Information Security Journal: A Global Perspective, 22:55–67, 2013

ISSN: 1939-3555 print / 1939-3547 online
DOI: 10.1080/19393555.2013.783952

Taylor & Francis
Taylor & Francis Group

An Analytical Study of Web Application Session Management Mechanisms and HTTP Session Hijacking Attacks

Shellie Wedman, Annette Tetmeyer, and Hossein Saiedian
Department of Electrical Engineering Computer Science, University of Kansas, Lawrence, Kansas, USA

ABSTRACT The HTTP protocol is designed for stateless transactions, but many Web applications require a session to be maintained between a Web browser and a server creating a stateful environment. Each Web application decides how its session is managed and needs to be able to trust the session identifier. However, it is possible for sessions to be hijacked, and an intruder can gain unauthorized access to the hijacked session. The purpose of this paper is to provide an analysis of current session management mechanisms and examine various hijacking techniques. The primary issues that will be addressed pertain to session management and the importance of securing the creation, deletion, and transmission of a session token. We provide a broader view of the session hijacking threat environment by analyzing existing Web application implementations to help demonstrate the need for session hijacking prevention. We will identify the session management areas that are targeted by attackers and will identify and examine various attacks that can lead to a session being hijacked.

KEYWORDS internet security, session management, session hijacking, HTTP programming, web browser vulnerability

1. INTRODUCTION

The threat of hijacked Hypertext Transfer Protocol (HTTP) sessions is a serious problem for Web applications. Three security risks are identified in the Open Web Application Security Project (OWASP Foundation, 2010b) Top 10 application security risks for 2010 that can lead to a session being hijacked:

- Broken authentication and session management (A3),
- Cross-site scripting (A2), and
- Cross-site request forgery (A5).

OWASP provides a risk-ranking methodology for each of the Top 10 application security risks, which include likelihood factors for attack vectors, weakness prevalence, weakness detectability, and a technical impact factor. Figure 1 is the

Address correspondence to Hossein Saiedian, Department of EECS, School of Engineering, University of Kansas, Lawrence, KS 66045, USA. E-mail: saiedian@eecs.ku.edu

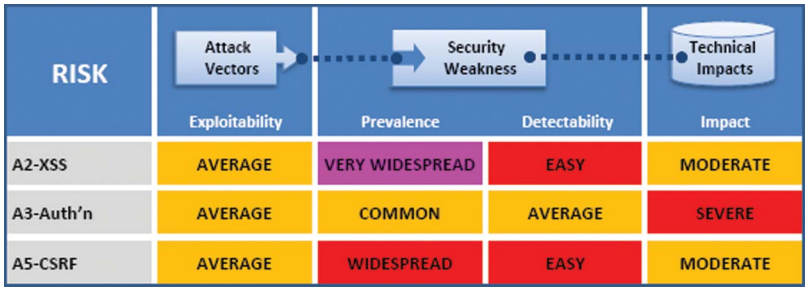

FIGURE 1 **OWASP Top 10 application security risk factor summary for 2010 (OWASP Foundation, 2010b) (color figure available online).**

risk factor summary for the security risks as determined by OWASP (2010b).

Broken authentication and session management ranks among the worst technical impact with a severe rating. Cross-site scripting and cross-site request forgery are both ranked the highest in security weakness prevalence (very widespread and widespread, respectively) but with moderate technical impact. All three of these risks can lead to a session being hijacked.

Vamosi (2009) describes several malware examples that have exploited a banking system's session management. One example given is the botnet Clampi; it waits for a user to log in to his or her bank account. After a valid session has been created, Clampi takes over the session by impersonating the legitimate user. The user has no knowledge that this attack has taken place. Another example (Vamosi) is the Trojan horse URLZone; it allows the attacker to hijack sessions by spoofing bank pages. Both are real life examples of a sophisticated hijacking for financial gain.

There are two types of sessions within the context of a Web application: browser (client) session and HTTP session. A browser session is managed by the client's computer. It begins when the browser is opened and terminates when the browser is closed (Shklar & Rosen, 2009). In contrast, the HTTP session is managed by the server, not the client or browser, and it provides a way to link the browser session to the server session (Shklar & Rosen). A Web application's HTTP session is "established to recognize requests that belong together and to associate these requests with (session) data stored at the server" (Jovanovic et al., 2006, p. 2). Both sessions described above are required because HTTP is a "stateless" protocol; this means that "once an initial communication exchange between a client and a server is completed, the connection between them is dropped" (Berghel, 2002). Since HTTP does not support sessions, the Web application must devise a way to implement and manage sessions.

1.1. Objectives

The main objective of this work is to provide a survey of session management threat environment. This survey will collect and examine the known problems and solutions to such problems. To complete the survey, the following issues will be addressed:

- What areas of session management are targeted in a session hijacking attack?
- What are the different types of attacks currently defined that lead to session hijacking?
- What are the mechanisms currently available to developers to secure against this type of attack?
- How are various Websites currently implementing session management? Provide a brief analysis to determine if Websites are insecure and show the threats involved in session management.

The next section reviews and analyzes session management. It is important to understand the mechanisms involved in session management before examining different hijacking attacks.

2. SESSION MANAGEMENT

Belani (2004) describes session management as the "techniques employed by Web applications to transparently authorize a user for every HTTP request without having the user repeatedly login." Essentially, the session has "the same significance as the user's original credentials" (Jovanovic et al., 2006, p. 2). These session management techniques are exploited by attackers to trick the server into

performing functions using hijacked credentials. The three areas within the session management process that will be analyzed are:

- Session token creation and deletion,
- Session token transmission, and
- Session token verification.

Each of these areas will be reviewed below.

2.1. Session Token Creation and Deletion

HTTP session token creation and deletion are key elements to determining the hijacking window. The goal is to minimize this attack window period. The window can be viewed to be as long as the user is actively and inactively using the Website. Another way to state this is:

– SD − SC = Aw
– SD = Session Deletion/Invalidation Time
– SC = Session Creation time
– Aw = Hijacking Attack Window

Session token deletion is the act of invalidating a token. Deletion occurs by the browser and the server. Actions that could lead to an HTTP session being destroyed are:

- Server session inactive timeout event,
- User logging out of Website,
- Technical difficulties which result is a session disconnect, and
- Browser session closed (this only invalidates the browser side of the session; the server session might still be valid which will be addressed later in the paper).

HTTP session token creation is the act of assigning a token to an authenticated user. This is performed by the server. Examples of when a session is created are:

- First visit to site,
- Explicitly logging into Website, and
- Entering a more secure area of a Website.

An example of the session inactive timeout being implemented with a short interval was found at a banking Website. The banking session timed out after 10 minutes of inactivity, and the following message was displayed for the user:

> As part of our strict commitment to online security, we automatically terminate your secure online session after an extended period of inactivity. This prevents unauthorized users from accessing your account information. It's just one of the many ways we strive to protect you and your personal information online.

The session cookie was still present on the user's computer, even after server session timeout. The session cookie is not recognized by the server; therefore, the user must be re-authenticated before accessing his or her account again. This indicates that specific HTTP session token is inactive on the server and therefore cannot be used in a hijacking attack. Table 1 offers a comparison between Website category and server timeout period. This timeout period is for inactivity only. The Websites that make financial transactions have a smaller session inactive period than sites that only store personal information.

After analyzing how Websites implement the log-out functionality, some sessions are invalidated only on the server, but the session cookies are not deleted from the browser. The session cookie will still be submitted with each page request after logging out, but the server determines that it is invalid and does not link session data to the ID. This enforces the point that the browser session is not implicitly linked to the server session. Figures 2 and 3 demonstrate this finding using a low-level security Website. Figure 2 is the JSESSIONID cookie while logged into the Website, and Figure 3 is the JSESSIONID cookie after logging out of Website. The values of both are the same, and

TABLE 1 **A comparison of website categories and timeout periods**

Website category	Connection type	Timeout period
Banking – account summary	https	10 minutes
E-commerce	https	30 minutes
Social networking site (logged in)	http	Long – specific time unknown
University (student account)	https	30 minutes
E-commerce checkout	https	10 minutes
Email account (logged in)	https	Long – specific time unknown
Parental control Website	https	Over 2 hours – specific time unknown

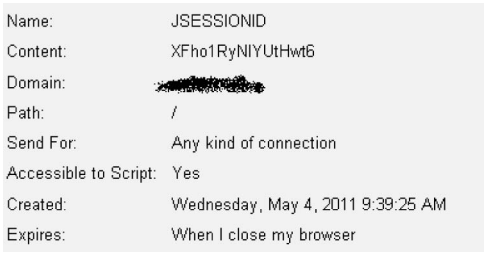

FIGURE 2 **JSESSIONID cookie after logging into Website.**

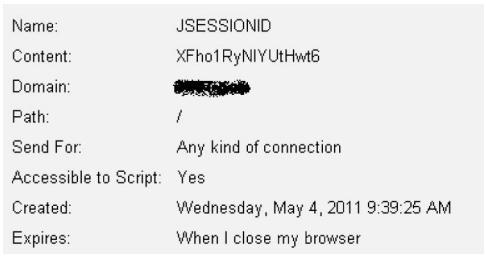

FIGURE 3 **JSESSIONID cookie after logging out of Website.**

the cookie was not deleted from browser even though it is invalid on the server.

The browser invalidates cookies based on the max-age attribute. When this cookie attribute is set to −1 it indicates a browser session cookie. This cookie will be deleted once the browser session is closed. Internet Explorer 8 shares a session between tabs and browser instances (Tulloch, Northrup, & Honeycutt, 2010). It is important to note that the browser session is not closed until all instances of the browser are closed, not just the tab running the Web application. That is, the session cookie stays active when the tab is closed if the browser application is still running. The Web application can be re-opened and the session will still be valid; the user will still be logged in. Another important note is that the browser session does not implicitly communicate with the server session and vice versa: the server is not notified when a browser session is closed, and the browser session is not notified when the server session is closed. This is important to note because the browser session might appear closed to the user because he or she is asked to log in again to access the site, but the previous HTTP session can still be hijacked because the server considers the previous session token active.

2.2. Session Token Transmission

The second area of session management to analyze is session token transmission. How tokens are implemented affects the way they are transmitted. Visaggio and Blasio (2010) state two of the main mechanisms for implementing session tokens are cookies and URL rewriting.

The first mechanism mentioned is the use of cookies. Vinod et al. (2008) describe some basic best practices for developers to follow when using cookies for storing the session token (Table 2). Cookies are transmitted via the HTML response and request headers in name/value pairs (Shklar & Rosen, 2009). The browser stores the transmitted cookies and includes them in all appropriate server requests. The browser determines which requests are appropriate by matching the requesting server domain and URL path to the stored cookies domain and URL (Shklar & Rosen).

The second mechanism for implementing session tokens is URL rewriting. URL rewriting method takes place when the server appends the session token to the end of all HTML links. The following URL example demonstrates URL rewriting:

http://www.xxx.com/.../search/newresults.jhtml;jsessionid=HUQ43LOI4TRS2CSGBIYMVCQ?searchType=city

The jsessionid is appended to all URL's before the request parameters. If cookies are disabled in the browser, the application server normally switches to URL rewriting to maintain the session (Fain, 2011). After trying various Websites, the majority gracefully require cookies to be enabled, not supporting URL rewriting. However, some of the Websites analyzed did not handle disabled cookies gracefully and continued to function but in an unstable way. Supporting URL rewriting functionality in a Website could lead to session hijacking if the user decides to copy and paste the URL into an email. For example, if the user posts the previously stated URL that includes the jsessionid, then an attacker can easily click on the posted URL and join the user's session. This is assuming that the Web application has no other security mechanisms in place to validate sessions. Furthermore, the browser history, cache, and bookmarks use the URL string to revisit Web pages. If the session token is included in the URL, it would also be saved. When using a public computer, anyone using the computer could potentially gain access to that session. Beginning with version 8, Internet Explorer offers a new option of *InPrivate* browsing that inhibits browsers from capturing and saving sensitive information (Tulloch, Northrup, & Honeycutt, 2010). This relies on the user to configure the browser to run in this mode. It does not help the Web application in determining if a session has been hijacked.

TABLE 2 **Best practices when using cookies (Vinod et al., 2008)**

Best practice for cookies	How it works	Why it helps
Use SSL protocol	Secures the transmission by encrypting data.	If a request packet is captured and SSL is used, then the data will be encrypted. If not it will be in clear text. Figure 4 is an example of a captured packet not using SSL.
Use nonpersistent (session) cookies	The *max-age* attribute on a cookie is used to create a nonpersistent cookie. When this attribute is omitted during cookie creation, then the browser will delete the cookie upon session termination (Tappenden & Miller, 2008).	If the cookie is persistent and has a long *max-age* period, then the session stays valid for a longer period and gives an attacker a larger window to guess a valid session token.
Use secure cookies	The *secure* attribute for a cookie indicates to the browser that the cookie should only be transmitted using a "secure" channel such as HTTPS (Tappenden & Miller, 2008).	Creating a *secure* cookie will stop an attacker from capturing an HTTP packet and being able to view the session token in clear text.
Use HTTP Only cookies	The http only attribute of a cookie is used by the browser to determine if the *document* object within any embedded JavaScript should be granted access to this cookie and its associated values (Tappenden & Miller, 2008).	The http only attribute will be discussed in the cross-site scripting section. It protects the cookies from being accessed by scripts running on the WEb page.

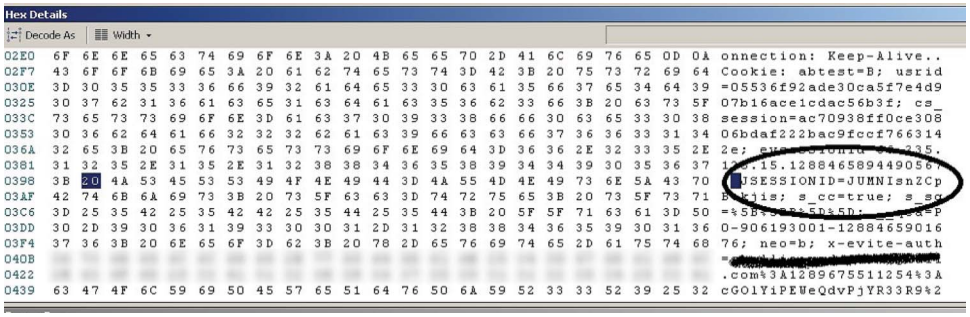

FIGURE 4 **A screen shot of a captured packet that displays the jsessionid in clear text for a Website (color figure available online).**

2.3. Session Token Verification

The third vulnerable area within session management is the verification of the session token sent by the browser to the server. This verification is done by the Web server container. Within the application server Tomcat 6, "statefulness will only work if the client returns that identifier back to the server, for each request that is part of that conversation. When each new request is received, a server attempts to locate this identifier, and use it to look up the conversational state associated with this client" (Chetty, 2009). Java, for example, has an implicit session object available to the controller code. This object is created for each request and is automatically populated with the session data that is tied to the session ID in the request.

Additional security mechanisms found on an e-commerce site that protects against a session being hijacked is requiring the person to re-login before performing higher level security functions such as checking out of a store. E-commerce sites have added the functionality of saving credit card information to make it easier for the user to purchase items, but this creates potential problems if a session is hijacked. One Website combats this issue by requiring the credit card number to be re-entered if the shipping address for the order is new to the account. Figure 5 displays this security policy as presented to the user at the time of purchase.

Another issue to address is the "Remember Me" feature. Figure 6 displays this functionality as you would see it on a Website. The Remember Me functionality extends the

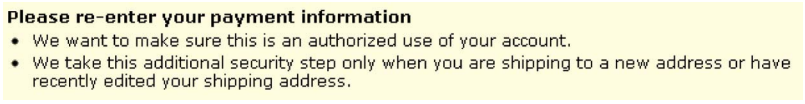

FIGURE 5 Security message display on an e-commerce site at time of purchase (color figure available online).

FIGURE 6 Keep me logged in functionality of a Website (color figure available online).

user's session for a longer period of time. The Website that offers this functionality displayed in Figure 6 actually sets the cookie value to be valid until 2021. Figure 7 displays the cookie values after logging in not using this feature. The cookie will expire when the entire browser is closed and the user will be required to log back in on next visit. Figure 8 displays the cookie values after logging in using this feature. The cookie will expire after 10 years. The user will not need to log in on their next visit to this site when using the same computer.

One way this area can be exploited is if the user's computer is physically accessible to an attacker such as at a workplace or using a public computer. This functionality enables the session to stay active even when the browser is closed. This functionality also leaves the server's session object valid for a much longer period of time. Websites that require a higher level of security usually do not implement the Remember Me feature (Mitchell, 2010). When inspecting a variety of Websites that would be categorized as requiring lower levels of security, most do offer this feature.

The next section examines specific hijacking attacks and analyzes the session management areas they exploit.

3. SESSION HIJACKING ATTACKS

Session hijacking attacks exploit the session management process in a way that an attacker avoids the authorization and verification of a Web application by impersonating a valid and verified user. This happens when an attacker captures a user's session token. There are various ways an attacker is able to achieve this. This section reviews a variety of different hijacking attack types and ways to protect against them.

3.1. Session Prediction

Session prediction is when an attacker is able to predict a valid session token. This exploits the first session management area of session token creation. When a session token is created in a nonrandom way, an attacker can analyze the Web application's cookies, giving the attacker a greater chance of predicting valid sessions. Figure 9 demonstrates a predictable session cookie. The BFSESS cookie has a value of 'Y.' This indicates that there is a session for the user, but there is no session ID cookie created.

The Website that generated the BFSESS cookie also stores the user name and password in another cookie. There are two concerns with this approach to session management: session prediction and storing of unencrypted

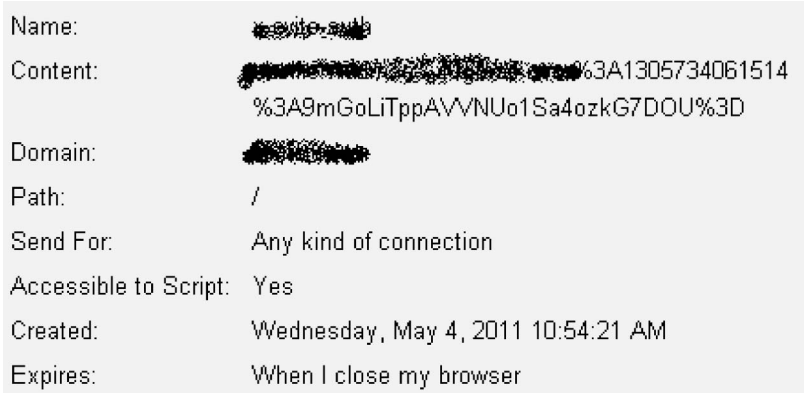

FIGURE 7 Screen shot of the Website cookie focusing on the "Expires" date when the Remember Me feature is not used during login.

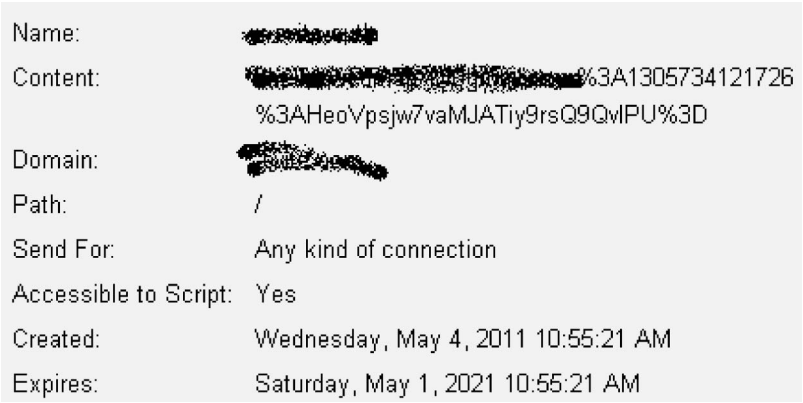

FIGURE 8 **Screen shot of the Website cookie focusing on the "Expires" date when the Remember Me feature is used during login.**

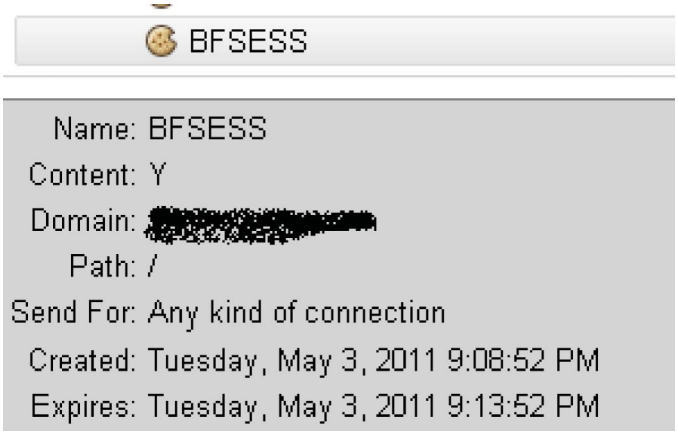

FIGURE 9 **Session cookie with the value of 'Y.' (color figure available online).**

passwords in cookies. Vinod et al. (2008) secure coding standards state that Web applications need to ensure that random numbers or numbers that are not easily guessed should be used as the session token. Using JBoss as an example, it allows for configurable session creation algorithms, but the default is MD5. SessionID length is configurable, but the default value is 16 characters. Using the default implementation should provide a strong enough sessionID that prevents session prediction. Using ASP.NET, for example, a valid session ID is the only data that need to be captured and forged by the attacker. If the attacker can predict or capture a valid session ID, he or she can use that session state (Esposito, 2004).

3.2. Session Fixation

Session fixation attacks happen when tokens are not regenerated between an HTTP and HTTPS transmission. Attackers are able to obtain the session token by intercepting a non-secure HTTP transmission (Visaggio & Blasio, 2010). Figure 10 demonstrates how a basic session fixation attack occurs. A new session token is created when a user first visits a Web application. After logging into the Web application, the session token stays the same. Therefore, the attacker could capture the session token, JSESSIONID, from the less protected area of the Web application.

When URL rewriting is being used for session management, it is easier for a user to give away a session

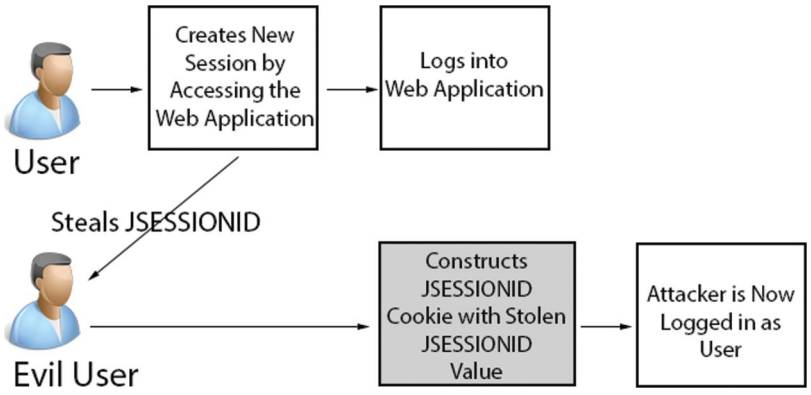

FIGURE 10 **Diagram of how a session fixation attack occurs (Mularien, 2010) (color figure available online).**

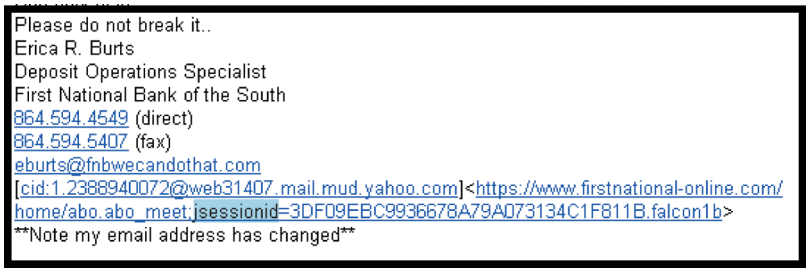

FIGURE 11 **Section of an email message with a link containing the jsessionid (color figure available online).**

ID without realizing it. The scenario was described earlier of when a user copies a URL into a message board. If URL rewriting is implemented, then the session ID is in plain view for an attacker to use. This can lead to a bigger issue if a session token is created in a nonsecure page and not regenerated upon login. The user might think he or she is sharing basic information about a product and then later log in to a personal account. If the session token is not regenerated, the attacker has hijacked the session and has access to sensitive personal information. An alternate way a fixation attack could occur is if the attacker posts or emails a URL with the session identifier appended. Figure 11 is from an actual spam email that has the jsessionid appended to the URL. The user, upon clicking on a link with the session ID appended, would potentially start using the attacker's session. The user is tricked into using the attacker's session. This specific version of a fixation attack could be stopped if the Web application checked more information in the request header. For instance, the origin or host variables will differ from the attacker's, and the Web application could detect the hijack. Another countermeasure identified by Mularien (2010) and OWASP (2011a) for session fixation is to simply regenerate session tokens after login or when switching from http to https and vice versa.

Based on an informal survey of a variety of sites, most do not regenerate session IDs. The only Website that did regenerate the session ID was a banking Website. The session cookie was generated during login, and it was configured to only be transmitted on encrypted connections. The five e-commerce Websites analyzed appeared to be generating a session ID upon first visit and not regenerating it after login. The other Websites analyzed do not fall into these two categories but do have a login and seem to not regenerate session IDs after login. These results are based on cookie analysis only. Figures 12 and 13 demonstrate this issue with one of the Websites analyzed. Figure 12 displays the cookie that is created by the Website before logging in. All pages are displayed using

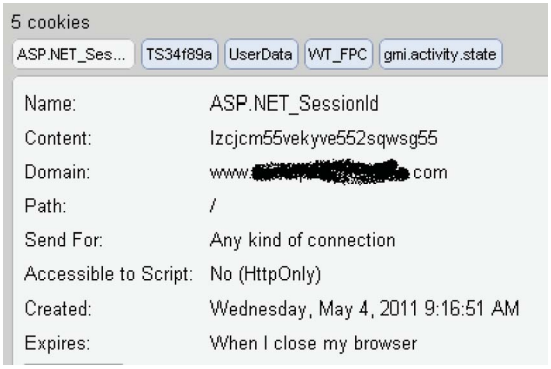

FIGURE 12 **ASP.NET_SessionId cookie created when browsing Website not logged in (color figure available online).**

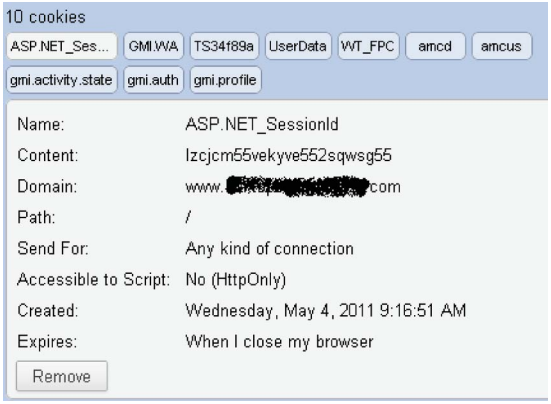

FIGURE 13 **ASP.NET_SessionId cookie after logging into Website (color figure available online).**

the HTTP protocol. Figure 13 displays the same cookie after logging into the Website. The login page uses HTTPS and, after logging in, the subsequent pages use HTTP. The ASP.NET_SessionId cookie value is the same for both cookies; therefore, this session ID was not regenerated going from HTTP to HTTPS.

3.3. Cross-Site Scripting Attacks

Cross-site scripting (XSS) attacks happen when an attacker is able to embed malicious scripts on a Web page. XSS is a potential problem for any Website that displays

untrusted user input. CERT (2000) states that Website developers can prevent these types of attacks from occurring by "filtering and encoding Web page input." This means that all user input should be encoded and filtered to not allow for embedded scripts to be displayed and executed on a Web page or stored in a database for later display. There are well-tested filtering functions already available to programmers. Enterprise Security API (ESAPI) is a Web application security control library offered by OWASP. The ESAPI framework includes default implementations of input validation controls. There are three different types of XSS attacks defined by OWASP (2010a):

- Stored,
- Reflected, and
- DOM injected.

Each type will be described in the following subsections.

Store XSS

Stored XSS is when the malicious script is entered into a form and saved into the Web application's database. This malicious script is then potentially displayed for an administrator or general site user. This can only happen if the input is not filtered and the output is not sanitized. The response Web page presented to the user would then execute the java script in the same domain as the Web page; that is, the malicious script is able to avoid restrictions in the same origin policy. The same origin policy is the core security policy in place for browser-side programming languages such as JavaScript. It limits scripts from accessing information from other domains (White, 2009). The script must be embedded into the Website to avoid this restriction. The user has no knowledge or warning that a script is running because the browser thinks the script is legitimate. With an embedded script, the attacker can easily obtain the user's cookies for that domain by calling the document.cookie function. Once the script has the cookies, it can transmit them to the attacker. If the session token is saved as a cookie, the attacker has captured the session and is able to perform a hijacking attack. The damage caused by this attack dramatically increases if an administrator is the one who runs the malicious java script. The httponly cookie attribute previously mentioned was defined by Microsoft as a deterrent to XSS attacks (Tappenden & Miller, 2008). Figure 11 is an example of a cookie with this attribute set. This attribute tells the client container that the cookie cannot be accessed through a script. IE8 incorporates a XSS filter, but it initially caused more vulnerabilities. Microsoft Security Bulletin MS10-002 said that there were seven vulnerabilities reported and the most critical ones had to deal with XSS attacks (Microsoft, 2010).

Reflected XSS

A reflected XSS attack is when the malicious script is reflected off a web server and sent to a victim. If a Web application displays part of the input in an error message, a malicious script could be reflected back to a victim. The browser would run the script without the victim knowing it. Reflected attacks can be sent to a victim through email or another Web page (OWASP, 2010a).

DOM Injected XSS

A Document Object Model (DOM) based attack occurs when the embedded script modifies the client's DOM environment. The DOM defines the logical structure of Web page documents. Any HTML or XML elements can be defined in the DOM. A DOM attack can add, delete, or modify elements in the DOM tree on the client side.

3.4. Cross-Site Request Forgery

Cross-site request forgery (XSRF) attacks happen when the user's browser session is exploited and a Web transaction is forged (Kerschbaum, 2007). Jovanovic et al. (2006) explains a XSRF attack as exploiting the trust that a Web application has with an authenticated user. Figure 14 helps to explain this attack. The user visits the attacker's Website or Web page that has the embedded malicious script while

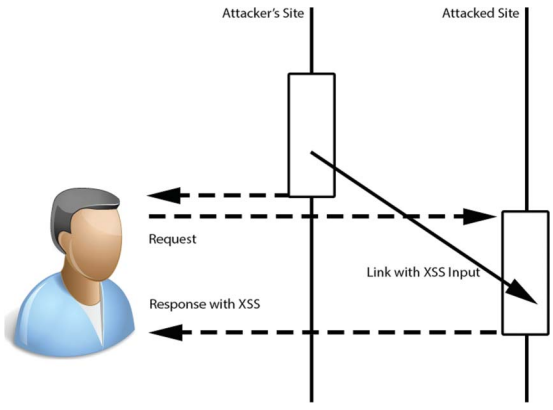

FIGURE 14 **Kerschbaum (2007) offers this depiction of interactions during XSRF attack (color figure available online).**

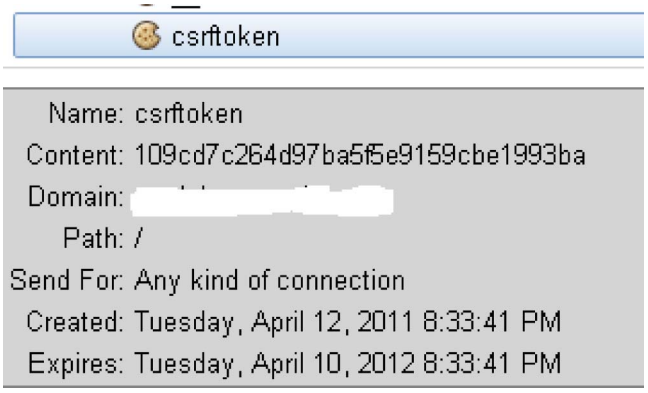

FIGURE 15 **Screen shot of a Website's csrftoken cookie (color figure available online).**

also having a session open with the attacked site. The cross-site request forgery attack takes place using the user's established session to forge a transaction without the user's knowledge. The attacked site thinks the request was sent by the user, but it was actually sent by the script from the attacker's Web site or a XSS embedded script.

Upon analysis, some Websites also create a csrftoken cookie (Figure 15 is a screen shot of a csrftoken cookie). This cookie is valid for one year, so when logging out and re-logging in the csrftoken remains the same. This creates a large window for an attacker to discover the value. The token is created and used in a double submit cookies technique to prevent CSRF attacks, which is a form of synchronized token pattern. OWASP (2011b) describes the double submit cookie technique as sending the csrftoken two different ways when submitting forms on a Website: the token is required to be sent in the form as a hidden value and as a cookie, and the values must match or it is a possible CSRF attack. Since a CSRF attack is unable to view the cookies from a Website because of the same origin policy, it is practically impossible for the attacker to submit the form with the csrftoken value set to match the cookie. OWASP (2011b) states that including authenticated session identifiers in HTTP parameters may increase the overall risk of session hijacking, but developers are still encouraged to use a synchronizer token pattern to help prevent CSRF attacks. OWASP has a CSRFGuard project which is a java library that implements a variant of the synchronizer token pattern. This guard performs the csrftoken check to ensure the form token matches the session csrftoken. If the tokens do not match, the request may be denied access (OWASP, 2011c).

Kerschbaum (2007) offers another solution to XSS. This solution forces all Web activity to originate at an entry page for a Website. All other pages are referred to as regular pages. Once a user enters the Website, a session is created. The regular pages ensure that the session is valid and that the referrer header is set to their Web application. If the referrer header does not match, the user is redirected to an entry page to reestablish authenticity. The assumption for this technique is that the referrer header cannot be forged. Assumptions like this present a weakness in the strategy.

3.5. Man in the Middle

A man-in-the-middle (MITM) attack occurs when an attacker is able to impersonate a legitimate server. The attacker acts as the MITM and receives all traffic from the client and forwards it to the legitimate server and vice versa. Figure 16 (modified from Xia and Brustoloni 2005) shows the sequence for a MITM attack. The Web client thinks it is communicating with the Web server, but it is really communicating with the MITM server that forwards the request to the Web server. The Web server sends the response to the MITM server, which has the ability to repackage it and forward it to the Web client.

This type of attack is successful with HTTP and HTTPS transmissions. If the MITM connects with the Web server using HTTPS, then it could read the response and forward it to the Web client using HTTP. The user would have to be aware that HTTPS is not being used. Another scenario is if the MITM server connects with the Web client using HTTPS and the user overrides the security warnings. This would allow the untrusted certificate to be used with the client. The attacker would be able to decode the information sent from the client's Web and encode the information and forward it to the Web server using the Web server's certificate. Xia and Brustonloni

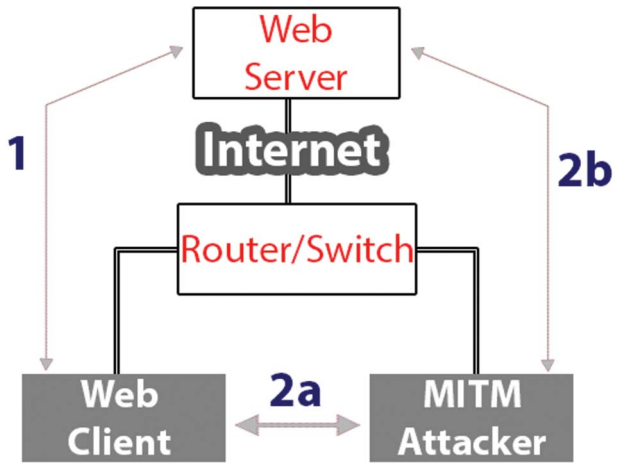

FIGURE 16 **MITM attack sequence (color figure available online).**

(2005) describe this attack and have created a prevention method. The technique helps prevent an attacker from becoming a MITM but does not help with recognizing a MITM server. The focus of this technique is end-user education. The education relates to stopping the user from making simple mistakes by displaying more information to a client container before the user makes potentially dangerous actions. One advantage to this technique since it is implemented in the client container is that it covers all Web applications, legitimate and malicious. This is a good addition to prevention, but it does not address coding practices to detect an impersonator.

4. SESSION SECURING MECHANISMS

Developers can secure sessions against the previously addressed hijacking attacks using a variety of mechanisms. A comprehensive overview of all securing mechanisms is beyond the scope of this paper, so only a small sampling of session securing tools and solutions will be reviewed in this section.

4.1. Session Prediction Mechanisms

Session tokens should be generated using long randomly generated numbers to deter prediction such as brute force attack guessing. The user's IP address can be stored to link the session token to a specific connection assuming that the attacker cannot use the same IP address. Both methods reduce the ability of an attacker to capture and forge the session token without adding significantly to developer complexity or overhead.

4.2. Session Fixation Mechanisms

Session token regeneration can mitigate session fixation attacks in a variety of forms. Session tokens from legitimate users should be regenerated when switching between HTTP and HTTPS sessions. This mitigates the chance that the session token has been intercepted and can be used to hijack the session. Phishing or malicious URL posts that include a session ID known to an attacker should also be regenerated upon login. This eliminates the ability of a malicious user to use the previous session ID. Session token expiration mechanisms can be enforced at the web server rather than relying on user timeout such as browser information. Without token regeneration, developers should employ mechanisms to check token header information.

4.3. Cross-Site Scripting Mechanisms

Several commonly used browsers provide built-in XSS mitigation tools. Internet Explorer 8 launched a XSS filter that is enabled by default. If an exposure to XSS is detected, then user will be alerted of the presence of a possible XSS attack and can choose to proceed. Google Chrome also launches a XSS Auditor which behaves in a similar manner and is enabled by default. Both tools do not eliminate XSS risk altogether, but they do provide enhanced security to the end user. This approach is limited to browser versions that support these tools, but not all computers are running the most updated version of every browser. For example, XSS filtering does not apply to Internet Explorer version 7 or earlier. Clearly, additional XSS mitigation strategies are needed.

Providing additional browser support has been proposed by Louw and Venkatakrishnan (2009) by integrating the BLUEPRINT tool with existing Web application. BLUEPRINT was able to "*demonstrate strong resistance to attacks, excellent compatibility with web browsers and reasonable performance overheads.*" Athanasopoulos et al. (2010) propose the xJS framework to mitigate XSS attacks without disrupting the client browsing experience and with minimal computational overhead.

4.4. Cross-Site Request Forgery Mechanisms

Mechanisms to mitigate CSRF can be broken into two categories: user and developer. User mechanisms are relatively easy to implement but rely completely on end-user education. Security unaware users may routinely leave browser sessions logged in, save passwords, or access several Websites at one time. Popular Web-based email applications may not default to strong security settings such as disabling links and images. Complexity is minimal to implement solutions for all of these cases but relies heavily on end-user education and awareness to provide a more secure environment that will mitigate CSRF hijacking attacks.

Developers must focus on different CSRF mitigation mechanisms. OWASP (2012) recommends countermeasures to reduce risk including adding "session-related information to the URL" and POST requests to increase hijacking attack complexity. Testing tools are available from open-source, widely available sources, as well as commercially developed tools.

4.5. Man-in–the-Middle Mechanisms

Beyond end-user education, we were unable to find a viable technique to prevent this type of attack. The server establishes a valid session with the MITM and the MITM establishes a valid session with the user. If this scenario is able to happen, how is the Web application supposed to determine that the session is with a legitimate user instead of a MITM attacker? Mechanisms for securing sessions against the hijacking attacks are summarized in Table 3.

5. CONCLUSIONS

The previous section reviewed a variety of HTTP session hijacking attacks. Each specific attack presents new challenges for Web developers to create secure code and to protect against threats. Table 3 summarizes the vulnerabilities discussed and techniques to minimize the risks. Web application session management's objective stated by OWASP (2011a) is to ensure "authenticated users have a robust and cryptographically secure association with their session." Therefore, if session management is required by a Web application, then the responsibility to provide a secure association with the session is inherently required. This seems to be the key to preventing HTTP session hijacking attacks.

The contributions of our work can be summarized as follows: (1) Providing an overview of session management and the security requirements of an application to create and maintain a Web application session; (2) Identifying various attacks that can lead to a session being hijacked and providing brief descriptions on how these attacks take place; (3) Consolidating and organizing related prevention techniques for each type of attack; and (4) Analyzing existing Web application implementations to help demonstrate the need for session hijacking prevention.

TABLE 3 **Summary of vulnerabilities reviewed in paper**

Hijacking attacks	Vulnerability	Mechanisms
Session token creation		
Session prediction	Session token value	1. Generate session tokens using long randomly generated numbers 2. Store IP address to link user connection and session token
Session fixation	Creation of session token on first visit and keep token entire visit.	Re-generate token value after logging into Website
Session token deletion		
	Session inactive timeout	• Implement a shorter session timeout on servers • Set cookie max-age to −1 • Communicate browser session timeout to server session to help keep in sync
	Browser session	1. Use of referrer header 2. Synchronizer token pattern (XSRF)
Session token transmission		
	Use of URL rewriting	Require the use of cookies Don't allow for URL rewriting
Session token verification		
Cross-site scripting (XSS)	User input displayed in browser	Input filter tool and output sanitizing tool Use of http only cookies
Cross-site forgery (XSRF)		Developers: Synchronizer token pattern Use of referrer header End users: Logging off and closing browser sessions up exit Do not save or remember passwords Segregate sensitive applications into separate browser sessions Use security features of Web-enabled email environments to disallow HTML tags by default
Man in the middle (MITM)		• End user education

5.1. Suggestions for Future Work

Future work in this area includes the following:

- *Real time detection of a session hijacking attack.* Security through coding may not be enough since a hijacking attack uses valid session tokens. A Web application needs to identify if a user is legitimate. The key to detection is accurately defining session hijacking signatures and creating a real time engine to detect if a user is being impersonated.
- *Integration of session management into Web server and browser applications.* Such integration technology would remove the need for individual Web applications to implement session management.

REFERENCES

Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E., and Karagiannis, T. (2010). xJS: Practical XSS prevention for Web application development. Proceedings of the 2010 USENIX Conference on Web Application Development, pp. 147–158.

Belani, R. (2004). Basic Web session impersonation. Symantec. Available from: http://www.symantec.com/connect/articles/basic-web-session-impersonation

Berghel, H. (2002). Hijacking the Web. Communications of the ACM, 45(4), 23–27.

CERT/CC. (2000). CERT advisory CA-2000-02 malicious HTML tags embedded in client Web requests. Available from: http://www.cert.org/advisories/CA-2000-02.html

Chetty, D. (2009). Tomcat 6 developer's guide. Birmingham, England: Packt Publishing.

Esposito, D. (2004, October 11). Session hijacking. Dr. Dobbs Journal.

Fain, Y. (2011). *Java programming 24-hour trainer* (p. 444). Indianapolis, IN: Wiley.

Jovanovic, N., Kirda, E., and Kruegel, C. (2006). Preventing cross-site request forgery attacks. Securecomm and Workshops, August 28–September 1.

Kerschbaum, F. (2007). Simple cross-site attack prevention. Third International Conference on SecureComm 2007, pp. 464–472, 17–21.

Louw, M. T. and Venkatakrishnan, V. N. (2009). Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. 2009 30th IEEE Symposium on Security and Privacy, pp. 331–346.

Microsoft. (2010). Microsoft security bulletin MS10-002. Available from: http://www.microsoft.com/technet/security/bulletin/ms10-002.mspx

Mitchell, S. (2010). Sams teach yourself ASP.NET 4 in 24 hours: Complete starter kit (1st ed.). Indianapolis, IN: Sams.

Mularien, P. (2010). Spring security 3. Birmingham, England: Packt Publishing.

OWASP Foundation. (2010a, October). Cross-site scripting (XSS).

OWASP Foundation. (2010b). OSWAP top 10 – 2010. The ten most critical Web application security risks.

OWASP Foundation. (2011a). Session management. Available from: https://www.owasp.org/index.php/Session_Management

OWASP Foundation. (2011b). CSRF prevention cheat sheet. Available from: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

OWASP Foundation. (2011c). CSRFGuard project. Available from: https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project

OWASP Foundation. (2012). Testing for CSRF (OWASP-SM-005). Available from: https://www.owasp.org/index.php/Testing_for_CSRF_(OWASP-SM-005)

Shklar, L. and Rosen, R. (2009). Web application architecture: Principles, protocols, and practices (2nd ed.). West Sussex, England: John Wiley & Sons.

Tappenden, A. and Miller, J. (2008). A three-tiered testing strategy for cookies. *1st International Conference on Software Testing, Verification, and Validation*, April 9–11.

Tulloch, M., Northrup, T., and Honeycutt, J. (2010). Windows 7 resource kit. Redmond, WA: Microsoft Press.

Vamosi, R. (2009, November). New banking Trojan horses gain polish. PCWorld.com. Available from: http://www.pcworld.com/article/182889/new_banking_trojan_horses_gain_poli-sh.html

Vinod, V., Anoop, M., Firosh, U., Sachin, S., Sangit, P., and Siddharth, A (2008). *Application security in the ISO27001 environment* (pp. 207–209). Ely, UK: IT Governance Publishing.

Visaggio, C. and Blasio, L. (2010). Session management vulnerabilities in today's Web. Security & Privacy, IEEE, 8(5), 48–56.

White, A. (2009). Java script programmers reference. Indianapolis, IN: Wiley.

Xia, H. and Brustoloni, J. (2005). Hardening Web browsers against man-in-the-middle and eavesdropping attacks. WWW '05 Proceedings of the 14th International Conference of World Wide Web, pp. 489–498.

BIOGRAPHIES

Shellie Wedman is an M.S. in Information Technology candidate at the University of Kansas. Her research interest is information security.

Annette Tetmeyer is a Ph.D. candidate in computer science at the University of Kansas. Her research interests include security requirements engineering, data mining, human computer interaction, and engineering education. In addition to experience in private industry, she has taught a variety of undergraduate and graduate engineering courses at the University of Kansas. She received her M.S. in Computer Science from the University of Kansas, her M.S. in Engineering Management from the University of Kansas, and her B.S. in Mechanical Engineering from Iowa State University.

Hossein Saiedian (Ph.D., Kansas State University, 1989) is currently a professor in the Department of Electrical Engineering and Computer Science at the University of Kansas (KU) and a member of the KU Information and Telecommunication Technology Center (ITTC). Professor Saiedian's primary area of research is software engineering and information security.