CHAPTER 7

SESSION HIJACKING PREVENTER

7.1 INTRODUCTION

Session hijacking is an exploitation of a valid web application session or session key, to gain unauthorized access to information or services in a computer system. Due to the stateless nature of the HTTP protocol, web applications require additional measures to keep their users authenticated. To achieve this criterion, session identifiers are used for the authentication process (Pujolle et al 2009). After a successful authentication, the web application generates the session ID, which will be transmitted to the client. Every HTTP request that contains this session ID is regarded as belonging to this particular user (Johns 2006). There are three methods of implementing session identifiers; they are URL query string, hidden form field and browser cookies. In this scenario, session hijacking is a type of attack through which the session ID will be hijacked by an attacker. After hijacking the legitimate user session ID, the attacker can impersonate as a legitimate user in the web application.

Session hijacking is possible through the session fixation attack or browser hijacking or background XSS propagation attack. To prevent session hijacking, the session ID should be properly handled without disclosing its value. In order to prevent session hijacking, a unique prevention approach is proposed and implemented; it is called session hijacking preventer in the WAPS-CIVS System.
7.2 SESSION HIJACKING PREVENTER ARCHITECTURE

The session hijacking preventer consists of three major modules, such as the session ID fixation preventer module, the browser hijacking preventer module and the background XSS propagation preventer module, as shown in figure 7.1. The common input interceptor module, intercepts all the client requests and the server responses and these will be sent to the individual prevention mechanism to analyse the request to prevent session hijacking in web applications.

In session ID fixation prevention, a dynamic session ID mapping technique is followed, and the entire prevention mechanism is implemented through the web services. This approach significantly protects the session ID cookies from the session fixation attacks. For the prevention, a dynamic session ID will be created and that will be attached to the original cookie, and it is set as httponly through the dynamic session ID mapping technique. Browser hijacking is another type of session hijacking attack in web application. To prevent browser hijacking, the one-time URL technique is followed. The one-time URL is created through the URL randomizer object using the nonce value, which will be created during the web application session. In web application, a common feature is the iframe and popup window (Phung et al 2009). This feature can be used to create an attack on the client side browser to steal the session, and is called the background XSS propagation attack. In the session hijacking preventer engine, one of the modules is to prevent this XSS propagation attack by tracking the sub-domain, which will be used to find the inclusion of the iframe and popup window.

In the prevention mechanism, all the prevention modules are implemented by means of web services to address the platform independent approach and to provide loosely coupling in the web application.
Figure 7.1 Session hijacking preventer architecture
7.3 PREVENTION OF SESSION ID FIXATION ATTACK

To prevent the session ID fixation attack, a dynamic session ID mapping technique is followed. In the implementation of the dynamic session ID mapping, the user request and response will be taken as inputs to prevent the session id fixation attack.

![Session ID fixation prevention engine](image)

**Figure 7.2 Session ID fixation prevention engine**

Whenever the user requests a web page, a request has to be initiated from the client. The initiated request is sent to the server, which brings back the response to the client. The intercepted request and response will contain header information and cookies, which contain the session id to maintain the user session. The internal architecture of the session ID fixation preventer engine is shown in Figure 7.2. Usually, the response from the web server is always along with a randomly generated unique session id to maintain a unique session for the single user session between the client and the server.
The input interceptor will intercept the HTTP request and HTTP response, and send this information to the cookies storage and mapping module, which it will store the original cookies value in the database, and generate one time dynamic id for each response, and attach it to the original session id value through the dynamic cookie rewriting technique. The dynamic id is generated for each client request in the web page on the server side, using the static id and request time, using the hash algorithm. This dynamic id is stored for each client-server session and attached to the original cookie, using the setHeader function of the servlet filter.

**Input:** User requested URL to the web server.

**Output:** Rewritten value of the DynamicID with the original id in the cookie.

**Algorithm:**

1. Read the URL requested.
2. Fetch the cookie of user request.
3. Generate dynamic id for each user request.
   \[
   \text{DynamicID} = \text{Hash (StaticID + ClientTime)} + \text{StaticID} + \text{ClientTime}
   \]
4. Attach dynamic id with original cookie and set it as httponly.
5. Store static and dynamic id in database for mapping.
6. Dynamic id is mapped for each user request in the web server.
7. If dynamic id does not match, send an error message to the user as session hijacking.

The returned cookie from the browser will also be rewritten back to the original value at the web service, before being forwarded to the web server. As the browser’s database does not store the dynamic id values of the cookies, so even the attacker can steal the cookies from the browser’s database, and the cookies cannot be used later to impersonate the users.
7.3.1 Dynamic ID Generation

Generate a dynamic random ID for unique session id using MD5 hash algorithm, by giving the static id and client request time as input. Dynamic ID contains the concatenation of Part1_String and Part2_String of static id and time details. Part1_String is formed using hash function by giving Static_ID and Client_Time as input. Part2_String is formed by the concatenation of Static_ID and Client_Time.

\[
\text{Part1\_String} = \text{Hash (Static\_ID + Client\_Time)} \\
\text{Part2\_String} = \text{Static\_ID + Client\_Time} \\
\text{DynamicID} = \text{Part1\_String + Part2\_String}
\]

When a user enters into a single client-server session, his session id is intercepted and stored in the database. If a client requests more number of pages in a single session with one unique id, then the database will save the request session id, domain that requested and path of the requested web application only once for whole session. Figure 7.3 shows the database table stored for each web server response.

![Database for session cookies](image_url)
Generate a dynamic random id for the unique session id, using the MD5 hash algorithm by giving the static session id and client request time as input. This dynamic id is generated for each client request in the web page on the server side, using the static id and request time using the hash algorithm. This dynamic id is stored for each client-server session and attached to the original cookie using the setHeader function of the servlet filter.

### 7.3.2 Dynamic session ID mapping

Using the dynamic session ID mapping technique, the dynamic id is added to the original cookie in the header field. Figure 7.4 shows the dynamic id and original session attached in the header field of the web server response.

![Figure 7.4 Dynamic session ID mapping](image)

The set-cookie header contains both the dynamic and original session id. The Dynamic session id in the header will set as the httponly cookie, so that it will not be displayed for the cross-site scripting code, and will not be given to the scripting code for the session hijacking attack in the web browser. Even if an attacker fetched the original session id through the
java script code for fixation, it won’t be helpful to steal the session. Figure 7.5 shows the cookies’ value that contains only the original session id.

![Welcome to Money Transfer](image1)

**Figure 7.5 Attacker httponly cookies**

Even if the original session id is fixed in URL link, the dynamic id will not be sent to the web browser. The attacker may crafted link with a original session id and send it to the user through email for session id fixation shown in Figure 7.6.

![link1 - Notepad](image2)

**Figure 7.6 Original session id crafted link**

If a legitimate user clicks on the specially crafted link, it will be redirected to authentication page. On the authentication page the user has to provide his own authentication credential, and log in his page, but the attacker session id will not be fixed to a user due to the dynamic id set in the header.
Both the attacker and user have different sessions, so a session fixation attack will be prevented. Figure 7.7 shows how the user session id differs from the attacker fixed session id.

7.4 BROWSER HIJACKING PREVENTION

The browser hijacking attack uses this ability to place a series of HTTP requests to the web application (Fu et al 2001). The application server cannot differentiate between regular, user initiated requests, and the requests that are placed by the script. The malicious script is therefore capable of acting under the identity of the user, and commits arbitrary actions on the web application.

```
javascript: function a(){for(i=1;i<3;i++){window.open("http://www.google.co.in");
self.focus();document.location="http://www.google.co.in";}}a();
```

This script will be given to the browser address bar, and it will send a continuous HTTP request to the web server and the server will respond to each request placed by the script. This script will execute the browser hijacking attack for google page, as shown in Figure 7.8.
Figure 7.8 Browser hijacking attack on Google page

The Browser hijacking attack can be prevented by generating one time URL with randomized nonce through URLRandomization. Figure 7.9 shows the browser hijacking prevention system. The attacking script submits one or more http requests to the server and potentially parses the server response. The basis for this attack is the attacker knowledge of the web application URLs.

Figure 7.9 Browser hijacking prevention system
The main idea of the proposed system is to enhance the application URLs with a secret component which cannot be known by the attacking JavaScript. As long as the server responds only to requests for URLs with a valid secret component, the attacker is unable to execute a browser hijacking attack. This approach uses a URL GET parameter, called rnonce, to implement the URL randomization. Only URLs containing a valid rnonce are treated as authorized by the web server. To conduct the actual randomization of the URLs by the URLRandomizer, a JavaScript object is included in every webpage. The URLRandomizer object contains a private variable that holds all the valid randomization data. During object creation, the URLRandomizer requests from the web server a list of valid nonces for the webpage URLs. This request has to be done as a separate HTTP request on runtime. Otherwise, the list of valid nonce would be part of the source code of the HTML page, and therefore, unprotected against browser hijacking attacks.

7.4.1 Extracting nonce Value

A nonce is a number generated for a specific use, such as session authentication (Yang and Shieh 1999). In this context, nonce stands for "number used once" or "number once". Typically, a nonce is some value that varies with time, although a very large random number is sometimes used. A nonce can be a time stamp, a visit counter on a web page, or a special marker intended to limit or prevent the unauthorized replay or reproduction of a file. An initialization vector is a nonce used for data encryption. The initialization vector, used only once in any session, prevents the repetition of sequences in an encrypted text. Identifying such repetitions can help an attacker break a cipher. This pseudo random number ensures that old communications between a client and a server cannot be reused in browser hijacking attacks. A browser hijacking attack is a network attack in which previous valid data transmission is repeated. The nonce is a server specified data string which should be uniquely generated each time a 401 response is returned by the
server. The 401 response that is sent back to the client includes the nonce generated by the server. The client should add this nonce to the header of the next requests. This nonce generated uniquely for each request of web application, is called a randomized nonce (Yadav and Sardana 2011) or rnonce.

### 7.4.2 Generate One-Time URL with rnonce value

All browser hijacking attacks have one characteristic in common: the attacking script submits one or more HTTP requests to the server and potentially parses the server’s response. The basis for this attack is, therefore, the attacker’s knowledge of the web application’s URLs. The main idea of the proposed countermeasure is to enhance the application’s URLs with a secret component rnonce. As long as the server responds only to requests for URLs with a valid rnonce, the attacker is unable to execute a browser hijacking attack. The generated rnonce value is attached to the URL, to generate URLRandomization for one-time URL. Figure 7.10 shows the one-time URL with the rnonce value.

![Figure 7.10 One-Time URL with rnonce value](image-url)
This nonce value is checked for every user request with original session id value. Browser hijacking attack occurs within single session value. By attaching nonce value, only one nonce value is valid for single user session. Figure 7.11 shows the database for different nonce value for single user session.

![Database for nonce value with single user session](image)

**Figure 7.11  Database for nonce value with single user session**

For every user request, original session id value is compared with nonce value attached is set true flag in database. If flag is true, change it to false and generate new nonce value and attached this new nonce value with header URL. Set this nonce value flag as true for this session id value. For remaining request from same session, the web server send error message as browser hijacking. Attacker script send continuous request to the web server with same session id but with different nonce value in header field. But those fields flag value are set as false for that session id and nonce value. So the server responds to the client as a browser hijacking attack for the remaining request.
If the original request comes from different session id values with the nonce, the web server will respond to that request in a new session as a different user. Figure 7.12 shows the server error for the browser hijacking attack by continuous requests to the web server through a vulnerable script.

### 7.5 BACKGROUND XSS PROPAGATION PREVENTION

Background XSS propagation can be prevented by forming a domain cluster for sub-domain switching. Each time the domain is switched in the cluster, the Document.domain property of the popup window differs from the original window. The DOM-Based XSS attack will not be possible through the sub-domain switching technique. Figure 7.13 shows the background XSS propagation prevention system.
Background XSS propagation attacks are prevented, by ensuring a no trust relationship between the pages induced by the same-origin policy, which exists as long as the document.domain property for every page differs. To achieve this trust removal introduce an additional sub-domain to the web application. Every link included into a webpage directs to a URL with a sub-domain, that differs from the domain of the containing webpage. As a result, every single page possesses a different document.domain value. In cases where a page A explicitly wants to create a trust relationship to a second page B, pages A and B can change their document.domain setting to exclude the additional sub-domain. All the added sub-domains map to the same server scripts. The sub-domain have no semantic function; they are only used to undermine the implicit trust relationship. If a malicious script rewrites all
URLs in a page to match the script’s document.domain value, the web application will still function correctly and a background propagation attack will again be possible. For this reason, the web server has to keep track of which mapping between the URLs and sub-domains has been assigned to a user session.

7.5.1 Preventing background XSS propagation

The background XSS propagation attack is a kind of DOM-Based XSS attack in web application. A Vulnerable XSS script will not be sent to the server side. It will access the information of users in the DOM tree of the web page. This attack occurred by opening the popup window, and staying in the background, when the user is browsing web page. The user cannot notice the popup window open in the background.

This popup window has the same DOM property as the original window because of the same origin policy. So the attacker popup window will access the users DOM tree and send valuable users information to the attacker. When a user clicks the above link, the attacker script will get executed, and open the original user window and the popup window in the background, with the same document.domain property. If the user enters credential information in the original web page, it will be accessed in the popup window through the DOM tree of that original page. Because of the same document property, the JavaScript in the popup window will access all the document values in the original user page and send this information to the attacker site. Figure 7.14 shows the popup window accessing all the information of the original user page.
7.5.2 Forming Domain Cluster

Webpages with the same origin implicitly trust one another. Because of this circumstance unwanted background windows are capable of inserting malicious scripts in pages that would be vulnerable. To remove this implicit trust between individual webpage that belong to the same web application, there is a need to ensure, that the no trust relationship between these pages induced by the same-origin policy exists. As long as the Document.domain property for every page differs, background XSS propagation attacks are impossible. To remove the implicit trust between web pages, each web page must be loaded with different domains for the same web application. In order to implement this approach, we have to form different sub-domains for one main domain. These sub-domains are formed as a single cluster, using the clustering option in the sun java application server for glassfish v2.

Node agents are created in the application server for forming the cluster agent. These node agents are started in the asadmin command prompt.
of the glassfish server. The target domain instance has to be configured for each sub-domain in the cluster, after starting of the node agent, and a running node agent has to be provided in the server. Then the instances are added to form a cluster to implement the sub-domain for web applications. Figure 7.15 shows the domain cluster created for the sun java system application server. These clusters are in the running state to implement sub-domain switching in web applications.

![Figure 7.15 Domain cluster formation](image)

### 7.5.3 Sub-domain Switching

The formed clusters with sub-domains are added to the target server by clicking the managing target in the application server for unique web applications. These sub-domains are all mapped to the same server scripts. Every link included into a webpage directs to a URL with a sub-domain, that differs from the domain of the containing webpage.

Consider web pages A & B, in cases where a page A explicitly wants to create a trust relationship with a second page B; pages A and B can change their Document.domain setting to exclude the additional sub-domain. Figure 7.16 shows the selection page for target sub-domain switching, in a particular web application.

![Figure 7.16 Selecting target domain for sub-domain switching](image)

All added sub-domains map to the same server scripts. Therefore, the URL http://ldomain1.localhost:8080/process.jsp points to the same resource, as for example, the URL http://ldomainn.localhost:8080/process.jsp. The sub-domains have no semantic function, and they are only used to undermine the implicit trust relationship. If a malicious script rewrites all URLs in a page to match the script’s Document.domain value, then the web application will still function correctly and a background propagation attack will again be possible. For this reason, the web server has to keep track of which mapping between URLs and sub-domains have been assigned to a user’s session.
For our implementation, the Glassfish web server is used, which allows the usage of wildcards in the definition of sub-domain names. Consequently, we had an unlimited supply of applicable sub-domain names. This allows a choice between random subdomain names or incrementing the sub-domain identifier (ldomain1.localhost:8080 links to ldomain2.localhost:8080 which links to ldomain3.localhost:8080).

**Figure 7.17 Preventing background XSS propagation**

On application servers that do not offer such an option, and where the number of available sub-domain names is limited, the web application has to be examined closely. It has to be determined how many sub-domains are required and how the mapping between URLs and sub domains should be implemented.

Figure 7.17 shows the implementation of preventing the XSS propagation in web application through the sub-domain switching technique.
7.6 FILE SYSTEM LOG ENTRY

Whenever a client provides input through the web application, a log entry is made for each request and response. If the session hijacking prevention system filter returns true, then the log entry is made along with the requested client details for further analysis. If the session filters false, then there will be no logs with respect to the input parameters. The report generation module can be further used to compare the effectiveness of the tool by using the two most important metrics of a high vulnerability coverage and a less false positive rate.

Figure 7.18 Sample file system log entry

Figure 7.18 shows a sample log file which is created when the attack occurs. The log files are maintained in the server, which consists of all the clients’ information. The log file can be used for storing the type of error that had occurred when the attack occurred.
7.7 RESULTS AND DISCUSSION

To evaluate the proposed system, a detailed analysis has been performed with different types of XSS inputs. In the analysis process, the designed system is integrated with the web application, and verifies the detection of the session ID fixation, browser hijacking and background XSS propagation.

7.7.1 Session ID Fixation Prevention Results

The designed system is evaluated with a set of HTTP requests for a period of one month, and the results are obtained to find the detection rate of the WAPS-CIVS system.

The proposed and designed Session ID fixation attack prevention system has the following advantages

- It prevents the session ID fixation attack without decrease the performance of the server compared to the existing approaches.
- Does not demand any change in the business logic.
- The entire system is designed by means of web services to meet loosely coupling with any web application.

The Session ID fixation attack prevention of WAPS-CIVS is compared with the static analysis approach, and the results of the comparison are shown in Table 7.1.
Table 7.1  Comparison of the session ID fixation attack Preventer with the static analysis method

<table>
<thead>
<tr>
<th>No of HTTP request submitted</th>
<th>Session Hijacking script</th>
<th>Detected by static analysis</th>
<th>Detected by WAPS-CIVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>45</td>
<td>38</td>
<td>33</td>
</tr>
<tr>
<td>Week 2</td>
<td>50</td>
<td>41</td>
<td>35</td>
</tr>
<tr>
<td>Week 3</td>
<td>60</td>
<td>57</td>
<td>49</td>
</tr>
<tr>
<td>Week 4</td>
<td>75</td>
<td>71</td>
<td>64</td>
</tr>
<tr>
<td>Week 5</td>
<td>65</td>
<td>59</td>
<td>51</td>
</tr>
</tbody>
</table>

The results show that the session ID fixation prevention system prevents session hijacking upto 96.2%, as shown in Figure 7.19.

Figure 7.19  Comparison of WAPS-CIVS with the static analysis method

To evaluate the performance of the session ID fixation prevention module, a set of XSS inputs were given to the implemented system and the response time is tabulated in Table 7.2. The response time of the web application is evaluated by with and without the WAPS-CIVS system. From the comparison it is seen, that the implemented session ID fixation prevention module completely prevents the hijacking of the session ID with XSS inputs to the web application. In order to prevent the session ID hijacking through
the WAPS-CIVS, it takes an average of 3ms more than the actual response time, as shown in Figure 7.20.

**Table 7.2 Response time comparison for session ID fixation module**

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Response Time for Session ID Fixation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without Prevention System (milli seconds)</td>
</tr>
<tr>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>Avg.</td>
<td>21.5</td>
</tr>
</tbody>
</table>

As a result, the developed system ensured that it would prevent session hijacking through the session ID fixation.

**Figure 7.20 Response time evaluation of the session ID fixation prevention module**
### 7.7.2 Browser Hijacking Prevention Results

The browser hijacking prevention module responded with “server error” for all the arbitrary malicious requests from the attacker. The designed system was evaluated with a set of HTTP requests for a period of one month, and the results are obtained to find the detection rate of the browser hijacking prevention in the WAPS-CIVS system.

The proposed and designed Session ID fixation attack prevention system has the following advantages:

- It prevents the browser hijacking attack with a simple approach, compared to the existing approaches.
- It is designed by means of web services, to meet the loosely coupling with any web application.
- It does not demand any additional tool in the client browser to prevent browser hijacking.

The browser hijacking prevention of the WAPS-CIVS is compared with the static analysis approach, and the results of the comparison are shown in Table 7.3.

<table>
<thead>
<tr>
<th>Week</th>
<th>No of HTTP request submitted</th>
<th>Browser Hijacking script</th>
<th>Detection by Static analysis</th>
<th>Detected by WAPS-CIVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>38</td>
<td>36</td>
<td>31</td>
<td>34</td>
</tr>
<tr>
<td>Week 2</td>
<td>35</td>
<td>31</td>
<td>24</td>
<td>29</td>
</tr>
<tr>
<td>Week 3</td>
<td>50</td>
<td>46</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>Week 4</td>
<td>60</td>
<td>56</td>
<td>52</td>
<td>55</td>
</tr>
<tr>
<td>Week 5</td>
<td>57</td>
<td>55</td>
<td>48</td>
<td>53</td>
</tr>
</tbody>
</table>
The results show that the browser hijacking prevention system prevents session hijacking up to 95.9%, as shown in Figure 7.21.

![Figure 7.21 Comparison of the WAPS-CIVS with the static analysis method](image)

To evaluate the performance of this module in the WAPS-CIVS, a set of malicious requests were sent to the web server at different times. The response of the web application with its response times are tabulated in Table 7.4.

**Table 7.4 Response time comparison for the session ID fixation module**

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Response Time Without Prevention System (ms)</th>
<th>Response Time With Prevention System (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>65</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>39</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>31</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>54</td>
<td>62</td>
</tr>
<tr>
<td>10</td>
<td>37</td>
<td>42</td>
</tr>
<tr>
<td>Avg.</td>
<td><strong>39.7</strong></td>
<td><strong>45.6</strong></td>
</tr>
</tbody>
</table>
In a real-time environment testing, the browser hijacking prevention system in WAPS-CIVS preventing the session hijacking, by an additional average response time of 5 milli seconds, as shown in Figure 7.22.

![Response Time Evaluation](image)

**Figure 7.22 Response time evaluation of the browser hijacking prevention module**

According to the result, the developed system ensured that it will prevent session hijacking through session ID fixation.

### 7.7.3 Background XSS Propagation Prevention Results

The background XSS propagation prevention module does not reveal any confidential data between the parent window and the child window. Through the sub-domain switching technique, the relationship between the parent and child window is maintained properly. The designed system is evaluated with a set of HTTP requests for a period of one month, and the results are obtained to find the detection rate of the background XSS propagation prevention in the WAPS-CIVS system.
The proposed and designed Session ID fixation attack prevention proves that,

- It prevents the background XSS propagation attack created through the iframe and popup window, compared to the existing approaches.

- It does not demand any change in the business logic and the entire system is designed by means of web services to meet the loosely coupling with any web application.

The background XSS propagation prevention of the WAPS-CIVS is compared with the static analysis approach, and the results of the comparison are shown in Table 7.5.

**Table 7.5 Comparison of the Background XSS propagation Preventer with the static analysis method**

<table>
<thead>
<tr>
<th></th>
<th>No of HTTP request submitted</th>
<th>Background XSS propagation</th>
<th>Detection by Static analysis</th>
<th>Detected by WAPS - CIVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>45</td>
<td>36</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>Week 2</td>
<td>25</td>
<td>22</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>Week 3</td>
<td>40</td>
<td>35</td>
<td>27</td>
<td>34</td>
</tr>
<tr>
<td>Week 4</td>
<td>35</td>
<td>29</td>
<td>34</td>
<td>27</td>
</tr>
<tr>
<td>Week 5</td>
<td>38</td>
<td>31</td>
<td>25</td>
<td>29</td>
</tr>
</tbody>
</table>

The results show that the background XSS propagation prevention system prevents session hijacking up to 93.5%, as shown in figure 7.23. In the results, without the WAPS-CIVS system, the web application is vulnerable to the session hijacking attack, which is created through background XSS propagation.
In order to verify the response time of the background XSS propagation prevention module in WAPS-CIVS, ten different malicious inputs are applied to the web application. At the end, the response times are recorded with and without the WAPS-CIVS of the background XSS propagation preventer, as tabulated in Table 7.6.

**Table 7.6  Response Time Comparison for the background XSS propagation module**

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Response time of background XSS propagation system</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response Time Without Prevention System (ms)</td>
<td>Response Time With Prevention System (ms)</td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>78</td>
<td>83</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>31</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>31</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>47</td>
<td>53</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>36.2</strong></td>
<td><strong>41.5</strong></td>
</tr>
</tbody>
</table>
For the detection of session hijacking, the prevention system needs to use an additional response time of an average of 5 milli seconds. But the prevention system completely blocks the communication of confidential data between the parent window and the attacker initiated popup window as shown in Figure 7.24.

![Response Time Evaluation](image)

**Figure 7.24 Response time evaluation background XSS propagation prevention module**

At the end, the response times of the WAPS-CIVS including the session ID fixation preventer, browser hijacking preventer, and background XSS propagation preventer are calculated and the figure 7.25 shows the overall response time of WAPS-CIVS system to prevent session hijacking attack.

This time difference is not significant, and is only in milliseconds. But this approach detects all types of vulnerable input like the user input, HTTP cookie and the server variables.
In addition, the entire system is designed and implemented based on web services; the preventer system is independent of the web application and the session hijacking preventer does not require any modification in the web application. Moreover, the prevention system can be used in any platform applications.

7.8 CHAPTER SUMMARY

Communication over the HTTP protocol may have different TCP connections. For every connection, the web server needs a method to recognize every user’s connections. The most useful method depends on a session ID that the Web Server sends to the client browser, after a successful client authentication. A session ID is normally composed of a string variable, and it could be used in different ways, as in the URL, in the header of the http requisition as a cookie, in other parts of the header of the http request, or in the body of the http requisition. This session ID can be hijacked through session ID fixation, browser hijacking and background XSS propagation.
In this design, every session hijacking mechanism is individually considered to prevent the hacking of a session. Every HTTP request and HTTP response are intercepted and passed to the session ID fixation prevention module, and it will store the original cookies value in the database, and generate one dynamic id for each response, and attach it to the original session id value through the dynamic session ID mapping technique. By setting the set-cookie header containing both the dynamic and original session id, the dynamic session id in the header will be set as the httponly cookie, so that it will not be used to hack the session. Another session hijacking attack is browser hijacking. This browser hijacking is prevented by generating a one-time URL with a randomized nonce for URL Randomization. For every user request, the original session id value is compared with the nonce value, which is a unique value, and it is attached to the URL to generate URLRandomization for a one-time URL. The attached nonce value is set the true flag in the database. If the flag is true, change it to false and generate a new nonce value and attach this new nonce value to the header URL. Set this nonce value flag as true for this session id value. For the remaining request from the same session, the web server sends an error message as browser hijacking.

The background XSS propagation attacks are prevented by ensuring a no trust relationship between the pages induced by the same-origin policy, which exists as long as the document.domain property for every page differs. To achieve this trust removal introduces an additional sub-domain to the web application. Every link included in a webpage directs to a URL with a sub-domain, that differs from the domain of the containing webpage. As a result, every single page possesses a different document.domain value. Hence, the attacker cannot steal any credential, and other confidential information from the legitimate web page.