffing the transmission traffic between them. After capturing the session token, the attacker can act like a legitimate user to the server and can have unauthorized access. Session sniffing is possible when the communica-

tion channel between the user and the web server is insecure. In other words, when the web service is not configured to use HTTPS connection, all data transmitted between the client and the server is in plaintext. Without encryption the transmission can be sniffed by any computer on a network through which the packet flows. For example, POST form elements of a login page contain the username and password and this data can be seen while sniffing plaintext transmission. After that, the HTTP response from the server might contain the `Set-Cookie` header including the SID.

Strong cryptographic functions seeded with strong, random key can be used for generating SIDs. A session identifier whose length is more than 128 bits can be considered as a safe one. Moreover, encrypting the session identifier can make the web based system secured to some extent. If the data transmitted between the server and the client is encrypted, the data captured from the transmission becomes unreadable. Consequently any kind of interception is not fruitful. However, any kind of encryption is not able to prevent session sniffing attacks. Only the use of strong cryptographic algorithm is effective.[33] Use of TLS communication for encrypted transmission can prevent this kind of attack. Error messages and stack traces presented by a web system on the occurrence of unexpected events also must be properly sanitized so that they do not reveal any information or SIDs. Oftentimes the malicious users are able to gather important data about the system by triggering errors. [33]

4.1.3 Cross Site Scripting

Web services always provide some methods for the security check in order to protect against information leakage. Cross site scripting (XSS) permits an attacker to bypass these security checks. In this method, an attacker can read or modify the session ID. This is achieved by uploading malicious code on a targeted website, and this piece of code makes the website to transmit data across sites or modify it. An attacker's ability to perform cross site scripting does not depend on the used session tracking solution i.e. whether the SID is stored in GET or POST data or cookies.

At first, the attackers target a website to be compromised. Then, they place malicious code on it with the help of search input, comment box, forums or any other places where the website accepts user input. A website is vulnerable to cross site scripting only when it fails to properly sanitize user inputs. A malicious user can utilize this cross site scripting vulnerability and make the server read or modify the GET or POST form elements or cookies. This is possible because the malicious code resides on the target site that has the permission to read or modify session data. Hence the code on the target site can direct the client to

perform an unauthorized task.[36]

In most cases, HTTP requests include the HTTP Referer field. HTTP Referer attribute keeps record of the page that redirected to the current page. Typically the Referer field contains the entire URL of the last page including the GET variables passed across pages. If session identifiers are passed as a GET variable, the HTTP Referer attribute will expose the SID to the next site that is visited after leaving the current page.To perform cross site scripting attack on sessions with GET form data, the malicious code may contain a link that refers to an external site. Once the client is redirected to this external site, this site is able to retrieve the SID from the HTTP Referer filed.[33]

For sessions handled with POST data, a code can be placed to add forms including a hidden element and this hidden form element contains the SID. Then the malicious code transmits the form with the hidden field to an external site. Furthermore, XSS attacks can also bypass the security policies of a browser related to cookie access. A web browser allows a domain to read or modify a cookie only if it is the one that set the cookie. In XSS the request to read or modify a cookie is being generated by the malicious code, placed in the actual domain, and therefore the browser is unable to detect any unauthorized access.[33] In addition, JavaScript is able to perform extensive attacks. With JavaScript, it is possible to set the `onsubmit` property and thereby change the page that accepts the form data upon submission.

Preventing the clients to input raw HTML code will mitigate the XSS attack possibilities. Any type of user input needs to be sanitized and escaped properly. When the user input is escaped, it will be displayed on the web page as literal code. Any form of link, HTML form or scripts posted by a user will become inactive after being filtered. Often it is recommended to use built-in filtering functions instead of implementing a new one. Parsing HTML is very complex and implementing a filtering function of one's own might result in some unnoticed parsing flaws.[33]

The level of security can be increased by setting the `secure` attribute of the cookie as true on a cookie based session. When the secure attribute is set to true for a cookie, the cookie will be transmitted only over a secured communication channel i.e. HTTPS connection. As a result, the unsecured webpages will not be able to access the cookies of the secured pages which will restrict the chance of XSS attack being successful.

To avoid exposing the SID from the HTTP Referer field in sessions using GET data, an additional level of redirection can be added before the client is led to the target page. The webpage containing the SID as a GET variable will be forwarded to an intermediate page and from there the client will be redirected

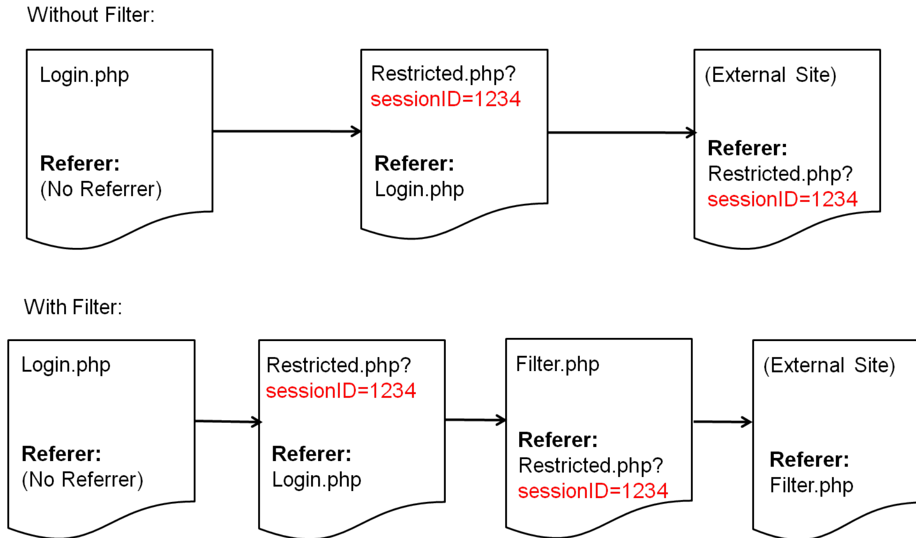


Figure 4.2: HTTP Referer Filtering [33]

to the actual destination page. In this way, the extra redirection will reset the HTTP Referer field of the destination page to the URL of the intermediate page. Therefore, the SID of the current page will not be passed to the destination page. The HTTP `Location` or `Refresh` header is useful for performing this action. This method is known as HTTP Referer filtering. [33]

Figure 4.2 presents an example of HTTP Referer filtering. First, the general scenario is shown when there is no filtering applied. It is visible that the HTTP Referer field in the external site contains the URL of the `Restricted` page along with the session identifier. When a filtering is applied between the `Restricted` page and the `External` page, the URL containing the session identifier is no more visible to the external site.

4.1.4 Session Exposure

Session exposure attack is related to the capture of the session token by browsing session storage in a shared host system. A web based server system needs to store the session data somewhere. In some cases, the server may store the session data or SIDs in a publicly-accessible location. Generally, PHP stores its session variables data in a particular directory that is the same for all the shared hosts.

In this public session store, the session tokens are saved in files. In some cases, the session data is stored as plain text in the files. This leads to a situation where any sharing host can browse the public session storage and the session data are exposed.[49]

Additionally, cookie-cache is a critical issue for shared host client systems. An attacker who is able to access the cookie cache can easily be able to read persistent cookies. If the system is publicly accessible then there exists a potential risk that consequent users will be able to browse the cookie cache of the previous users.[33]

To mitigate the damage caused by a session exposure attack, a web based system should be designed in such a way that only a session identifier is stored on the client machine and the server maintains the other session data including the user credentials. To prevent the exposure of the session identifiers from a cookie-cache, persistent sessions should be avoided. In this way, the session cookie will not be stored on the client cache. Another method to limit the chance of session exposure is to automatically expire a session after a short period of inactivity. This will reduce the lifetime of a session and therefore, reduce the probability of it being hijacked. Storing the session identifiers in a secure location of the server with access control is also a good practice.

4.2 Session Fixation

Besides session hijacking, another major session attack is known as session fixation. Session fixation arises in systems where the attacker is able to specify the session identifier for the session of a legitimate user. In this attack, the attacker at first sets the application user's session ID to an explicit value. Server generated session ID may sometimes be forged due to a weak cryptographic algorithm or easily guessable session ID. The victim then authenticates himself to the application server using the session ID fixed by the attacker. Once the user is authenticated, the attacker can use the predefined session ID to access the application server and impersonate the victim user.

In figure 4.3 an overall idea of session fixation is shown. First, the attacker creates a crafted link to connect to the web server `www.example.com` by setting the session ID to a fixed value of `1234`. The attacker then presents the link to the client by posting it on a public place or using some other vulnerability. At some point, the client clicks on the link and connects to `www.example.com` with the fixed value of 1234 for the session ID by presenting his credentials. When a session is established between the client and the server, the attacker can access

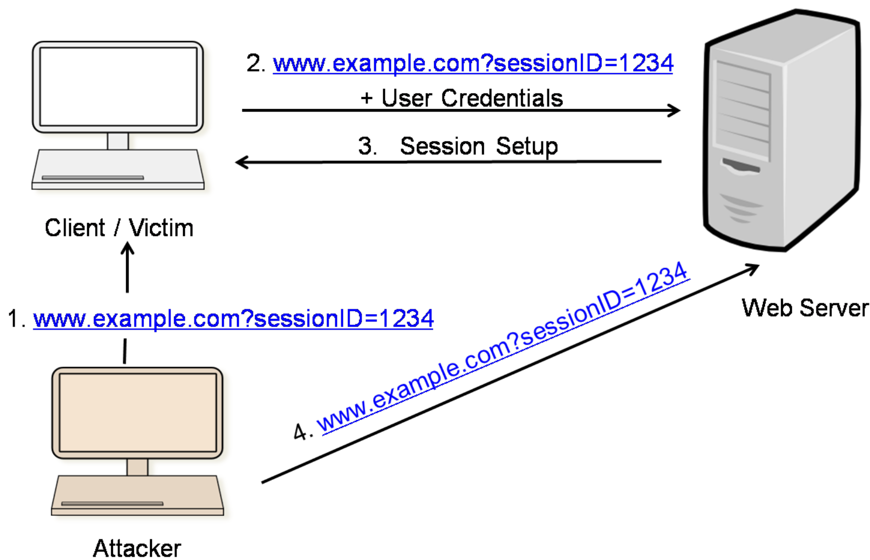


Figure 4.3: Session Fixation

the client's session with the known session ID.

Basically, session fixation is a three step attack. The steps of a session fixation attack is shown in Figure 4.4. To make a session fixation attack, the attacker generates a link of the target website along with a chosen session identifier. This is the first step of a session fixation attack. Sometimes, an attacker needs to initiate a session with the server before authentication and pass on the URL of that session to the user to authenticate. It those cases, the attacker needs to maintain the trap session at a regular interval. In the second step, the attacker needs to present the session trap to the victim user and fix the session. Typically the attacker posts the link on a public location. To be on a safer side, the attacker may use the XSS vulnerability of the target website, if possible, and place the link on the target site. Apparently a victim user may click on the specific link and try to log into the system. This is the last step of the fixation attack where the victim accesses the fixed session. Since the crafted link is encoded with a SID, the web server will not try to overwrite it with its own. Consequently the user will be able to log into the system with the attacker specified SID. Now the session becomes accessible to the attacker even without authentication. [22]

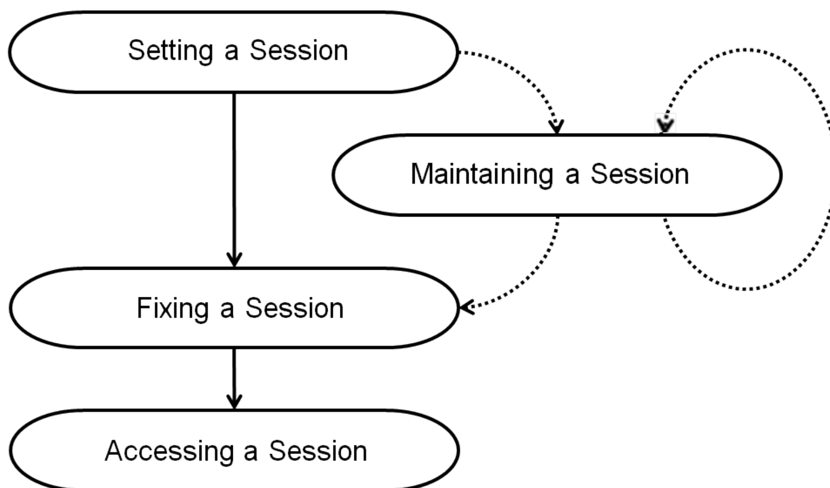


Figure 4.4: Session Fixation Steps

4.2.1 Setting a Session

Session fixation attack is potentially effective in permissive web applications. A permissive web application will accept the session identifier presented by the client and create a new session with the given SID if the session does not exist. That is, the web server will not try to assign a server generated SID. On the other hand, a strict web application will only accept session identifiers generated by the server and override any given session identifier with its own one.[33, 22] Permissive web applications are vulnerable to pre-assigned session identifiers while strict ones are not.

In a session fixation attack on a permissive application, the attacker can fix a session ID of his own choice and the web application will adopt this SID when the victim presents it. However, in a strict system, the attacker has to create a session with the web server and then make a trap session with the SID generated by the server. If the server has implemented idle time out for an inactive session, then the attacker might need to refresh the trap session at regular intervals.

4.2.2 Fixing a Session

Session fixation attack does not depend on the session tracking solution. Session data propagated through GET or POST form data are mostly vulnerable to this

kind of attack. However, cookie based sessions are also at potential risk. Some client browsers do not follow properly the standards of creating and reading cookies according to the RFC 2965.[24] Therefore, these browsers contain flaws that make them vulnerable to session fixation attacks. The session identifier of a session fixation attack can be fixed in many ways.

HTTP Response Splitting

Using the user input vulnerabilities the web server of an application can be made to modify the HTTP response and perform some malicious task. Murphey[33] presents a scenario, where a web page redirects to another page whose URL is embedded with a variable named `productno` and the value of this variable is taken as input from the client. The HTTP response of that web page will contain the following line, when the value of the `productno` is `AB1234` :

```
Location: http://www.example.com/index.html?productno=AB1234
```

Now lets assume that the input for the variable `productno` is provided as following:

```
AB1234%0d%0aSet.Cookie:%20sessionID%3d4567
```

Here the value given as argument to the variable `productno` is encoded with hexadecimal data where `%0d%0a` decodes to `CR-LF (end of line)`, `%20` to `space` and `%3d` to an `equal sign(=)`. Once the variable is decoded and embedded to a redirect URL, it looks like this:

```
Location: http://www.example.com/index.html?productno=AB1234

Set.Cookie: sessionID=4567
```

The given input has added an additional header to the HTTP response of the server and this header will cause the receiving client browser to set a cookie with the specified SID. This type of attack is known as HTTP response splitting. The malicious user can also set an expiration date for the cookie and consequently the cookie will be stored on disk of the client machine. As a result, the session ID will be persistent for a longer period. This kind of HTTP splitting attack can include many lines to the HTTP response.

This kind of attack cannot be successful for the servers that escape or replace the

`CR-carriage return, \r, LF-line feed` or `end of line` characters. These characters allows a malicious user to control the rest of the headers and body of the response as well as appending additional responses. Any application that does not allow these characters as input are free from the HTTP splitting attack.[33] Furthermore, a preventive measure can be taken into consideration while encoding a user input into the URL for HTTP headers. The URLs can be encoded before appending them into the critical HTTP headers such as `Location` or `Set-Cookie`.

Client-side Attacks

Cross site scripting attack is a kind of client side attack. In XSS the attacker can set the session token and also post malicious code to the target webpage. This attack becomes possible in the web systems that fail to perform proper sanitation and filter of the user input. Moreover, sometimes the vulnerability is caused by a filtering process that fails to filter out some input types. Even though the output of the filtering process is not a correctly formatted HTML file, some browsers are made to handle them. Thus sometimes XSS attacks are successful in some specific browsers.

For example, in a cookie based session XSS can be used to bypass the security checks of a browser. The browser allows a site to read or modify a cookie if it has been set by a site within the same domain. Using cross site scripting an attacker can upload a customized link with malicious code on the targeted website, and when the victim clicks on the link provided by the attacker, the malicious program runs on the client-side. Since the code is uploaded on the site which has set the cookie, the browser will allow the malicious code to access the cookie and modify it. The attacker can use programs to set the value of the cookie value of his own choice.[33] For session propagated with POST data, an attacker can create a form with hidden elements containing a specific SID and post the code on a public location. For systems using the GET data as session tracking, a XSS attack can be launched by creating a specially crafted link that encodes the attacker's chosen session ID.

4.2.3 Accessing a Session

After the victim user has performed authentication for the trap session, the system will consider the SID as a valid one. Since the user session is associated with a SID known by the attacker, he can present the chosen SID to the system which is considered to be a valid one and acquire the access permissions of the

victim. This attack can become extremely harmful when the user does not log out of the system and the attacker can use the victim's identity for a longer time.[22]

4.2.4 Countermeasures

The core of the session fixation attack being successful is the server's nature to adopt any pre-defined session identifier. The permissive nature of the web applications allows a system to behave like this. For this reason, the web applications can only provide preventive measures to protect against the session fixation attack.[22] The web applications should only accept session identifiers generated by the web server i.e. behave like strict web applications. The web server should always ignore any session identifier provided by the client.

The best way to prevent session fixation is to inactivate the session that is active during authentication and create a new session with a fresh session ID after a successful login. In this way the attacker cannot authenticate a session with a predefined session ID. The attacker can gain knowledge about a valid session identifier only when he is a legitimate user of the system. Therefore, he will not be able to create a session trap with a valid SID. Even if the attacker can fix a session with a specific SID, the SID will be overridden after the authentication and the attacker will not be able to access the new session. Checking the stability of the user agent header and the network portion of the IP address can also restrict the session fixation attempt.[33] Another effective method for highly sensitive applications is to check the SSL client certificate. Using this SSL client certificate, a server can check if the session was originally initiated with the same client using the same certificate.[33, 22]

4.3 Other Attack Possibilities

In addition to session hijacking and session fixation, there exists some other attack possibilities. Some of them are presented here briefly.

4.3.1 Brute Force Attack

Brute force attack is simple in method. It appears when an attacker repeatedly tries all possible session identifiers until a valid one is found. This is potentially

an effective way of acquiring SIDs. If the systems are not designed carefully to avoid the brute force attack, this can become significant. Strong pseudo random numbers with enough entropy size can be used to prevent the brute force attack. Care has to be taken to while calculating the effective entropy size. The entropy size should be calculated based on the domain of possible identifiers instead of the size of the buffer containing the ID.[33]

4.3.2 Unlimited Session Expiration

Unlimited or too long session timeout may cause an unauthorized session access attack. Long session expiration gives an attacker time to predict or exploit a valid session ID and gain unauthorized access to that session eventually. Moreover, sometimes SIDs are logged and cached in proxy servers. If an attacker can capture the sessions that have not expired yet, he can use those sessions too. To prevent this kind of attack, session ids should have smaller timeout window.

CHAPTER 5

Session Management in PHP

PHP is a recursive acronym and it stands for PHP: Hypertext Preprocessor. It is a widely-used general-purpose scripting language that is mostly intended for creating dynamic web content. PHP can be deployed on most web servers and operating systems. It is an open source tool for server-side scripting, command line scripting and for creating desktop applications. One of the most significant features of PHP is its ability to support a wide range of databases. This gives the opportunity to write database-enabled web pages in PHP.[27] Moreover, each request in PHP is handled independently and the objects used and resources opened do not live longer than a single request. Therefore, it is easy to distribute PHP application across server clusters. The current version of PHP is PHP5.

In server side scripting, PHP scripts are embedded inside HTML documents. PHP codes are enclosed in starting and ending processing instructions `<?php` and `?>`. This allows to move into and out of PHP mode. When an HTML file is parsed in the web server, any PHP script embedded in it is processed by the PHP engine installed on the web server. After that each PHP script is replaced by the corresponding script output and the generated HTML is sent back to client. The client can only see the results of the script but not the actual script. The overall architecture of a PHP web application is presented in Figure 5.1.

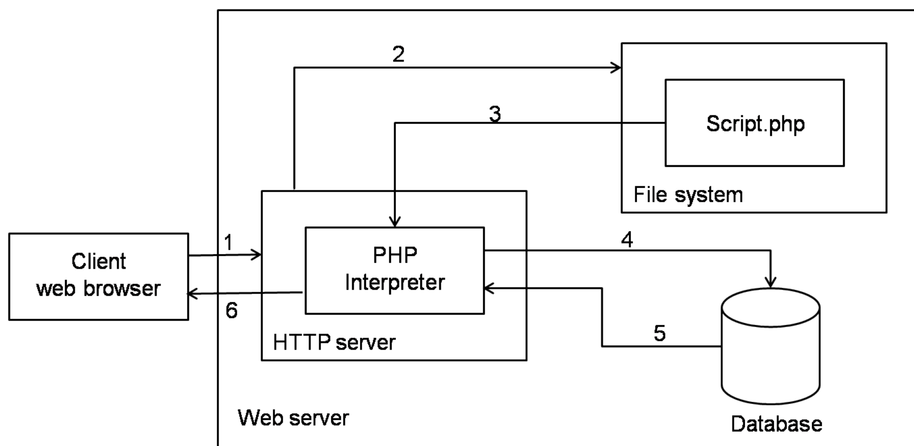


Figure 5.1: Architecture of PHP Web Applications

5.1 Handling Session

The best way to maintain session in PHP is to use the built-in session handling mechanism. Sessions in PHP allow to preserve session data on the web server across subsequent requests. A typical HTML website cannot pass data from one page to another one. Any information used in a page is forgotten upon a new page. In PHP, a session is maintained for each user by creating a unique session ID (SID) and storing different session variables based on this SID. PHP uses cookies or URL encoding as session tracking mechanism. A unique SID for each user helps to differentiate between two different user's data. Storing data across pages in PHP is useful in a way that all the interaction are hidden from the user. Moreover, these session data are temporary and, in most cases, they are destroyed once the user has left the website and the session.

Figure 5.2 shows the message passing sequence of default PHP session management module and how the session ID is used as an index to the session variables stored on the server. The session variables are the state information related to a user and stored on the web server. The state information of a user on the server can be located using the session ID. For example, simple session information in PHP can be thought of as setting the value of a variable on a web page and then recalling the variable on the next page. PHP uses the global array `$_SESSION` to hold all the session variables. A simple session variable can be in the form `$_SESSION['var']`. The associative array `$_SESSION` is able to store any kind of information e.g. arrays, objects, strings and values.

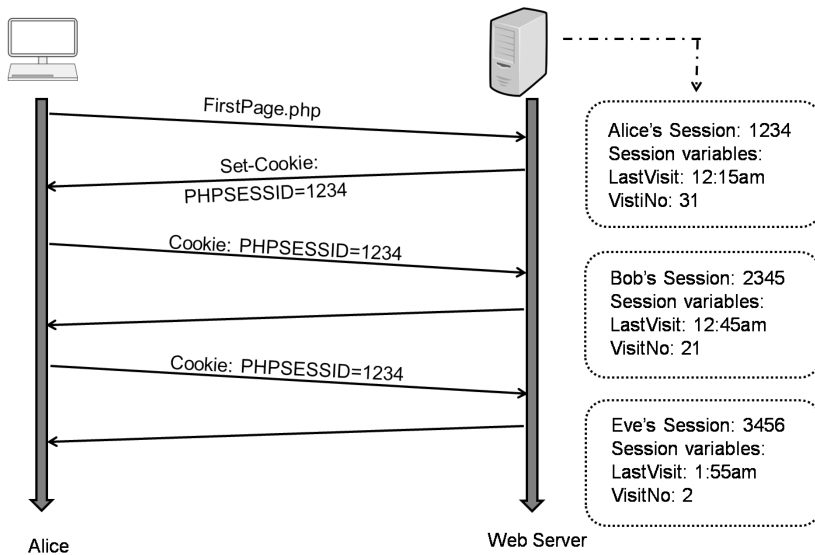


Figure 5.2: Session ID and Session Variables

Figure 5.3 shows how a typical session based web application works with PHP session management module. At first, the login page is used to prompt the user for credentials. The user passes the credentials to the server with an HTML form using the POST method and this information is gathered by the setup script. Once authenticated, the setup script sets up a session of the user with the help of the session management module. After initiating the session, the server generates a welcome page and redirects the user to that page. The welcome page and other application pages can retrieve the session variables from the global array `$_SESSION`. The session is maintained between the user and the server as long as the user is requesting any of the application pages. When the user decides to log out of the system, the server calls a logout script to destroy the user session and redirects the user to a receipt page which does not use session. In the following sections, we will give examples of different functionalities of PHP session management.

5.2 Creating Session

A PHP session needs to be initiated before any user specific information can be stored in it. It is done at the beginning of the script, before any HTML or text

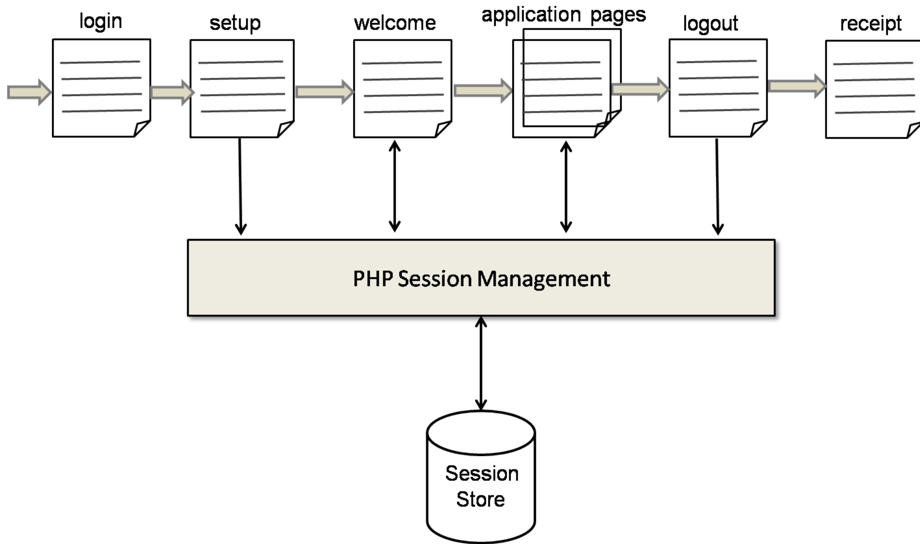


Figure 5.3: A Typical Session based Application in PHP [25]

is sent. The function `session_start()` first checks if the client request includes a session identifier, in this case known as `PHPSESSID`. If the client already has a SID, the associated session data can be retrieved and accessed via `$_SESSION`.

If the request does not include any SID, then PHP will create the user session in the session storage of the web server and associate a SID for that session. This will allow the server to start saving user specific information. After this, PHP sets caching headers and the `Set-Cookie` header. The control flow of session creation and maintenance is presented in Figure 5.4.

Listing 5.1: Creating Session `start.php`

```
1  <?php
2  //starting a PHP session
3  session_start();
4  //setting a session variable
5  $_SESSION['status'] = 'beginning';
6  //showing a status message
7  echo 'Starting a session.';
8  ?>
```

In Listing 5.1, first we create a session for the user by calling the function

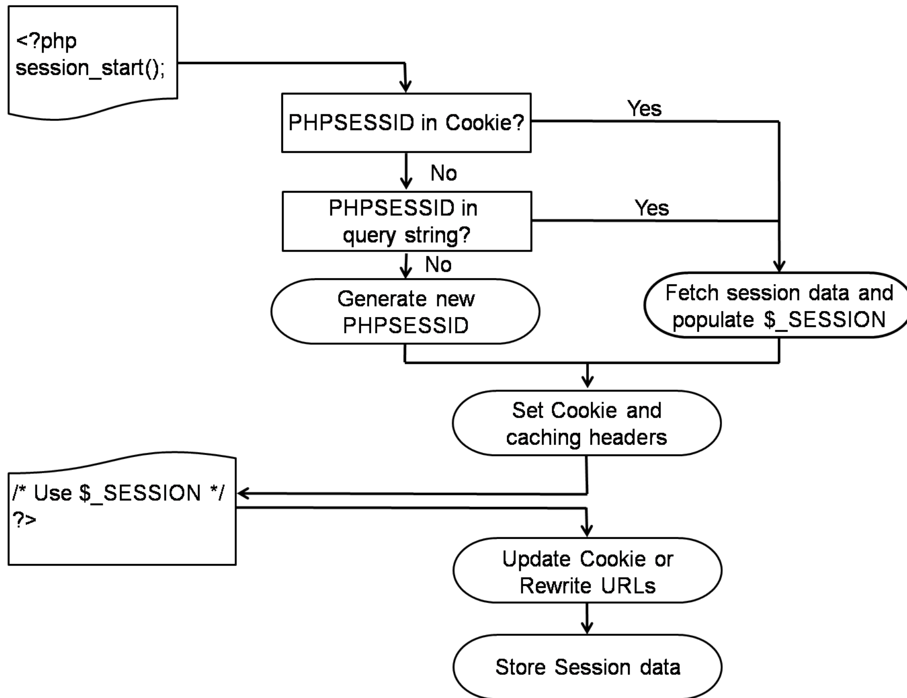


Figure 5.4: Overview of Session Creation and Session Handling [44]

`session_start()` and then set the value of a session variable named `status`. When the user continues to the next page `access.php`, it will reload the value of the variable `status` set in the page `start.php`.

In Listing 5.2, we can see that the function `session_start()` is used in the beginning. By calling this function, the stored session data is loaded into `$_SESSION` for use. If a session is not active already, a new session will begin when a variable is added to the global array `$_SESSION`. Every page that uses the session needs to place the function `session_start()` at the beginning of the HTML page code. Nothing can be sent to the client browser before calling this. Otherwise, the program will cause error. The reason is that headers can be sent to the browser only once. Calling the function `session_start()` or sending text also sends headers to user browser. An alternative approach is to buffer the output in an internal buffer. The output buffer is activated by calling `ob_start()` and no output will be sent from the script until the buffer is flushed using `ob_end_flush()`.[27]

Listing 5.2: Accessing Session Variable `access.php`

```
1 <?php
2 //begining a session
3 session_start();
4 //recalling the session variable
5 echo 'Current status is '.$_SESSION['status'];
6 ?>
```

Changing the value of a session variable is easy. Assigning a new value to the associative array `$_SESSION` using the session variable as key will update the session variable. A variable is created and stored in the session with a view to use it in the future. Before using the session variable it is necessary to check if it already exists. The PHP function `isset()` takes any variable as argument and checks if it has been already assigned a value. With the help of a single session variable and this function we can generate a simple visit counter on a webpage, as given in Listing 5.3.

Listing 5.3: Page Visit Counter `visit.php`

```
1 <?php
2 //begining a session
3 session_start();
4 //checking the session variable 'visit_no'
5 if(!isset($_SESSION['visit_no']))
6     $_SESSION['visit_no'] = 1;
7 else
8     $_SESSION['visit_no']++;
9
10 echo 'Page visited: '.$_SESSION['visit_no'];
11 ?>
```

When the script in Listing 5.3 is run for the first time, it will check if a session variable named `visit_no` is created before or not. Since the script is running for the first time, no session variable is stored yet. Therefore, a new session variable will be created with the value 1. When the page is visited later, it will find that a variable with the name `visit_no` already exists and then it will increment its value by one.

5.3 Session ID

The PHP session module creates a cryptographic session identifier when the function `session_start()` is called for the first time. By default, the session ID is passed to the client in a cookie. If cookies are not enabled on the client machine, the session ID is propagated by embedding into the URL. Additionally, IDs can be propagated by GET or POST requests. Moreover, PHP allows access to the current session identifier with the help of the global constant `SID`.

The PHP produces the session ID by using a hash function, MD5 by default. The MD5 hash function produces 128 bit long hashes. PHP uses hexadecimal representation for the hash output. As a result, the PHP session ID created using MD5 hash function is a 32 bytes long hexadecimal. The hash function used to create the SID in PHP can also be configured to use other hash functions. The `session.hash_function` property of PHP in the configuration file `php.ini` controls the hash function being used to create the SIDs. If the SHA-1 hash function is used, then the SIDs will be 160 bits long.

The function `session_id()` is useful for retrieving the current session ID. It can be also used for generating a new ID. When the function is called without any argument, it will return the session ID of the currently initiated session. If the server needs to create a specified session ID, this function can be called with the value of the new ID as parameter. Using the function `session_start()` to create the session ID when a session is initialized is not enough. The session ID needs to be regenerated whenever there is a change of privilege. PHP provides the `session_regenerate_id()` function to re-create the ID of a current session. Even though a new session ID is created, the session data of the current session is always maintained. This function is useful for avoiding possible session ID interceptions or fixation by malicious users.

5.4 Session Cookie

Cookies are used to store small amount of data on the client browser. They allow to track or identify users across multiple requests and multiple sessions. Cookies are part of the HTTP header. They can be set in PHP using the `setcookie()` function that must be called before any output is sent to the client browser.

The function `setcookie()` can be called with many arguments:
`setcookie(name, value, expire, path, domain, secure, httponly)`.
All the arguments except the `name` are optional. The `name` contains the name

of the cookie and it is used as the identifier of the cookie. The `value` contains the cookie value that needs to be stored and `expire` contains the date when the cookie will expire. When an expiration date is not set, it is treated as session cookie that can be removed once the browser is closed.

Listing 5.4: Setting a Cookie `setCookie.php`

```
1  <?php
2  //checking if the cookie exists
3  if(isset($_COOKIE['ViewTime']))
4      echo 'Recently Visited on: '.
5        $_COOKIE['ViewTime'];
6  else
7      echo 'Have not visited recently.';
8
9      //now setting the cookie for current visit
10     //one month in seconds
11     $month = 24*60*60*30 + time();
12     //setting the cookie
13     setcookie('ViewTime',date("d/m/y G:i"),$month);
14     ?>
```

Listing 5.4 gives an example where a cookie is set to the value of the current time and it expires within one month. The idea is to track the recent visit time of the client and store it in the cookie. Any user re-visiting the page within a month can see the last time he has visited the site. A cookie sent to the web server from the client is automatically loaded into the global array named `$_COOKIE`. The name of the cookie is used as a key to the associative array to retrieve the value of the cookie. In Listing 5.4, we use the `isset()` function to check if the cookie still exists. If the cookie exists, then the date and time of the recent visit is shown.

Listing 5.5: Destroying Cookie

```
1  <?php
2  //previous day in seconds
3  $yesterday = time() - 24*60*60;
4  //setting the cookie
5  setcookie('ViewTime', date("d/m/y G:i"), $yesterday);
6  ?>
```

A cookie can be destroyed by setting the same cookie again with the expiration date fixed as an early date. This way, whenever the cookie is checked next time, it will be expired and is not considered as a valid one. Web applications destroy

the cookie using this method when a user explicitly logs out of the server. The following example in Listing 5.5 shows how we can destroy the cookie we set in Listing 5.4.

5.5 Storing Session Data

Most PHP sessions are cookie based. Generally, when a session is initiated, a cookie is sent to the client browser along with a unique session ID (SID). In order to keep personalized user data on the server, the user specific information needs to be stored somewhere. Depending upon the number of users and size of user specific data, the data can be stored into a file or a database server. By default, the session variables are stored in a local file on the web server corresponding to the unique SID. The session file contains the session information in an array. Whenever a session variable is required, the server retrieves the variables from the file named with the client SID.

Listing 5.6: Viewing All Session Variable `showAll.php`

```
1  <?php
2  //begining a session
3  session_start();
4  //recalling all the session variable
5  foreach($_SESSION as $key => $value){
6      echo 'The value of $_SESSION[\''.
7        $key.'\'] is \''.$value.'\'<br/>';
8  }
9  ?>
```

Changing the value of a session variable is easy. Assigning a new value updates the session variable. PHP allows to store any kind of variable as session data i.e. a simple variable, array, object, etc. The content of the session data is serialized and stored somewhere externally as a binary string after every request. The string is retrieved from the storage and unserialized when the session data is used in the next request. The session data in PHP is not capable of storing any resources e.g.connection handles. In PHP, the session storage for a user is exclusively locked when a script calls the function `session_start()` and the session data is exclusively held by that process until the script closes the session by calling `session_close()` or `session_write_close()`, when the request is finished. When concurrent requests are made for the same session, they are handled sequentially while accessing the session data.[2]

By default PHP stores the session data in the file system. This default file storage for the session data can be changed by altering `session.save_handler` in the `php.ini` configuration file. If the option is set as `mm`, then the session data will be storeds in the memory. Storing the session data in memory increases the performance. Sessions can be stored in a database also. Database storage provides greater scalability and better managebility of sessions. A distributed database can be used to synchronize session data between distributed servers. Another option is to store the session data in a PHP native database named SQLite.[27]

5.6 Destroying Session

Often the session is destroyed when a user quits the application. Although the session data is temporary and it does not always need to be cleared explicitly, in some cases it is necessary to remove the session data. The function `session_destroy()` is used to destroy all the session data that are associated to the current session on the server. However, this function does not unset any global variables tied to the session or the corresponding cookie in the client. In order to clear all the registered session variables, the function `session_unset()` is used. A typical script that can destroy the current session along with all the session variables is listed below:

Listing 5.7: Destroying a Session `endSession.php`

```
1 <?php
2 //begining a session
3 session_start();
4 //free all session variable
5 session_unset();
6 //destroying the session
7 session_destroy();
8 ?>
```

In Listing 5.7 `session_unset()` will free all the session variables that are tied to the current session and a call to `session_destroy()` will reset the entire session.

From security point of view, it is recommended to invalidate the session on the server side as well as on the client side when the user is logged out of the system. In order to do this, the session data entry of the user is deleted from the server session storage and the cookie, residing on the client, is also invalidated.

5.7 Controlling Session Lifetime

Even though the session data are removed from the server by using the function `session_destroy()` when a user logs out, a server can never be sure that the user will always log out. Therefore, it is required to use the PHP built-in garbage collection mechanism that cleans up the unterminated, unused session files from the server. This feature helps to remove the redundant session files from the server and reduces the risk of session exposure. The PHP garbage collection mechanism has two parameters: `session.gc_maxlifetime` and `session.gc_probability`. Both of these parameters can be defined in the configuration file `php.ini`.

The `gc_maxlifetime` parameter defines the period of inactivity in seconds. The PHP engine runs the garbage collection process when a session is initialized and investigates each session. The sessions that have not been accessed for a specified amount of time are removed from the session storage. In a file based session management mechanism, the update time of the session file is used as an indicator of last access. Therefore, PHP needs to modify the update time of the session file when session variables are written or read as well. The other parameter `gc_probability` defines the percentage probability that the garbage collection process will be activated. Since the garbage collection process can increase the processing load on the server with a high numbers of users, a balanced percentage value is set for this property depending on the requirements of the application.

Moreover, the default PHP setting will keep a session cookie active indefinitely until the client browser is closed. This behavior can be changed by altering the value of `session.cookie_lifetime` in the `php.ini` file. The value of this option defines the lifetime of the cookie sent to client. [27]

5.8 Session Storage Security

PHP's native session handling functions can only provide a framework. It is always the responsibility of the application developers to use the provided framework functions properly and to implement a secure and reliable session management. The security of the session data lies on the security of the session ID. By default, PHP creates a file for each session in the temporary directory. But the location of the session storage is configurable. In order to provide security, the used directory and the session files need to be protected with authorized access. It is also possible to create a customized session store for maintaining

the session data.

The PHP interpreter allows the application developer to configure some of the options that control the key aspects of the built-in session management module behavior. The configuration options are presented in brief in following[27] :

- `session.save_handler` specifies the method used by PHP store and retrieve session variables. The default value is `files` that use session files. A useful value for this options is `user` that uses user defined handlers for storing and retrieving session data.
- `session.use:cookies` specifies if the session handling module will use cookies to transmit the SID and add cookies as a possible session tracking mechanism.
- `session.use_only_cookies` specifies cookies as only available session tracking mechanism in the session management framework. When this option is enabled, the session handling mechanism using embedded URLs gets disabled.
- `session.cookie_lifetime` can be used to specify the maximum lifetime of the session cookies that are generated by the server.
- `session.cookie_path` specifies the path to which the cookies are restricted. By default, session cookies are available to all server paths.
- `session.cookie_domain` is used to set the domain of the cookie. By default, the domain is restricted to the host name of the server generating the cookie.
- `session.cookie_secure` makes it possible to add the `secure` option of the cookie to all generated cookies.
- `session.cookie_httponly` specifies the `HttpOnly` option of the cookie for all newly created cookies.
- `session.use_trans_sid` is used to turn on or off the option of transparent rewriting of URLs while using URL-based session IDs.
- `session.hash_fucntion` selects a hash algorithm that is used for creating the session identifiers.
- `session.referrer_check` restricts the creation of sessions to requests that have the HTTP Referer header field set.

The security of the PHP session handling framework depends on the consistent and secure setting of all the above parameters in the PHP interpreter engine.

Chapter 6

Best Practices

While web based session management is important for tracking users, the most critical issue is the ability to bind user authentication and access control to unique sessions and to develop a secure session management mechanism to avoid possible session vulnerabilities. The basic rule of a secure session management is to secure all of its components. The starting point is the security of the authentication methods and user credentials, so that an intruder cannot access any resource using the user credentials. Following a successful authentication, an application formally sets up a session. Generally a session is associated with the user through a session ID that needs to be cryptographically strong and secure to protect against session attacks. The confidentiality and integrity of the session data needs to be provided both on the server and client side. By protecting the user sensitive information against security attacks, it is ensured that no unauthorized entity can read or write a user's data without the permission of the user. Even if the session token or state information is compromised, the application must be able to handle it gracefully i.e. minimize the effects of the attack.

The security of the network connection between the user and the server is another critical issue, and to protect this connection, the authentication information and session data can be transmitted over a secured connection. Moreover, care has to be taken so that an attacker cannot take control of the server and cause the service to be unavailable to its user. It is not a good idea to compose

different security schemes into a new scheme to provide security because more often the composed scheme includes some unnoticed security vulnerabilities due to lack of expertise. The security of the system also should not be dependent on the secrecy of the protocol but only the secrecy of the keys.

It is difficult to implement a secure session management mechanism because there exists no common guideline for it and there is no single solution that suits best in all situations. A system needs to provide the right amount of security according to need because security can be cumbersome for the user and costly for the service provider. Despite several security risks associated with session management, there are some basic simple but effective recommendations to follow that can greatly enhance the security of the session handling method. The following sections discuss the existing best practice recommendations in different areas of the session management mechanism.

6.1 Authentication

Most web applications maintain user specific session data and provide controlled access to sensitive resources. For this reason, the applications require their user to identify and authenticate while accessing the restricted parts of the application. A secure authentication method is the entry point of a secure session management in the application. A secure authentication system along with a mechanism to protect the user credentials can prevent the session from invasion. In a way, a secure authentication can also enhance the security of the session. Web applications can use a TLS connection while transmitting the user credentials from the client to the server. Since the connection will be encrypted in TLS, the credentials will be safe from being tampered or sniffed by an attacker.

When a user is authenticated, the server typically creates a session token or session identifier to track the user and maintain his authenticated status. This session token then acts as a temporary password for the user session. When a web application uses a TLS connection for sensitive resources with authentication, the web application must be designed properly to handle the security tokens. While moving back and forth between secured and unsecured areas of the application, it is important to handle the session in a way which ensures that no security token is transferred over unencrypted HTTP connections.

Another possible way to handle the secured and unsecured area of a web application is to consider the secured and unsecured area as two different services and to maintain two different sessions for them. As a result, when a user switches

from a secured area to a unsecured area, a new session token will be generated. It enables user tracking but does not allow the user to access the sensitive resources. Furthermore, when the user wants to access the secured area, there is a change of privilege and the user can be prompted for his credentials and, thereby, a new session will be created after successful authentication that gives access permission to the restricted resources.

One effective way to provide security is to check the TLS client certificate. The client can acquire a TLS certificate to prove his identity. By presenting the client certificate and responding to an encrypted message, the client is able to prove his identity without even revealing his secret key. This type of authentication is able to mitigate the problem of user credentials being captured by an attacker. However, the private key is a long series of random binary data which is not possible to remember and thus restricts the client's mobility to the devices where the key is stored.

6.2 Token Handling

Generally a secure session management system includes two different types of tokens. Authentication token is maintained to allow authorized users to access protected data of the web application. However, session tokens are used to maintain the state of the authorized users across multiple HTTP requests. Most often the web applications use a single session token to maintain both the authenticated status and state information of the user. Securing the session token prevents the attacker from gaining the access to sensitive resources and personal state information.

However, it is recommended that a system should not rely only on session tokens for accessing the protected data. When both an authentication token and a session token is used in a system, an attacker cannot access the sensitive resources by stealing the session token only. An attacker has to capture the authentication token to get the permissions of the valid user. In this type of systems, if the authentication token is secured against being hijacked, the secured area of the system will be safe. When an attacker tries to access protected data using only the session token, he will be redirected to the authentication page and prompted to provide the credentials.

Since session tokens are used to maintain the authentication status and the user specific information, they can be thought of as a kind of temporary passwords. Therefore, a session token needs to fulfill certain properties to maintain the security level of the application. The key factors of a session token or ses-

sion identifier is its randomness and length. A session identifier needs to be a unique, strong, long, unpredictable, random number. It is better to use a cryptographically strong algorithm seeded with a strong random key to generate the strong long random session ID. When session IDs are generated, they must have sufficient entropy to prevent prediction. It is recommended that application generated session identifiers must have at least 128 bits of entropy. A session identifier whose length is more than 128 bits can be considered safe from being predicted.

When a web based application accepts session IDs that are generated by the client, the application becomes prone to session fixation attack where the attacker can fix a session for a valid user with a chosen session ID. Therefore, it is a good practice to generate the session ID on the server side. The session ID generation technique can be designed in such a way that only the server is able to create a valid session ID. In this way, the application can ensure that no session ID will be accepted as valid if it is generated by some party other than the server. The session tokens presented by the client always need to be validated to ensure that the session token is generated by the server and it is valid at the time of client request. Another requirement of a session token is that the user credentials are not encoded within the session token or identifier. If it is done, then this makes it equivalent to the basic identification scheme.

Since the session token resides on the client end, the user's ability to modify the session data can lead to security vulnerabilities. Therefore, web applications do not want their users to be able to modify their own session data. The confidentiality and integrity of the session data residing on the client side means that no party except the server is able to interpret or manipulate the session data. To accomplish this, an application can digitally sign and encrypt the user session data before passing it to the client with a strong cryptographic algorithm. Alternatively, the server can sign the transmitted session data in order to prevent any tampering on the client side and verify the session data when from the client within requests. Encrypting the session identifier can make the web based system secure to some extent.

When a cookie is used as a session tracking solution, the domain of the cookie needs to be set carefully. It is a good practice not to set the domain of the cookie to the top level but make it as specific and narrow as possible. When a server receives a cookie with a session ID, the server needs to verify the domain before accepting it. Care has to be taken while setting the path of the cookie also. In order to be able to transfer the cookie over a strong encrypted connection, the `secure` attribute of the cookie needs to be set. If this attribute is not set, the cookie will be transmitted over an unsecured channel.

6.3 Session Data

Often session data contains sensitive information of the user. For this reason, securing the session data is a critical issue in both client side state management system and server side state management system.

It is a good practice to avoid storing any protected and sensitive data using client side state management methods. In client side state management methods, the session token resides on the client end, and therefore, the protected data is more prone to tampering or hijacking. It is a better practice to adapt proven strong cryptographic techniques to encrypt and sign the session tokens. Invented new cryptographic techniques may always end up being broken. Even when the session token is encrypted, it gives a malicious user the opportunity for trying to find out the encryption key. If a malicious user is able to figure out the encryption key, then he is able to cause serious damage to the compromised system. Data that is used for unsecured user specific decisions are safer to be stored in the client side. A web application also needs to avoid client side caching of pages with sensitive data.

When server-based sessions are used and most of the session data is stored on the server database or file system, special care has to be taken to protect the data on the server. The session information on the server can be stored in such a way that the attacker cannot acquire enough knowledge to access the session even when the session storage is being compromised. For example, passwords should not be stored as plaintext or even encrypted text in the server storage. It is better to use one-way password hashes with salt. When the hash of the user credentials are saved on the server database, an attacker who has compromised the server database cannot get the information to create a user session.

From the security point view, it is safe for a web application not to rely on the client side e.g. web browser to invalidate the session data or cookies. It is always better to handle this functionality on the server side. Therefore, the server should have a mechanism to destroy the session data and invalidate the user session, when a user chooses to leave the system or the user is inactive for a reasonable amount of time.

In order to avoid any unauthorized use of the session, a web server needs to validate and filter user input coming from all sources. A proper sanity check on all user input can mitigate the probability of any cross site scripting attacks. It is better to avoid performing any validation on the client side.

When the session data is stored in a file system on the server, it is not wise to store the files in a publicly accessible place. The session files needs to be stored

in a restricted place where only the server can access them. With controlled access and restricted storage place, the session data remains safe from being captured by an attacker. A better approach is to store the session data in private temporary file areas in the server for each client of the application.

6.4 Session Lifetime

Session lifetime is an important aspect of secured session management. A session token is generated when a user visits a website for the first time. Then the session is maintained by following one of the session tracking solutions. At some point, the session is terminated either by the client or the server. Unterminated sessions are vulnerable to session attacks. It is always safer to provide a log out option to allow a user to destroy his current session deliberately. Therefore, a server has to destroy the existing session data and session token (both in server side and client side) when the user logs out of the system. Sometimes it is also necessary to revoke a session at the server side under some situations e.g. in case of a compromised session. Therefore, the application needs to provide some mechanism that enables the server to revoke a specified session. Maintaining a table for user specific sessions can be a way to handle this. Another effective way to terminate the sessions of a particular user is to maintain a random salt for each user. By changing the salt of the user whose session has been compromised, the server can invalidate all the active sessions of that user.

It is always recommended to avoid any long term secrets in a secure session management. Session lifetime should always be short to minimize the period of exposure of any compromised session. For this same reason, it not recommended to use persistent cookies for storing any security sensitive session information. Often persistent cookies are stored on the disk that is easier to access for a user than the temporary session cookies residing in the browser memory. It is safe to automatically expire the existing session tokens and to recreate them from time to time. Prevention measures are also necessary to prevent any unauthorized cookie lifetime extension.

Since the server cannot be sure that a user will always log out the system, the server needs to check for inactive sessions at regular intervals. After a maximum period of inactivity, the server can expire the current session and ask the user to re-authenticate. Inactive timeout is a useful way of cleaning up the stale sessions and preventing session exposure attacks. Moreover, despite of a session being active, the server cannot determine whether the session is still being used by a legitimate user. Therefore, an application can implement an absolute timeout that restricts the overall duration of a session and requires user credentials to

regenerate a new session after timeout.

Furthermore, a user should always be re-authenticated before performing any critical and sensitive operation. Whenever an authentication is performed, the server will invalidate the previous active sessions and will create a fresh session ID for a new session. Regenerating the session token after each change in user privileges helps the application to protect against any session fixation attack.

Chapter 7

Conclusion

The most important aspects of a secure session management mechanism are the ability to bind an incoming request to the session it belongs to, to determine where and how the session state can be stored, and to find out measures to protect the mechanisms from attacks. This thesis focuses on analyzing these key aspects of a secure session management in a single server environment. First of all, we have examined the basics of HTTP session management and outlined the various existing options for maintaining application state and handling session data with HTTP session. Then we have covered the possible session attacks and discussed the countermeasures that can be adopted to prevent against session vulnerabilities. In addition, this thesis shows the general steps of implementing a session with PHP and discusses the implications of manipulating some of the session management configuration options on the security level of the application. In the end, we have studied the available best practices for maintaining a secure session.

The stateless behavior of HTTP requires web application developers to use additional methods for maintaining state and user specific session information. Both stateful and stateless mechanisms can be used with HTTP to manage sessions. Most often web applications use sessions to impose security restrictions and encapsulate other state information. Thereby, the security of the session management module of web applications is a critical issue. By accessing the session of a legitimate user, an attacker can obtain the access permissions of

the user and perform any kind of valid operations. For this reason, the tendency of launching attacks on the session management process is increasing day by day. Often the session handling processes of web applications are weak in implementation. Vulnerabilities in the session management process can cause serious damage since they generally maintain important and sensitive data of the web based systems. Additional care and preventive measures are required in existing mechanisms to ensure sufficient level of session security.

Even though there exists many different ways of implementing a session management mechanism, each implementation method of session tracking has its specific benefits and limitations. This thesis analyzes the advantages and disadvantages of each method from a security point of view. Furthermore, handling the sessions securely can be delicate with many session vulnerabilities and little known flaws. A lack of knowledge of the existing session attack possibilities can easily leave the web application exposed to a security compromise. For this reason, we have given a detailed description of the possible session exploitations and some techniques that can be used to prevent invasions and minimize the impact of attacks. Additionally, the use of session handling framework is an important design issue in the complex web applications. PHP developers have access to a powerful and robust session management module through the proper use of PHP's built-in session handling mechanism. We have covered the features of PHP session management that can help to create a reliable and secure web application and to provide a better experience for the application users. Finally, developing a secured session management is not a difficult task if some simple best practices are followed in every aspect of state maintenance. In this thesis we have presented the available best practice recommendations that web applications can implement to ensure that their session handling is secure and their client data is protected. A web application developer in a single server environment should pay attention to creating and storing of session identifiers, transmitting the information between the server and the client, and deleting the session when it is no longer needed either due to user logout or too long idle time.

There is no perfect solution available for secure session management. The level of security required by a web application depends on its intended usage. Moreover, every session tracking solution has its advantages and disadvantages and it is important for a web application developer to understand the various existing solutions of session management as well as their restrictions. Despite of several security risks associated with session management, there exist some simple but effective best practice recommendations that can greatly enhance the security of session management. We hope that this thesis can be considered as an overall guideline for maintaining a secure session in single server environments. However, a strong session management is only one element of a secure web application. To mitigate the various possible session attacks, good web application

design practices are also essential. Several defense-in-depth measures are necessary for securing web applications against session attacks and violations of the system.

7.1 Future Work

In this thesis, the security analysis performed on the different aspects of session management has been totally theoretical, depending on studying the research papers, RFCs and books and examining existing open source PHP applications. As a future work, it would be better if the session attacks, described in chapter 4, can be launched in a live environment in order to investigate which vulnerabilities cause these attacks. Moreover, the stated countermeasures for protecting against session vulnerabilities can also be the subject of further study to verify if they are effective in preventing the session attacks and, thereby, help in improving the security of the session management mechanism.

The discussion of implementing a secured session management process in this thesis is focused on the PHP framework only. It would be better to analyze the security provided by other implementation frameworks as a future work. A comparison between different session management modules can also be useful.

Furthermore, all the best practice recommendations accumulated in this thesis can be used to design a new session management library that will combine all the the security measures stated to prevent against the security vulnerabilities and to avoid security pitfalls in every step of session management. Thereby, a web based system can be developed using this session management module to provide secure session handling.

Bibliography

[1] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 517–524, New York, NY, USA, 2008. ACM.

[2] Gaylord Aulke. PHP is not Java: session management whitepaper. Zend Technologies, November 2007. http://static.zend.com/topics/0200-T-WP-1107-R1-EN-PHP-is-not-Java-Seesions-in-PHP.pdf [referred 25.05.2011].

[3] J.R. Basto Diniz, C.A.G. Ferraz, and H. Melo. An architecture of services for session management and contents adaptation in ubiquitous medical environments. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1353–1357. ACM, 2008.

[4] T. Berners-lee, R. Fielding, and H. Nielsen. RFC 1945: Hypertext transfer protocol-HTTP/1.0. Request for comments, IETF, 1996. http://www.ietf.org/rfc/rfc1945.txt.

[5] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Andrew D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10, May 2007.

[6] Raffaele Bolla, Riccardo Rapuzzi, and Matteo Repetto. Handling mobility over the network. In *Proceedings of the 4th International Conference on Future Internet Technologies*, CFI '09, pages 16–19. ACM, 2009.

[7] M. Cao, T. Xing, and C. Wang. Implementation of web security & identity scheme based on session & online table. In *Computer Science & Education,*

2009. ICCSE'09. 4th International Conference on, pages 1278–1283. IEEE, 2009.

[8] CERT. CERT advisory CA-2000-02 malicious HTML tags embedded in client web requests, February 2000. http://www.cert.org/advisories/CA-2000-02.html [referred 25.05.2011].

[9] M.E. Chalandar, P. Darvish, and A.M. Rahmani. A centralized cookie-based single sign-on in distributed systems. In *ITI 5th International Conference on Information and Communications Technology, 2007 (ICICT 2007)*, pages 163–165, Dec 2007.

[10] Nico L. De Peol. Automated security review of php web applications with static code analysis. Master's thesis, University of Groningen, May 2010.

[11] Pax Dickinson. Top 7 PHP security blunders, December 2005. http://articles.sitepoint.com/article/php-security-blunders [referred 25.05.2011].

[12] T. Dierks and E. Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2. Request for comments, IETF, August 2008. http://tools.ietf.org/html/rfc5246.

[13] D. Endler. Brute-force exploitation of web application session IDs. *iDEFENSE*, 2001. Retrieved from http://www.cgisecurity.com/lib/SessionIDs.pdf [referred 25.05.2011].

[14] EUROSEC. Web application session management, 2007. http://www.secologic.org/downloads/web/070212_Secologic_SessionManagementSecurity.pdf [referred 25.05.2011].

[15] EUROSEC, GmbH, Chiffriertechnik, and Sicherheit. Session management in web applications, 2005. www.eurosec.com [referred 25.05.2011].

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol-HTTP/1.1. Request for comments, IETF, 1999. http://www.ietf.org/rfc/rfc2616.txt.

[17] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004, August 2004.

[18] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617: HTTP authentication: Basic and digest access authenication. Request for comments, IETF, 1999. http://www.ietf.org/rfc/rfc2617.txt.

[19] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*. USENIX Association, 2001.

[20] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, 36:147–158, August 2006.

[21] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance dramatically improved by caching SSL session keys. In *Workshop on Internet Server Performance*, Wisconsin, USA, 1998.

[22] M. Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, page 7, Dec 2002. http://www.acrossecurity.com/papers/session_fixation.pdf [referred 25.05.2011].

[23] Balachander Krishnamurphy, Jeffrey C. Mogul, and David M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the eighth international conference on World Wide Web*, WWW '99, pages 1737–1751, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[24] D.M. Kristol and L. Montulli. RFC 2965: HTTP state management mechanism. Request for comments, IETF, 2000. http://www.ietf.org/rfc/rfc2965.txt.

[25] David Lane and Hugh E. Williams. *Web Database Applications with PHP and MySQL*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2004.

[26] A.X. Liu, J.M. Kovacs, C.T. Huang, and M.G. Gouda. A secure cookie protocol. In *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, pages 333–338. IEEE, 2005.

[27] Nuno Lopes, Mehdi Achour, Friedhelm Betz, Antony Dovgal, Hannes Magnusson, Georg Richter, Damien Seguy, and Jakub VranaBakken. *PHP Manual*. http://php.net/manual/en/index.php [referred 25.05.2011].

[28] Nuno Loureiro. Programming PHP with security in mind. *Linux Journal*, 2002, October 2002.

[29] Oliver Masutti. Distributed web session management. Master's thesis, University of Zurich, Oct 2000. http://www.ifi.uzh.ch/archive/mastertheses/DA_Arbeiten_2000/Masutti_Oliver.pdf [referred 25.05.2011.

[30] Z. Miller, D. Bradley, T. Tannenbaum, and I. Sfiligoi. Flexible session management in a distributed environment. In *Journal of Physics: Conference Series*, volume 219, page 042017. IOP Publishing, 2010. http://iopscience.iop.org/1742-6596/219/4/042017/ [referred 25.05.2011].

[31] S.J. Murdoch. Hardened stateless session cookies. In *Sixteenth International Workshop on Security Protocols*, Cambridge, UK, 2008. http://www.cl.cam.ac.uk/~sjm217/papers/protocols08cookies.pdf [referred 25.05.2011].

[32] L. Murphey. Secure web-based authentication. 2004. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.3893&rep=rep1&type=pdf [referred 25.05.2011].

[33] Luke Murphey. Secure session management: Preventing security voids in web applications, Jan 2005. http://www.sans.org/reading_room/whitepapers/webservers/secure-session-management-preventing-security-voids-web-applications_1594 [referred 25.05.2011].

[34] R. Nixon. *Learning PHP, MySQL, and JavaScript*, chapter 13. O'Reilly Media, 2009.

[35] Gunter Ollmann. Web based session management - best practices in managing HTTP-based client sessions. http://www.technicalinfo.net/papers/WebBasedSessionManagement.html [referred 25.05.2011].

[36] OWASP. The open web application security project. https://www.owasp.org/index.php/ [referred 25.05.2011].

[37] Chris Palmer. Secure session management with cookies for web applications, Sep 2008. http://www.isecpartners.com/files/web-session-management.pdf [referred 25.05.2011].

[38] J.S. Park and R. Sandhu. Secure cookies on the web. *Internet Computing, IEEE*, 4(4):36–44, 2002.

[39] Guy Pujolle, Ahmed Serhrouchni, and Ines Ayadi. Secure session management with cookies. In *Proceedings of the 7th international conference on Information, communications and signal processing*, ICICS'09, pages 689–694, Piscataway, NJ, USA, 2009. IEEE Press.

[40] J. Salowey, H. Zhou, and Tschofenig H. Eronen, P. RFC 5077: Transport layer security (TLS) session resumption without server-side state. Request for comments, IETF, January 2008. http://tools.ietf.org/html/rfc5077.

[41] V. Samar. Single sign-on using cookies for web applications. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99)*, pages 158–163. IEEE, 2002.

[42] G. Schlossnagle. *Advanced PHP programming*, chapter 14. Sams Publishing, Indianapolis, IN, USA, 2003.

[43] C. Shiflett. PHP security. *ApacheCon.* (Las Vegas, USA, 2004) http://shiflett.org/php-security.pdf [referred 25.05.2011].

[44] C. Shiflett. *Essential PHP security.* O'Reilly Media, Inc., 2005.

[45] Pekka Silvekoski. Client side migration of authentication session. Master's thesis, Aalto University, Jan 2010.

[46] D. Sklar and A. Trachtenberg. *PHP Cookbook*, chapter 11. O'Reilly Media, 2006.

[47] SMF. Simple machines forum. http://wiki.simplemachines.org/smf/Main_Page [referred 25.05.2011].

[48] C. Snyder and M. Southwell. *Pro PHP security.* Apress, 2005.

[49] Przemek Sobstel. PHP session security, 2007. http://segfaultlabs.com/files/pdf/php-session-security.pdf [referred 25.05.2011].

[50] H. Song, H. Chu, and S. Kurakake. Browser session preservation and migration. *Poster Session of WWW, Hawai, USA*, pages 7–11, 2002.

[51] Sanna Suoranta, Jani Heikkinen, and Pekka Silvekoski. Authentication session migration. In *NORDSEC 2010, The 15th Nordic Conference on Secure IT Systems*, Espoo, Finland, October 2010.

[52] Codex Documentation Team. Wordpress codex. http://codex.wordpress.org/ [referred 25.05.2011].

[53] Martyn Tovey. Predictable session IDs. NetCraft, January 2003. http://news.netcraft.com/archives/2003/01/01/security_advisory_2001011_predictable_session_ids.html [referred 25.05.2011].

[54] L. Welling and L. Thomson. *PHP and MySQL Web development, Fourth Edition.* Addison-Wesley, 2009.

[55] R. Ye, A. Chan, and F. Zhu. Efficient cookie revocation for web authentication. *IJCSNS*, 7(1):320, 2007.

[56] Chuan Yue, Mengjun Xie, and Haining Wang. An automatic HTTP cookie management system. *Comput. Netw.*, 54:2182–2198, September 2010.