SessionSafe: Implementing XSS Immune Session Handling*

Martin Johns

Security in Distributed Systems (SVS)
University of Hamburg, Dept of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
johns@informatik.uni-hamburg.de

Abstract. With the growing trend towards the use of web applications the danger posed by cross site scripting vulnerabilities gains severity. The most serious threats resulting from cross site scripting vulnerabilities are session hijacking attacks: Exploits that steal or fraudulently use the victim's identity. In this paper we classify currently known attack methods to enable the development of countermeasures against this threat. By close examination of the resulting attack classes, we identify the web application's characteristics which are responsible for enabling the single attack methods: The availability of session tokens via JavaScript, the pre-knowledge of the application's URLs and the implicit trust relationship between webpages of same origin. Building on this work we introduce three novel server side techniques to prevent session hijacking attacks. Each proposed countermeasure removes one of the identified prerequisites of the attack classes. SessionSafe, a combination of the proposed methods, protects the web application by removing the fundamental requirements of session hijacking attacks, thus disabling the attacks reliably.

1 Introduction

Web applications as frontends for online services enjoy an ever growing popularity. In addition, a general direction towards web applications replacing traditional client side executables can be observed during the last years. Email, banking, project management or business services move from specialized programs to the web browser.

In close connection to this trend, web application vulnerabilities move from being mere annoyances towards posing severe threats. With companies like Google and Yahoo starting to integrate various web applications under one authentication process the impact of single vulnerabilities even increases, as one weakness could now endanger a whole range of different applications. One of the most common threats is session hijacking, an attack method that targets the victim's identity. Session hijacking is often feasible because web applications frequently

* This work was supported by the German Ministry of Economics and Technology (BMWi) as part of the project "secologic" , www.secologic.org.

D. Gollmann, J. Meier, and A. Sabelfeld (Eds.): ESORICS 2006, LNCS 4189, pp. 444–460, 2006.

suffer from cross site scripting (XSS) vulnerabilities. In most cases one single XSS vulnerability suffices to compromise the entire application.

Even though XSS is known at least since the year 2000 [4], this class of weaknesses is still a serious issue. In 2005 alone 167 XSS vulnerabilities have been reported to the BugTraq mailing list. Also, the year 2005 brought a shift to more sophisticated XSS attacks: In July 2005 Anton Rager presented the XSS proxy, a tool that allowed for the first time a systematic, semi automatic exploitation of XSS vulnerabilities [18]. Wade Alcorn described in September 2005 an XSS virus that is able to self propagate from server to server, provided all these servers run the same vulnerable web application [2]. Finally, in October 2005 the self replicating "mySpace XSS worm" infected more than one million user profiles [19]. While the cause of XSS vulnerabilities is almost always insufficient output sanitization in connection with handling of user provided input strings, ensuring the absence of this flaw is of growing difficulty. Today's web applications are complex. They often consist of numerous different server side technologies, legacy code and third party components. Thus, enforcing consistent input handling a non-trivial task. In this paper we describe a novel approach to protect web applications against XSS session hijacking attacks. Instead concentrating on user input, we disable session hijacking by removing the attacks' basic requirements.

The remainder of the paper is organized as follows. Section 2 discusses web application security topics that are relevant for the proposed methods. Section 3 describes and classifies currently known XSS session hijacking attacks. In Section 4 follows a description of our countermeasures; those countermeasures then will be discussed in Section 5. Finally, after looking at related work in Section 6, we conclude in Section 7.

2 Technical Background

2.1 Session Management

Because of http's stateless nature [7] web applications that require authentication need to implement additional measures to keep their users authenticated. To achieve this, session identifiers (SID) are used: After a successful authentication the web application generates the SID and transmits it to the client. Every following http request that contains this SID is regarded as belonging to this particular user. Thus the SID is a credential that both identifies and authenticates the user. The protection of this information is therefore essential for the security of the application. There are three methods of implementing SIDs: inclusion of the identifier in the URL, communication of the identifier via POST parameters or storing the identifier in browser cookies:

URL query strings: The SID is included in every URL that points to a resource of the web application: `<a href="some_page?SID=g2k42a">...</a>`

POST parameters: Instead of using hyperlinks for the navigation through the application HTML forms are used. In this case, the SID is stored in a hidden form field. Whenever a navigation is initiated, the according HTML form is submitted, thus sending the SID as part of the request's body.

Cookies: Using cookies for SID storage is broadly used in today's web applications. Web browser cookies technology [14] provides persistent data storage on the client side. A cookie is a data set consisting at least of the cookie's name, value and domain. It is sent by the web server as part of an http response message using the `Set-Cookie` header field. The cookie's domain property is controlled by the URL of the http response that is responsible for setting the cookie. The property's value must be a valid domain suffix of the response's full domain and contain at least the top level domain (tld) and the original domain. The cookie's domain property is used to determine in which http requests the cookie is included. Whenever the web browser accesses a webpage that lies in the domain of the cookie (the domain value of the cookie is a valid domain suffix of the page's URL), the cookie is automatically included in the http request using the `Cookie` field.

2.2 Cross Site Scripting

Cross Site Scripting (XSS) denotes a class of attacks. These attacks are possible, if a web application allows the inclusion of insufficiently filtered data into web pages. If a web application fails to remove HTML tags and to encode special characters like `"`, `'`, `<` or `>` from user provided strings, an attacker will be capable of inserting malicious script code into webpages. Consequently, this script code is executed by the victim's web browser and runs therefore in the victim's security context. Even though XSS is not necessarily limited to JavaScript, attacks may also use other embedded scripting languages, this paper focuses its description of attacks and countermeasures on JavaScript. While completely removing scripting code through output filtering is feasible to disarm XSS threats posed by more obscure client side scripting languages like VBScript, this procedure is not an option in the case of JavaScript. JavaScript is ubiquitous, deeply integrated in common DHTML techniques and used on the vast majority of websites. A rogue JavaScript has almost unlimited power over the webpage it is contained in. Malicious scripts can, for example, change the appearance of the page, steal cookies, or redirect form actions to steal sensitive information (see Section 3 and [9] for further details).

2.3 JavaScript Security Aspects

JavaScript contains semantics of object oriented programming as well as aspects that are usually found in functional languages. In this paper we describe JavaScript from an object orientated point of view. JavaScript in webpages is executed "sandboxed": It has no access to the browser's host system and only limited access to the web browser's properties. JavaScript's capabilities to manipulate the appearance and semantics of a webpage are provided through the global object `document` which is a reference to the root element of the page's DOM tree [11]. A script can create, delete or alter most of the tree's elements. JavaScript 1.5 has been standardized by ECMA as "ECMAScript" [6] in 1999.

The same-origin policy: The "same-origin policy" defines what is accessible by a JavaScript. A JavaScript is only allowed read and/or write access to

properties of windows or documents that have the same origin as the script itself. The origin of an element is defined by specific elements of the URL it has been loaded from: The host, the port and the protocol [8]. While port and protocol are fixed characteristics, JavaScript can influence the host property to mitigate this policy. This is possible because a webpage's host value is reflected in its DOM tree as the `domain` attribute of the `document` object. JavaScript is allowed to set this property to a valid domain suffix of the original host. For example, a JavaScript could change `document.domain` from `www.example.org` to the suffix `example.org`. JavaScript is not allowed to change it into containing only the top level domain (i.e. `.org`) or some arbitrary domain value. The same-origin policy defines also which cookies are accessible by JavaScript.

Public, privileged and private members in JavaScript objects: A little known fact is, that JavaScript supports information hiding via encapsulation. The reason for this obscurity is that JavaScript does not provide access specifiers like "private" to implement encapsulation. Encapsulation in JavaScript is implemented via the scope of a variable. Depending on the context in which a variable or a method is created, its visibility and its access rights are defined [6]. From an object oriented point of view this translates to three access levels: "public", "privileged" and "private" [5]: "Public" members of objects are accessible from the outside. They are either defined by prototype functions [6] or created as anonymous functions and added to the object after object creation. Either way: They are created within the global scope of the object's surroundings. Public methods cannot access private members. "Private" members are only accessible by private or privileged methods in the same object. They are defined on object creation and only exist in the local scope of the object. Private methods cannot be redefined from the outside after object creation. "Privileged" methods are accessible from the outside. They can read and write private variables and call private methods. Privileged methods have to be defined on object creation and exist therefore in the local scope of the object. The keyword `this` is used to export the methods to the global scope, so that they can be accessed from outside the object. If a privileged method is redefined from the outside after object creation, it will become part of the global scope and its state will change therefore to "public".

3 A Classification of XSS Session Hijacking Attacks

All currently known XSS session hijacking attack methods can be assigned to one of the following different classes: "Session ID theft", "Browser Hijacking" and "Background XSS Propagation".

3.1 Session ID Theft

As described in Section 2.1, web applications usually employ a SID to track the authenticated state of a user. Every request that contains this SID is regarded as belonging to the authenticated user. If an attacker can exploit an XSS vulnerability of the web application, he might use a malicious JavaScript to steal

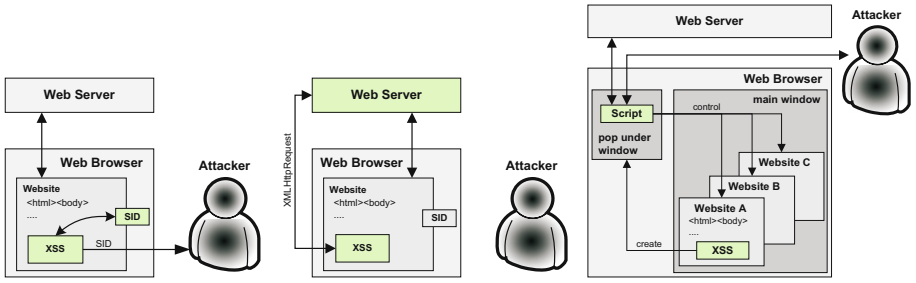


Fig. 1. The three classes of XSS session hijacking attacks [24]

the user's SID. It does not matter which of the methods described in Section 2.1 of SID storage is used by the application - in all these cases the attacking script is able to obtain the SID. The attacking script is now able to communicate the SID over the internet to the attacker. As long as the SID is valid, the attacker is now able to impersonate the attacked client [13].

3.2 Browser Hijacking

This method of session hijacking does not require the communication of the SID over the internet. The whole attack takes place in the victim's browser. Modern web browsers provide the XMLHttpRequest object, which can be used to place GET and POST requests to URLs, that satisfy the same-origin policy. Instead of transferring the SID or other authentication credentials to the attacker, the "Browser Hijacking" attack uses this ability to place a series of http requests to the web application. The application's 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 commit arbitrary actions on the web application. In 2005 the so called "mySpace Worm" employed this technique to create a self replicating JavaScript worm that infected approximately one million profiles on the website myspace.com [19].

3.3 Background XSS Propagation

Usually not all pages of a web application are vulnerable to cross site scripting. For the attacks described above, it is sufficient that the user visits only one vulnerable page in which a malicious script has been inserted. However, other attack scenarios require the existence of a JavaScript on a certain webpage to work. For example, even when credit card information has been submitted it is seldom displayed in the web browser. In order to steal this information a malicious script would have to access the HTML form that is used to enter it. Let us assume the following scenario: Webpage A of the application is vulnerable against XSS whereas webpage B is not. Furthermore, webpage B is the page containing the credit card entry form. To steal the credit card information, the

attacker would have to propagate the XSS attack from page A to page B. There are two techniques that allow this attack:

Propagation via iframe inclusion: In this case, the XSS replaces the displayed page with an iframe that takes over the whole browser window. Furthermore, the attacking script causes the iframe to display the attacked webpage, thus creating the impression that noting has happened. From now on every user navigation is done inside the iframe. While the user keeps on using the application, the attacking script is still active in the document that contains the iframe. As long as the user does not leave the application's domain, the malicious script is able to monitor the user's surfing and to include further scripts in the webpages that are displayed inside the iframe. A related attack is described in [18].

Propagation via pop under windows: A second way of XSS propagation can be implemented using "pop under" windows. The term "pop under" window denotes the method of opening a second browser window that immediately sends itself to the background. On sufficiently fast computers users often fail to notice the opening of such an unwanted window. The attacking script opens such a window and inserts script code in the new window's body. The new window has a link to the DOM tree of the original document (the father window) via the `window.opener` property. This link stays valid as long as the `domain` property of the father window does not change, even after the user resumes navigating through the web application. The script that was included in the new window is therefore able to monitor the user's behavior and include arbitrary scripts in web pages of the application that are visited during the user's session.

4 Countermeasures Against Session Hijacking

In the next Sections we propose countermeasures against the described session hijacking attacks. Each of these countermeasures is designed to disarm at least one of the specified threats.

4.1 Session ID Protection Through Deferred Loading

The main idea of the proposed technique is twofold: For one, we store the SID in such a way that malicious JavaScript code bound by the "same-origin policy" is not able to access it any longer. Secondly, we introduce a deferred process of loading the webpage, so that security sensitive actions can be done, while the page is still in a trustworthy state. This deferred loading process also guarantees the avoidance of timing problems.

To successfully protect the SID, it has to be kept out of reach for any JavaScript that is embedded into the webpage. For this reason, we store the SID in a cookie that does not belong to the webpage's domain. Instead, the cookie is stored for a different (sub-)domain that is also under the control of the web application. In the following paragraphs the main web application will reside on `www.example.org`, while the cookies will be set for `secure.example.org`. The domain `secure.example.org` is hosted on the same server as the main web

application. Server scripts of the main web application have to be able to share data and/or communicate with the server scripts on `secure.example.org` for this technique to work. On the `secure` domain only two simple server scripts exist: `getCookie.ext` and `setCookie.ext`. Both are only used to transport the cookie data. The data that they respond is irrelevant - in the following description they return a 1 by 1 pixel image.

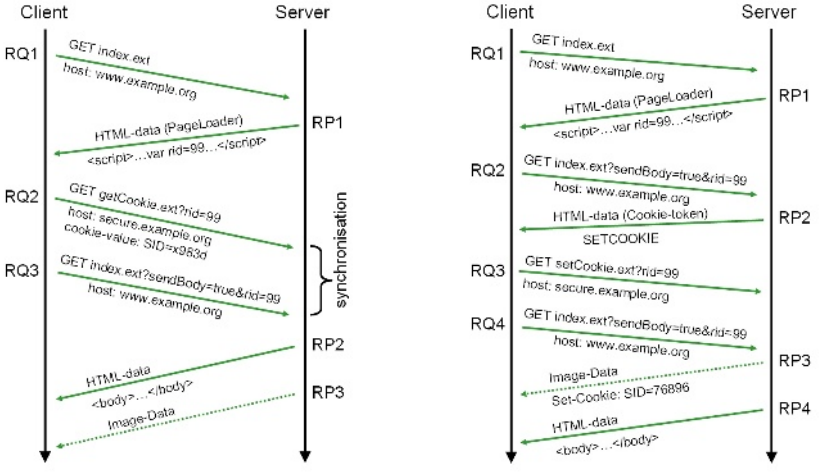
To carry out the deferred loading process we introduce the "PageLoader". The PageLoader is a JavaScript that has the purpose to manage the cookie transport and to load the webpage's content. To transport the cookie data from the client to the server it includes an image with the `getCookie.ext` script as URL. For setting a cookie it does the same with the `setCookie.ext` script. To display the webpage's body the PageLoader requests the body data using the XMLHttpRequest object. In the following specifications the abbreviations "RQ" and "RP" denote respectively "http request" and "http response".

Getting the cookie data. The process of transferring an existing cookie from the client to the server is straight forward. In the following scenario the client web browser already possesses a cookie for the domain `secure.example.org`. The loading of a webpage for which a cookie has been set consists of the following steps (see figure 1.a):

1. The client's web browser sends an http request for `www.example.org/index.ext` (RQ1).
2. The web server replies with a small HTML page that only contains the PageLoader (RP1).
3. The PageLoader includes the `getCookie.ext` image in the DOM tree of the webpage. This causes the client's web browser to request the image from the server (RQ2). The cookie containing the SID that is stored for `secure.example.org` is included in this request automatically.
4. The PageLoader also requests the webpage's body using the XMLHttpRequest object (RQ3). This http request happens parallel to the http request for the `getCookie.ext` image.
5. The web server waits with the answer to RQ3 until it has received and processed the request for the `getCookie.ext` image. According to the cookie data that this request contained, the web server is able to compute and send the webpage's body (RP2).
6. The PageLoader receives the body of the webpage and uses the `document.write` method to display the data.

The web server has to be able to identify that the last two http requests (RQ2 and RQ3) where initiated by the same PageLoader and therefore came from the same client. For this reason the PageLoader uses a request ID (RID) that is included in the URLs of the request RQ2 and RQ3. The RID is used by the web server to synchronize the request data between the domains `www` and `secure`.

Setting a cookie: The usually preceding process of transferring existing cookie data from the client to the server, as described above, is left out for brevity. With this simplification the setting of a cookie consists of the following steps (see figure 1.b):



a. Getting a cookie b. Setting a cookie

Fig. 2. schematic view of the processes

1. The client's web browser sends an http request for `www.example.org/index.ext` (RQ1).
2. The web server replies with the PageLoader (RP1) and the PageLoader subsequently requests the body data (RQ2).
3. The web server computes the request RQ2. Because of the outcome of the computation the server decides to place a cookie. The server replies with "`SETCOOKIE`" to the PageLoader's request for the body data (RP2).
4. The PageLoader receives the "`SETCOOKIE`" token and includes the `setCookie.ext` image in the DOM tree of the webpage. This causes the client's web browser to request the image from the server (RQ3).
5. The PageLoader also requests the webpage's body once more (RQ4). This http request happens parallel to the http request for the `setCookie.ext` image.
6. The web server receives the request for the image and includes the cookie data in the response (RP3). The web server marks the RID as "used" (see below).
7. The web server waits with the answer to RQ4 until it successfully delivered the `setCookie.ext` image to the client. After the image request has been processed the body data gets sent (RP4).

There is an important timing aspect to take into consideration: The PageLoader should not display the HTML body data before the cookie setting process is finished, and the web server should never reply more than once to a `setCookie.ext` request containing the same RID value. Otherwise, the security advantage of the proposed method would be lost, because after the HTML body data is displayed in the client's browser a malicious JavaScript might be executed. This script then could read the DOM tree to obtain the full URL of the `setCookie.ext` image and communicate this information via the internet to the

attacker. If at this point of time the web server still treats this image URL (more precise: the RID value) as valid, the attacker would be able to successfully request the image including the cookie data from the web server. If no invalidation of the RID happens, the described technique will only shift the attack target from losing the cookie value to losing the RID value. For the same reason the RID value must be random and of sufficient length in order to prevent guessing attacks. Because of the restrictions posed by the same-origin policy, the cookies stored for `secure.example.org` are not accessible by JavaScript embedded into a page from `www.example.org`. Furthermore, JavaScript is not allowed to change `document.domain` to `secure.example.org` because this value is not a valid domain suffix of the original host value `www.example.org`. The `secure` subdomain only contains the two specified server scripts for cookie transportation. The reply data of these server scripts does not contain any dynamic data. Thus, an XSS attack on `secure.example.org` is not feasible. Therefore, the proposed technique successfully prevents cookie stealing attacks without limiting cookie usage.

4.2 One-Time URLs

To defend against browser hijacking (see 3.2) we have to remove the fundamental basis of this attack class. Every browser hijacking attack has 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 which cannot be known, obtained, or guessed 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.

To determine the requirements for successful URL hiding we have to examine the abilities of rogue JavaScript. The secret URL component has to satisfy the following limitations:

- It has to be unguessable.
- It must not be stored in an HTML element, e.g. a hidden form field. JavaScript can access the DOM tree and therefore is able to obtain any information that is included in the HTML code.
- It must not be stored in public JavaScript variables. All JavaScript code in one webpage exists in the same namespace. Therefore, a malicious script is able to execute any existing JavaScript function and read any available public variable.
- It must not be hard coded in JavaScript. Every JavaScript element (i.e. object, function or variable) natively supports the function `toString()` which per default returns the source code of the element. Malicious script could use this function to parse code for embedded information.

– It has to be valid only once. Otherwise, the attacker's script could use the value of `document.location` to emulate the loading process of the displayed page.

Thus, the only place to keep data protected from malicious JavaScript is a private variable of a JavaScript object. In the following paragraphs we show how this approach can be implemented. We only describe this implementation in respect of randomizing hyperlink URLs. The randomization of HTML forms is left out for brevity - the applicable technique is equivalent.

The URLRandomizer Object: Our 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 we introduce the URLRandomizer, a JavaScript object included in every webpage. As introduced above, the URLRandomizer object contains a private variable that holds all valid randomization data. During object creation the URLRandomizer requests from the web server a list of valid nonces for the webpage's 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 XSS attacks. The URLRandomizer object also possesses a privileged method called "go()" that has the purpose to direct the browser to new URLs. This method is called by hyperlinks that point to URLs that require randomization:

```
<a href="#" onclick="URLRandomizer.go('placeOrder.ext');">Order</a>
```

The "go()" method uses the function parameter and the object's private randomization data to generate a URL that includes a valid rnonce. This URL is immediately assigned to the global attribute `document.location` causing the client's web browser to navigate to that URL. Listing 1 shows a sketch of the URLRandomizer's go() function. In this code "validNonces" is a private hashtable containing the valid randomization data.

```
this.go = function(path){
    nonce = validNonces[path];
    document.location =
         "http://www.example.org/"+path+"?rnonce="+nonce;
}
```

Listing 1.1. sketch of the URLRandomizers go() function

Timing aspects: As mentioned above, the URLRandomizer obtains the valid randomization data from the server by requesting it via http. This leads to the following requirement: The URL that is used to get this data also has to be randomized and limited to one time use. It is furthermore important, that the URLRandomizer object is created early during the HTML parsing process and that the randomization data is requested on object creation. Otherwise, malicious JavaScript could examine the source code of the URLRandomizer to obtain the URL for the randomization data and request it before the legitimate

object does. As long as the definition and creation of the URLRandomizer object is the first JavaScript code that is encountered in the parsing process, this kind of timing attack cannot happen.

Entering the randomized domain: It has to be ensured that the first webpage, which contains the URLRandomizer object, was not requested by a potential malicious JavaScript, but by a proper user of the web application. Therefore, an interactive process that cannot be imitated by a program is required for the transition. The natural solution for this problem is combining the changeover to one-time URLs with the web application's authentication process. In situations where no authentication takes place CAPTCHA (Completely Automated Public Turing-Test to Tell Computers and Humans Apart) technology [23] can be employed for the transition. If no interactive boundary exists between the realms of static and one-time URLs, a malicious JavaScript would be able to request the URL of the entry point to the web application and parse its HTML source code. This way the script is able to acquire the URL that is used by the URLRandomizer to get the randomization data.

Disadvantages of this approach: The proposed method poses some restrictions that break common web browser functionality: Because it is forbidden to use a random nonce more than once, the web server regards every http request that includes a invalidated nonce as a potential security breach. Depending on the security policy such a request may result in the termination of the authenticated session. Therefore, every usage of the web browser's "Back" or "Reload" buttons pose a problem because these buttons cause the web browser to reload pages with invalid nonces in their URLs. A web application using one-time URLs should be verbose about these restrictions and provide appropriate custom "Back" and "Reload" buttons as part of the application's GUI. It is also impossible to set bookmarks for URLs that lie in the randomized area of the web application, as the URL of such a bookmark would contain an invalid random nonce. Other issues, e.g. the opening of new browser windows, can be solved using DHTML techniques. Because of the described restrictions, a limitation on the usage of one-time URLs for only security sensitive parts of the web application may be recommendable.

Alternative solutions: Some of the limitations mentioned above exist because the proposed URLRandomizer object is implemented in JavaScript. As described above the separation of two different JavaScript objects running in the same security context is a complex and limited task. Especially the constraint that a random nonce can be used only once is due to the described problems. An alternative approach would be using a technology that can be separated cleanly from potential malicious JavaScript. There are two technologies that might be suitable candidates: Java applets [22] and Adobe Flash [1]. Both technologies have characteristics that suggest that they might be suitable for implementing the URL randomizing functionality: They provide a runtime in the web browser for client side code which is separated from the JavaScript runtime, they possess interfaces to the web browser's controls, they are able to export functionality to JavaScript routines and they are widely deployed in today's web browsers. Before

implementing such an solution, the security properties of the two technologies have to be examined closely, especially in respect of the attacker's capability to include a malicious Java or Flash object in the attacked web page.

4.3 Subdomain Switching

The underlying fact which is exploited by the attacks described in Section 3.3 is, that webpages with the same origin implicitly trust each other. Because of this circumstance rogue iframes or background windows are capable of inserting malicious scripts in pages that would not be vulnerable otherwise. As years of security research have taught us, implicit trust is seldom a good idea - instead explicit trust should be the default policy. To remove this implicit trust between individual webpages that belong to the same web application, we have to ensure that 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 achieve this trust removal, we introduce additional subdomains to the web application. These subdomains are all mapped to the same server scripts. Every link included into a webpage directs to a URL with a subdomain that differs from the domain of the containing webpage. For example a webpage loaded from `http://s1.www.example.org` only contains links to `http://s2.www.example.org`. Links from `s2.www.example.org` would go to `s3.www`... and so on. 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 subdomain.

Tracking subdomain usage: As mentioned above, all added subdomains map to the same server scripts. Therefore, the URL `http://s01.www.example.org/order.ext` points to the same resource as for example the URL `http://s99.www.example.org/order.ext`. The subdomains 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 which mapping between URLs and subdomains have been assigned to a user's session.

Implementation aspects: The implementation of the subdomains is highly dependent on the application server used. For our implementation we used the Apache web server [16] which allows the usage of wildcards in the definition of subdomain names. Consequently, we had unlimited supply of applicable subdomain names. This allows the choice between random subdomain names or incrementing the subdomain identifier (`s0001.www` links to `s0002.www` which links to `s0003.www` and so on). On application servers that do not offer such an option and where therefore the number of available subdomain names is limited, the web application has to be examined closely. It has to be determined how many subdomains are required and how the mapping between URLs and sub-

domains should be implemented. These decisions are specific for each respective web application.

5 Discussion

5.1 Combination of the Methods

Before implementing the countermeasures described in Section 4, the web application's security requirements and environment limitations have to be examined. A combination of all three proposed methods provides complete protection against all known session hijacking attacks: The Deferred Loading Process prevents the unauthorized transmission of SID information. Subdomain Switching limits the impact of XSS vulnerabilities to only the vulnerable pages. Furthermore Browser Hijacking attacks that rely on the attacker's capability to access the content of the attack's http responses are also prevented as the XMLHttpRequest object is bound by the same origin policy: With Subdomain Switching in effect the attacking script would have to employ iframe or image inclusion to create the attack's http request. One-Time URLs prevent all Browser Hijacking attacks. Furthermore Session Riding [20] attacks would also be impossible as this attack class also relies on the attacker's prior knowledge of the application's URLs. It is strongly advisable to implement all three methods if possible. Otherwise, the targeted security advantage might be lost in most scenarios.

5.2 Limitations

As shown above, a combination of the countermeasures protect against the session hijacking attacks described in Section 3. However, on the actual vulnerable page in which the XSS code is included, the script still has some capabilities, e.g altering the page's appearance or redirecting form actions. Thus, especially webpages that include HTML forms should be inspected thoroughly for potential weaknesses even if the described techniques were implemented.

The described techniques are not meant to replace input checking and output sanitation completely. They rather provide an additional layer of protection to mitigate the consequences of occurring XSS vulnerabilities.

5.3 Transparent Implementation

An implementation of the proposed methods that is transparent to existing web applications is desirable. Such an implementation would allow to protect legacy applications without code changes.

Deferred Loading: There are no dependencies between the deferred loading process and the content of the application's webpages. Therefore, a transparent implementation of this method is feasible. It can be realized using an http proxy positioned before the server scripts: The proxy intercepts all incoming and outgoing http messages. Prior to transferring the request to the actual server scripts,

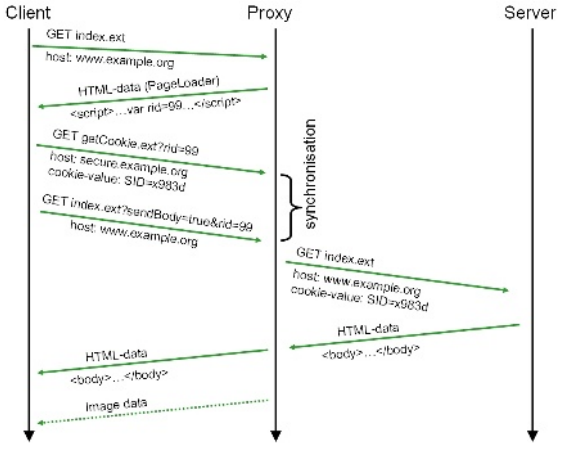


Fig. 3. Transparent implementation of the "get cookie" process

the "get cookie" process is executed (see figure 3). Before sending the http response to the client, all included cookies are stripped from the response and send to the client via the "set cookie" process.

One-Time URLs and Subdomain Switching: We only describe the problems in respect to One-Time URLs. Most of these issues also concern Subdomain Switching, while Subdomain Switching does not pose additional difficulties. A transparent implementation of these methods also would employ proxy like functionality. All incoming requests are examined whether their URLs are valid, i.e. contain a valid random nonce. All outgoing HTML data is modified to use the specified URLs. Implementing such a proxy is difficult because all application local URLs have to be rewritten for using the randomizer object. While standard HTML forms and hyperlinks pose no special challenge, prior existing JavaScript may be harder to deal with. All JavaScript functions that assign values to `document.location` or open new windows have to be located and modified. Also all existing `onclick` and `onsubmit` events have to be rewritten. Furthermore, HTML code might include external referenced JavaScript libraries, which have to be processed as well. Because of these problems, a web application that is protected by such a solution has to be examined and tested thoroughly. Therefore, an implementation of the proposed methods as a library for hyperlink and form creation is preferable.

5.4 Future Work

It still has to be specified how the proposed methods can be integrated into established frameworks and application servers. Such an integration is the prerequisite for examinations concerning performance issues and backwards compatibility. Recently we finished developing a transparent solution as described in Section 5.3 for the J2EE framework [24]. This implementation will be the basis for further investigations .

6 Related Work

The first line of defense against XSS attacks is input filtering. As long as JavaScript code is properly stripped from all user provided strings and special characters are correctly encoded, XSS attacks are impossible. However, implementing correct input filtering is a non-trivial task. For example, in 2005 an XSS input filter was added to the PHPNuke content management system that still was vulnerable against numerous known XSS attack vectors [3].

Scott and Sharp [21] describe an application level firewall which is positioned in front of the web server. The firewall's ruleset is defined in a specialized security policy description language. According to this ruleset incoming user data (via POST, GET and cookie values) are sanitized. Only requests to URLs for which policies have been defined are passed to the web server. The Sanctum AppShield Firewall is another server side proxy solution [13]. AppShield executes default filter operations on all user provided data in order to remove potential XSS attacks. Opposed to Scott and Sharp's approach, AppShield requires no application specific configuration which makes it easy to install but less powerful.

Kirda et al. proposed "Noxes", a client side personal firewall [12]. Noxes prevents XSS induced information leakage, e.g. stealing of cookie data, by selectively disallowing http requests to resources that do not belong to the web application's domain. The firewall's ruleset is a combination of automatically constructed and manually configured rules. Noxes does not offer protection against bowser hijacking attacks.

"Taint analysis" is a method for data flow tracking in web applications. All user controlled data is marked as "tainted". Only if the data passes sanitizing functions its status will change to "untainted". If a web application tries to include tainted data into a webpage a warning will be generated. Taint analysis was first introduced by Perl's taint mode [15]. In 2005 Huang et al. presented with WEBSSARI a tool that provides static taint analysis for PHP [10].

Microsoft introduced an "http only" option for cookies with their web browser Internet Explorer 6 SP1 [17]. Cookies that are set with this option are not accessible by JavaScript and therefore safe against XSS attacks. The http only option is not standardized and until now there are no plans to do so. It is therefore uncertain if and when other web browsers will implement support for this option.

7 Conclusion

In this paper we presented SessionSafe, a combination of three methods that successfully prohibits all currently known XSS session hijacking attacks.

To achieve this, we classified currently known methods for session hijacking. Through a systematic examination of the resulting attack classes, we identified the basic requirements for each of these attack methodologies: SID accessibility in the case of Session ID Theft, prior knowledge of URLs in the case of Browser Hijacking and implicit trust between webpages in the case of Background XSS Propagation.

There are only two instruments provided by the web browser architecture that can be used to enforce access restrictions in connection with JavaScript: The same-origin policy and private members in JavaScript objects. Using the knowledge gained by the classification, we were able to apply these security mechanisms to remove the attack classes' foundations: To undermine the SID accessibility, the SID is kept in a cookie which belongs to a different subdomain than the main web application. To achieve this, we developed a deferred loading process which allows to execute the cookie transport while the web page is still in a trustworthy state. To undermine the pre-knowledge of the application's URLs, valid One-Time URLs are hidden inside private members of the URL-Randomizer JavaScript object. Finally, additional subdomains are introduced by Subdomain Switching, in order to create a separate security domain for every single webpage. This measure employs the same-origin policy to limit the impact of XSS attacks to the vulnerable pages only. Consequently, each proposed countermeasure removes the fundamental necessities of one of the attack classes, hence disabling it reliably. By preventing session hijacking, a large slice of the attack surface of XSS can be removed.

The proposed countermeasures do not pose limitations on the development of web applications and only moderate restrictions on web GUI functionality. They can be implemented as an integral component of the application server and thus easily be integrated in the development or deployment process.

References

1. Adobe flash. Website <http://www.adobe.com/products/flash/flashpro/>.
2. Wade Alcorn. The cross-site scripting virus. Whitepaper, <http://www.bindshell.net/papers/xssv/xssv.html>, September 2005.
3. Maksymilian Arciemowicz. Bypass xss filter in phpnuke 7.9. mailing list BugTraq, <http://www.securityfocus.com/archive/1/419496/30/0/threaded>, December 2005.
4. CERT/CC. Cert® advisory ca-2000-02 malicious html tags embedded in client web requests. [online], <http://www.cert.org/advisories/CA-2000-02.html> (01/30/06), February 2000.
5. Douglas Crockford. Private members in javascript. website, <http://www.crockford.com/javascript/private.html>, last visit 01/11/06, 2001.
6. ECMA. Ecmascript language specification, 3rd edition. Standard ECMA-262, <http://www.ecma-international.org/publications/standards/Ecma-262.htm>, December 1999.
7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, June 1999.
8. David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, November 2001.
9. Jeremiah Grossman. Phishing with super bait. Presentation at the Black Hat Asia 2005 Conference, <http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-grossman.pdf>, October 2005.

10. Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.
11. Philippe Le Hégaret, Ray Whitmer, and Lauren Wood. Document object model (dom). W3C recommendation, <http://www.w3.org/DOM/>, January 2005.
12. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks, security. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.
13. Amit Klein. Cross site scripting explained. White Paper, Sanctum Security Group, <http://crypto.stanford.edu/cs155/CSS.pdf>, June 2002.
14. D. Kristol and L. Montulli. Http state management mechanism. RFC 2965, <http://www.ietf.org/rfc/rfc2965.txt>, October 2000.
15. LarryWall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.
16. Ben Laurie and Peter Laurie. *Apache: The Definitive Guide*. O'Reilly, 3rd edition, December 2002.
17. MSDN. Mitigating cross-site scripting with http-only cookies. website, <http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp>, last visit 01/23/06.
18. Anton Rager. Xss-proxy. website, <http://xss-proxy.sourceforge.net>, last visit 01/30/06, July 2005.
19. Samy. Technical explanation of the myspace worm. website, <http://namb.la/popular/tech.html>, last visit 01/10/06, October 2005.
20. Thomas Schreiber. Session riding - a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH, <http://www.securenet.de/papers/Session_Riding.pdf>, December 2004.
21. D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of 11th ACM International World Wide Web Conference*, pages 396 – 407. ACM Press New York, NY, USA, 2002.
22. Sun. Java. Website <http://java.sun.com/>.
23. L. von Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *Proceedings of Eurocrypt*, pages 294–311, 2003.
24. Christian Weitendorf. Implementierung von maßnahmen zur sicherung des web-session-managements im j2ee-framework. Master's thesis, University of Hamburg, 2006.