

secologic

Web Application Session Management

secologic Whitepaper

Document History

Version	Date	Editor	Reviewer	Status	Remarks
0.9	2006-05-09	Thomas Apel		Draft	
0.10	2006-10-13	Thomas Apel		Draft	
0.11	2006-11-04		P.Hildenbrand		
0.12	2006-12-20	Thomas Apel	Albert Vetter	Draft	
0.13	2007-01-10	Thomas Apel		Draft	
1.0	2007-01-22	Thomas Apel			

Project codename: secologic

Created by: secologic Consortium

File Name: 070122_Secologic_SessionManagementSecurity_Draft.doc

Copyright 2007, secologic

Table of Content

1 Introduction4

2 Typical Session Attacks5

3 Session Identification Mechanisms6

 3.1 General Considerations6

 3.2 Cookies.....7

 3.3 URL Parameter9

 3.4 URL Path Portion10

 3.5 Hidden Form Fields.....10

4 General Session Management Aspects12

 4.1 Session State Storage.....12

 4.2 Session Timeout12

 4.3 Logging13

5 Session Management in Web Frameworks14

 5.1 SAP NetWeaver Application Server Java.....14

 5.2 SAP NetWeaver Application Server ABAP.....14

 5.3 Microsoft ASP.NET16

 5.4 PHP16

 5.5 Tomcat **Fehler! Textmarke nicht definiert.**

6 Conclusion.....19

7 References.....22

1 Introduction

HTTP is a stateless protocol. So, for a plain HTTP request (i.e. without additional information included), a web application is not able to recognize whether this request is somehow related to previous requests or not. It is not able to relate the request to a session that tells the application, for example, what items are in a user's shopping basket or in what stage of a registration process a user is.

The combination of technical mechanisms for solving this task is usually called session management. The most important questions around these mechanisms are:

- How to identify what session an incoming request belongs to?
- Where and how to store the session state?
- How to protect the mechanisms from attacks?

The most popular mechanism is to generate a unique session identifier and store it in a cookie. This identifier can be used for identification as the web browser will send the identifier cookie along with all subsequent HTTP requests to the issuing web application.

A reason where session management mechanism have to fulfill especially important tasks, is when it comes to authenticated sessions. The usual mechanism is to perform an initial authentication for example via username and password or SSL client certificate and to treat the user as authenticated in subsequent requests as long as the requests appear to belong to the same session as they contain the right session identifier. In such scenarios, weaknesses in the session management mechanisms might allow attackers to impersonate other users.

This paper gives an overview of the standard session management techniques. It discusses the advantages and disadvantages of the different approaches, points out common pitfalls and gives recommendations on how to avoid them [3][6]. The evaluation is mainly based on the following key security requirements for session management mechanisms:

- All information that is sufficient for accessing a session must be protected from theft.
- This information must also be resistant against brute force guessing.
- Session data must be stored in a way that is safe from manipulation.

The paper is outlined as follows: Section 2 gives an overview on typical session attacks. Sections 3 and 4 form the main part of this paper. They discuss mechanisms for session identification and general aspects like logging and session state storage. Section 5 discusses session management implementations in popular web application frameworks. Section 6 offers a short summary and a list of best practice recommendations for implementing safe session management mechanisms.

2 Typical Session Attacks

Typical attacks on web application sessions can be divided roughly into two categories: Those that hijack the whole session and those that trick the client browser into triggering actions within the original session.

Session Hijacking is usually performed using one of the following attack techniques:

- Eavesdropping gives an attacker access to the session identifier. In most cases, the knowledge of the session identifier is sufficient to gain access to the running session. Eavesdropping attacks on the session identifier are especially dangerous if the session identifier also acts as a user authentication credential or is based on a username and password combination. The best protection against all eavesdropping-based attacks is to perform all communication over SSL-encrypted connections.
- With Cross-Site Scripting (XSS) [1][7][13], an attacker injects script code (usually JavaScript) into web content delivered by the web application. This is done by exploiting insufficient output filtering in the application. For injecting the script, there are two different mechanisms. The script can either be permanently stored within the application, for example, within an article in a discussion board. The other method uses volatile injection, typically via URL parameters in specially crafted links that are offered to the victim. Independent of the mechanism, the injected script code is then executed by the browser of the victim, where it performs malicious actions. As an example, the code could access the session identifier and transmit it to the attacker's web server. Currently there is no proven protection mechanism against such attacks besides preventing XSS within the application using appropriate output encoding. An advanced protection mechanism that claims to work even if the application is vulnerable to XSS is proposed by Martin Johns in [4] and [5].
- With Session Fixation [7][8], the attacker tricks the victim's browser and the web application into using a session with an identifier chosen by the attacker. In this case, the attacker does not need to steal the identifier, he already knows it. As a protection measure, web applications should change the external representation of the session identifier on each change of privilege in the session.

Tricking the client browser into triggering actions within a valid session is called Cross-Site Request Forgery (CSRF) [17][14] also known as Session Riding. Such attacks are especially dangerous if a session is authenticated. Such attacks can be performed in two ways:

- The easiest way is to include requests to the target application in another web site. This attack works if the browser automatically sends the session identifier (e.g. in a cookie) with each request. In that case, when the victim visits the attacker's web site, the victim's browser will issue a request with proper session identification credentials to the targeted application. Such a request can then perform arbitrary actions on behalf of the user. The application can defend against such attacks by implementing some kind of challenge response mechanism. For example, it can use random tokens in hidden form fields for requests that trigger important actions. As the attacker does not know this token, he consequently cannot forge a valid request.
- The other method for performing requests on behalf of the victim requires a Cross-Site Scripting vulnerability in the target application. An injected script is used to perform arbitrary requests within the current session.

3 Session Identification Mechanisms

The purpose of session identification is to allow a web application to identify related incoming requests as such.

The following mechanisms for session identification are common:

- Unique identifier in cookie.
- Unique identifier as URL parameter.
- Unique identifier in path portion of URL.
- Unique identifier in hidden form field.

The following subsections describe how these mechanisms work in detail, discuss their advantages and drawbacks, and give recommendations on how to use them safely.

3.1 General Considerations

For all mechanisms chosen session identifier must fulfill certain requirements. It must not only be unique in order to prevent collisions between multiple sessions, it must also prevent guessing of valid identifiers of currently running sessions. This translates to the following two technical requirements a good session identifier must fulfill if it is also used as authentication token:

- It must be cryptographically random.
- It must be long enough (128 bit are a good value).

Mechanisms that perform the mapping between the user's requests and session on the basis of session identifiers are especially susceptible to session hijacking. If an attacker gains knowledge of the session identifier he can perform arbitrary requests from his computer that will be linked to the victim's session.

As an mitigation strategy against attacker that were able to obtain a session identifier, it is recommended to change the identifier on all changes in the privilege level. This is for example when a anonymous session is changed to an authenticated session or if a normal user changes to an administrative permission level by providing authentication credentials.

The following mechanisms are often presented as additional security measures for preventing session hijacking in cases where the attacker gained knowledge of the session identifier. However, these mechanisms can either be manipulated or introduce complications that make them worthless:

- The first mechanism is to bind the session to client's IP address. From a security point of view this is an effective measure as long as the attacker is physically in another network than his victim. In such scenarios it is usually not possible to establish TCP connections with forged IP addresses. If attacker and victim are located within the same physical network, the attacker is in most cases able to overtake the IP address of his victim, which renders IP address bindings useless. Moreover, this mechanism might also reject valid requests as the assumption that all requests from the same client have the same source address is not always true. For example, for clients behind load-balancing proxies, the server will see requests from different source IPs. An additional drawback is that the mechanism is useless when the attacker and

his victim are situated behind the same proxy server. From the web application's point of view, their requests originate from the same IP address. Summarized, IP address bindings cannot be regarded as suitable security mechanism.

- The application could bind the session to HTTP header information like the user agent string or content type accept header. The underlying assumption is that all requests come from the same browser that always sends the same user agent string or content type accept headers. However, the gain is limited such header fields are far from being unique (many users use the same browser) and they can easily be forged by an attacker in a way that it matches the one of his victim. For example, the string could be stolen together with the session identifier. In addition, the user agent header might change within a session. Again, the reason could be load-balancing proxies that add different information to the string.

3.2 Cookies

For most situations, the best way for sending session identifiers along with each page request are cookies. Cookies are small bits of data that a web site can store within the browser. The web application sets the cookie by sending a special HTTP header:

```
Set-Cookie: SessionID="d58f2738ff3dbbfe912c1735bd51e199"; \
    Version="1"; Path="/shop"; Domain="shop.example.com"
```

In the example the server sets a cookie with the name SessionID and the value d58f2738ff3dbbfe912c1735bd51e199. Using the parameters "path" and "domain", the application can define for the own domain where the client should send the cookie to. In this example the browser will send the cookie only when it requests a resource from the server shop.example.com and the directory /shop or one of its subdirectories:

```
GET /shop/index.html HTTP/1.1
Host: shop.example.com
Cookie: SessionID="d58f2738ff3dbbfe912c1735bd51e199"; \
    Version="1"; Path="/shop"; Domain="shop.example.com"
```

There are two forms of cookies: persistent cookies and session cookies. Persistent cookies have the additional parameter "expires" that specifies the date and time when the browser should delete the cookie. Cookies without the "expires" parameter are called session cookies. Persistent cookies are stored on the user's local disk and get deleted when they expire (or by explicit user action), session cookies get deleted when the user closes his browser.

For storing session identifiers, only session cookies should be used. That way the session identifier is removed from the client when the browser is closed, even when the user does not explicitly logout and the application has no chance to unset the cookie. This prevents attack scenarios where the attacker gains to the users cookie store after he closed the browser window. Access to the cookie store might be possible for example on public terminals or when multiple users share a common account.

Storing the session identifier in cookies has the following advantages:

- The browser automatically sends the cookie value with each request to the application. The application does not need to include the identifier in all links or forms like it is necessary for URL or form-based mechanisms. Cookies even work when parts of the application consist of static HTML pages.
- Session identifier changes work over multiple browser windows. When the application changes the session identifier, the new identifier is automatically available to all open browser windows within the same process. For the same reason, that the cur-

rent identifier is automatically sent with each request, there are also no problems that might break the back or reload buttons.

- Normally, cookies do not leave traces in the browser history or proxy logs, even if the connection is not SSL encrypted, since the corresponding HTTP header is usually not logged. However, this depends on the web server configuration.
- Session cookies are automatically purged when the user closes his browser. They do not leave persistent traces on the client machine.

The disadvantages of the cookie mechanism are as follows:

- The fact that the cookie with the session identifier is sent automatically with each request to the application makes this mechanism vulnerable to CSRF attacks from external sites. Malicious sites can include references to the web application for example in an image tag that does not point to an image but to an URL of the attacked web application. The victim's browser will issue this request to the attacked web application. If the user is currently logged on and authenticated to the application, the browser will send the user's valid session identifier along with this request. With the authenticated request, the attacker can perform application actions on behalf of his victim.
- The cookie is vulnerable to session identifier theft via XSS as they can be accessed via JavaScript. However, Microsoft introduced an additional cookie attribute "HttpOnly" that allows servers to set cookies that cannot be accessed via JavaScript. Details of this attribute are explained below in this section.
- Another important disadvantage of session identifiers in cookies is that users might have cookies turned off in their browsers. In such cases the only possibilities are to insist that the user must turn on cookies or to offer a fallback solution like passing the session identifier as URL parameter.

In order to protect session identifiers in cookies against theft through XSS, it is advisable to use Microsoft's "HttpOnly" attribute [12] for such cookies. However, setting this option does not solve all problems around cookie theft through XSS:

- Unfortunately, this proprietary Microsoft feature is not implemented in all browsers. Microsoft introduced this feature with Internet Explorer 6.0 SP1. KDE Konqueror supports it since version 3.2.0. Mozilla Firefox currently does not support this option. For the current state of Firefox's support see Mozilla Bug 178993 for details [11]. The support status for Apple Safari and Opera could not be determined. Browsers that do not support this attribute will not protect the cookie against JavaScript access.
- The second problem is that the "HttpOnly" attribute alone does not solve the problem. An injected script can issue an HTTP trace request. If the server supports the TRACE command, the server will echo the request including the cookie [2]. That way a malicious script can circumvent the restriction of the "HttpOnly" attribute. So, in order to secure the cookie it is important to ensure trace requests do not get answered on the web server and proxy server in addition to setting the "HttpOnly" attribute in the cookie.

A cookie attribute that does not help against XSS attacks but can help against eavesdropping is "secure". It is defined in RFC2109 and RFC2965 [9][10] and is supported at least by current versions of Mozilla Firefox and Microsoft Internet Explorer. When the "secure" attribute is set, the browsers will send the cookie only over SSL connections. So, when the application shall perform its communication exclusively over SSL, it is advisable to set the "secure" attribute for cookies with session identifiers in order to prevent accidental transmission over unencrypted connections.

For protection against CSRF attacks from external sites, the application should only use HTTP post requests for all important actions. The forms should include a hidden field with

a random token. When the server receives the request, it can check whether the received token matches the one that was sent with the form. Attackers that include CSRF attack requests into this external web sites have no access to the form, previously sent to the original user, and the included random token. As a consequence, they cannot forge the request.

In general, cookies are the best way for transmitting session identifiers. It is recommended to use only session cookies and to set the “HttpOnly” attribute. For applications only available via SSL connections, it is also recommended to set the “secure” attribute. Moreover, anti-CSRF measures should be implemented when using cookies. In order to prevent session fixation attacks, the session identifier should be changed on each change of privilege level (especially after an initial authentication step) and the application should not accept any session identifiers in the URL if cookies are used as the only method for transmitting the identifier.

3.3 URL Parameter

Another way to transmit the session identifier along with HTTP requests, is to pass it as an URL parameter. Browsers must pass the session identifier with every request. So, it is necessary that the web application includes the identifier in each link on the pages it delivers. URLs that pass a session identifier might have the following form:

```
http://www.example.com/shop.php?PHPSESSID=d58f2738ff3dbbfe912c1735bd51e199
```

In general, passing the session identifier in the URL has the following advantages:

- The mechanism is immune to CSRF attacks that are leveraged through external websites. Such attacks do not work because the attacker does not know the session identifier and the browser of the potential victim does not send it automatically.
- The mechanism is independent of browser settings. Passing parameters along with URLs is a feature that is always supported by all browsers. That is the reason why URL parameters are a fall-back solution to cookies.

On the other hand, there are also disadvantages:

- The session identifier in the URL is visible. It appears in the HTTP referrer header sent to other websites when the user follows an external link from within the application. It appears in log files of proxy and web servers as well as in the browser history and bookmarks. Moreover, users might copy the URL including the identifier to a mail and send it to others while they are still logged in. This practice is not unusual and will often allow unauthorized access to the application.
- The mechanism is vulnerable to session identifier theft through XSS. The identifier is included in every link within the application. Thus, it is also accessible by injected scripts. Unlike the “HttpOnly” mechanism for cookies, no mechanism exists for restricting script access to links in the current page.
- All links within the web application must include the session identifier. Either the application itself must take care of that or this task must be handled by the application’s framework.
- The mechanism is susceptible to session fixation attacks. For authenticated session a recommended counter-measure is change the identifier during the logon. However, for anonymous sessions, there does not exist any mitigation mechanisms that can be recommended for practical use.

It must be noted that the problem of URL parameters appearing in proxy log files and in the referrer header can be solved by using SSL connections for all requests. When using SSL, the

proxy can only see the encrypted data (as long as it does not act as an SSL terminator). Furthermore, when following links to unencrypted sites, the browser omits the referrer header if the original site was retrieved via SSL.

Note that the session identifiers will also appear in the log files of the destination web server where the application is running. All administrators that have access to these log files while the session is still running are potentially able to overtake such a session. This is relevant in scenarios where there is a separation between application administrator and web server administrator. In scenarios where the complete web server including the application is administered by a single person, this is usually not relevant.

In general, passing the session identifier as an URL parameter is a suitable solution if cookie-based solutions are not feasible. It is common to use the URL parameter approach as fall-back solution for clients that do not support cookies. It is recommended to use SSL, in order to prevent at least that the session identifier appears in proxy logs.

3.4 URL Path Portion

An alternative to URL parameters for transmitting the session identifier is to include it in the path portion of the URL. Such an URL might look as follows:

```
http://www.example.com/d58f2738ff3dbbfe912c1735bd51e199/shop.php
```

The mechanism is very similar to transmitting the session identifier as an URL parameter. Also from the security perspective these mechanisms can be considered basically the same, with one important difference: Application might not expect the path portion of a URL to contain sensitive authorization data. For example the integrated phishing filter of Microsoft Internet Explorer 7.0 transmits all visited URLs to Microsoft for examination, whether the requested page is a known phishing web site.

In order to protect sensitive data, the filter strips all URL parameters from the URL before it is sent to Microsoft over an SSL-encrypted connection. However, session identifiers in the path portion of the URL will not be recognized as sensitive and will not be stripped. The filter will transmit the currently valid session identifier to the Microsoft analysis server. For most applications it must be assumed that a fairly high number of users will have this feature activated, so that session identifier in the path portion of an URL does not appear to be a safe solution. It is recommended to use URL parameters instead.

3.5 Hidden Form Fields

An alternative to putting the session identifier in some parts of the URL is to transmit it in a hidden form field within an HTTP post request.

The mechanism has the following advantages:

- In contrast to get request parameters, hidden form fields are transmitted in the request body and do not appear in proxy logs or referrer headers. Moreover, users cannot accidentally copy them to mails.
- The mechanism is immune to CSRF attacks that are leveraged through external web-sites. Such attacks do not work because the attacker does not know the session identifier and the browser of the potential victim will not send it automatically.

- The mechanism is independent of browser settings. Hidden form fields are always available and unlike cookies, they cannot be turned off by the user. That is the reason they can be used as fall-back solution for post requests when cookies are turned off.

The disadvantages of the mechanism are:

- As the session identifier appears in the HTML page, the mechanism is vulnerable to session identifier theft via XSS.
- The identifier must be explicitly included in each form. So, the mechanism does not work with static pages within the application.

In addition to these security related disadvantages, the most important functional disadvantage is that hidden form fields are restricted to HTTP post requests. The mechanism does not work for HTTP get requests. Unfortunately, all embedded objects like images, frames, and iframes cannot be included with post requests. Resources that are referenced in HTML documents with for example within the tags “img”, “iframe”, etc. are always retrieved by the browser via HTTP get requests. The only alternatives are to include these objects without any session information, which also means that these objects are accessible without authentication, or to resort to mechanisms like session identifier in URL parameters for those request. However, the latter option would unite all the disadvantages of session identifiers in hidden form fields with those of session identifiers in URL parameters.

So using hidden form fields for transmitting the session identifier is limited to those application fields where no session state information is required for performing the requests for embedded objects. This includes in most cases that no authentication checks are possible for such resources.

4 General Session Management Aspects

4.1 Session State Storage

All data that needs to be available to the application across different requests within the same session is called session state or session state data. Examples for such data are shopping basket content, intermediary results of database queries, as well as authorization state information. For storing such data between requests, there are in general two possibilities:

- Sending the state information back to the client. With the next request the current state is transmitted to the server again.
- Keeping the necessary data structures on the server. No session data (except the referencing identifier) is transmitted to the client.

Sending the session state data back to the client is generally not recommended. From a security perspective, the main problem is that the session state can easily be manipulated on the client side if it is not protected appropriately.

In scenarios where session state information must be transmitted back to the client, it is generally possible to sign all transmitted session state information in order to prevent changes on the client side. When the server receives session state information within requests from the client, it can check the signature in order to verify that the data was not altered.

A functional drawback is that the size of the information that can be stored by round-tripping it to the client side is limited through the available bandwidth. For session state data that consists of only a few bytes this does not matter. However, in scenarios with larger session state data structures, re-transmitting them with each request becomes impractical.

Keeping the session state strictly on the server is the recommended solution. Whether the session state is stored in the server RAM or whether it is serialized and written to a persistent storage is irrelevant from a session management security point of view. Independent of the chosen mechanism for storing session state data, it is important to ensure that the session state information is sufficiently protected against illegal access on the server side.

4.2 Session Timeout

All web applications may use two different kinds of session timeouts after which the session gets invalidated: a relative and an absolute timeout.

A relative timeout is used to terminate the session after a certain time of inactivity. The purpose of this timeout is to remove sessions of users that finished using the application, but did not explicitly terminate the session. That way the server can cleanup stale session objects. In addition, it reduces the timeframe where a stolen session identifier can be used for hijacking a session. Relative timeouts are mainly intended to prevent attacks with session identifiers that might be found on public computers when the user forgot to log out.

Furthermore, an application may also implement an absolute timeout that limits the overall duration of a session. This measure prevents that attackers keep hijacked session alive forever by sending dummy requests at regular intervals.

Using both kinds of timeout is recommended. However, it is important to recognize that advanced attacks, like automated XSS for example, happen within fractions of seconds in an automated fashion, so that inactivity timeouts do not help.

How long the exact timeout period is chosen, heavily depends on the application, its typical usage, its environment, and its security requirements. For value considerations, twelve hours for the absolute timeout and one hour for the idle timeout might be a good starting point. The twelve hours allow to keep a session alive for a whole work day. The one hour idle timeout is a good starting point for balancing forced re-logins and security requirements. However, the exact times must be chosen for each system individually and might differ substantially from the given times.

Moreover, note that it is good practice to use a two-step mechanism for implementing timeout mechanisms. In the first step after the first timeout occurred, the application can demand the user to provide his authentication credentials again. If this does not happen, after a second amount of time, the actual session-state will be removed. This allows to lock a users session and prevent misuse, but still allows a user to rejoin his initial session if he provides his authentication credentials again.

4.3 Logging

The following session-management-related events can be logged:

- A session was created.
- A session was terminated (explicit, through timeout).
- A session for the received identifier does not exist.
- The peer's IP address changed within the session.

Logging a session's start and end time might help in post-incident examinations when it is known or suspected that an attack occurred. Knowing start and end times of sessions can relate other events and forming a wider picture of the incident.

The event that the application receives a request with a session identifier for which no valid session exists is security relevant as it might indicate an attack attempt. For example brute force guessing of session identifiers would lead to many error messages about non-existing sessions. On the other hand, this event might also be caused by harmless operations such as sending requests belonging to sessions that are already timed out. Another cause for such messages would be users that access the application through bookmarks that contain a session identifier in the URL. So, not the log events itself, but only an abnormal accumulation can be taken as a sign for a potential attack.

Whether changes in the peer's IP address should cause any worries, depends on the application environment. Such a change might be sign for session hijacking. However, if clients use dynamic IP addresses, changing these addresses during the session can be part of the normal operation. This applies to corporate WLAN networks as well as many hosts that connect to the Internet with dial-up or DSL connections. Another scenario, where such changes are normal, are users behind load balanced proxy servers. For them, the IP address might change with every request. So, overall IP address changes cannot be taken as reliable sign for session hijacking, but nevertheless should be logged as they can offer valuable information when they are correlated to other events.

Summarized, the value of session-management-related logging events is rather limited. Nevertheless, there are events that worth logging, as they can help to reconstruct incident details.

5 Session Management in Web Frameworks

The following sections present the session management implementations of popular frameworks for web applications.

5.1 SAP NetWeaver Application Server

The SAP NetWeaver Application Server provides session handling facilities as part of its framework services. Stateful applications only request this feature but do not have to implement it on their own.

These services also include abilities to store session related information internally with the session object.

There are different kinds of SAP session identifiers used to uniquely identify user sessions residing on either the SAP NetWeaver AS ABAP or JAVA system on a specific application server.

Session identifiers can be represented in the HTTP header as cookies or as part of the URL in order to enable stateful sessions without cookies (URL-rewriting scheme). If cookies are used, the cookies will be created with the domain attribute set to the hostname contained in the client request.

Sessions will only be created if a user does access a stateful application.

Sessions will expire automatically after a configurable period of time. The default timeout value is 30 minutes.

In addition SAP NetWeaver application servers may use additional tokens to support load-balancing. More information about SAP NetWeaver session identifiers can be found in the SAP online help system [15].

5.1.1 SAP NetWeaver Application Server ABAP

On the SAP NetWeaver AS ABAP, the session identifier is called sap-contextid. The sap-contextid is created using the following template:

```
<sap-abap-sid> = URL-encode( SID:ANON:<sap-server-name>
:<internal-id>-<mode> )
<sap-server-name> = <host-name> _<sap-system-id> _<sap-
instance-number>
```

The internal id of the context id is based on a random value and a time value to guarantee the uniqueness of the create session identifier.

The session timeout can be controlled by maintaining the parameter `rdisp/plugin_auto_logout`.

5.1.1.1 Session Identifier as Cookie

If using cookies for session tracking the cookie contains the following:

```
Cookie: sap-contextid=<sap-abap-sid>
```

5.1.1.2 Session Identifier in URL

As opposed to standard J2EE URL-rewriting, the ABAP session IDs, when part of the URL, are represented in a proprietary encoding scheme (SAP URI parameter encoding) basically consisting of a Url64-encoded list of name-value pairs (each value itself URL-encoded) enclosed in parentheses (“()”) embedded between URL-path segments, based on the template shown here:

```
<uri-path-segment> "( <Url64-encoded-name-value-pairs> )"
"/" ...
<Url64-encoded-name-value-pairs> = Url64-Encode( <name-
value-pairs> )
```

where `<name-value-pairs>` is a list of ampersand-separated name-value-pairs analogous to the standard HTTP query parameters (following the question mark (“?”) at the end of the URL) in the form `<name>=URL-Encode(<value>)`. See also RFC 2396; section 3.4 “Query Component”.

```
/sap(bD11bmcmcz1NWVNJRA==)/myapplications/foo/bar
```

Where the Url64-encoded string `bD11bmcmcz1NWVNJRA==` translates to the following list of name-value-pairs (in general, the “value”-part of each pair is URL encoded):

```
l=eng&s=MYSID
```

The name of the name-value-pair denoting the session identifier is `s`. There may be other name-value-pairs present such as `l` (for sap-language). Only single-character names are used in order to save space when the strings are included into the URL.

The format of the session identifier itself is not changed.

5.1.2 SAP NetWeaver Application Server Java

The session identifier on the SAP NetWeaver AS Java is named `JSESSIONID`.

The general format of the `JSESSIONID` is:

```
( <sapservername> ) ID<internal_id>End
<sap-server-name> = <host-name> _<sap-system-id> _<sap-
instance-number>
```

The internal id consists of a time based value and random value and is URL encoded.

Just like the SAP NetWeaver AS ABAP both cookies and URL rewriting may be used. In the case of URL rewriting, the session id will be appended to the URL, like shown below:

```
/test/myapp.html;JSESSIONID=<sessionid>
```

The default session timeout is 30 minutes but can be changed per application by maintaining appropriate values in the `web.xml` of the application.

5.2 Microsoft ASP.NET

Microsoft provides since the release of ASP.NET 1.0 a method for a secure, reliable and comfortable web application session management. Unfortunately ASP.NET contains some other methods for session management which can not be recommended for critical business applications. We give a short overview about the different concepts with an assessment from the viewpoint of security. We start with a remark about disambiguation. In the documentation of ASP.NET the term “application state” is used for information which is available to all users of a web application. “Session state” is used only for information which is available for a specific session. In particular “session state” management differs from “user profile management” and does not require necessarily an authentication.

A detailed and very fruitful resource is the document “Improving Web Application Security: Threats and Countermeasures”¹ in Microsofts Patterns and Practices Series.

5.2.1 The Internal State Object “HttpSessionState”

An instance of the class HttpSessionState provides a secure, server based session state management. An items in the session state is structured as a key value pairs. Some default session variables are provided and managed by ASP.NET. However the programmer can add application-specific session objects, for example the content of a shopping cart or a list of last visited products in the shop. The most basic requirement for a session state variable is that it can be serialized.

The programming syntax for handling session variables is very simple. For example the statement `Session("email") = "foo@bar.com"` (Visual Basic) or `Session["email"] = "foo@bar.com"` (C#) can be used do manage the email address of the user.

Each session is labeled with a 120-bit session-ID which satisfies reasonable security requirements. The default tracking method for this session-ID is the ASP.NET session cookie. Alternatively cookieless sessions are supported but not emphatic recommended in Microsoft’s documentation.

In the Web.config file several global session parameters are controled. For example the code fragment

```
<configuration>
  <system.web>
    <sessionState timeout = "20" / >
  </system.web>
</configuration>
```

sets the session timeout to 20 minutes.

1 Improving Web Application Security: Threats and Countermeasures Roadmap, J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla and Anandha Murukan, Microsoft Corporation, June 2003. Available as download at <http://www.msdn.com> .

The session state can be stored in the memory of the server (`<sessionState mode="InProc">`) or in a database (`<sessionState mode = "SQLServer" ...>`). The InProc version is faster but does not support load balancing and server fail overs.

5.2.2 The Viewstate

The Viewstate of an ASP.NET application is a http hidden field which can contain state information. It is a BASE64 encoded internal representation of state variables. The viewstate is not encrypted and at least in ASP.NET 1.0/1.1 not secured against spoofing. There are obvious security issues with viewstate. Disclosure and spoofing of internal data (for example database connection strings with passwords) are not unusual.

The viewstate can be protected with MACs (message authentication codes) to avoid spoofing or protected with encryption to avoid spoofing and disclosure. Both countermeasure are tricky in web farms due to the difficult key management.

The most important recommendation to viewstate is to stop using it. The page directive „enableViewState=false“ should disable the viewstate, however unfortunately some controls ignore it and use the viewstate on their own.

We strongly recommend to examine the viewstate hidden field of an application in a security test.

5.2.3 Cookies

Cookies a client based management. In the presence of the secure session management mechanismn HttpSessionState we regard the storing of internal session states in cookies in as a serious design flaw. Only the session identifier can be transmitted as a cookie.

A reasonable usage for cookies is the storing of user specific information which exists longer than a single session. For example a localization variable to welcome a user in his favourite language. Those cookies are persistent cookies and not session cookies. Be aware that persistent cookies are relevant for privacy concerns.

5.3 PHP

PHP offers a built-in session handling mechanism [14]. The most important function of PHP's native session handling is `session_start()`. It checks whether a request includes a session identifier. If there is an identifier, it provides all session data in the global array `$_SESSION` to the application. If the request does not include any identifier, PHP generates one and creates a new record for storing the session data.

However, it is important to note that PHP's native session handling functions only provide a framework. It is still the developers responsibility to use the provided framework functions properly in order to create a safe and secure session handling. For example improper usage may result in application vulnerable to session fixation. In order to prevent this, it is recommended to create a new session identifier upon each change of privilege level with the function `session_regenerate_id()`. As an additional counter-measure, it is recommended to disable the acceptance of session identifiers in URLs if this is feasible.

By default PHP uses hexadecimal encoded 128 bit session identifiers and stores the actual session data in the file system. For each session PHP creates a file in a configurable directory.

By default the systems temporary directory is used. It is important to secure the used directory and the session files from authorized access. For one, the session file might contain sensitive data but more important, the name of the session file contains the session identifier. As an alternative, it is possible to hook in a custom session store that interacts with the framework via a predefined interface.

The following PHP interpreter options allow configuring key aspects of the built-in session management framework's behavior:

- `session.use_cookies` specifies whether the session management framework will use cookies as a possible technique for transmitting the session identifier.
- `session.use_only_cookies` allows specifying the session management framework will only use cookies for transmitting the session identifier. If this option is enabled, session identifiers in URLs get disabled.
- `session.cookie_lifetime` allows to specify the maximum lifetime of generated session cookies on the browser. By default session cookies are used for transmitting the session identifier.
- `session.cookie_path` specifies the path to which session identifier cookies are restricted. By default session cookies are not restricted to paths. They are available to all server paths.
- `session.cookie_domain` allows specifying the domain to be set in session identifier cookie. By default the domain is restricted to the host name of the server that generates the cookie.
- `session.cookie_secure` allows adding the cookie option "secure" to all generated session identifier cookies.
- `session.cookie_httponly` allows adding the HttpOnly option to all generated session identifier cookies.
- `session.use_trans_sid` turns the transparent rewriting of URLs for URL-based session identifiers on or off.
- `session.hash_function` allows choosing a hash algorithm for creating the session identifiers. The default choices are MD5 (128 bits) and SHA-1 (160 bits).

The consistent and secure setting of all these parameters for all PHP applications lies in the responsibility of the application developers and the Web server administrator configuring the PHP interpreter engine. Therefore setting up a set of PHP development rules is a good idea.

6 Conclusion

All presented mechanisms have their specific advantages and disadvantages. There is no perfect solution available. Their suitability for a specific application heavily depends on the intended usage scenario.

In general, the following points can be regarded as golden rules for web-based session management:

- The application must not be vulnerable to XSS. All of the presented mechanisms can be circumvented if an attacker finds a way to inject code into the applications web pages. Most of the mechanisms are vulnerable to session hijacking via XSS, and all are vulnerable when it comes to CSRF via XSS. It is important to note that the use of SSL does not provide protection against XSS attacks.
- The use of SSL is strongly recommended. This prevents eavesdroppers from stealing authentication credentials and session identifiers. Moreover, it prevents attackers from manipulating transmitted data on the wire. When using SSL exclusively, cookies should use the “secure” attribute to prevent their transmission over plain HTTP.
- The application should perform important actions only via HTTP post requests. The corresponding HTML form should include random tokens in hidden fields, so that the web application can check the authenticity of the request. This helps to prevent CSRF attacks from external sites.
- The session state information should only be stored on the server within the application framework. This is the best method for preventing manipulations on the client side. If it is necessary to send the session state back to the client, it must be protected against malicious modifications on the client.
- The applications should use long (128 bit is a good value) and cryptographically random session identifiers. If session identifiers are too short or the values are not sufficiently random, an attacker can find valid session identifiers with an exhaustive brute-force search.
- If the session identifier is transmitted as part of the URL, it must be changed on each change in the user’s privilege level. This helps to prevent session fixation attacks.
- Storing session identifiers in cookies is the most appropriate solution for most scenarios. If cookies are not available, the session identifier can be transmitted in URL parameters and hidden form fields.
- The application should always set the “HttpOnly” attribute for cookies and the trace method should be turned off on the web server running the application. This helps to prevent cookie theft via XSS for certain browsers.

The following table summarizes the characteristics of the different mechanisms and related recommendations:

Mechanism / Evaluation criteria	Cookie	URL parameter	URL path portion	Hidden Form Field
Authentication token theft through eavesdropping	Can be prevented with SSL. The attribute “secure” can be set and all plain HTTP requests can be redirected to HTTPS.	Can be prevented with SSL. Plain HTTP requests can be redirected to HTTPS.	Can be prevented with SSL. Plain HTTP requests can be redirected to HTTPS.	Can be prevented with SSL. Plain HTTP requests can be redirected to HTTPS.
Authentication token theft via Cross-Site Scripting	Possible. Can be mitigated with attribute “httponly” and turning off of trace command. Output encoding is nevertheless mandatory.	Possible. Using output encoding against XSS is mandatory.	Possible. Using output encoding against XSS is mandatory.	Possible. Using output encoding against XSS is mandatory.
Session fixation	Sending link with predefined session identifier is not possible if cookies are used exclusively. Manipulating cookies requires more sophisticated attacks. Change identifier on change of privilege level, especially logon.	Is issue. Requires change of session identifier on change of privilege level, especially logon. No practical countermeasures for anonymous sessions.	Is issue. Requires change of session identifier on change of privilege level, especially logon. No practical countermeasures for anonymous sessions.	Sending link with predefined session identifier is not possible if hidden form fields are used exclusively. No realistic attack scenarios.
CSRF (Session Riding) via other web sites	Must be prevented with per-request random tokens.	Not an issue.	Not an issue.	Not an issue.
CSRF (Session Riding) via Cross-Site Scripting	Possible. Using output encoding against XSS is mandatory.	Possible. Using output encoding against XSS is mandatory.	Possible. Using output encoding against XSS is mandatory.	Possible. Using output encoding against XSS is mandatory.
Traces in log files of web server or proxy.	No traces in standard setups.	Traces on web servers and proxies. SSL prevents traces in proxies.	No traces.	No traces.
Unintended disclosure of session identifier due to user activities	Not possible.	Gets disclosed when users publish links.	Gets disclosed when users publish links.	Not possible.
Disclosure of session identifier in referrer header	Not possible.	Is an issue. Can be prevented with SSL.	Is an issue. Can be prevented with SSL.	Not possible.
Traces on client side	Yes, while browser instance is running.	Permanent traces in history, and cache.	Permanent traces in history, and cache.	Post data might be cached within the running browser instance.
Requires URL rewriting in links	No	Yes	Yes	Yes

Mechanism / Evaluation criteria	Cookie	URL parameter	URL path portion	Hidden Form Field
Possible to change SID over multiple windows	Yes	No	No	No
Requirements on client side	Cookies must be supported and turned on.	No special requirements.	No special requirements.	No special requirements.

Table 6-1: Mechanism Characteristics and Recommendation Overview

7 References

- [1] CERT Advisory CA-2000-02 – *Malicious HTML Tags Embedded in Client Web Requests*, February 2000, URL: <http://www.cert.org/advisories/CA-2000-02.html>
- [2] Jeremiah Grossman: *Cross-Site Tracing (XST)*, January 2003, URL: http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf
- [3] Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster: *Dos and Don'ts of Client Authentication on the Web*, Proceedings of the 10th USENIX Security Symposium, August 2001, URL: <http://www.pdos.lcs.mit.edu/papers/webauth:sec10.pdf>
- [4] Martin Johns: "SessionSafe: Implementing XSS Immune Session Handling", European Symposium on Research in Computer Security (ESORICS 2006), Gollmann, D.; Meier, J. & Sabelfeld, A. (ed.), Springer, LNCS 4189, pp. 444-460, 2006, URL: http://www.informatik.uni-hamburg.de/SVS/papers/2006_esorics_SessionSafe.pdf
- [5] Martin Johns, Justus Winter: "RequestRodeo: Client Side Protection against Session Riding" in Proceedings of the OWASP Europe 2006 Conference by Piessens, F. (ed.), Report CW448, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium, 2006, URL: http://www.informatik.uni-hamburg.de/SVS/papers/2006_owasp_RequestRodeo.pdf
- [6] Paul Johnston: *Authentication and Session Management on the Web*, November 2004, URL: http://www.westpoint.ltd.uk/advisories/Paul_Johnston_GSEC.pdf
- [7] Amit Klein: *Cross-Site Scripting Explained*, June 2002, URL: <http://crypto.stanford.edu/cs155/CSS.pdf>
- [8] Mitja Kolšek: *Session Fixation Vulnerability in Web-based Applications*, December 2002, URL: http://www.acrosssecurity.com/papers/session_fixation.pdf
- [9] D. Kristol, L. Montulli: *HTTP State Management Mechanism*, RFC2109, February 1997, URL: <http://www.ietf.org/rfc/rfc2109.txt>
- [10] D. Kristol, L. Montulli: *HTTP State Management Mechanisms*, RFC2965, October 2000, URL: <http://www.ietf.org/rfc/rfc2965.txt>
- [11] Mozilla Bugzilla: *MSIE-extension: HttpOnly cookie attribute for cross-site scripting vulnerability prevention*, Bug 178993, URL: http://bugzilla.mozilla.org/show_bug.cgi?id=178993
- [12] MSDN: *Mitigating Cross-Site Scripting with HTTP-only Cookies*, URL: http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp
- [13] The OWASP Foundation: *Cross Site Scripting (XSS) Flaws*, URL: <http://www.owasp.org/documentation/topten/a4.html>
- [14] PHP Group, *PHP Manual*, section CXLVII Session Handling Functions, URL: <http://www.php.net/manual/en/ref.session.php>
- [15] SAP Library: *SAP Web Dispatcher*, section Session Identifiers, URL: http://help.sap.com/saphelp_nw2004s/helpdata/en/93/33b504f33cb9468bf35f8fdb11294/frameset.htm
- [16] Thomas Schreiber: *Session Riding – A Widespread Vulnerability in Today's Web Applications*, SecureNet GmbH, December 2004, URL: http://www.securenet.de/papers/Session_Riding.pdf

- [17] Peter W: *Cross-Site Request Forgeries*, Butgtraq mailing list, June 2001, URL:
<http://www.tux.org/~peterw/csrf.txt>