KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Improving session security in web applications

Bram Bonné

Thesis voorgedragen tot het
behalen van de graad van Master
in de ingenieurswetenschappen:
computerwetenschappen

Promotoren:
dr. L. Desmet
Prof. dr. F. Piessens

Assessoren:
Ir. P. Philippaerts
G. Poulisse

Begeleiders:
P. De Ryck
N. Nikiforakis

Academiejaar 2010 – 2011

Preface

I'd like to thank Lieven Desmet and Frank Piessens, my promotors, and Philippe De Ryck and Nick Nikiforakis, my supervisors, for suggestions and feedback. Lastly, thanks go out to all people who tested the extension.

Bram Bonné

Contents

Abstract

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.

Samenvatting

Nederlandstalig abstract

List of Figures and Tables

List of Figures

List of Tables

Glossary

cookie A piece of text made up of one or more name-value pairs stored by a website in the browser. Cookies are set by a website upon first visit, and sent by the browser on every request.. 3

CSRF Cross-Site Request Forgery, an attack in which JavaScript is used to execute malicious code in the victim's browser. 17

DOM Document Object Model, used by programming languages like JavaScript to access elements in an XML document or on a web page. vii, 5, 13

HTTP HyperText Transfer Protocol, the protocol which is used by web browsers to communicate with web servers. 3

HTTPS HyperText Transfer Protocol Secure, a secure variant of HTTP, which uses encryption when transmitting data. viii

LCG Linear Congruential Generator, a generator for pseudorandom numbers based on recurrence. 6

MAC Message Authentication Code, a short piece of information providing data integrity and authenticity for a message. 7

MitM Man in the Middle, used to describe an attacker that is able to intercept traffic between the victim and some other principal. 13

session cookie A cookie containing a session identifier. 4, 16

session fixation An attack in which the attacker tries to enforce a SID upon the victim, so he is able to take over the session later on. 5, 17

session hijacking An attack in which the attacker tries to capture the victim's SID, to be able to take over his session. 4, 14, 17

session ID Session Identifier, a unique string of text used by the server to identify a user. 3, 12, 14

SID see 'session ID'. vii, 3, 18

SOP Same Origin Policy, the policy which states that DOM objects may only be accessed by principals having the same origin as the object. 5, 12

SSL Secure Socket Layer, used in HTTPS to encrypt traffic between a client and a server. 18

URL Uniform Resource Locator, a string that specifies the location of a resource. 4, 15

XSS Cross-Site Scripting, an attack whereby the attacker injects JavaScript into a legitimate page that will be executed by another user's browser when the user visits the page. 6, 9, 13, 15, 17

Chapter 1

Introduction

Chapter 2

About web sessions

When a user visits a website, it is often needed for the web application to remember which user it is interacting with. For this reason, the concept of a 'web session' was invented. In this chapter we will see how web sessions work, and how their security can be improved.

2.1 How web sessions work

To understand the need for web sessions, we first have to look at how web browsers communicate with web servers. The browser requests web pages by issuing HTTP requests to a web server [34]. The server subsequently responds to every request with an HTTP response containing the requested page. Unfortunately, HTTP is a stateless protocol, which means that the web server has no way of knowing whether two different requests come from the same user. Because of this, a mechanism is needed on top of HTTP to enable stateful communication between a web server and a client. This mechanism is known as a web session.

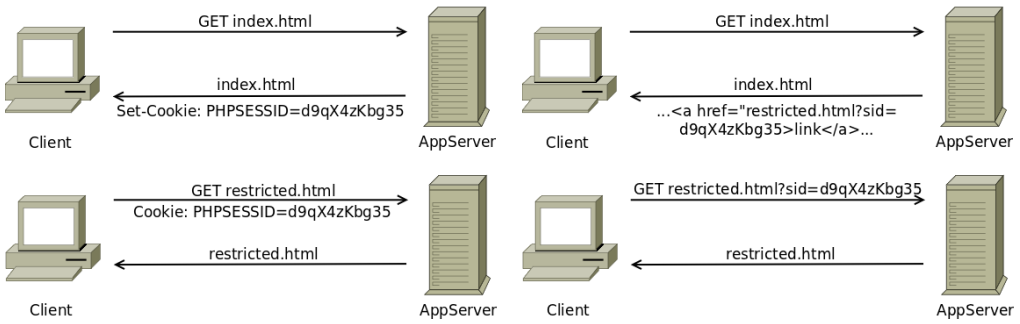
Web sessions work as follows:

1. When the web server receives its first HTTP request from a particular client, it creates a *session identifier* (also called a session ID or SID) that it associates with this client. It then sends the newly generated SID to the client as part of the HTTP response.

2. In subsequent communications, the client attaches the SID it received to every HTTP request it issues to the server. Because the web server has associated this SID with a particular client, it will know who it is interacting with.

There are three ways in which session identifiers can be attached to requests and responses [26]. The first one, which is most common, makes use of cookies [33, 42]. Cookies are strings consisting of multiple name-value pairs which are set by the web server using the `Set-Cookie` HTTP header. Upon receiving a cookie, the browser stores it for a specified amount of time. It then attaches this cookie to

every subsequent request made to the server[1] that set the cookie by using HTTP's `Cookie` header. This process is graphically depicted in Figure 2.1a. It is clear that cookies are a very convenient mechanism for managing session identifiers. Because cookies can also be used for other purposes than session management, we will make a distinction between *cookies* (which can be used for all sorts of state information) and *session cookies* (which are cookies that store a SID) in this text.



(a) via cookies (b) via URL rewriting

Figure 2.1: Session management

The second possibility to include session information is via *URL rewriting*. In this case, the web server appends the session ID as a parameter to every served URL that points to a page on the same server. Thus, when the user clicks a link on the served page, the request that is made contains the session ID as a parameter, and the server will know who made the request. This process is graphically depicted in Figure 2.1b.

The third possibility is very similar to URL rewriting, but uses POST instead of GET parameters [22]. Here, the session ID is included as a `<form>` element. When the user submits the form, the session ID is sent along with the request.

In this text, we will mostly focus on SIDs in cookies, because they are by far the most common. However, when appropriate, we will include information about session IDs in URLs.

There are two important things to note about session IDs. Firstly, there is no standardized way of doing session management. This means that different web applications will use different SID names, and that they will generate SID values in different ways. Secondly, SIDs are almost[2] always used exclusively for authentication. This means that the SID only identifies the client, while all other state is saved at the server side. The server then associates the SID with the other state information stored for that particular session.

[1] The actual access control policy is a bit more complicated and will be discussed in section 2.2.

[2] We use the term 'almost' because there is no standardized way of using SIDs. It can however safely be assumed that the majority of the pages encountered on the Web will use SIDs for authentication only.

2.2 Accessibility of session identifiers

In this section, we look into the ways session identifiers can be accessed, and who is allowed to access them. This will be important when we look at the session hijacking and session fixation attacks in chapter 3, where the attacker tries to access a victim's SID.

The access policy for cookies states that a cookie is sent only to those pages that have the same (sub)domain as the page that set the cookie. Moreover, the page to which a cookie is sent must be on a path that is a suffix of the page that set the cookie [47]. For SIDs that make use of URL rewriting, the situation is different. Here, the SID is only attached to requests that are the result of the user clicking a link on a web page, and only when the originating web page explicitly included the SID as a parameter in the link.

SIDs can also be accessed at the client side. For this, the *Document Object Model* (or *DOM*) is used. The Document Object Model provides a way for programming languages like JavaScript to access elements on a web page. It allows an SID to be accessed both when it stored in a cookie (via the `document.cookie` property) as when a link containing the SID is present in a web page (directly via the `href` attribute of the `<a>` element containing the link). For DOM objects, the *Same Origin Policy* (or *SOP*) is enforced [47]. This policy states that web pages that want to access a certain DOM object must have the same origin as this object. The origin is defined as the tuple `<protocol,domain,port>`. Thus, the Same Origin Policy essentially allows only access to DOM objects which are on the same domain as (or on a subdomain of) the principal trying to access them. Moreover, it is required that the accessing principal uses the same protocol and port as the protocol and port where the DOM object originated from.

2.3 Keeping web sessions secure

It is important to ensure that a session identifier may only be known by the web server and the client that is identified by it. If an attacker is able to know a client's SID, he can use it to impersonate the client in the web application (which is exactly what the session attacks we will describe in chapter 3 try to do). Because of this, we describe in this section the properties a secure SID should possess.

2.3.1 Unguessable by an attacker

An attacker should not be able to guess the value of a user's session ID. For this, the following properties are necessary:

Randomness

To be unguessable, a session identifier should appear like it could be any random string of text. Random in this case means that session identifiers should have [40, 14, 13]:

high entropy The higher the number of bits that are necessary to represent a string, the higher the string's entropy is.

low correlation If SIDs are correlated, an attacker is able to derive (part of) SIDs that will be generated from SIDs which were already generated. As a consequence, an attacker would be able to predict a victim's session ID from his own session ID.

a high number of possible values This is related to both high entropy, and to sufficient length (which will be discussed shortly hereafter).

It must also be noted that it is not sufficient for session IDs to be only statistically random: they have to be cryptographically random [16]. This means that more than just an LCG is needed to generate the SIDs.

Sufficient length

To prevent successful brute forcing attacks, wherein an attacker exhaustively tries lots of possible values, a session ID should be of sufficient length. The expected number of seconds required to guess any one valid session identifier is given by the equation [41]:

$$\frac{2^B + 1}{2A \cdot S}$$

where B is the number of bits of entropy in the SID, A is the number of guesses an attacker can try each second, and S is the number of valid SIDs at any given time.

OWASP recommends a session ID length of at least 128 bits [41].

2.3.2 Unavailable to an attacker

If an attacker is able to read the value of the victim's SID, the victim's session is compromised. Because of this, it is of utmost importance that the SID value is never visible to anyone but the web server and the legitimate user. In this section, we only describe some properties to achieve this. We will go into more detail about making cookies unavailable to an attacker in section 4.3.

SIDs that are set via cookies are, by default, transmitted as clear text. This means that, in an insecure channel like the Internet, an eavesdropper is able to intercept the cookie value. An eavesdropping attacker can thus take over a user's session. Cookies can also be sent over a TLS connection [12]. In this case, the requests and responses (and thus, the cookie value) are encrypted, and the attacker is unable to extract the cookie wen viewing one of these messages. Care must be taken, then, that the cookie value is never sent in the clear; otherwise its value can still be compromised. To ensure that this is the case, the `secure` flag should be enabled when setting a cookie [16, 32]. This flag demands that the browser may never send the cookie over an unencrypted connection. In section 4.1.1, we will go into more detail about secure connections.

When an SID which is set via a cookie will never need to be accessed via JavaScript, it is best to enable the `HttpOnly` flag when setting the cookie [40]. When

this flag is enabled, the browser will only allow the cookie to be accessed via HTTP. Accessing a cookie set using the `HttpOnly` flag via JavaScript will be disallowed. This prevents attackers from using XSS attacks (more about XSS attacks in section 3.1) to capture the SID value. With URL rewriting, it is not possible to state that the SID may not be available to client-side JavaScript. Indeed, a link containing the SID will always be available as a DOM object, making the SID vulnerable to XSS attacks. In section 4.1.1, we will go into more detail about `HttpOnly` cookies.

2.3.3 Short lifetime

A third way of making sessions more secure is to ensure that they have a limited lifetime [16]. This has two advantages:

- The shorter the lifetime of a SID, the less time an attacker has to brute force it.
- In case the attacker was able to capture the SID, the lifetime of the SID determines how long the attacker is able to use it for impersonating the victim.

Additionally, limiting the lifetime is needed because a user can not be relied upon to manually log out every time he stops using a web application [41].

Limiting the SID's lifetime

The lifetime of a cookie can be limited by using the `expires` attribute when setting the cookie. However, care must be taken that the session is also expired at the server side, because the attacker could otherwise still use the cookie after it expired at the victim's browser [31].

Another way of limiting a SIDs lifetime is presented by K. Fu et al. [16]. By including a timestamp as part of the SID value, a server can determine whether the SID is still valid without having to save the SID's expiration time as part of its state. It is important to note that this requires the SID value to be signed by the server (using a MAC). Otherwise, an attacker could just change the timestamp part of the SID to extend its lifetime. A disadvantage of this approach is that there is no possibility of revoking SIDs without keeping extra server state.

Renewing the SID

Fortunately, the user does not have to re-authenticate every time he gets a new session ID. Only three steps have to be performed by the server to provide a client with a new SID:

1. Generate a new session ID.
2. Associate the new SID with the existing user. For this, any server side state that was attached to the old SID should be attached to the new SID instead.

3. Make sure the client uses the new SID instead of the old one. This can be done by attaching a `Set-Cookie` header to the next HTTP response, containing the new SID value. The browser will notice that the cookie name is the same as the old SID, and will therefore update the old value. Alternatively, when URL rewriting is used, the web server has to make sure that all links on subsequent web pages served to this client include the new SID value.

2.3.4 Other best practices

Problems occur in lots of web applications because they implement their own session management. As we will see in section 4.1.2, lots of web frameworks have thoroughly tested session management already built-in. Because it is very easy to make mistakes when providing security, it is recommended to make use of an existing session management framework.

Secondly, K. Fu et al. argue that the use persistent cookies should be avoided when using cookies for authentication [16]. Persistent cookies are cookies that are saved on disk at the client-side, instead of just being saved in main memory. The security advantage non-persistent cookies have over persistent cookies is that they can not leak via file access on the system. To create a cookie that is non-persistent, the `expires` field should be omitted.

Chapter 3

Session attacks

Now that we know how web sessions work, we look into ways in which attackers are able to abuse them. In this chapter, we will see how an attacker can abuse the web sessions concept to impersonate a legitimate user on a website.

3.1 Background: cross-site scripting

Cross-site scripting (or XSS) is not a session attack in itself. Instead, it is the exploitation of a vulnerability in the way user input is handled within certain web applications. The attack can be of great use when executing one of the actual session attacks described afterwards.

In a cross-site scripting attack, the attacker exploits a vulnerability in a website to inject his own JavaScript code into this page [11]. The code will then be executed in the browser of any user that loads the page with the injected code.

3.1.1 Variants of cross-site scripting

There are two forms of cross-site scripting: persistent and non-persistent XSS. In this subsection, we describe their differences and give some examples of possible attack scenarios for each of them.

Persistent (stored) XSS

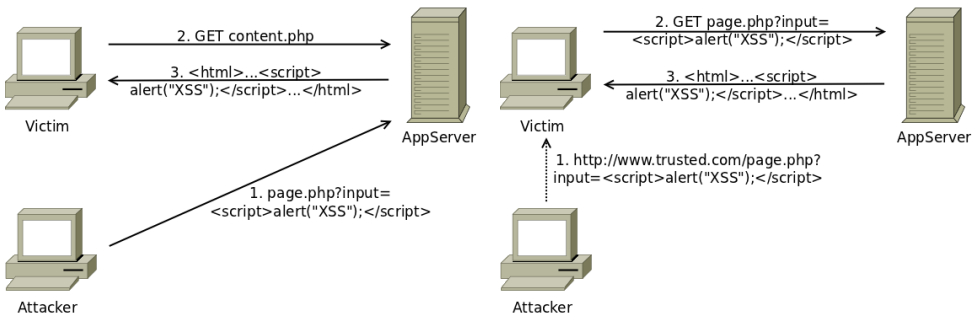
In a persistent attack, the attacker makes the web application store the script code in a database. The complete scenario goes as follows:

1. The attacker provides his malicious code as input to the web application. The web server stores this input (and thus the script code) in the database to be able to provide it to clients later on.

2. The victim requests a page containing content generated by the attacker. The server returns the page containing the attacker's JavaScript.

3. The victim's browser thinks that the script code is part of the requested web page and executes it.

This attack variant is graphically illustrated in figure 3.1a.



(a) persistent (b) non-persistent

Figure 3.1: The cross-site scripting attack

An example of a web application that will store user input in a database to serve it to other clients later on is a bulletin board (phpBB[1] is a well-known example). In such a web application, a user can create forum posts which can be read by other users. When an attacker inserts script code into a forum post, the code will be executed by the browser of every user that reads the post.

A more recent example is found in social networking sites like Facebook[2]. Here, a user can place information on his own page, or on the pages of other people. This information will then be served to the user's friends.

Non-persistent (reflected) XSS

In a non-persistent attack, the script code is never stored at the server side. Instead, code that was part of a request is immediately reflected back to a client's browser as part of the response page. The complete scenario is as follows:

1. The attacker tricks the victim into opening a malicious link containing script code.

2. The victim opens the link, and unknowingly makes a request containing script code to the server.

3. The server reflects this script code back to the victim as part of the response.

4. The victim's browser thinks that the script code is part of the requested web page and executes it.

[1] http://www.phpbb.com/
[2] http://www.facebook.com

This attack variant is graphically illustrated in figure 3.1b.

An example of this is a search form, where a website returns the query the user searched for, without stripping any script data that might be inside [51]. If doing a search by going to the URL `http://www.trusted.com/search?q=query` makes the website return "x results for *query*", the attacker can replace *query* by script code that will be executed by the browser of the client that opens the link. The attacker can then trick a user into clicking the malicious link, causing the JavaScript code to be executed in his browser.

Another example of a scenario where a web application might return data present in the URL is the 'Page not found' error page [28]. Here, the name of the page that could not be found is often included as part of the error message. Thus, an attacker can craft a link wherein a page with name `<script>alert("XSS succeeded");</script>` is requested. The browser of a client opening this link will then execute the JavaScript code.

3.1.2 Including the script code

There are multiple ways in which script code can be included in a web page. Since all of these can be leveraged by an attacker to inject his code, it is important that web developers are aware of all of them. T. Jim et al. give a good overview of different approaches to including JavaScript code [21]. We describe the most important ones here:

between `<script>` tags This is the most obvious way of including JavaScript in a web page. All code between these tags will be executed as soon as the browser encounters it.

using the `<script>` tag's `src` parameter This parameter can be set to point to an external piece of JavaScript. As such, an attacker is able to make a website load JavaScript code from another domain. This method of injecting script code gives the attacker the advantage that the actual injected string is shorter, bypassing message length limits. An even more important advantage for an attacker using this method is described in section 3.1.3.

as the URL of the background image An attacker can insert a tag similar to `<div style="background-image: url(javascript:alert('XSS');" />` or `<style>.bar{background-image:url("javascript:alert('XSS');");}</style>` (where `bar` is the class of an object in the page) to make the browser execute JavaScript, thinking it is loading the background image.

using the `onload` parameter of the `<body>` tag This tag is used for code that should be executed once the browser has completed loading the page.

It must also be noted that attackers can use different possible encodings to inject JavaScript, so as to circumvent server-side JavaScript checkers, while still being able to execute at the client side [21]. For example, the encoding of (parts of) a web page can be changed using the `encoding` and `charset` attributes of various HTML

tags [20]. Some browsers also allow strings to contain JavaScript that is split over multiple lines [21]. Lastly, JavaScript can be split over multiple `<![CDATA[...]]>` tags, making it much harder to detect.

3.1.3 The danger of cross-site scripting

One danger of persistent XSS is immediately clear: websites that do not belong to an attacker can still be abused by the attacker to execute malicious code at a user's browser. The user, thinking that a trusted site will only execute trusted code, can fall victim to the attacker without expecting it.

There is, however, a bigger problem which applies to both persistent and non-persistent XSS: the problem of cross-domain interactions. Normally, the attacker's code would be subject to the SOP (described in section 2.2), making him unable to access elements of a domain that he does not own. However, when the attacker's code is able to execute from within a trusted domain (as is the case in an XSS attack), this code may access the elements belonging to the trusted domain [29]. This gives the attacker the ability to read information from, and write information to, elements in the trusted domain.

The situation is even worse when the attacker's code is loaded from a different domain (because the attacker injected the script code by using the `<script>` tag's `src` attribute, for example). When this is the case, the script code is also allowed to communicate with the domain it was loaded from [47], i.e. the attacker's domain. Because of this, the attacker is able to capture information from a trusted domain and subsequently send this information to his own domain. This will be a major attack vector for both the session hijacking and the session fixation attacks, as we will see in the next sections.

3.2 Session hijacking

In the session hijacking attack, an attacker tries to take over a victim's session by capturing his session ID. By using this SID himself, the attacker can make the server think that he is the same person as the victim. First, we describe the attack scenario. Afterwards, we will see how the attacker can capture the victim's session ID.

3.2.1 Attack scenario

The session hijacking attack works as follows [40]:

1. The victim establishes a new session at the server. This is done automatically by the victim's browser either when he first visits the page or when he logs in (depending on the web application).

2. The attacker captures the session ID that corresponds to the victim's created session by using one of the methods described in section 3.2.2.

3. The attacker makes a request to the server, attaching the captured session ID as his own. Because the server has no other means of distinguishing between the victim and the attacker, it thinks that it is the victim that made the request. Thus, the attacker is now able to impersonate the victim at the server.

These steps are graphically illustrated in figure 3.2.

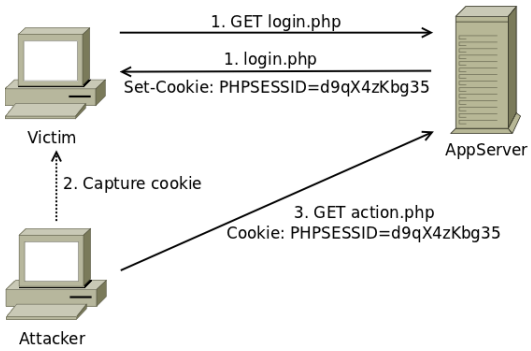


Figure 3.2: The session hijacking attack

3.2.2 Capturing the session ID

There are lots of possible ways in which an attacker can capture a victim's session ID. We list the most important ones here.

Via a (passive) man-in-the-middle attack

In a man-in-the-middle (or MitM) attack, an attacker is able to read traffic that passes between the victim and the server. This is usually the case because the victim and the attacker are using the same network unencrypted WiFi network, or because they share the same network hub.

As mentioned in section 2.3.2, the session identifier is, by default, sent over the network as an unencrypted string of text. This is the case both when using cookies as when making use of URL rewriting. Because of this, an attacker able to read all network traffic can easily extract the SID from the request URL or `Cookie` header in one of the user's requests, or even from the response page or the `Set-Cookie` header in the server response setting the cookie [1].

This problem has been known for some time, and has recently again received some attention thanks to tools like Hamster [18] and Firesheep [7], which make a session hijacking attack almost trivial when the victim is using an insecure network.

Via cross-site scripting

The XSS attack described in section 3.1 can also be used to steal a victim's session ID. For this, the attacker uses the `<script>` tag's `src` attribute to load script code from his own domain into a web page on the domain he wants to acquire the cookie

from. Because the script code is loaded from within a page on the target domain, the attacker has access to both the target domain's cookie via the `cookie` attribute of the `domain` DOM object (for SIDs set in cookies), and to all links on the current page (for SIDs used via URL rewriting). However, since the script code was loaded from the attacker's domain, it is also allowed to communicate to that domain (see section 3.1.3 and [29] for more information). Because of this, the attacker is able to forward the target domain's cookie, or an URL present on a page of the target domain, to his own domain, effectively capturing the client's SID.

Via the referer header

A third possible SID leak occurs when the browser attaches a `referer` header to requests going to a different domain than the one it originated from [16]. The `referer` header is used to transmit the website from which the request-URL was obtained [15]. If SIDs are included in the URL (as is the case with URL rewriting), the website receiving the `referer` header as part of the request can extract the client's SID for the web application that contained the link.

If, for example, a social networking website manages sessions via URL rewriting, an attacker could perform a session hijacking attack by sharing a link to his own website. When another user clicks the link, a request is made to the attacker's web server. This request contains the URL of the current page, and thus the user's SID for the social networking website, in the `referer` header, making it available to the attacker.

Sometimes, the `referer` header is suppressed in the network or by the browser, especially for cross-domain requests [4]. Unfortunately, the percentage of requests where the `referer` header is stripped is still small enough to consider leaking of SIDs via this channel a significant threat.

Via the user

Lastly, it is also possible that the user unknowingly leaks his own session identifier. This is the case when the user shares a link to a page of a web application that uses URL rewriting [25], for example via email or via a social networking site. When another user clicks the shared link, this second user will automatically take over the first user's session. Thus, the second user essentially performs a session hijacking attack on the first user, possibly without even realizing it.

This problem, together with the possibility of leaking SIDs via the `referer` header, provides a strong argument against using URL rewriting for session management. There are, however, also some advantages of using URL writing instead of cookies, as we will see in section 4.1.1.

3.3 Session fixation

In the session fixation attack, as in the session hijacking attack (section 3.2), an attacker's goal is to be using the same session as a victim. However, instead of

capturing the victim's session ID (as is the case with session hijacking), the attacker forces the victim to use a SID that is known in advance. We will first describe the steps necessary to execute this attack. Afterwards, we list the ways in which the attacker can force a victim to use a certain session ID.

3.3.1 Attack scenario

The session fixation attack works as follows [31]:

1. The *attacker* establishes a new session at the server. He does this by sending a request that doesn't include a SID, which will cause the server to attach a newly generated SID to the response. Some servers also accept random SIDs[3] [46]. In this case, the attacker can just make up a new SID, and no request needs to be made.

2. The attacker forces the victim to use the newly created session ID. We will see how this can be done in section 3.3.2.

3. The victim uses his credentials to log in at the server. The SID that was injected at the client's web browser will automatically be attached to the request. There now exists a session at the server, identified by the SID known by the attacker, in which the victim is logged in.

4. The attacker makes a request to the server, attaching the captured session ID as his own. This makes the server think that it is the victim that made the request. Thus, the attacker is now able to impersonate the victim at the server.

These steps are graphically illustrated in figure 3.3.

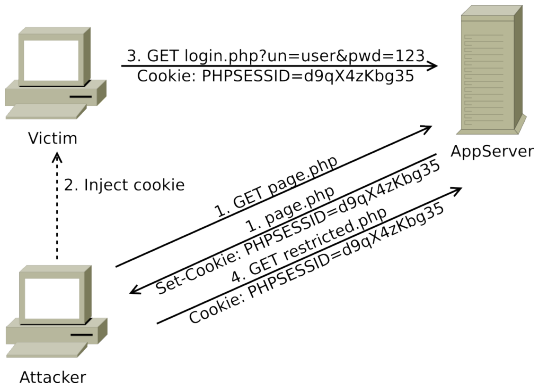


Figure 3.3: The session fixation attack

[3] By random SIDs, we mean that these SIDs don't need to be generated by the server in advance. A random SID can be sent by a user on his first request to make the server 'adopt' this SID for this user.

3.3.2 Injecting a session ID

In this section, we describe the most important attack vectors which can be used by an attacker to inject a session ID into the victim's browser.

Via GET or POST parameters

If the target website accepts session ID's in URLs (see section 2.1), the attacker can craft a link of the form `http://www.target.com/login.php?PHPSESSID=d9qX4zKbg35`, where he chooses the desired session ID [23]. He then sends this link to the victim, or places it on the target website as part of a XSS attack. When the victim clicks the link, the request sent to the target server contains the attacker's SID in the URL, causing the web application to think it is the victim's session ID. Alternatively, the attacker can force the victim to visit the URL by using a HTTP redirect [15] to make the victim's browser automatically load the page [46]. This requires the attacker to be able to perform a XSS attack, or the victim to visit the attacker's page.

In case the target website uses POST instead of GET parameters for session management, the link can be replaced by an automatically submitting form (as we will see in section 3.4.2) [31, 5].

Note that, for a website to accept session IDs via GET or POST parameters, it doesn't have to do its session management via URL rewriting by default. Indeed, multiple frameworks provide URL rewriting as a fallback for browsers that don't support cookies, causing them to be vulnerable to session fixation via URL rewriting [43, 9, 19].

Via cross-site scripting

If an attacker is able to inject script code into the target site (using one of the methods described in section 3.1.2), he can use this script code to set or replace the session cookie with the desired value. This is done by editing the `document`'s `cookie` property [31] or by using the `cookie.write()` function [23].

Via the `<meta>` tag

When an attacker is unable to inject script code (for instance, because `<script>` tags are stripped from user input), it is often still possible to abuse returned user input for setting cookies. Instead of using JavaScript, the attacker can use the HTML `<meta>` tag to set the cookie. For this, he injects the following line of HTML code into the target web page:

```
<meta http-equiv=Set-Cookie content="PHPSESSID=d9qX4zKbg35">
```

where the name and value of the session cookie are chosen by the attacker [31].

Via HTTP response splitting / header injection

There is another attack which can be used by an attacker to inject a cookie on the victim's machine. In a HTTP response splitting attack, the attacker tricks the

victim's browser (or an in-between proxy) into thinking that two HTTP responses were sent by the target server, whereas both are actually part of the same HTTP response [30]. The interesting part for the attacker is that the contents of one of the two responses can be chosen by himself. Because of this, an attacker is able to insert a `Set-Cookie` header containing the desired session identifier into the response, effectively injecting the SID into the client's browser [23].

Via subdomain cookie setting

Sometimes, an attacker is able to take over a subdomain of the target website because it is more vulnerable than the parent domain, or because he has legitimate access on the subdomain. Normally, setting a cookie on a subdomain would not have any effect on the parent domain. Indeed, as we described in section 2.2, cookies will only be used for the same domain as the one that set them, or for one of its subdomains. Thus, a cookie set on a subdomain will not be used for its parent domain. This makes it seem like performing a session fixation attack on a parent domain is not possible from within a subdomain.

There is, however, a possibility to set a cookie for a parent domain by using the cookie's `domain` parameter [32]. As an example, consider the case where the attacker was able to take over the domain `vulnerable.target.com`. He can then use this domain to set a cookie for `target.com`, and all of its subdomains, by specifying the cookie as `PHPSESSID=d9qX4zKbg35;domain=.target.com` (notice the '`.`' preceding `target.com`). This effectively allows an attacker that has access to a subdomain to execute a session fixation attack on a parent domain [31].

A related problem occurs when a cookie with the same name is set for both the parent domain and for subdomain. When the user visits a page on the subdomain, his browser will attach both the cookie for the parent domain and the cookie for the subdomain to the request. Unfortunately, since cookies don't contain the `domain` attribute when they are sent from the client to the server, the server has no way of distinguishing between these two cookies. What makes matters worse is that the order in which both cookies are sent differs between browsers. We tested the behavior of the Firefox, Opera and Chrome browsers, and found that:

- Firefox sends the subdomain cookie first, and the parent domain cookie second.
- Opera sends the parent domain cookie first, and the subdomain cookie second.
- Chrome sends the cookies in alphabetical order, regardless of their domain.

3.3.3 Other dangers of session fixation

It seems that the only benefit an attacker has from performing a session fixation attack is that he can take over the session of another user. There is, however, another issue: an attacker is also able to force a victim to be logged in under the attacker's account. He can do this by logging in first, and subsequently forcing the victim to use the attacker's session ID. This has two major implications [4]:

- An attacker can track the victim's actions on the target web application by making use of logging functionality offered by this application. For example, most major search engines offer the option to log the user's search history[4], allowing an attacker who is able to perform a session fixation attack to access this highly sensitive information [2].

- On domains that allow the embedding of trusted scripts, it creates the ability for an attacker to execute XSS attacks. Until recently, iGoogle[5] offered the ability to embed trusted scripts on your own personal homepage [4]. Because of this, an attacker who is able to perform a session fixation attack has the ability to offer scripts to a victim from within the `google.com` domain. He does this by adding a script to his own homepage, and subsequently imposing his own SID upon the victim's browser. Thus, the next time the victim visits iGoogle, the attacker's home page will be loaded, and the script will be executed from within the `google.com` domain.

As we can see, the session fixation attack is more venomous than would be expected when first looked upon. A partial client-side solution to session fixation is described in chapter 5.

3.4 Cross Site Request Forgery

The cross site request forgery (also called CSRF or session riding) attack is different from the session hijacking and session fixation attacks in the sense that the attacker will not try to completely take over a victim's session. Instead, the attack leverages the victim's browser's implicit authentication to make requests in the name of the client. This is accomplished by compelling the victim's browser into issuing the request. A possible threat exists, for example, when the victim is logged in at the website of his bank. In this case, the attacker can use the victim's implicit authentication to transfer money from the victim's account to his own account. Before we see the ways in which the attacker can make a victim's browser issue requests, let us first look at the complete attack scenario.

3.4.1 Attack scenario

The CSRF attack is made up out of the following steps (assuming that the victim is already authenticated at the target website) [44]:

1. The attacker forces the victim's browser to send a request to the server (we will see how this happens in section 3.4.2). It is the attacker that chooses the contents of this request.

[4] Google, for example, offers an overview of all your searched queries at http://www.google.com/searchhistory/.

[5] http://www.google.com/ig

2. The browser, thinking that a legitimate request is performed by the victim, automatically attaches the victim's authentication information. This authentication information can be in the form of a SID, `HTTP Auth` credentials, SSL information, or even the user's IP address [24, 54].

3. The browser sends the request to the target server. This server uses the request's authentication information to determine that the request was made by the victim. Thus, the server executes any action that was requested by the attacker as if the victim requested it.

The different steps of the CSRF attack are graphically illustrated in figure 3.4.

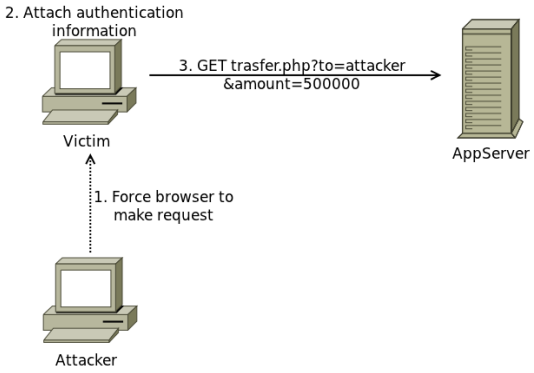


Figure 3.4: The cross site request forgery attack

3.4.2 Forcing the browser to make a request

As was the case with the previous attacks, there are several possibilities to execute a CSRF attack. We describe these methods in this section. Table 3.1 summarizes for each of them whether they can be used for GET requests, POST requests, or both.

Via crafted URLs

The simplest method is to craft an URL containing the desired request and its GET parameters. When the victim clicks the URL, the browser will automatically issue the request together with its parameters. Tricking the user into clicking the URL can be done either by phishing [4], or by embedding the URL in a trusted web page in a manner similar to the XSS attack described in section 3.1. Because URLs can only hold GET parameters, this method can not be used for issuing requests that contain POST parameters.

Via the `<img>` tag

An attacker can also abuse the `<img>` HTML tag for issuing requests. The `<img>` tag is normally used for including images in a web page. To download an image from the

server where it is stored, the client's web browser has to issue a request. However, instead of providing the URL to an actual image, the attacker can provide a request URL containing GET parameters, causing the victim's browser to issue the request when trying to load the image [24]. If the attacker is able to insert images into a trusted website (as is often the case in bulletin boards), he can make a victim issue the request when he visits the trusted page containing the image [4]. Similarly to the previous method, this method can only be used for requests that contain GET parameters.

Via forms

When the target web application uses POST instead of GET parameters, a HTML form can be used to perform the request. When a form is submitted, the browser issues a request containing the form elements and their values as parameters.

An attacker can create a form containing elements that have the desired values. He must then force the victim's browser to submit the form, which can be done by either tricking the victim into submitting the form manually, or by submitting the form using JavaScript.

Tricking the victim into submitting the form can be relatively easy: all the attacker has to do is assure that the victim clicks the submit button. This can be achieved by hiding all form elements except for the submit button, pretending that the button serves some other purpose. This approach is taken even further by Mao et al. [35]: they noticed that, even when it is required that the *actual* submit button of the legitimate site is pressed (as is the case when some CSRF countermeasures [26, 22] are in place), a victim can still be tricked into triggering a request it didn't want the browser to perform. This is done by including an iframe in which only part of the original form is visible into the attacker's website. The hiding of elements on the legitimate page is possible because iframe parents have access to all elements in the child iframes, irrespective their origins [47]. Their result can be seen in figure 3.5.

An attacker can also make the form automatically submit in the victim's browser. For this, he needs JavaScript capabilities on the page where the form is located. An attacker has these capabilities when the form is hosted on the attacker's domain (which then needs to be visited by the victim), or when the form is displayed on a page where the attacker can execute a XSS attack (see section 3.1). To automatically submit the form, a line of JavaScript similar to `document.forms[0].submit()` should be inserted in the HTML code of the page [26].

Via asynchronous requests

If the attacker is able to perform a XSS attack, he can inject JavaScript code which uses the `XMLHttpRequest` object to perform an asynchronous HTTP request. The complete specification of this object is available at [50]. We only provide the following small example of its use [49]:

```
var client = new XMLHttpRequest();
client.open("POST", "transfer.php");
```

(a) the original form

(b) a malicious page including only parts of the form in an iframe, trying to trick a user into clicking the button

Figure 3.5: Tricking a user into submitting a form [35]

```
params = "to=attacker&amount=50000";
client.setRequestHeader("Content-type", "application/
                                        x-www-form-urlencoded");
client.setRequestHeader("Content-length", params.length);
client.setRequestHeader("Connection", "close");
client.send(params);
```

Creating an asynchronous GET request is similar. Differences are that the parameters are now appended to the URL instead of passed to the `send()` function, and that the HTTP headers don't have to be set explicitly.

Via active network attacks

If the connection between the client and the server is not encrypted, an active network attacker (or 'active MitM') doesn't need the victim's browser to submit the request: he can simply modify any request sent by the victim to contain the URL and parameters he wants [4].

	GET requests	POST requests
Link	X	
`<img>` tag	X	
Form		X
Async. request	X	X
Active attack	X	X

Table 3.1: Different methods for forcing a browser to make a request

Table 3.4 summarizes for each of the discussed methods whether it can be used to issue GET requests, POST requests, or both. As we noted in section 3.3.2, however, some websites which normally use one of those two methods also accept requests issued using the other method [54].

3.4.3 CSRF and the same origin policy

The SOP (described in section 2.2) limits JavaScript access to DOM objects that have the same origin as the page which contains the script. Although this prevents JavaScript from accessing DOM objects from another domain, it does not prevent JavaScript from making requests to another domain [10]. Thus, the same origin policy does not have any effect on CSRF attacks.

Chapter 4

Session attack countermeasures

In this chapter, we discuss some general countermeasures to the session hijacking and session fixation attacks. We defer the discussion of more specific countermeasures to section 5.7, where we talk about countermeasures related to our own solution.

4.1 Server-side countermeasures

4.1.1 General solutions

Secure connections

Cookies vs. URL rewriting

4.1.2 Session security in web application frameworks

Often, web applications are built on top of a web application framework. A web application framework (or WAF) provides a web developer with the core functionality of a web application [45]. This core functionality typically consists of elements like user session management, data persistence, and templating systems used to dynamically render web pages. It is upon the foundations provided by these frameworks that many dynamic web applications are built.

In this section, we describe the measures that are taken in some widely used WAFs to ensure session security against session hijacking and session fixation attacks.

4.2 Client-side countermeasures

4.3 Other Solutions (and their problems)

Chapter 5

A client-side solution to session fixation

In this chapter, we propose a client-side solution to the session fixation attack described in section 3.3. To our knowledge, there exists no other practical client-side solution to this attack.

Out of the methods for injecting a session ID described in section 3.3.2, we consider XSS and `<meta>`-tag injection the most important. These injection attacks have a high severity rating [52] and lots of websites are vulnerable [6]. Ideally, these problems would be solved by finding a solution to XSS attacks. Unfortunately, as we saw in section 4.3, solutions to XSS are often lacking. Other attack vectors, such as URL rewriting, subdomain cookie setting and response splitting are considered out of scope because either a client-side solution would be unable to distinguish between legitimate SIDs and forged SIDs, or because they exploit a bug at the browser or proxy level.

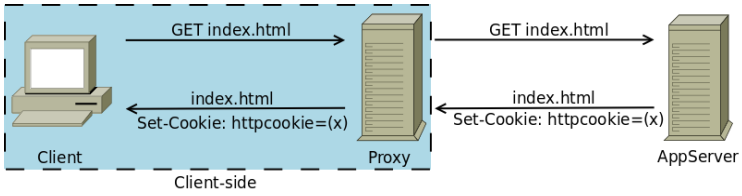
We first discuss the client-side policy that was developed to counter session fixation. Afterwards, we implement this policy as a Firefox add-on, and we provide a thorough evaluation. Lastly, we describe how the add-on was extended to also provide a solution to the session hijacking attack.

5.1 Principle

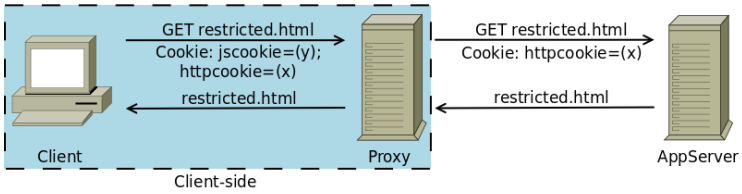
The reasoning behind the client-side solution developed is that session IDs will never be set over an untrusted channel, only to be requested over a trusted channel later on. We consider HTTP to be trusted, since this channel is controlled entirely by the web server. As an untrusted channel we consider elements in the web page itself, such as JavaScript and `<meta>` tags, because they often contain user input. The assumption made is thus that most websites will set their session identifiers via HTTP, and that websites that don't will never request the SID via this channel. As we will see in section 5.4, this assumption is valid for all practical use.

The solution has the form of a proxy that is located at the client-side. As a basic policy, we choose to only allow cookies in outgoing HTTP requests if they were

previously set via a HTTP response from the server. This policy is depicted in figure 5.1. When a new SID is sent to the client via HTTP, the proxy remembers this SID. When an outgoing request is sent to the server, the proxy checks all outgoing SIDs. If one of these was not set via HTTP, it is removed from the request. This prevents all cookies set via JavaScript or `<meta>` tags from being used over HTTP.



(a) An incoming response containing a new SID.



(b) An outgoing request containing a cookie set via HTTP, and one set via JavaScript. The JavaScript cookie is stripped from the request by the proxy.

Figure 5.1: Client-side solution to session fixation

We want to apply the policy only to session cookies, instead of all cookies. To see why, consider the scenario where a user can set the theme of the current web page by clicking a button[1]. Because this theme can immediately be applied (by changing the page's css style on the fly), there is no need to send an extra HTTP request to refresh the page. To make this style change persistent, a cookie has to be set in the user's browser. To avoid the need for another HTTP request and response, this cookie is set using JavaScript. Subsequent requests will however send this cookie over HTTP, to make sure that the returned page is using the correct style. If no distinction is made between session cookies and other cookies, sending this cookie over HTTP would not be allowed by our policy. Correctly identifying which cookies are session cookies is the subject of the next section.

The proposed policy effectively mitigates the attack vectors we considered to be in scope. Cookies set from JavaScript are marked as untrusted, which mitigates the cross-site scripting attack vector, both within one domain as for sites sharing cookies across subdomains. A second attack vector is the injection of cookies through the `<meta>`-tag. Since these cookies do not come from a `Set-Cookie` header, they too are considered untrusted. As with the cross-site scripting attack vector, attacks within one domain and across subdomains are mitigated. In section 5.4, we show

[1]The website `http://www.last.fm`, for example, allows a logged in user to set his theme by clicking the 'paint it black' link at the top of the site.

that dismissing untrusted session cookies has no impact on the user experience. The complete policy is summarized in table 5.1.

	Normal Cookie	Session Cookie
Trusted Channel	Allowed	Allowed
Untrusted Channel	Allowed	Not Allowed

Table 5.1: The client-side policy for preventing session fixation

5.2 Identifying session identifiers

Because there is no standardized way of doing session management (see section 2.1), we have to look for patterns which are common among session identifiers. Based on the properties of secure web sessions listed in section 2.3, and on an algorithm defined by Nikiforakis et al. [40], we identify a cookie to be a session cookie if it has one of the following properties:

- The name of the cookie is in a list of known session identifiers, such as `phpsessid`, `aspsessionid` and `jspsession`. Most frameworks implementing a session mechanism have default names for their session IDs, which are included in this list.
- The name of the cookie contains the substring 'sess' and the value of the cookie is a sufficiently long string containing both numbers and characters.
- The cookie value passes a randomness test. For this, the relative entropy of the cookie value is computed based on the Shannon Entropy measure [27]. Afterwards, it is calculated how many bits would be needed to encode the string [40].

5.3 Implementation

The solution was first prototyped as a lightweight HTTP proxy written in Python, and later implemented as an add-on for Mozilla Firefox. We first discuss the advantages and disadvantages of an implementation as a browser extension. Afterwards, we discuss the relevant parts of the Firefox architecture. We conclude with the specifics about the implementation itself.

5.3.1 HTTP proxy or browser extension?

Implementing the policy as a browser extension has numerous advantages over an implementation as a HTTP proxy. The most important advantage is visible when an encrypted (HTTPS, see section 4.1.1) connection is used. In this case, a browser extension is already behind the regular SSL endpoint. A HTTP proxy, however, should provide its own SSL endpoint if it wants to intercept secured traffic. This

means that it should handle encryption, handshakes and certificates – which are all very difficult to get right – separately from the browser. The other option is to not let the proxy intercept HTTPS traffic, rendering it incapable to protect this kind of traffic against session fixation. This would be a major security compromise, since SSL traffic is deemed to be more secure than normal (unencrypted) traffic.

A second advantage lies in the fact that an extension can differentiate between the normal and private browsing modes in the browser. Thus, it can make sure normal surfing cookies are never mixed with private surfing cookies. A proxy would allow cookies that were set during a private browsing session to pass through in a normal browsing session.

An additional advantage is that a browser already has an up-to-date list of top level domain names. This is convenient when checking whether a website is trying to set a cookie for a valid parent domain (and not a top level domain).

Lastly, installing a browser extension – and especially a Firefox add-on – is trivial, allowing us to reach a broad audience.

There are also some disadvantages a browser extension has compared to a proxy. The most obvious one is that a particular extension implementation will only be usable by at most a few browsers, while a proxy can protect all HTTP traffic that passes through. Moreover, some browsers do not provide support for extensions at all.

Secondly, on some browsers, extensions are able to interfere with one another. This allows rogue browser extensions to thwart security extensions [3].

5.3.2 The Firefox architecture

Mozilla Firefox allows to extend the browser using a combination of JavaScript and XML. These can access the browser's XPCOM components, which offer access to various browser features.

In order to implement our client-side protection technique, the following capabilities are needed:

- Inspect incoming HTTP(S) requests, to be able to track what trusted session identifiers are set.
- Persistently store information, to be able to keep information on persistent session identifiers.
- Inspect and modify outgoing HTTP(S) requests, to be able to strip out untrusted session identifiers.

These capabilities are provided by the following Firefox components:

- The `http-on-examine-response` observer allows us to intercept HTTP responses before they are processed. Whenever a response is received, this object's `observe()` method is executed [38].

- The `http-on-modify-request` observer allows us to intercept and modify HTTP requests before they are sent. Whenever a response is received, this object's `observe()` method is executed [38].

- The `storageService` allows us to persistently store data in a SQLite database [39]. This service is also used by Firefox internally to store data on the local machine.

Additionally, Firefox provides an interface that examines a hostname and determines the longest portion that should be treated as though it were a top-level domain (TLD) [37]. This is convenient when checking whether a cookie is being set for an allowed domain.

It must be noted that the previously listed 'required capabilities' are currently only partially available in all other browsers. Google Chrome, for example, had no support for intercepting HTTP requests and responses at the time of writing, although this feature is currently on their wishlist [8]. To be able to port the add-on to other browsers, these browsers should implement similar interfaces.

5.3.3 Implementation as a Firefox add-on

The proposed policy was implemented as a Firefox add-on, available for Firefox 3.5 or higher[2]. We describe the internals of the add-on in this section.

To detect when a trusted session ID is set, the add-on searches for a `Set-Cookie` header in all incoming HTTP responses. When it finds that a cookie is being set, it checks whether the cookie is a session identifier, using the algorithm described in section 5.2. If the cookie contains a `domain` attribute, the value of this attribute is checked for validity in order to prevent cross-site cooking attacks [53] against the add-on. For this, the add-on assures that `domain` parameters in cookies are always parent domains or subdomains of the website the cookie was received from. Also, Firefox' top-level domain list [37] is used to make sure no website can set a cookie for a top-level domain (such as *.co.uk*).

When all requirements are satisfied, the cookie is stored in a separate cookie jar implemented as a SQLite database. Storage handled using asynchronous writes, to reduce the delays introduced by the add-on (see section 5.4.3). To make sure that session IDs are also available before their write operation is complete, cookies are temporarily stored in main memory until they have been written to the database.

To filter untrusted session IDs, every outgoing HTTP request is intercepted by the add-on. Before a request is sent to the server, its `Cookie` header is inspected. For each cookie, the add-on checks whether the cookie represents a session identifier. If this is the case, the add-on's separate cookie jar is queried to make sure that the cookie is trusted. If this is not the case, this particular cookie is stripped from the request. After having repeated this process for all cookies, the request is released with the modified `Cookie` header.

[2]This add-on is available for download at https://addons.mozilla.org/firefox/addon/nofix/.

5.4 Evaluation

The add-on was evaluated in three parts. The first part evaluates the add-on's functional correctness. The second part evaluates its impact in real-world browsing scenarios. Lastly, the third part evaluates the add-on's performance, and thus – together with the second part – its impact on everyday web browsing.

5.4.1 Correctness

To make sure that the implementation behaves in accordance with the specification, a simple HTTP server was created. This server issues both cookies in the way a session fixation attack would, as in the way a legitimate website would. It is then checked that only the trusted session identifiers are returned to the server on subsequent requests, while untrusted session identifiers are stripped. The implementation passed all these correctness tests.

5.4.2 Real-world impact

An evaluation of the add-on's impact on the user experience is difficult to automate. Because of this, we asked several test persons to use the add-on during their daily web browsing activities. Apart from collecting logging data from the test persons, we asked them to pay close attention to see if any websites would break or behave differently. The experiment ran for 20 weeks.

From the log files, we found that in a considerable fraction (19.53%) of the requests, session cookies were stripped. We would assume that, with such statistics, the extension leads to a severely degraded user experience. Surprisingly, the only problems that were mentioned by the testers had other causes such as malfunctioning websites or other poorly written add-ons. Not a single problem mentioned in these 20 weeks was due to the policy applied by our add-on. We will discuss the causes for this behavior in section 5.5.

Because of the very little real-world impact of this solution, it would even make sense for Firefox to implement the described policy by default. In the past, the Firefox team has already taken bold steps to increase security, even though some websites might have been broken afterwards. [47, 36]

5.4.3 Performance

5.5 Discussion

Our client-side countermeasure against session fixation is very effective with no impact on the user experience and a minimal impact on the siteâĂŹs behavior in the form of false positives. In this section, we describe why the add-on blocks so many cookies, and why this does not affect the web application's behavior or the user experience.

One reason is that a web application might not need cookies that are set via JavaScript to be available in HTTP requests. Indeed, when a cookie is set via

JavaScript, chances are quite high that this cookie will afterwards only be accessed via JavaScript. However, a JavaScript cookie will also always be sent over HTTP, and thus stripped by the add-on. The reason that this doesn't impede a web application's behavior is that when the web application tries to access such a cookie using JavaScript, it is allowed to do so by our policy. Thus, a web application accessing untrusted cookies via JavaScript only will keep working as expected.

Another reason for the stripped cookies is that the add-on also exhibits some false positives. False positives are cookies that should be allowed to be sent over HTTP, but which were erroneously stripped from the request. The cookies that we discovered to be false positives when implementing and evaluating the add-on were all related to web analytics services. Web analytics services provide a way for web developers to gather statistics about their website. Two examples of web analytics services are Google Analytics[3] and Yahoo! Web Analytics[4]. To be able to track a visitor's behavior on a website, these services set cookies to measure the time spent on the website, and to uniquely identify a visitor [48, 17]. This last category of cookies often has a value that is random, and because the web analytics code is embedded as JavaScript code within the developer's web page, this type of cookie is set via JavaScript. Hence, these cookies will be marked as 'untrusted' by the add-on, and will as such be stripped from all HTTP requests. Stripping these cookies only affects the web developer, which is why none of the add-on testers noticed a degraded user experience.

If we want our policy to be widely adopted, we should make sure that the previously mentioned false positives are accounted for, even if they don't impede the user experience. Because of this, we also provide a whitelist of cookies that shouldn't be stripped in the add-on.

5.6 Extending with session hijacking protection

The add-on was also extended with a solution to the session hijacking attack. To do this, we made an even stronger distinction between HTTP cookies and JavaScript cookies: in our extended policy, session cookies that are set via HTTP are not allowed to be accessed by JavaScript. This prevents session cookies from being stolen via XSS attacks.

The policy extension is implemented in our add-on by adding the `HttpOnly` flag to every incoming HTTP cookie which is marked as a session cookie. When this flag is enabled, the browser will only allow the cookie to be accessed via HTTP (see section 4.1.1 for more information). Thus, we let the browser handle the blocking of HTTP session cookies over JavaScript.

The extension to our policy is related to the session hijacking solution proposed by Nikiforakis et al. [40] (which will be discussed in section 5.7.1), where HTTP

[3] An overview of the features available in Google Analytics can be found on http://www.google.com/analytics/features.html.

[4] An overview of the features available in Yahoo! Web Analytics can be found on http://web.analytics.yahoo.com/features.

cookies are kept in a separate cookie store, outside of the browser. The difference lies in the fact that our add-on *does* allow the browser access to the cookies. The only access our extended policy prevents is from JavaScript to session cookies.

5.7 Related work

5.7.1 SessionShield

Chapter 6

Conclusion

Bibliography

[1] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[2] Michael Barbaro and Tom Zeller Jr. A Face Is Exposed for AOL Searcher No. 4417749, 2006.

[3] Adam Barth, Adrienne Porter Felt, and Prateek Saxena. Protecting browsers from extension vulnerabilities. *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2010.

[4] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, page 75, 2008.

[5] Will Bontrager. Form Submissions Without Submit Buttons, 2005.

[6] Mason Brown, Bob Martin, and Steve Christey. CWE/SANS TOP 25 Most Dangerous Software Errors Version 2.0, 2010.

[7] Eric Butler. Firesheep, 2010.

[8] Chromium Developers. API Wish List.

[9] Craig Condit. JSESSIONID considered harmful, 2006.

[10] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. *Lecture Notes in Computer Science*, 5965/2010:18–34, 2010.

[11] Giuseppe A. Di Lucca, Anna Rita Fasolino, M. Mastroianni, and Porfirio Tramontana. Identifying cross site scripting vulnerabilities in Web applications. In *Web Site Evolution, Sixth IEEE International Workshop on (WSE'04)*, volume 1550-4441, pages 71–80. IEEE Comput. Soc, 2005.

[12] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[13] D. Eastlake 3rd, J. Schiller, and S. Crocker. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[14] Stephen Farrell. Leaky or Guessable Session Identifiers. *IEEE Internet Computing*, (December 2009):88–91, 2011.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[16] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 19–19. USENIX Association, 2001.

[17] Google Developers. Cookies & Google Analytics, 2011.

[18] Robert Graham. SideJacking with Hamster, 2007.

[19] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*, chapter 19. Apress, 2008.

[20] Richard Ishida. Declaring character encodings in HTML, 2010.

[21] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. *Proceedings of the 16th international conference on World Wide Web - WWW '07*, pages 601–610, 2007.

[22] Martin Johns. SessionSafe: Implementing XSS immune session handling. *Computer Security–ESORICS 2006*, pages 444–460, 2006.

[23] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable Protection Against Session Fixation Attacks. *ACM Symposium on Applied Computing*, 2011.

[24] Martin Johns and Justus Winter. RequestRodeo: client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5–17, 2006.

[25] Paul Johnston. Authentication and Session Management on the Web. *SANS Institute InfoSec Reading Room*, 2004.

[26] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, August 2007.

[27] Ian Kaplan. Shannon Entropy, 2002.

[28] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[29] Amit Klein. Cross site scripting explained. *Sanctum White Paper*, (June), 2002.

[30] Amit Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. *Whitepaper, Sanctum Inc.*, January 2004.

[31] M Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, (Id):1–16, 2002.

[32] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Proposed Standard), February 1997. Obsoleted by RFC 2965.

[33] David M. Kristol. HTTP Cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology (TOIT)*, 1(2):151–198, November 2001.

[34] James F. Kurose and Keith W. Ross. *Computer networking: a top-down approach*. Pearson/Addison-Wesley, 4th edition, 2008.

[35] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. *Financial Cryptography and Data Security*, pages 238–255, 2009.

[36] Mozilla Developers. XMLHttpRequest allows reading HTTPOnly cookies, 2009.

[37] Mozilla Developers. nsIEffectiveTLDService, 2010.

[38] Mozilla Developers. Observer Notifications, 2010.

[39] Mozilla Developers. Storage, 2010.

[40] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. *securitee.org*, pages 1–14, 2010.

[41] OWASP. Insufficient Session-ID Length, 2009.

[42] Joon S. Park and Ravi Sandhu. Secure cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, 2000.

[43] PHP Developers. Passing the Session ID, 2011.

[44] T Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. *Whitepaper, SecureNet GmbH*, 2004.

[45] Matt Schwartz. Web application framework, 2010.

[46] Chris Shiflett. Session Fixation, 2004.

[47] Kapil Singh, Alexander Moshchuk, HJ Wang, and W. Lee. On the incoherencies in web browser access control policies. In *2010 IEEE Symposium on Security and Privacy*, pages 463–478. IEEE, 2010.

[48] Andrew F. Tappenden and James Miller. Cookies: A deployment study and the testing implications. *ACM Transactions on the Web (TWEB)*, 3(3):1–49, June 2009.

[49] Binny V A. Using POST method in XMLHTTPRequest(Ajax), 2010.

[50] Anne van Kesteren. XMLHttpRequest. Last call WD, W3C, November 2009. http://www.w3.org/TR/2009/WD-XMLHttpRequest-20091119/.

[51] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42. Citeseer, 2007.

[52] Jeff Williams and Dave Wichers. OWASP top 10, 2010.

[53] Michal Zalewski. Cross Site Cooking, 2006.

[54] William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention, 2008.

Fiche masterproef

Student: Bram Bonné

Titel: Improving session security in web applications

Engelse titel: Verhoogde veiligheid van sessies in webapplicaties

UDC: 004.49

Korte inhoud:
Hier komt een heel bondig abstract van hooguit 500 woorden.

Thesis voorgedragen tot het behalen van de graad van Master in de ingenieurswetenschappen: computerwetenschappen

Promotoren: dr. L. Desmet
Prof. dr. F. Piessens

Assessoren: Ir. P. Philippaerts
G. Poulisse

Begeleiders: P. De Ryck
N. Nikiforakis