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:
Computer Science

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

Assessoren:
Ir. P. Philippaerts
G. Poulisse

Begeleiders:
P. De Ryck
N. Nikiforakis

Academiejaar 2010 – 2011

© Copyright K.U.Leuven

Without written permission of the promotors and the authors it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail `info@cs.kuleuven.be`.

A written permission of the promotor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

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.

Abstract

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 subsequently included in every request made by the browser. 3, 30

CSRF Cross-Site Request Forgery, an attack in which JavaScript is used to execute malicious code in the victim's browser. 21, 31

CSS Cascading Style Sheets, used for defining the appearance of a web page. 23

DOM Document Object Model, used by programming languages like JavaScript to access elements in an XML document or on a web page. viii, 5, 16, 24, 39, 48, 51

DoS Denial of Service, an attack which causes a service to be unavailable to legitimate clients. 50

HTML HyperText Markup Language, the language used to structure web pages. 14, 19, 22

HTTP HyperText Transfer Protocol, the protocol which is used by web browsers to communicate with web servers. vii, 3, 7, 22, 28, 29, 39, 40, 53

HTTPS HyperText Transfer Protocol Secure, a secure variant of HTTP, which uses encryption when transmitting data. viii, 39, 40, 56

IP Internet Protocol, the protocol used for routing datagrams over the Internet. An IP address identifies a machine on the internet. viii, 22, 30

IRC Internet Relay Chat, a protocol used for creating chat rooms on a centralized server. 33

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

login session fixation A popular variation of the session fixation attack, in which an attacker tricks the user into logging in with a predefined SID. 21, 27, 42

MAC Message Authentication Code, a short piece of information providing data integrity and authenticity for a message. 8, 28, 39, 40

MitM Man in the Middle, used to describe an attacker that is able to intercept traffic between the victim and some other principal. 16, 20, 25, 29, 31, 39, 50

NAT Network Address Translation, often used for allowing Internet access to multiple machines via the same IP address. 30

phishing an attack wherein a victim is tricked into thinking the attacker is a trusted party. 13, 25

PSID Proxy Session Identifier, a second session ID which is attached and checked by a proxy to counter session fixation attacks. 42

session cookie A cookie containing a session identifier. 4, 19, 54

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. vii, 21, 53

session hijacking An attack in which the attacker tries to capture the victim's SID, to be able to take over his session. 17, 21

session ID Session Identifier, a unique string of text used by the server to identify a user. viii, 3, 15, 17, 53

SID see 'session ID'. vii, viii, 3, 22

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, 14, 24

SSL Secure Socket Layer (also known as TLS, or Transport Layer Security), used in HTTPS to encrypt traffic between a client and a server. 7, 22, 29, 33

TLD Top Level Domain, the last part of a URL, which determines the part of the domain that registered in the name space's root zone. Examples are `.com`, `.org` and `.co.uk`. 57

URL Uniform Resource Locator, a string that specifies the location of a resource. viii, 4, 18, 25, 30

web application framework a framework providing the core functionality of web applications, which provides the foundation for web developers when building web applications. 32

XSS Cross-Site Scripting, an attack wherein the attacker injects JavaScript into a legitimate page that will be executed by another user's browser when that user visits the page. 11, 16, 18, 21, 23–25, 28, 40, 41, 45, 48, 53, 61, 63

Chapter 1

Introduction

Over the last few years, the web has shifted from being a collection of pages containing static information to a dynamic and fully interactive platform. Where the Internet was once used only as an information repository, today it powers complex web applications, developed both to replace programs that were once running locally on a user's computer, and to provide whole new functionality that is possible only on the web. For this, web protocols to be used and extended in ways they were never imagined to be.

More than in the past, these web applications deal with sensitive personal information. Thanks to the emergence of web-based mail applications akin to Google's GMail and Microsoft's Hotmail, and social networks like Facebook and Netlog, a great deal of user information is stored on the servers of web applications. Moreover, shops have moved to the on-line world, and payments can be made on-line by using on-line banking or credit cards. Because many web applications handle such sensitive information, security of a user's information and identity is of utmost importance.

User authentication is handled in most web applications via the concept of web sessions. These allow users to use a web application without having to enter their login credentials for every action. Unfortunately, sessions have many security weaknesses. OWASP, a leading organization in the field of web application security, rates 'Broken Authentication and Session Management' as the third most important web application security risk [117].

In this thesis, the security of sessions in web applications is examined. For this, we first study some well-known, and some lesser known methods in which user sessions can be abused by an attacker. We then present some generalized methods for protecting against these attacks. Afterwards, we conduct a thorough examination of the application of these protections in web frameworks and in more specific attack countermeasures. We conclude by developing a client-side solution to the session fixation attack. This is, to our knowledge, the first attempt to solving session fixation at the client side.

Chapter 2

About web sessions

When a user visits a website, the web application often needs to remember which user it is interacting with. For this reason, the concept of 'web sessions' 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 [67]. 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. Consider for example an on-line shop, where a user can fill its virtual shopping cart with different items available on the website. Without state, the web application is not able to remember which items were associated with a user on subsequent page requests. 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.

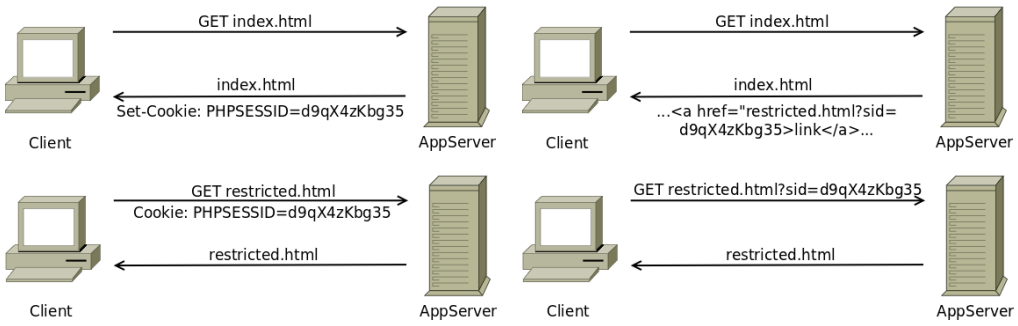
Web sessions work as follows:

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, every request issued by the client to the server contains the received SID. 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 [57]. The first one, which is most common, makes use of cookies [66, 82]. Cookies are strings consisting of multiple name-value pairs which are included in

the `Set-Cookie` header of a HTTP response by the web server. 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 domain[1] the cookie was set for. This is done by including the cookie as the value for HTTP's `Cookie` header. The 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 session identifier) 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 URL occurring in a response that points to a page on the web server's own domain. 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 [53]. 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 only as an enabler of server side storage, and not as the storage itself. This means that the SID only identifies

[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 only to identify the user.

the user, 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.

2.2 Accessibility of session identifiers

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 [97]. This means that a cookie set by the page `a.example.com/s/p.html` is sent by the browser to both `b.a.example.com/s/p2.html` and to `a.example.com/s/ss/p3.html`, but *not* to `example.com/s/4.html` or `a.example.com/p5.html`.

Cookies 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 SIDs that are stored in a cookie to be accessed via the `document.cookie` property. For DOM objects, the *Same Origin Policy* (or *SOP*) is enforced [97]. This policy states that web pages that want to access a certain DOM object (in this case, the cookie) 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.

For SIDs that make use of URL rewriting, the server side access policy 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. Thus, the server can decide by itself for every other web page whether the session identifier should be included in the request. At the client side, the session identifier is again available via the DOM, this time via the `href` attribute of the `<a>` element containing the URL. Thus, at the client side, the SOP again applies.

2.3 Keeping web sessions secure

It is important to ensure that a session identifier can only be known by the web server and the web browser of the user that is identified by it. If an attacker is able to know a user's SID, he can use it to impersonate the user within the web application. This can have severe consequences in for example a webmail application[3], where an attacker could use the SID to read and send e-mail on behalf of the user. Another example is the on-line shop, where an attacker could fill the user's virtual shopping cart with unwanted items, or even purchase items for himself by using the user's credit card information. To prevent an attacker from knowing a user's SID, the

[3] Examples of widely-used webmail applications are Google's GMail (`http://mail.google.com`) and Microsoft's Hotmail (`http://www.hotmail.com`).

SID should possess some properties: it should be unguessable and unavailable to an attacker, and its lifetime should be limited. We describe each of these properties in this section.

2.3.1 Unguessable by an attacker

If an attacker is able to guess a user's SID, he can compromise his session. To prevent an attacker from guessing the user's SID, 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 [77, 39, 36]:

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 might be able to derive (part of) SIDs that will be generated from SIDs which were already generated. As a consequence, an attacker is able to predict a victim's session ID from his own session ID. Furthermore, an attacker might be able to derive past SIDs from SIDs that are currently issued.

a high number of possible values This property follows from having both high entropy and 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 [43]. This means that more than just a LCG [20] 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 [80]:

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

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, a leading organization in the field of web application security[4], recommends a session ID length of at least 128 bits [80].

[4]More information about the OWASP project can be found at https://www.owasp.org/.

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 legitimate user and the web servers on the domain the SID was set for. In this section, we only describe some default properties to achieve this. We will go into more detail about making cookies unavailable to an attacker in section 4.1.

SIDs that are set via cookies are, by default, transmitted as clear text. This means that, on an insecure channel like the Internet, an eavesdropper is able to intercept the cookie value. Such an attacker can thus take over a user's session. Interaction between a client and a web application can also happen over a secure (TLS, but most often referred to as SSL) connection [99, 25]. In this case, the requests and responses (and thus, the cookie value) are encrypted, and an attacker is unable to extract the cookie from a captured message. When using secure connections, care must be taken 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 [43, 64]. This flag demands that the browser only sends the cookie over encrypted 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 [77]. 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. With URL rewriting, it is not possible to state that the SID should not be available to client-side JavaScript. Indeed, a link containing the SID will always be available as a DOM object.

2.3.3 Short lifetime

A third way of making sessions more secure is to ensure that they have a limited lifetime [43]. 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 to successfully impersonate the victim.

Additionally, limiting the lifetime of a session identifier makes sure that a user will be logged out eventually. This is convenient in case a user forgets to log out when he stops using the web application [80].

Limiting the SID's lifetime

The lifetime of a cookie can be limited by using the `expires` attribute when setting the cookie [66]. This attribute indicates the date that the cookie stops being valid.

On this date, the user's web browser should stop sending the cookie to the server. The lifetime for SIDs in URLs or forms can be limited by simply not including them in any URL or form at the next response from the server to the client. For either of these session mechanisms, care must be taken that the session is also invalidated at the server side. Otherwise, an attacker could still use the cookie to impersonate the victim, even after it expired at the victim's browser [62].

Another way of limiting the lifetime of a session identifier is presented by K. Fu et al. [43]. By including a timestamp as part of the SID value, the server can determine whether an SID is still valid without having to save the SID's expiration time as part of its state. It is important to note that, because the timestamp is appended to the normal session identifier, this approach requires the complete SID value (including the timestamp) to be signed by the server. Otherwise, an attacker could just change the timestamp part of the SID to extend its lifetime. The signing can be easily achieved by letting the server create a MAC of the entire SID, and appending this MAC to the SID value. 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 which was attached to the old SID should be attached to the new SID instead [113].
3. Make sure the client uses the new SID instead of the old one. When cookies are used for session management, this can be achieved by attaching a `Set-Cookie` header containing the new SID value to the next HTTP response. The user's browser will notice that the cookie's name is the same as that of the old SID, and will therefore update the old value. When URL rewriting is used, the web server can make the client's browser use the new SID by making sure that all links on subsequent web pages served to this client include the new SID value.
4. Invalidate the old session identifier.

2.3.4 Using a session management framework

Problems occur in lots of web applications because they implement their own session management. As we will see in section 4.2, 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. Using such a framework does not completely relieve the web developer of all tasks associated with session management: he must still make

sure that the framework is configured correctly, and that channels other than HTTP (for example, JavaScript) do not pose any security issues.

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 exploit security issues in session management mechanisms in order to impersonate a legitimate user on a website. The discussion of mitigating these attacks is deferred until the next chapter.

3.1 Background: cross-site scripting

Cross-site scripting (or XSS) [24] 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 web page to inject his own JavaScript code into this page. The injected code will then be executed in the browser of any user that loads the vulnerable page.

3.1.1 Variants of cross-site scripting

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

Persistent (stored) XSS

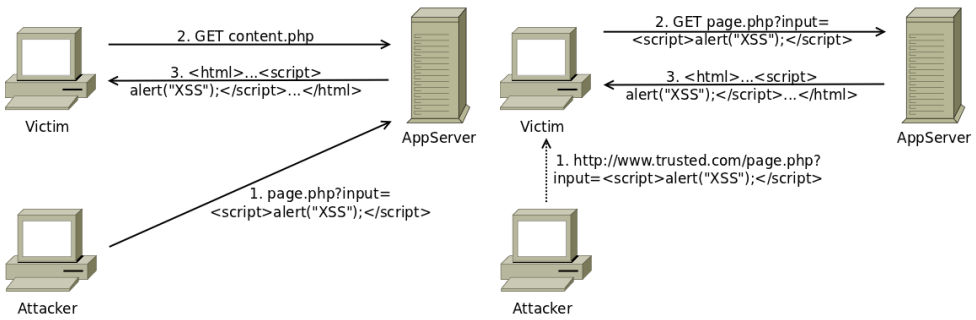
In a persistent XSS attack, the attacker misuses a website's functionality that allows users to provide their own content. This allows him to make the web server persistently store his script code, and to make it serve this code to other users later on. The complete scenario goes as follows:

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 its database.

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

3. The victim's browser sees the script code as part of the requested web page and executes it.

This attack variant is graphically illustrated in Figure 3.1a.



(a) persistent (b) reflected

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 of web applications serving user generated content 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.

Reflected (non-persistent) XSS

In a reflected XSS attack, the script code is not stored at the server. 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.

[1] phpBB is free and open source bulletin board software, available at http://www.phpbb.com/

[2] Facebook is currently the most popular social networking site. It can be found at http://www.facebook.com/

3. The server reflects this script code back to the victim as part of the response.
4. The victim's browser sees the script code as part of the requested web page and executes it.

This attack variant is graphically illustrated in Figure 3.1b.

An example of a web page returning user input is a search form which displays the search query as part of the response. If it does this without stripping any script data that might be embedded in the query, the web page is vulnerable to reflected XSS attacks [111]. To see why, consider the scenario where doing a search for *query* by going to the URL `http://www.trusted.com/search?q=query` makes the website return "x results for *query*". In this case, 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 [59]. 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 provided by the attacker.

Tricking a user into clicking a link can be achieved by executing a phishing attack [23]. In such an attack, the victim is lead on to believe that the attacker is a known trusted party (e.g. the victim's bank). The attacker, can then ask the victim to click a link under a false premise. Because the victim thinks he can trust the source of the link, he will be inclined to click it. If the link contains malicious data (as is the case in a reflected XSS attack), the victim will unknowingly send this data to the server.

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 [52]. We describe the most important ones, including one that wasn't presented in [52], 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. Furthermore, because the attacker keeps control over the source of the script code now, he is able to edit this code afterwards.

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 for the particular object.

using JavaScript handlers for page elements It is possible to specify script code that should be triggered when a certain action occurs on an element in a web page. For example, the `onload` parameter of the `<body>` tag is used to specify a JavaScript function that should be executed when once the browser has completed loading the page. Two other examples are the `onclick` and `onblur` parameters, which can be specified for different kinds of tags, with `<p>` (which indicates a paragraph) and `<a>` (which indicates a link) being two possibilities. These parameters specify, for a certain object on the page, respectively the JavaScript function to be executed when the object is clicked, and the function to be executed when the object loses focus.

via binary objects If an attacker is able to include a binary object into the page, he can use this object to execute JavaScript code [11]. Flash applets, for example, allow running JavaScript code by using either the `getURL()` or the `fscommand()` function [79, 100, 83].

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 [52]. For example, the encoding of (parts of) a web page can be changed using the `encoding` and `charset` attributes of various HTML tags [50]. Some browsers also allow strings to contain JavaScript that is split over multiple lines [52]. 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 which 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 reflected XSS attacks: 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), the code is allowed to access elements belonging to that domain [60]. This gives the attacker the ability to read information from, and write information to, elements in the trusted domain. To see why this is such a big issue, consider the case where an attacker injects code into a trusted domain that reads information from this domain,

and subsequently sends the information to the attacker's web server. Sending this data can be done, for example, by having JavaScript open a window containing a page from the attacker's domain with the data as a GET parameter [60]. These cross-domain interactions are 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 the victim's session ID. He then uses the SID to make the server think that he is the victim. This causes him to be able to, for example, read the victim's e-mail in a webmail application, change the victim's information on a social networking website, or acquire the victim's credit card information in an on-line shop. 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 [77]:

1. The victim establishes a new session at the server. This is done automatically either when he first visits the page or when he logs in (depending on the web application), as described in the previous chapter.

2. The attacker captures the session ID that corresponds to the victim's created session. The methods that can be used to do this will be described in the next section.

3. The attacker makes a request to the server, attaching the captured session ID as his own. Because the server has no other effective[3] means of distinguishing between the victim and the attacker, it thinks that it is the victim who 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.

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.

[3] We use the term 'effective' here because, as we will see in section 4.1, there are some other – unsatisfying – means for distinguishing between different clients.

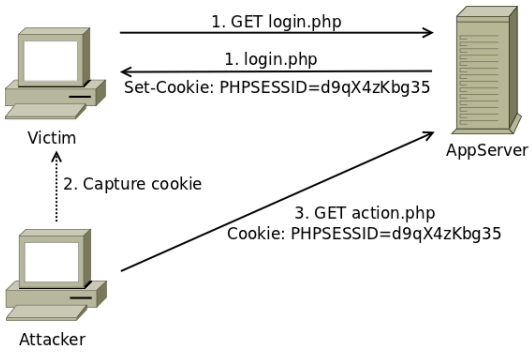


Figure 3.2: The session hijacking attack for a web application that uses cookies

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. A common example is when the victim and the attacker are sharing the same network hub, or when they are both connected to the same WiFi hotspot without using WPA2 encryption [4].

As was 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 or form elements. Because of this, an attacker able to read all network traffic can easily extract the SID from the `Cookie` header, request URL or request data in one of the user's requests, or even from the `Set-Cookie` header or the response page 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 [46] and Firesheep [16], which make a session hijacking attack almost trivial when the victim is using an insecure network.

Via cross-site scripting

The XSS attacks can also be used to steal a victim's session ID. For this, the attacker uses one of the methods described in section 3.1.2 to inject script code 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, the script is also able to send data to the attacker's domain (see section 3.1.3 and [60] 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

If SIDs are included in the URL (as is the case with URL rewriting), the SID can also leak via the HTTP `referer` header. The `referer` header is used to transmit

the website from which the request-URL was obtained to the web server that the request is issued to [40]. When the browser attaches a `referer` header to requests going to a different domain than the one it originated from, 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 to the requested page [43].

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 [8]. 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 [56], 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 4.1.6.

3.3 Session fixation

In the session fixation attack, as in the session hijacking attack, 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 [62]:

1. The *attacker* establishes a new session at the server. He does this by sending a request that does not include a SID, which will cause the server to attach a

newly generated SID to the response. Some servers also accept crafted SIDs[4] [95]. 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. The methods that can be used to do this will be described in the next section.

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 to 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. As such, the attacker is able to impersonate the victim at the server.

These steps are graphically illustrated in Figure 3.3.

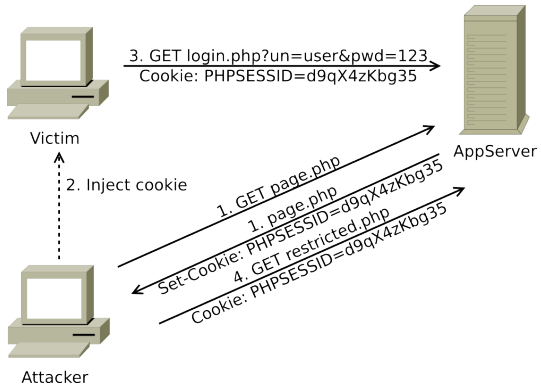


Figure 3.3: The session fixation attack for a web application that uses cookies

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 [54]. He then sends this link to the victim, or places it on the target website as part of an XSS attack. When the victim clicks

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

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 an HTTP redirect [40] to make the victim's browser automatically load the page [95]. This requires the victim to visit the attacker's page, or the attacker to be able to perform an XSS attack.

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) [62, 14].

Note that, for a website to accept session IDs via GET or POST parameters, it does not 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 [84, 19, 48].

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 [62] or by using the `cookie.write()` function [54].

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 [62].

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 an 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 [61]. 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 [54].

Via an active man-in-the-middle attack

In an active man-in-the-middle attack, the attacker can not only eavesdrop on the victim's communication with the server, but can also intentionally modify this

communication. This is the case, for example, when the attacker is able to take over the victim's gateway to the Internet (e.g. an Internet router or a proxy).

Since an active MitM is able to modify responses from the server to the victim, he can alter the `Set-Cookie` header such that it contains the desired session identifier. The victim's browser will see this as the server issuing a new session cookie, and will store the cookie accordingly.

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, the possibility to set a cookie for a parent domain by using the cookie's `domain` parameter [66]. As an example, consider the case where the attacker is 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 [62].

A related problem occurs when two cookies with the same name are set for both the parent domain and the subdomain. When the user visits a page on the subdomain, his browser will attach both cookies 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 the parent domain and the subdomain cookie. 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.

It has been proposed that cookies include their attributes when they are sent to the server [65]. Unfortunately, as we can see from the previous results, this standard has not yet been widely implemented, even though it has existed for over a decade already.

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 [8]:

- 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[5], allowing an attacker who is able to perform a session fixation attack to access this highly sensitive information [5].

- On domains that allow the embedding of trusted scripts, it creates the ability for an attacker to execute XSS attacks. Until recently, iGoogle[6] offered the ability to embed trusted scripts on your own personal homepage [8]. 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.

A distinction must be made between the previously described scenario, wherein an attacker tricks the user into logging in with a predefined SID, and the scenarios described in this section. We will use the term *login session fixation* to refer to the first variation, while using just *session fixation* to refer to the general class of session fixation attacks.

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 an attacker executing a CSRF attack does 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 a 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.

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

[6] `http://www.google.com/ig`

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) [93]:

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

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 [55, 120].

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.

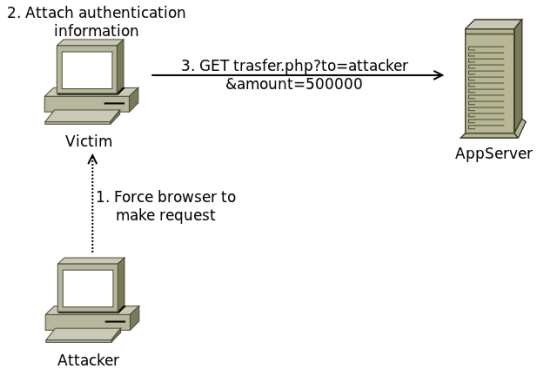


Figure 3.4: The cross site request forgery attack

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. In this section, we describe the most important methods an attacker can use for compelling the victim's browser into issuing a cross-site request.

Via the `<img>` tag

The `<img>` HTML tag is normally used for including images in a web page. However, the `<img>` tag can also be abused by an attacker for issuing cross-site requests. For this, instead of providing the URL to an actual image, the attacker provides a request URL containing GET parameters. This will cause the victim's browser to issue the

request when it tries to load the image as part of the current page [55]. 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 [8]. The `<img>` tag can only be abused by an attacker for issuing requests that contain GET parameters.

Via cascading style sheets

Cascading style sheets (or CSS, not to be confused with XSS) are used for defining the appearance of a web page. They include the colors, fonts, and placements for elements on a web page. Similar to using the `<img>` tag for issuing requests, the `<link>` tag, which is normally used to load external CSS styles can be used to issue requests [49]. For this, the attacker sets the URL of the style sheet to the request URL containing the GET parameters. This will cause the victim's browser to issue the request when it tries to load the style sheet for the page. As was the case with the previous method, cascading style sheets can only be abused for issuing requests that contain GET parameters.

Via forms

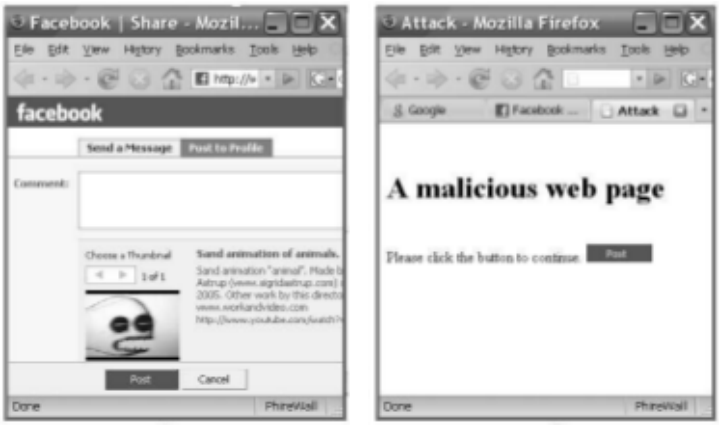
When the target web application uses POST instead of GET parameters for the request the attacker wants to execute, an HTML form can be used to perform the request. When a form is submitted, the browser issues a request containing the form elements with their values as POST parameters.

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 having the form submit automatically using JavaScript.

Tricking the victim into submitting the form can be relatively easy: all the attacker has to do is assure that the victim will click the submit button. This can be achieved by hiding all form elements except for the submit button, and making it seem like the button serves some other purpose. This approach is taken even further by Mao et al. [68]: they noticed that, even when it is required that the *actual* submit button of the trusted site is pressed (as is the case when some CSRF countermeasures [57, 53] are in place), a victim can still be tricked into triggering the browser to make a request that he did not want it to perform. This is done by including an iframe[7] containing the original form into the attacker's website, while making only the form's 'post' button visible to the victim. The hiding of other parts of the form is achieved by resizing and auto-scrolling the iframe. The result of executing such an attack in order to make a victim unwillingly post something on his own Facebook profile page is shown 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

[7] An iframe is an HTML tag which allows a HTML page to be embedded within a parent HTML document.



(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 [68]

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 an XSS attack (see section 3.1). To access the form, the attacker can use the `forms` DOM element, which contains a list of all forms on the page. Automatically submitting the form can then be done by inserting a line of JavaScript similar to

```
<script> document.forms[0].submit() </script>
```

into the HTML code of the page [57].

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 [22]. Thus, the same origin policy does not have any effect on CSRF attacks.

3.4.3 Other ways of forcing the browser to make a request

Technically, cross site request forgery attacks always automatically issue a request to a different domain. In this section, we describe some other methods an attacker can use to make a victim's browser issue a request. Note that, although these methods can be associated with cross site request forgery attacks, they are technically different, because they do not issue a request from one domain to a different domain, or because they can not be used to issue a request automatically.

Via crafted URLs

The simplest method for an attacker to make a victim's browser issue a request is to trick the victim into clicking a crafted 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 [23], 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 asynchronous requests

If the attacker is able to perform an 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 [110]. We only provide the following small example of its use [108]:

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

An asynchronous GET request can be made in a similar fashion. 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 explicitly set.

Via an active man-in-the-middle attack

An active network attacker (or 'active MitM') does not 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 [8]. However, as we noted when describing the session hijacking and session fixation attacks, an attacker is likely to have better options for attack when he is an active MitM. Indeed, an active attacker will always have the ability to intercept a victim's SID, causing him to be able to issue requests in name of the victim himself.

Table 3.1 summarizes for each of the methods discussed in this and the previous section whether requests can be automated, and whether cross-domain requests are allowed to be made. The table also lists whether a certain method 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 [120].

	GET requests	POST requests	automatable	cross-domain
`<img>` tag	X		X	X
CSS	X		X	X
Form		X	X	X
Crafted URL	X			X
Async. request	X	X	X	
Active MitM	X	X		X

Table 3.1: Comparison of different methods for forcing a browser to make a request

Chapter 4

Session attack countermeasures

In this chapter, different countermeasures to the session hijacking and session fixation attacks are discussed, together with a detailed analysis of their shortcomings. We first describe some general security measures that a web developer can take to secure his web application. Next, we inspect different web frameworks to see how well they apply these principles. Afterwards, we give an overview of standalone countermeasures, both at the server side and at the client side, that were developed over the years. We close with a brief description of some solutions to cross-site scripting attacks, because of their importance in session attacks. We do not consider solutions to cross site request forgery here, since these would lead us too far.

4.1 General security measures

Different security measures may be taken by web application and web framework developers to secure their web applications. In this section, we discuss to what extent these measures provide security against session hijacking and session fixation attacks.

4.1.1 Renewing the session identifier

The practice of renewing the session identifier (see section 2.3.3) can provide excellent protection against login session fixation attacks. Indeed, if the web application renews the session identifier every time the authentication status of a user changes, the SID which was enforced by the attacker will cease being valid once the victim has logged in. As an example, renewing the SID in PHP can be accomplished by using the following code snippet [86, 54]:

```
if ($authentication_successful) {
    $_session["authenticated"] = true;
    session_regenerate_id();
}
```

To make sure that the session identifier is changed on every authentication change, it can be made to contain specific user information (like the username). In this case, when the authentication state changes, the session identifier has to change with it.

Note that such an approach needs a MAC, created by the server using its private key (as described by Fu et al. [43]), to be included as part of the SID. Otherwise, an attacker would still be able to simply replace the username in the value of the SID himself.

Renewing the session identifier regularly is also benefical for preventing session hijacking attacks. As was discussed in section 2.3.3, the shorter the lifetime of a session identifier, the more difficult it will be to capture, and the less useful it will be if it has been captured [43].

Keeping in mind the previous discussion, one might assume that it is best to renew a session identifier as often as possible. Consider, however, the case where the session identifier is renewed on each request. In this case, every request would result in a new session cookie being set at the user's browser, with the old session cookie being invalidated. When issuing concurrent requests (for example, when simultaneously loading different images on a web page), the web application would only acknowledge the first request that arrived at the server. The second request would contain the (by then) invalidated session identifier, causing it to be rejected. Because of this, the browser would be allowed to issue only one request at the same time. Similarly, this would pose a problem when browser plugins communicate with the server, as well as with asynchronous requests. Another drawback of this approach is that web applications making use of multiple servers (to allow for load balancing) would have to synchronize their users' SIDs on every request [21]. Otherwise, a user would have to re-authenticate every time a page is served by a different server. A last problem with this approach is that the server would have to issue an extra write to the database on every request, in order to update the SID.

4.1.2 Using HttpOnly cookies

When setting a cookie in the user's browser, a web server can use the cookie's `HttpOnly` flag to indicate that the browser should only allow access to this cookie via HTTP. As a consequence, script code will not be able to access or edit the value for such a cookie, and session hijacking via XSS attacks can be prevented [71].

Session fixation, on the other hand, is *not* solved by using HttpOnly cookies. Indeed, when the attacker sets the cookie before the web server does, *he* is the one who can decide whether the `HttpOnly` flag is set. Thus, the web developer would have to make sure that a cookie is always set before an attacker can set it.

Moreover, even if the web server is able to set the (HttpOnly) session cookie before the attacker does, the attacker is still able to perform a successful session fixation attack via XSS. As Singh et al. have shown, it is sometimes still possible for an attacker to manipulate a HttpOnly cookie via JavaScript after it has been set [97]. Fortunately, our experiments show that the issue of Firefox allowing cookie writes to HttpOnly cookies has since been solved.

Nevertheless, we discovered yet another technique which allows an attacker to circumvent the `HttpOnly` browser policy. As we described in section 3.3.2, the attacker is often still able to set a cookie with the same name as the session cookie for the parent domain. This causes the browser to send both the parent domain

cookie and the subdomain cookie when a page on the subdomain is accessed. Since the `domain` attribute is not attached to cookies in the request, the server has no way of distinguishing between the injected and the legitimate cookie. As such, we must conclude that, while marking a cookie as `HttpOnly` does prevent an attacker from accessing the cookie via JavaScript (thus mitigating the session hijacking attack via XSS for these cookies), it does not prevent him from using XSS to inject a chosen value for this cookie into the victim's browser.

4.1.3 Secure connections

Secure (SSL) connections [99] can be used to make sure that a session identifier can not be intercepted by a passive MitM attacker. In this case, as already noted in section 2.3.2, care must be taken that cookies can never be sent over an insecure connection. This can be done by setting the `secure` flag for cookies that will be used only over an SSL connection.

Recently, some work has been done to make deployment of HTTPS more widespread [47, 51]. Unfortunately, there are still some drawbacks to using HTTPS for every page [1]. At the server side, HTTPS is very costly: every connection needs computationally intensive SSL operations to be performed. At the client side, browser caching works differently under SSL, and websites have to be completely transmitted before they can be rendered (because the validity is checked for the entire page).

4.1.4 Checking request headers

Similar to the `Cookie` header we discussed in section 2.1, a HTTP request can contain many other headers [40]. Some of these headers provide information that can be used to identify a user. For example, the `User-Agent` header contains information about a user's browser and operating system, while the `Accept-Language` header lists the languages the user is willing to receive.

A web server can gain some extra certainty about whether it is interacting with the user corresponding to the session ID in the request by comparing some of the header values to those in the last request that contained the same session ID. Indeed, because an attacker is probably using a configuration which is different from the victim's, the attacker's requests will have different header values. Thus, when certain header values differ between requests, it is possible that a session hijacking or session fixation attack occurred.

The question is then: which headers could be considered to give enough user-specific information, without changing over time? The `User-Agent` header is obviously the best candidate. Unfortunately, web proxies are known to modify this header [96]. Another header which could be considered is the `Accept` header, which lists the types of content the user's browser will accept. The problem with this header is that in Internet Explorer, its value can change over time [96]. The `Accept-Language` header could be checked. However, different browsers are likely to contain the same default value (`en-US`) for this header. The same goes for the `Connection` and

`Cache-Control` headers. Other headers are too susceptible to change, and should not be used for the purpose of asserting authentication.

Another problem with using HTTP headers for this purpose is that they are easily guessed by an attacker. Indeed, for most headers, only few options are possible, with even less options being very probable. Furthermore, even in the case that an attacker would not be able to guess the header values, he only needs to read a single request (to whatever server) from the victim to know what header values to use. He can get a request from the victim by intercepting it, or by tricking the victim into visiting his own website.

Thus, while checking request headers might raise the barrier for attackers, it is by no means a complete solution against any of the described session attacks.

4.1.5 Checking the IP address

Similar to request headers, the IP address can be used by the web server to ensure it is interacting with the same user as before. Unfortunately, this approach also suffers from some problems. Firstly, IP addresses can be guessed or captured in much the same way as HTTP headers. An attacker can then easily change the source IP address for his own packets to impersonate the victim. Secondly, when both the victim and the attacker are behind the same NAT proxy (as is often the case in session hijacking attacks over a wireless network), they are using the same IP address [54]. In this case, the server can not distinguish the attacker and the victim based on IP address. Lastly, requiring that the IP address stays the same over time can also cause some problems to the legitimate user: with users changing the location of their notebook or cell phone, the IP addresses of these devices will change when roaming, causing them to be denied access to a web application that checks their IP address. Moreover, some networks only issue dynamic IP addresses to their users. Because of these reasons, it can not be assumed that a user is uniquely tied to a single IP address.

As was the case for checking other request headers, checking the IP address can not be considered a complete solution to any session attack.

4.1.6 Cookies vs. URL rewriting and form elements

As we described in section 2.1, there are three major methods for doing session management: via cookies, via URL rewriting and via form elements. We compare, from a security perspective, the advantages and disadvantages of these methods here.

Two major disadvantages of URL rewriting were already discussed in section 3.2.2: firstly, the session identifier can leak when a link is shared with someone else (for example, via e-mail or a social networking site) [56]. A leak can also occur when the `referer` HTTP header is included in a request to another website: since the URL in the referer header contains the user's session identifier, this identifier is visible to the other website [43].

A disadvantage which is shared by both the 'URL rewriting' and 'form elements' methods is that the injection of a session identifier (with the objective of executing a

session fixation attack) requires little effort from an attacker. Indeed, as we saw in section 3.3.2, such an injection attack is reasonably straightforward.

There is, however, also an advantage of choosing URL rewriting or form elements over cookies, in particular when looking at the cross site request forgery attack. Recall that, to execute a CSRF attack, an attacker tricks the victim's browser into issuing a request. For this, he has to create a URL or a form containing the right GET/POST parameters for the attack. However, if the SID is one of the parameters that must be included in the request, the attacker does not know all required parameters, and is therefore not able to create a complete request [56].

It is clear that choosing which method to use for managing sessions requires careful weighing of the advantages and disadvantages of each method. It could be argued that cookies provide more security since they make leaking of session identifiers less likely. Indeed, it is often recommended to use cookies instead of URL rewriting for session management [121, 109]. An even better option is to use a combination of cookies and POST or GET parameters. This is indeed what many CSRF countermeasures try to do [57, 53]. In addition, some (mobile) web browsers don't support cookies, requiring the use of either POST or GET parameters when session management is needed.

4.1.7 Using an alternative to web sessions

There are also other – in some cases more secure – methods for a web server to know which user it is interacting with. We describe three alternatives to web sessions here.

Logging in for every request

Arguably the most secure way to determine whether a user is who he claims to be, is to make him enter his credentials for every request. It is obvious that this is very cumbersome for the user, who has to go through the process of logging in every time he requests a new web page. It is, however, good practice to require the user to log in for certain actions [114]. Indeed, consider the case where an attacker was able to take over a victim's temporary session. If no login is required to change the user's password, the attacker can completely take over the victim's account by changing the password to a value only he knows. Similarly, if the attacker is able to change the victim's e-mail address without having to enter the password, he can use the website's password recovery feature to have a password for the user's account sent to his own mailbox.

HTTP-Auth

In HTTP Authentication [41], a separate HTTP header (called `Authorization`) is used to transfer the user's credentials on every request. These credentials are often cached by the browser to relieve the user from having to enter them every time he requests a web page. A disadvantage of this approach is that the username and password are sent encoded (with the Base64 algorithm) but not encrypted, causing them to be available to a passive MitM if no secured connection is used. Another

disadvantage is that, since HTTP Authentication is completely handled by the HTTP stack, it is a much less flexible approach than web sessions. For example, there is no easy way for the user to log out (since the browser caches his credentials), and the server-side HTTP stack needs to have full access to the user database [1].

HTTP-Digest

HTTP-Digest is a variant of HTTP-Auth that uses encryption instead of just Base64 encoding [41].This has the advantage that the user's credentials can not be intercepted by a passive MitM attacker. Unfortunately, this method is still vulnerable to *active* MitM attacks. It also suffers from the same inflexibility issues that are associated with HTTP-Auth.

SSL client certificates

When secure sessions (see section 4.1.3) are used, mutual authentication can be achieved when both server and client possess a SSL certificate [82]. The problem with this approach is that many clients don't have certificates, and that certificate management is still too difficult for most regular users [115]. Moreover, a user needs to have its certificate installed on every device he wants to use to access the web application, which is not practical in the current world where people use smartphones and public computers to access web applications.

4.2 Session security in web application frameworks

Often, web applications are built on top of a web application framework. A web application framework provides a web developer with the core functionality of a web application [94]. 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 frameworks to ensure security against session hijacking and session fixation attacks. We consider only the renewing of the SID, the use of secure connections, and the use of GET and POST parameters for session management, because these are the security measures that are most eligible to be included in a web application framework. The list of frameworks is by no means complete[116], but gives a good overview of how popular frameworks handle session attacks.

4.2.1 Tomcat

Apache Tomcat[1] is a much-used software implementation of the Java Servlet and JavaServer Pages technologies. It powers the websites of a.o. WalMart, Wolfram Research and CiteSeerX [107].

[1]More information about Tomcat is available on its website: `http://tomcat.apache.org/`.

Renewing of the session identifier on authentication is automatically done since Tomcat 6.0 [106]. In previous versions (since 5.5), it is possible to enable this behavior by setting the `changeSessionIdOnAuthentication` configuration attribute in Tomcat's Authenticator Valve [105].

Tomcat uses cookies for session management, but also accepts SIDs that are included in the URL. URL rewriting is also used by default when the client's browser does not support cookies. It is possible for a web developer to disable this behavior, and to obligate Tomcat to only accept SIDs in cookies. This is, however, rather cumbersome, because it requires the implementation of a filter that intercepts requests and disables the session IDs from their URLs [19].

Secure connections can be managed managed by either the JSSE or the APR SSL implementation. To enable Tomcat to use secure connections, the web developer must set the `protocol` attribute of the Connector configuration entry to use one of these two implementations [104]. A cookie can be set with the `secure` flag by using the `setSecure()` method of the `Cookie` class [103]. Session cookies that are set using an HTTPS connection automatically have the `secure` flag enabled [44].

4.2.2 Alfresco

Alfresco[2] is a complete content management system that runs on a J2EE application server like Tomcat [107]. It is used by companies like France AirForce, Cisco, Fox and KLM [3].

Alfresco neglects to renew a user's session identifier when he logs in. We tested the behavior using Alfresco's demo server[3] and found that a session fixation attack is, indeed, possible. Further investigation showed that a bug which addresses this issue was already reported [63]. Unfortunately, the bug already dates from October 2008 and has not been solved since. Requesting more information about session management in Alfresco on their IRC channel or in the Alfresco forums proved fruitless.

If Alfresco is deployed on top of Tomcat, session management and secure connections work in much the same way as they do in Tomcat.

4.2.3 Ruby on Rails

Ruby on Rails[4] (or RoR, for short) is a web framework written in the in 1995 conceived Ruby programming language. It is used in popular web applications like Twitter, Groupon and Github [89].

Renewing the session identifier is not automatically done on each authentication state change in RoR. There is, however, a supported way of implementing this: invalidating the old SID can be done by adding `reset_session` to the `SessionsController#create` action [114]. The official documentation advertises

[2] More information about Alfresco is available on its website: http://www.alfresco.com/.
[3] An account for the demo server can be obtained from http://www.alfresco.com/try/.
[4] More information about Ruby on Rails is available on its website: http://rubyonrails.org/.

this solution as only requiring one line of code, but mentions that session state must still be manually copied.

RoR supports only cookie-based session management by default. If the web developer wants his web application to use URL rewriting instead, he needs to specifically enable this [70].

To make a RoR web application use secure connections for certain pages, the `ssl_requirement` plugin[5] can be used. This plugin allows to specify for which pages the HTTPS protocol should be used [98]. Making sure that certain cookies are only sent over secured connections only requires the configuration option `ActionController::Base.session_options[:secure]` to be set to `true` (this option is set to `false` by default).

4.2.4 Django

Django[6] is a web framework built using the Python language. It is used by a.o. Ars Technica and The Washington Post [27].

Django renews the session identifier automatically when a user logs in. For this, two possible cases are considered [29]:

- If the login request contained a session identifier that corresponds to another logged in user, or if the request did not contain a SID at all, a completely new session is created.

- If the login request contained a session identifier that is not yet associated with a logged in user, a new *SID* is created. This SID is then associated with the state corresponding to the old SID [30]. The old SID is removed from the database, so it can not be used in the future to access the session information.

SIDs are only accepted via cookies in Django. This is a very clear-cut design decision [28], and requires the web developer to write his own middleware if he wants the web application to use POST or GET parameters instead [38].

Secure connections are handled by the underlying web server, and configuring certain pages to require HTTPS connections should be done in the configuration files for the web server. However, to make sure that Django will set the secure flag for all session cookies, the setting `SESSION_COOKIE_SECURE = True` should be added to Django's `settings.py` [6]. This configuration setting is disabled by default [48].

4.2.5 CherryPy

CherryPy[7] is another web framework built using Python. It tries to make building web applications as similar as possible to developing any other object-oriented Python program.

[5] The `ssl_requirement` plugin is available at https://github.com/rails/ssl_requirement.
[6] More information about Django is available on its website: http://www.djangoproject.com/.
[7] More information about CherryPy is available on its website: http://cherrypy.org/.

The CherryPy documentation claims that CherryPy provides protection against session fixation attacks [17]. In reality, however, only crafted session identifiers[8] are prevented. Indeed, it is still possible for an attacker to establish a session by visiting the web application himself, and to impose this session on a victim. To execute a successful login session fixation attack, the only thing required is that some data is tied to the attacker's session, since CherryPy only considers a session identifier to be permanent if some server state is attached to it. The fact that CherryPy is vulnerable to session fixation attacks was already discovered earlier by Schrank et al. [92].

CherryPy manages sessions exclusively via cookies [17]. As a consequence, session identifiers will not be accepted as GET or POST parameters.

As with Django, secure connections are handled by the underlying web server. By default, CherryPy does not set the `secure` flag for any cookies. To enable this flag for all session cookies, the session object should be initialized (by calling `cherrypy.lib.sessions.init()`) with the `secure` parameter set to `True` [17].

4.2.6 PHP

PHP[9] is a scripting language used for web development. While PHP is technically not a web framework, we include a discussion of PHP's session module in this section because PHP is estimated to be the server scripting language for over 75% of all websites [112].

Since PHP only supports the concept of sessions, and not that of users, there is no way for PHP to know when the authentication state has changed. As such, PHP does not renew the session identifier automatically when needed. Session fixation prevention can however be easily implemented by calling the `session_regenerate_id()` function every time a user's authentication state changes (e.g. in the login function), similar to the code snippet provided in section 4.1.1.

By default, PHP uses cookies for session management, but also accepts session identifiers passed via URLs [48]. To change this behavior, the line

```
php_flag session.use_trans_sid off
```

should be set in the *web server's* `.htaccess` configuration file. Alternatively, the line

```
ini_set('session.use_trans_sid', false)
```

could be added to the PHP code. Note that the previous example only works when an Apache-based web server is used to serve the PHP pages (which is most often the case) [9].

To use secure connections, PHP must be compiled with the `-with-openssl` parameter. To make sure that PHP sets the `secure` flag for all session cookies, the line

```
session.cookie_secure = 1
```

[8] Recall from section 3.3.1 that a crafted session identifier is an SID which was not generated by the server in advance.

[9] More information about PHP is available on its website: http://php.net/.

should be added to the `php.ini` config file [85]. Alternatively, the `session_set_cookie_params()` function could be called with the parameter `secure` set to `true` for every request [87]. By default, cookies don't include the `secure` flag.

4.2.7 Drupal

Drupal[10] is a complete content management system written in PHP. It powers the websites of The Economist, Symantec, and even The White House [33, 88].

Drupal automatically renews the session identifier when the user's authentication state changes [35]. It does this by calling PHP's `session_regenerate_id()` function [32]. While this has always been the default behavior of Drupal, some bugs were still present in versions preceding Drupal 5.9 and 6.3 [31].

Drupal makes sure that PHP's underlying session mechanism will only accept session IDs via cookies. It does this by calling the previously described `ini_set()` function with the relevant parameters during initialization [34]. Unfortunately, problems are still known to occur for some web hosts, for which the relevant parameters should be added to PHP's `.htaccess` file manually [37].

Secure connections and secure cookies in Drupal are handled via PHP's mechanisms. However, Drupal's 'Secure Login' module can be used to force certain pages to be loaded over a HTTPS connection [42]. To make sure that every cookie set over HTTPS has the `secure` flag enabled, this module also sets the `session.cookie_secure` flag to `true` since Drupal 7.

4.2.8 Overview

A summary of session security in the discussed web application frameworks is given in Table 4.1. As we can see, the popular high-level frameworks in our list (RoR, Drupal, Django) provide pretty good protection against session attacks, which reinforces our recommendation to use a high-level framework whenever possible (see section 2.3.4). Using a framework, however, does not relieve the web developer from making sure that all components are configured correctly.

	Renews SID on auth	Only accepts cookie-based SIDs	`secure` flag for SSL session cookies
Tomcat	since version 6	implementable	default
Alfresco	no	implementable	default
RoR	configurable	yes	configurable
Django	yes	yes	configurable
CherryPy	no	yes	configurable
PHP	no	configurable	configurable
Drupal	since version 5.9/6.3	yes	via module

Table 4.1: Comparison of session security in different web application frameworks

[10] More information about Drupal is available on its website: http://drupal.org/.

4.3 Server-side countermeasures

In this section, we describe some standalone server-side countermeasures to session hijacking and session fixation. These should be deployed separately, and are not part of any web framework.

4.3.1 Deferred loading (SessionSafe)

SessionSafe is a combination of different solutions to session attacks proposed by Johns et al. in 2006 [53]. One of these, called 'deferred loading', was created specifically for session hijacking.

The reasoning behind deferred loading is that cookies should be separated from the content of a page, to prevent an attacker from using the page content to steal the cookie. This is achieved by setting the cookie on a different subdomain (`secure.example.org`) than the web page itself (`www.example.org`). A page is then requested as follows (see Figure 4.1a):

1. Instead of getting the page directly, the user's browser requests a 'page loader' from the server. This page loader is a small HTML page that contains logic which will be executed at the client side to fetch the actual page.

2. The page loader makes the browser send the user's cookie to `secure.example.org`, and a request for the actual page to `www.example.org`. In both requests, the browser includes the same *request ID*, which was sent by the server as part of the page loader.

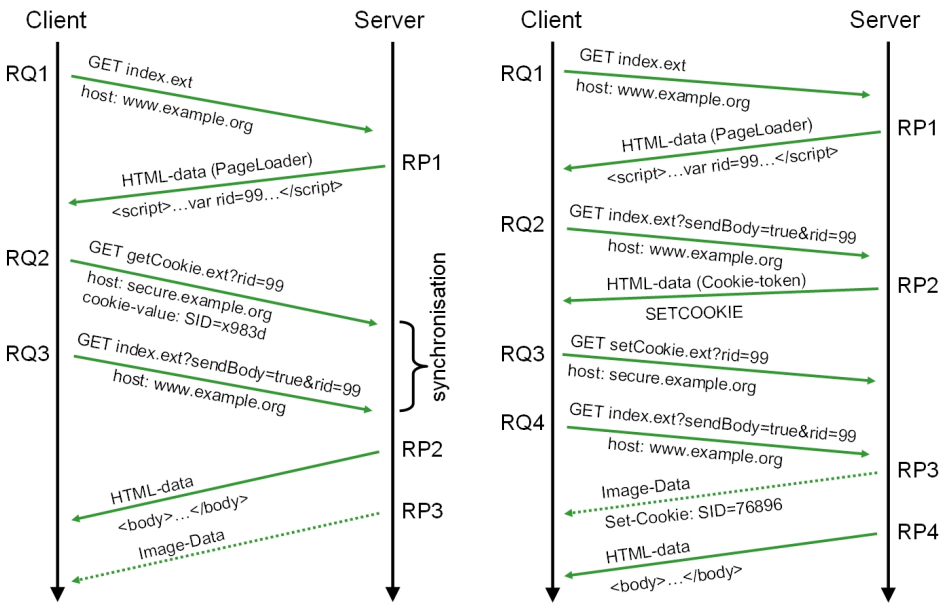
3. The web server checks that the cookie is correct, and that both requests were issued with the same request ID (and thus by the same page loader). If this is the case, it sends the requested web page as a response, and it invalidates the request ID for future requests.

4. The page loader displays the actual web page in the user's browser.

Because the cookie needs to be set for a different subdomain than the one the page is served from, it can not simply be included in the `Set-cookie` header of the HTTP response. Instead, an 'image' is included on the returned page, which is served from the domain `secure.example.org`. When the user's browser requests the image from this domain, the server is able to set the cookie in the response. The process of setting a cookie is graphically depicted in Figure 4.1b.

Deferred loading prevents session hijacking via XSS attacks. Indeed, when an attacker is able to inject script code in a web page on the `www.example.com` domain, this code has no access to cookies stored issued by `secure.example.com`. Because the page content itself is always served from the `www.example.com` domain, an attacker will never have the ability to access content on the `secure.example.com` domain.

Note that this approach does not prevent an attacker from executing a session fixation attack. Indeed, although script code is unable to access cookies stored



(a) Requesting a page (b) Setting the cookie

Figure 4.1: Deferred loading of a cookie [53]

in a different subdomain, it *is* able to set cookies for its parent domain, as was discussed in section 3.3.2. This allows an attacker to inject script code in a page hosted on `www.example.org` that sets a cookie for the domain `example.org`. Since `secure.example.org` is also a subdomain of `example.org`, requests made to this domain will contain the injected cookie.

4.3.2 SessionLock

SessionLock, proposed by Adida et al. in 2008 [1], tries to solve session hijacking by making session identifiers only available at the client side. For this purpose, fragment identifiers are used.

Fragment identifiers are mentioned in the URL specification [10] as a means for making the user's web browser scroll to a certain part of a web page. The fragment identifier is included in the URL as the part that follows the `#` character. For example, the URL `http://www.example.org/document.html#paragraph4` has `paragraph4` as its fragment identifier. Loading this URL in the browser will cause the browser to request *document.html* from the web server, and to jump to the element on that page which is identified by the string `paragraph4`. If no such element is available on the page, the web browser will simply ignore the fragment identifier part of the URL. Fragment identifiers are not sent to the web server on a request, but are instead handled by the browser to jump to the correct part of the page once it is loaded. The fragment identifier is available to client-side JavaScript via the

`document.location.hash` DOM element.

It is in such a fragment identifier that SessionLock stores the user's SID. Interaction between the user's browser and the server happens through the following steps (also depicted in Figure 4.2):

1. The user authentication happens over a secure (HTTPS) connection. If the authentication is successful, the web server sets the SID as a *secure* cookie, and redirects the user to a URL that includes the SID as its fragment identifier.

2. Subsequent requests are made over an insecure (HTTP) connection, and thus do not include the cookie set in the previous step. Instead, the user's browser attaches to every request a MAC, generated with the (secret) SID, of the tuple `(request-URL, timestamp)`. To avoid having to modify the user's browser to execute these steps, a small piece of JavaScript code is included in the page. This JavaScript code intercepts all user requests, and attaches a timestamp and MAC to every request before it is forwarded to the server.

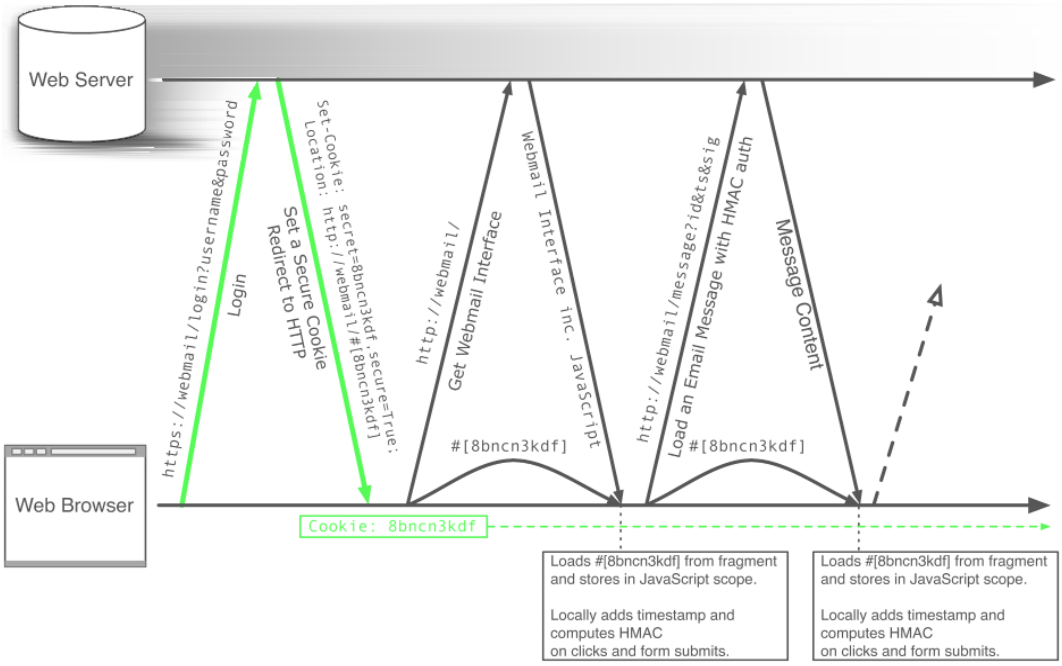


Figure 4.2: The SessionLock protocol, with SSL data indicated in green [1]. Note how the use of the fragment identifier effectively creates a client-only channel from one page to the next.

By making session identifiers only available at the client side, a MitM attacker is not able to capture the SID since it is never sent over the network. The authentication happens over a secure connection because otherwise, the SID would still be sent over an unencrypted channel when it is set by the server. A variant which uses the

Diffie-Hellman protocol [26] instead of encrypted connections when authenticating is also presented in the paper [1].

A major advantage of this approach is that, while users use HTTPS to authenticate (and thus never have their password sent in clear text), a secure connection is not required for subsequent requests. This eliminates the biggest drawback associated with using SSL, namely that of high costs (see section 4.1.3).

To make sure that the SID is remembered at the client side, JavaScript code is included in every page returned by the server. This code rewrites all URLs on the page to include the SID as a fragment identifier. Note that the server cannot rewrite these URLs itself, because then the SID would be visible on the network when the page is returned to the client. In case the SID does get lost, JavaScript can still recover it by opening an invisible iframe which loads the URL `https://example.org/login/recover` over a secure connection. Within this secure connection, the secure cookie which was set in step 1 will be available. Thus, all the JavaScript code has to do is get the SID value from this cookie, and reload the current page with the SID appended as the fragment identifier.

It is important to note that this session hijacking countermeasure only protects against passive MitM attacks and against the SID leaking via the `Referer` header. It does *not* protect the user from accidentally leaking his SID by forwarding a URL [1]. Moreover, it does *not* protect the user from active attacks. Indeed, since the SID is stored in a way which makes it available to JavaScript, an attacker performing an XSS attack will still be able to access the SID [21]. For the same reason, the proposed countermeasure does not protect against session fixation either.

Another disadvantage of this approach is that it is only usable for URLs that can be rewritten using JavaScript. Indeed, when binary objects (i.e. Flash) are used to generate dynamic links, URL rewriting will not work [21].

4.3.3 One-time cookies

In a paper written by Dacosta et al. [21], a solution to session hijacking is proposed wherein session identifiers are replaced by one-time cookies which are renewed on every request. This is done as follows:

1. The user authentication happens over a secure (HTTPS) connection. In this authentication, a secret seed and a session secret are generated.

2. Subsequent requests are made over an insecure (HTTP) connection. In the i-th request, the user's browser attaches the outcome of a hash function applied i times to the secret seed. Also attached is the current sequence number (i, in this case), the user's username, a nonce, and – most importantly – a MAC, generated with the session secret established as part of the authentication, of the triplet *(username, request-URL, hash outcome)*.

It is crucial that the server only allows requests containing a sequence number higher than the last one received. This prevents attackers from replaying captured one-time cookie information. Since the attacker does not know how to generate the next

one-time cookie from a previously captured one, and because a previous cookie can not be replayed, the approach renders a captured cookie essentially worthless. Moreover, the request-URL is included in the MAC to make sure that a certain one-time cookie can only be used to access one specific resource.

To prevent the session seed and session secret from leaking via XSS attacks, both are made inaccessible to JavaScript code. This is viable for two reasons: firstly, the client modification that is needed for one-time cookies to work is implemented as a browser extension, which makes it able to separate the authentication information from the page itself. Secondly, because one-time cookies will be exclusively used for authentication purposes, other cookies that need to be accessible via JavaScript can still be used in the regular way.

Aside from preventing session hijacking attacks, this approach will also prevent session fixation attacks. Indeed, as soon as the user authenticates, a new session secret and secret seed – both unknown to the attacker – are established. This is similar to renewing the SID on every authentication change.

Because the sequence number is part of the request, the server knows how many times the hash function should be applied to the secret seed. Naturally, the server does not need to apply the hash function i times for every request. The computational overhead can be greatly reduced by remembering the previous result, and applying the hash function only $i - i'$ times (where i' is the number of times the function was applied to get the previous result) to this value.

As was the case with SessionLock, one-time cookies also use HTTPS for authentication and HTTP for all other requests, combining the security of HTTPS with the speed of HTTP.

Possible disadvantages of an approach that uses a new SID for every request were already discussed in section 4.1.1. However, these do not apply to this particular solution. Firstly, because future SIDs do not need to be explicitly issued by the server, problems with concurrent requests can be avoided by equipping every request with a higher sequence number than the previous one. Similarly, asynchronous requests can be issued with higher sequence numbers than the last issued request, and browser plugins can be modified to use one-time cookies computed by the browser. Secondly, server synchronization is less of a problem because the policy of the servers could be relaxed to allow any sequence number in a request, as long as it is higher than the last one they received. In this case, subsequent requests by a client could be made to different servers, while synchronization between servers would be limited to the moments a user authenticates. Note that this policy would permit an attacker to replay a one-time cookie, as long as he issues the request to a server different from the one the user issued his request to. Fortunately, the damage is limited by the fact that the one-time cookie can only be used to access one specific resource, thanks to the request-URL which is included as part of the MAC.

Compared to using regular session IDs over an unencrypted connection, both the client and the server need to perform more computations, and both have to keep more state. However, as is shown in the paper, using secure connections for every request is still much more computationally intensive [21]. Thus, requiring some extra computation compared to using regular session IDs over an unencrypted

connection could be considered a minor trade-off to be able to provide secure user authentication.

The paper also describes that in the current implementation, both the server and the client need to be adapted to support one-time cookies. The authors are, however, exploring the idea of modifying one-time cookies to provide a version that does not require a browser extension [21].

4.3.4 Session fixation solution by Johns et al.

Johns et al. [54] present a solution to login session fixation attacks in the form of a server-side reverse proxy. This allows web applications that have already been deployed in the past to be patched against session fixation afterwards. The proxy works by introducing a second session identifier, called the 'proxy session identifier' (or PSID), which is tightly secured against login session fixation. The proxy works as follows (see Figure 4.3):

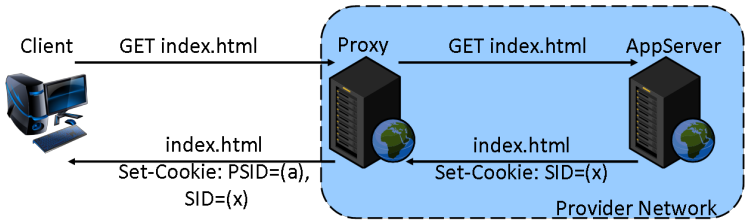
1. When a request without a PSID is received by the proxy, it is regarded as the user's first request. In this case, the proxy strips all authentication data from the request and generates a new PSID value. The new PSID is then included in the `Set-Cookie` header of the server's response when the response passes through the proxy.

2. When a cookie is set by the server (via the `Set-cookie` header), the proxy associates the PSID that was sent in the request (or the new PSID, if no PSID was present in the request) with the new SID from the server.

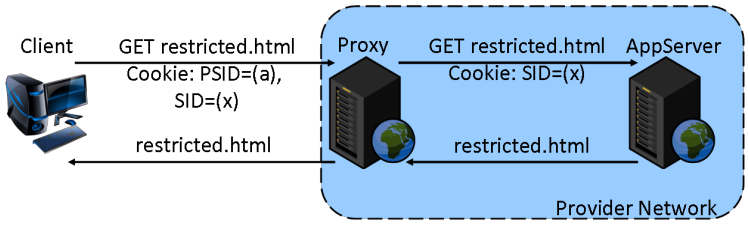
3. When a request containing a SID is received by the proxy, it checks whether the attached PSID is valid for the attached SID. If it is, the PSID is stripped, and the request is forwarded to the web server. If the PSID is not valid, or if no PSID is present, both the PSID and the server's SID are stripped from the request before it is forwarded. This causes a request with an invalid PSID to be void of any authentication data when it arrives at the server.

4. To make sure that the PSID is secure against login session fixation attacks, a new PSID is generated – and sent to the user – whenever the user changes its authentication state. Note that 'changing the authentication state' is not the same as 'receiving a new SID from the server', since it is possible that the server does not correctly renew the SID on every authentication state change.

To see why this solution protects against login session fixation, consider the scenario where the attacker was able to successfully force a SID and PSID upon the victim's browser. As soon as the victim logs in, his PSID is changed by the proxy. When the attacker subsequently tries to make a request using the victim's SID, he is only able to include the old PSID into the request. Because the association between the SID and the old PSID has been invalidated, the proxy will strip the SID from the request before forwarding it to the server. This causes the attacker's request



(a) Introducing the PSID



(b) Verifying the PSID

Figure 4.3: Protecting against login session fixation with a server-side proxy [54]

to behave as if it was an unauthenticated request, and prevents the attacker from taking over the victim's session.

To allow the proxy to detect when a user's authentication state changes, the web developer has to configure the proxy with the names of the GET and POST parameters that contain password data. When such a parameter is present in a request, the proxy considers it as an authentication request and renews the user's PSID.

The paper mentions that the same solution could also be applied at the framework level, where it is often known which parameters contain password data. The disadvantage of such an approach would be that the solution is then tied to a specific framework, instead of universally deployable. Moreover, if the framework provides session management, a much better solution is to make it renew the SID on every authentication state change itself, instead of introducing a second SID (i.e. the PSID).

4.4 Client-side countermeasures

In this section, we focus on session hijacking countermeasures at the client side. The advantage that these approaches have over server-side countermeasures is that they protect a user against session hijacking, even when the web application itself is not secure. Because of this, it is the *user* who can make sure that he is protected against this kind of attack, without having to rely on the web developers of all web applications he uses to implement session hijacking protection. Client-side countermeasures to session *fixation* attacks were, to our knowledge, non-existent at the time of writing [13].

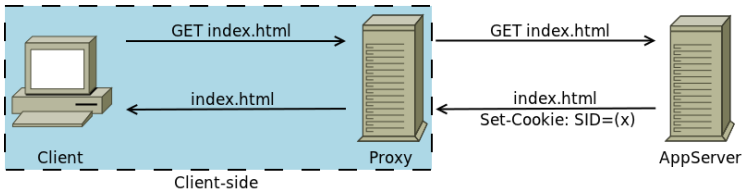
4.4.1 SessionShield

SessionShield is a client-side proxy providing protection against session hijacking attacks. It was proposed by Nikiforakis et al. in 2010 [77]. SessionShield is based on the idea that session identifiers should not be available to code running in a user's web browser.

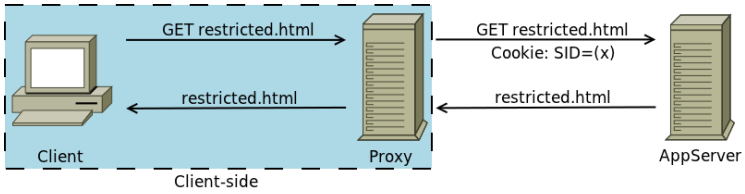
The proxy keeps session cookies unavailable to attackers by storing them outside of the browser. It does this as follows (see Figure 4.4):

1. When a session cookie is set by the server (via the `Set-Cookie` header), the proxy stores this cookie in his internal database, and strips it from the response before it is sent to the client.

2. On every outgoing request, the proxy queries its internal database using the domain of the request as the key. All cookies that are present in the database for this domain are then added to the outgoing request.



(a) An incoming response containing a new SID.



(b) An outgoing request by the browser does not contain any cookies. The proxy attaches the cookies for this domain that were stored in the database.

Figure 4.4: Protecting against session hijacking with a client-side proxy

To identify which cookies are session cookies, SessionShield uses a sophisticated SID detection algorithm. This algorithm considers a cookie to be a session cookie in one of the following cases:

- The name of the cookie is in a list of known session identifiers, such as `phpsessid`, `aspsessionid` and `jspsession`.
- The name of the cookie contains the substring "sess", and its value is more than 10 characters long.
- The value of the cookie passes a 'SID probability check', which consists of checking the string's information entropy, its correlation with the uniform distribution, and the number of dictionary words contained in it.

SessionShield's rationale is the same as that of `HttpOnly` cookies. However, the approach is taken one step further: SessionShield makes sure that cookies will never be visible in the browser, preventing an attacker from reading cookies even if he is able to seize control over the victim's browser in any way. This not only prevents session hijacking via XSS attacks, it also prevents rogue browser extensions from capturing a user's session cookie [102]. Another advantage SessionShield has over HttpOnly is that it works at the client side, allowing the user to be secure even when the web application fails to set the `HttpOnly` flag for all session cookies.

Note that SessionShield does not protect against session fixation attacks. Indeed, while SessionShield attaches additional cookies from his own cookie store, it does not strip any cookies it received from the browser. Because of this, an attacker is still able to inject a cookie in the user's browser, which will then be forwarded via SessionShield to the web server.

A disadvantage of implementing SessionShield as a proxy occurs when different browsers are using SessionShield at the same time. In this case, the proxy will attach cookies that were set for one browser to the other browser's outgoing requests, causing the user to be authenticated in both browsers when he has only explicitly logged in with one of them. Moreover, clearing private data in the browser, or using the browser's 'private browsing' functionality, has no effect on SessionShield, causing the user to be authenticated even when he expects not to be. We will discuss other advantages and disadvantages of implementing a client-side solution to session attacks as a browser extension instead of a proxy when we develop our own solution in the next chapter.

4.4.2 Noxes

Noxes, developed by by Kirda et al. in 2006, is a client-side solution to mitigate cross-site scripting attacks [59], and should therefore technically be described in the next section. However, since it is developed specifically to prevent information leakage from one domain to another, we discuss it here.

Noxes works in a similar way to a personal firewall like ZoneAlarm[11] or Windows Firewall[12]. However, instead of controlling the connections for all processes running on a user's machine, it only handles requests made by the user's browser. This allows Noxes to offer more fine-grained controls which are specifically tailored to web applications. For example, whenever a JavaScript request is trying to send certain information to a domain that is not known by Noxes, the user is presented with a prompt that asks for user confirmation. The user is able to accept or deny the request, and to create a firewall rule for similar future requests.

Such an approach is obviously very cumbersome: the user will have to explicitly allow every request that is made to a new domain. Because of this, Noxes includes a

[11] ZoneAlarm is available from http://www.zonealarm.com/.

[12] Windows Firewall is included by default on every version of Microsoft Windows since Windows XP Service Pack 2. More information is available on http://technet.microsoft.com/en-us/network/bb545423.aspx.

```
<html>
...
<img src="http://attacker.com/bit0_0.jpg">
<img src="http://attacker.com/bit0_1.jpg">
<img src="http://attacker.com/bit1_0.jpg">
<img src="http://attacker.com/bit1_1.jpg">
...
<script>
    for [i=0 to 100] {
        if (cookiebits[i] == 0)
            <contact http://attacker.com/biti_0.jpg>
        else if (cookiebits[i] == 1)
            <contact http://attacker.com/biti_1.jpg>
    }
</script>
</html>
```

Figure 4.5: Pseudo code for a JavaScript-based attack that transmits the SID to an attacker's domain using only static links (based on [59])

list of safe scenarios. When such a scenario occurs, the request is allowed without consulting the user. The following scenarios are considered safe:

Requests to the same domain When the base domains for the request and the page the request originated from are the same, information leakage to another domain is not possible. Noxes checks whether a request originated from the same domain by comparing the `Referer` HTTP header of the request to the domain of the requested web page.

Static links When links do not contain dynamic components, an attacker is unable to embed the victim's SID as part of the URL. Indeed, an attacker can only create the URL after he has read the domain's cookie value, which then has to be dynamically included in the URL.

Static links do not completely prevent a SID from leaking to another domain. Because of this, Noxes includes two exceptions to the second 'safe rule'. Firstly, consider the scenario where an attacker includes in the web page a static link to his own domain for each of the two possible bit values for every bit in the SID. He is then able to use JavaScript to load the links that correspond to the individual bits in the SID, one by one (see Figure 4.5). Because of this, Noxes allows maximally k static links to the same external domain, where k is a customizable threshold.

A second exception to the second 'safe rule' is made for pop-up and pop-under windows to a different domain. Using these, an attacker can open a window containing a static link to a page on his own domain, while setting the window's title to the value of the victim's cookie. This makes him able to read the cookie value from

within his own page, by using JavaScript on his own page to access the value of the `document.title` attribute. Because of this, Noxes presents an alert message to the user when it detects that a pop-up or pop-under window to a different domain is opened, regardless of the link being static.

Unfortunately, Noxes' complete policy still allows an attacker to leak the victim's SID to his own domain. Indeed, as Nikiforakis already noted [77], an attacker can create an XSS attack which, instead of `<script>` tags, injects an HTML-tag that statically references a URL to the attackers domain including the SID value. Furthermore, an attacker is still able to make the victim's browser load multiple pages of the originating domain, with every page sending maximally k characters (k being the maximum of allowed outgoing links to an external domain) of the victim's SID to the attacker's server.

Another problem with Noxes is that it uses the `Referer` header to check whether a request comes from a different domain than the one of the requested web page. As we saw in section 3.2.2, the `Referer` header is often stripped [8]. This will cause Noxes to present an alert even for some requests that go to the same domain as the one they originated from.

It can be concluded that Noxes is too complex to be considered a long-time solution against session hijacking attacks. Firstly, user interaction is needed for many requests, which is often cumbersome for the user. Moreover, less technically inclined users will not be able to judge whether a request is safe or not. Secondly, as can be seen from the 'safe rule' exceptions, lots of possible attack vectors will have to be accounted for by the firewall. This will cause Noxes to be often lacking in security.

4.4.3 Dynamic tainting and static analysis

Similar to Noxes, the solution developed by Vogt et al. [111] that we describe here is actually meant to mitigate data leakage in XSS attacks. It uses a combination of dynamic data tainting and static analysis to prevent sensitive data (e.g. session cookies) from leaking to a different domain than the one it originated from.

Tainting is a technique which makes sure that sensitive data keeps being regarded as such, even when it is passed around. For this, sensitive data is given a 'taint', which is propagated when (part of) the tainted data is assigned to a different variable, or used as part in an arithmetic or logic operation. For example, consider the following snippet of JavaScript code, wherein `document.cookie` is tainted:

```
var arr = []; // arr.length = 0
if (document.cookie[0] == 'a')
    arr[0] = 1;
if (arr.length == 1)
    y = 'a';
```

Because `document.cookie` is tainted, `arr` will be tainted as soon as the first if-test is executed. Upon executing the second if-test, `y` is also tainted, which correctly indicates that information about the sensitive (cookie) data is present in this variable.

When it is detected that a web page tries to send tainted data to a third party, the user is asked to allow or deny the actual transmission of this data, much in the same way as was the case with Noxes. Sending of tainted data to a third party can be achieved in many ways. Some examples of the methods the countermeasure detects are changing the location of the current web page, changing the source of an image, and automatically submitting a form [111].

Unfortunately, several hidden channels that can be used to transmit sensitive information remain undetected by this solution [77]. Moreover, Russo et al. described how the protection technique can be circumvented by both encoding secret information into the structure of the DOM tree and exploiting tree navigation [91]. Another disadvantage of this solution stems from the fact that it needs to ask for user confirmation. This leads to the problems which were already discussed when describing Noxes.

4.4.4 Overview

In Table 4.2, an overview of the different server-side and client-side session attack countermeasures is given. From this table, we can draw two conclusions: firstly, while lots of work has been done to mitigate session hijacking attacks, the session fixation attack is often overlooked. Secondly, none of the discussed countermeasures handle session fixation attacks entirely at the client side. In fact, we do not know of any countermeasure to session fixation that only requires modifications to be made at the client side [13].

	Needs modification	Prevents session hijacking	Prevents session fixation
Deferred loading	server	via JavaScript	no
SessionLock	server	via MitM & `Referer`	no
One-time cookies	server & client	yes[a]	yes
Johns et al.	server	no	yes
SessionShield	client	via the browser	no
Noxes	client	via JavaScript (incomplete)	no
Dynamic tainting	client	via JavaScript (incomplete)	no

[a] When using the same server for every request, session hijacking is prevented entirely. When using different servers, an attacker is prevented from accessing a resource different from the one the victim used the one-time cookie for.

Table 4.2: Comparison of different session attack countermeasures

4.5 Script injection countermeasures

Over the years, different solutions to XSS have been proposed, both at the server and at the client side. In this section, we describe the general principles behind these approaches, without going into too much details.

4.5.1 Server-side countermeasures

At the server side, two general categories of XSS countermeasures exist. The first one is that of *input validation*, or *input filtering*, as is available in solutions like Sanctum's AppShield [60]. In such an approach, all script code is stripped from user input, in order to prevent any user-submitted executable code to be sent to the browser. This is done either by removing certain substrings (e.g. `<script>` tags), or by replacing characters with their HTML-safe counterparts (e.g. replacing `<` by `&lt;` and `>` by `&gt;`), which will be converted back to the original character once they are rendered by the browser.

A second category of server-side XSS countermeasures tries to detect user-injected script code when it is returned to the victim. One solution that does this is XSS-Guard, developed by Bisht et al. [11]. In this solution, two versions are generated for every returned page: one without, and one containing user input. XSS-Guard then compares both pages to see if there are any differences in JavaScript between the two pages. If differences are found, the relevant parts are stripped from the response before it is sent to the client. This prevents JavaScript in the user-input from being returned to the client.

Another such solution, called SWAP, was presented by Wurzinger et al. in 2009 [118]. It works by assigning a so-called 'script ID' to every piece of JavaScript generated by the server. When script code is found in a server response, it is checked whether it has a valid script ID associated to it. If it does, the response is forwarded to the client. If the piece of code does not correspond to a valid script ID, the response is discarded, and the user is presented with a screen that notifies him of the suspected XSS attack.

For both categories of server-side countermeasures, a mechanism that automatically detects JavaScript is needed: the first category needs it to find JavaScript in user input, whereas the second category needs it to detect JavaScript in a returned page. A difficulty in detecting JavaScript is that it has to be able to recognize all different ways in which script code can be injected (see section 3.1.2 and [52]), including all possible encodings and taking into account differences in various browsers (called *browser quirks*). AppShield approaches this difficulty by using pattern matching on known attack vectors [60], whereas both XSS-Guard and SWAP use Firefox' HTML parser to detect which parts of a web page contain script code that will be executed by Firefox [11, 118]. While the first approach will possibly overlook some attack vectors, the second approach will cause every piece of HTML that will trigger code to be executed in the Firefox web browser to be detected. A disadvantage of the second approach, however, is that it does not take into account differences between various browsers. This makes it possible that, even if Firefox does not consider a particular piece of HTML code to be valid JavaScript, another browser might execute it as script code. Hence, an attacker will still be able to bypass the proxy by injecting JavaScript code which is specific to the Internet Explorer web browser.

A disadvantage shared by both approaches in the second category is that they do not prevent binary Flash objects from being injected. Thus, an attacker will still be able to use Flash to inject JavaScript on a page, regardless of the countermeasure

being in use [11, 79].

Another disadvantage, limited to SWAP, is that an attacker will still be able to inject script code in order to execute a DoS attack. Indeed, because script code is only detected *after* it has been injected at the server, an attacker is still able to inject script code. Moreover, because server responses containing injected code are discarded entirely by SWAP, all responses containing the attacker's injected code will be prevented from being sent to the client. As such, injecting script code now allows an attacker to render a page unavailable to the users of a website.

4.5.2 Client-side countermeasures

At the client side, four important countermeasures have been developed to mitigate cross-site scripting. Two of them, Noxes and dynamic tainting, were already discussed in the previous section. We discuss the remaining two countermeasures here.

The simplest countermeasure, called NoScript, allows the user to choose for each piece of script code whether it should be allowed to be executed [69]. It also provides specific protection to XSS attacks. Furthermore, allows other plugins and embeddings (e.g. Java and Flash) to be blocked. By default, it uses a whitelist approach, which implies that all scripts are blocked by default unless the user manually specifies an exception for a particular piece of JavaScript. As was already described in the context of Noxes, this is very cumbersome, especially for less technically inclined users. Moreover, it has been shown that an XSS vulnerability on a NoScript domain can be used to run JavaScript from any website, despite NoScript being enabled [81].

A second countermeasure, called HProxy and developed by Nikiforakis et al. [78] was originally developed to prevent SSL stripping attacks. In such an attack, the attacker acts as an active MitM to force users to communicate over an insecure channel with the web server. To detect such attacks, HProxy compares different HTML and HTTP elements in the response from the server that could be misused by a MitM attacker to what they looked like in previous page responses. What is relevant for our discussion is that one of these elements is JavaScript code, which is checked as follows:

1. When the first response is received by the server, a second request is made by HProxy. The first response is then compared to the second one in order to identify the dynamic JavaScript parts of the web page, i.e. the parts of the script code that differ on every page request.

2. For all subsequent responses, it is checked whether any non-dynamic JavaScript on the page differs from the JavaScript returned in the first response. If it does, the web page is marked as 'unsafe', and the response is dropped.

While HProxy is definitely a step in the right direction, enabling users to be secure against XSS attacks without requiring support from the web application, it does have some disadvantages. Firstly, current versions of web pages are always compared to the first version that was received. This causes updated versions of websites to be blocked. Moreover, it prevents the detection of attacks that occur the first time a

page is loaded when HProxy is in use. A second disadvantage is that HProxy suffers from false positives: the paper mentions that, even with advanced JavaScript change detection[13] in place, 3% of legitimate server responses are mistaken for malicious responses.

4.5.3 Hybrid approaches

Hybrid approaches mitigate XSS attacks by letting the server define a policy, while making the client check whether a returned page adheres to this policy.

'Signed scripts', proposed by Mozilla [90], provide a way for web servers to digitally sign the scripts they send to a client. The client is then able to verify that the scripts which are present in a web page are actually issued by the web server, and not modified in transit by an attacker. Currently, signed scripts are used by the Firefox web browser to allow a script to request extended privileges (e.g. modifying the browser's preferences). However, signed scripts are also used by HProxy, described earlier, to reduce the number of false positives by always allowing a correctly signed script to be executed.

Another hybrid approach is that of 'Browser Enforced Embedded Policies' (or BEEP for short), developed by Jim et al. in 2007 [52]. Here, the server includes an additional JavaScript function in the response, which will enable the browser to check that the page it is included in does not include injected code. Whenever a browser that enforces BEEP encounters a script while rendering a page, it invokes this special function with an identifier of the JavaScript code as its argument. If the function returns `true` then the script is deemed acceptable and will be executed; otherwise it will be ignored. The browser describes two possible policies that can be implemented in such a JavaScript function: *whitelisting*, in which the function returns `true` if the SHA1 hash of its argument corresponds with one of the 'safe hashes' included in a whitelist, and *DOM sandboxing*, in which parts of the web page that include user content are marked by setting their `class` attribute to `noexecute`. In this second policy, the function checks that its argument is not part of such a sandbox, and returns `false` if it is, preventing execution of the JavaScript code.

We conclude from this section that XSS protection is very hard to get right. Oftentimes, all different attack vectors are handled separately, which causes some of them to be unavoidably forgotten. Secondly, many XSS countermeasures require server-side cooperation, which renders them unusable for the majority of the web from the user's perspective. The approaches that are made work exclusively at the client side, on the other hand, either require extensive technical knowledge (NoScript), or suffer some amount of false positives (HProxy).

[13] The advanced JavaScript change detection consists of the practices described in this section, together with the whitelisting of signed scripts, as it is described in the next section. Without the whitelisting in place, false positives occurred for about 8% of legitimate responses

Chapter 5

A client-side solution to session fixation and session hijacking

In this chapter, we propose a client-side solution to the session fixation attack described in section 3.3. This proposed solution was also submitted as a paper, co-authored by Philippe De Ryck, Nick Nikiforakis, Lieven Desmet and Frank Piessens, to the W2SP 2011 conference [13]. This chapter includes some of their wordings and data.

The reason for developing a solution at the client side is that the user incentive for using a web application that is secure is often larger than the web developer incentive for creating one [54]. To our knowledge, there exists no other practical client-side solution to this attack [13].

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 [117] and lots of websites are vulnerable [15]. Ideally, these problems would be solved by finding a complete solution to XSS attacks. Unfortunately, as we saw in section 4.5, solutions to XSS are often lacking, and few have been developed that work exclusively at the client side. 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.

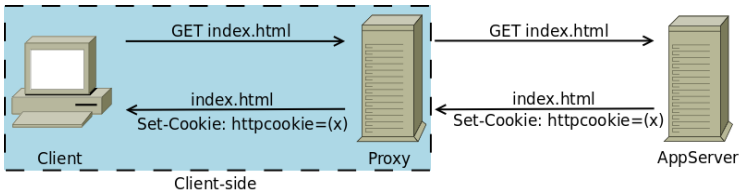
We first discuss the client-side policy that was developed to counter session fixation. Afterwards, we implement this policy as a Firefox add-on, and we provide a thorough evaluation. Lastly, we describe how the add-on was extended to also provide a solution to the session hijacking attack.

5.1 Principle

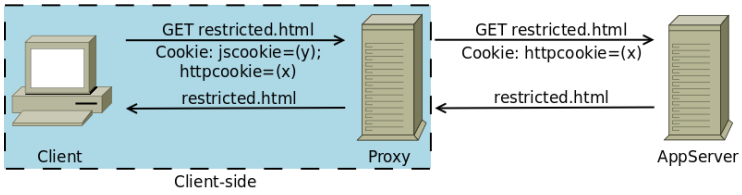
The reasoning behind the client-side solution developed is that session IDs will never be set over an untrusted channel, only to be requested over a trusted channel later on. We consider HTTP to be trusted, since this channel is controlled entirely by the web

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 an 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.

[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.

Similarly, the policy is only applied to session cookies set via an untrusted channel, that are sent over a trusted channel afterwards. Although we assume that cookies which are requested over a trusted channel will never be set over an untrusted channel, cookies can still be set over an untrusted channel and be requested over this channel afterwards. Because we don't want to break web applications which set cookies via JavaScript and subsequently request them via JavaScript, we allow this type of behavior.

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

	Regular 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 the SID detection algorithm defined by Nikiforakis et al. [77] (see section 4.4.1), 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[2] 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 [58]. Afterwards, it is calculated how many bits would be needed to encode the string [77].

[2]Based on [77], we consider 10 characters to be sufficiently long.

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[3]. 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 an HTTP proxy. The most important advantage is visible when an encrypted (HTTPS, see section 4.1.3) connection is used. In this case, a browser extension is already behind the regular SSL endpoint. An HTTP proxy, on the other hand, 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. Moreover, since a proxy has no way of distinguishing between the various applications using it, different browsers would be allowed to use each other's cookies.

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.

A second disadvantage is that, on some browsers, extensions are able to interfere with one another. In Firefox, for example, all extensions share the same namespace. This allows rogue browser extensions to thwart security extensions [7, 102].

[3]This add-on is available for download at https://spideroak.com/share/IJZGC3I/pub/home/bram/Openbaar/NoFix.xpi. It can also be tested by booting the live CD included with this thesis.

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 [74].
- 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 [74].
- The `storageService` allows us to persistently store data in a SQLite database [75]. This service is also used by Firefox internally to store data on the local machine [13].

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) [73]. 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 [18]. 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. 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 [119] 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 [73] 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.

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. [97, 72, 13]

5.4.3 Performance

The user experience can also be negatively impacted by a large overhead induced by client-side protection mechanisms [13]. We quantify this performance overhead by performing two experiments.

The first experiment consisted of timing the delays that were introduced by the add-on during an everyday web browsing session. For this, timers were added to the add-on code. We measured that the average delays for requests (where it is checked whether a cookie is allowed) and responses (where cookies are set) were 1.25 ms and 0.25 ms, respectively.

The second experiment[4] that was performed compares the *page* load times in Firefox with and without the add-on enabled for the top 1000 most visited pages on the Internet, according to Alexa [2]. For this experiment, we used a setup similar to the one used to evaluate SessionShield [77], where the network inconsistencies are eliminated by hosting all the pages locally. A fake local DNS server and fake local web server are used to serve the captured pages. For additional resources (e.g. images, scripts), the local web server responds with a 404 'page not found' status code. The actual experiment consists of using the ChickenFoot add-on [12] to automate the process of starting a timer, instructing Firefox to load a page, and stopping the timer. This scenario is repeated three times for every web page, both with and without the add-on enabled. The average page-load time without session fixation protection is 195.407 ms. When our add-on is enabled, the average page-load time is 198.242 ms [13]. Thus, the average overhead introduced by our add-on is approximately 3 milliseconds, which is negligible compared to the more than 0.5 seconds average loading time for pages on the Internet [76].

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 site's behavior in the form of false positives (i.e., regular cookies which are incorrectly marked as session cookies) [13]. 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.

[4]This experiment was performed by Nick Nikiforakis [13].

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 does not 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 [13]. 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[5] and Yahoo! Web Analytics[6]. 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 [101, 45]. 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 should not be stripped in the add-on.

An approach using heuristics will also always exhibit some false negatives. However, when a SID is incorrectly marked as a regular cookie in our session ID detection mechanism, this means that it does not satisfy the safety requirements for SIDs presented in section 2.3.1 [77]. Because of this, the SID would already be susceptible to being guessed by an attacker, which will still be a problem even if the user would be secure against session fixation attacks. When the web developer fixes the problem of insecure session IDs, the session fixation problem will automatically be solved by our solution.

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

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

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

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 read via HTTP (see section 4.1.2 for more information). Thus, we let the browser handle the blocking of HTTP session cookies over JavaScript. Figure 5.2 shows the adapted version of Figure 5.1a, with the solution extended to provide session hijacking protection.

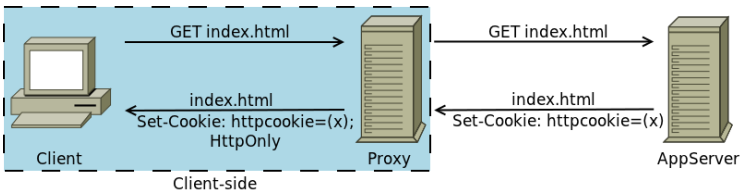


Figure 5.2: The extended client-side solution, providing both session fixation and session hijacking protection

5.7 Comparison to other session attack countermeasures

In this section, we compare our solution to some of the countermeasures discussed in the previous chapter.

5.7.1 Renewing the session identifier

Renewing the session identifier was mooted in the previous session to be the best solution against session fixation attacks. Unfortunately, this solution can only be implemented at the server side. Because of this, Johns et al. suggest that their proxy protecting against session fixation could also be implemented at the client side, renewing the PSID instead of the SID to protect the user [54]. It would then be expected that the proxy makes sure that an attacker can not take over a victim's session, since he does not know the victim's PSID. Unfortunately, this is not the case. To see why, consider the scenario depicted in Figure 5.3. In this scenario, the attacker uses HTTP response splitting (as described in section 3.3.2) to inject a cookie. Because the proxy sees the SID as a normal HTTP session cookie, he will attach a valid PSID to this cookie, and send it to the victim. When the victim logs in, he attaches both SID and PSID, of which the proxy will only forward the first one to the web server. The victim is now logged in with the fixated SID, as before. Since the attacker's requests do not have to pass through the proxy, he can use the fixated SID to act as the victim on the server.

Note that in the previous discussion, we only talked about injecting a SID via HTTP response splitting. The reason for this is that the proxy has been developed

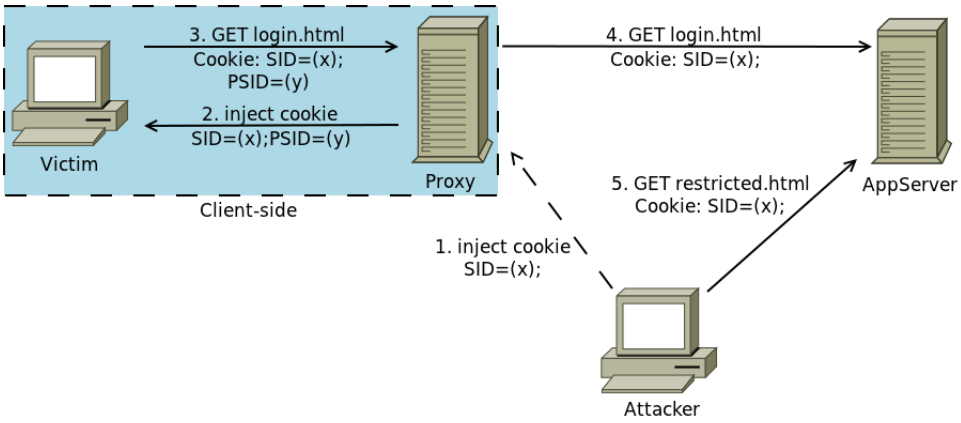


Figure 5.3: Attack on a client-side implementation of the proxy proposed by Johns et al.

specifically for HTTP cookies, which will prevent it from detecting cookies being set via other channels such as JavaScript or `<meta>` tags. In fact, this causes a client-side implementation of the proxy to effectively act like our own solution: session cookies which have been set over a channel different from HTTP will not have a valid PSID attached to them, and will as such be stripped from the requests by the proxy.

It is not possible for a client-side proxy to be configured for every possible web application. Because of this, the paper argued that such a solution should be able to automatically identify session cookies. Our solution uses a SID detection algorithm exactly for this purpose.

5.7.2 HttpOnly

Apart from being used as the mechanism to make our solution able to prevent JavaScript access to HTTP session cookies, HttpOnly was a great influence to the development of our session fixation solution. Indeed, our solution can be seen as the dual of HttpOnly cookies: where the `HttpOnly` flag makes sure that HTTP cookies are unavailable to untrusted channels, our solution makes these untrusted channels unable to set cookies which will be sent over HTTP.

A difference between HttpOnly and our solution is that the `HttpOnly` flag must be explicitly set at the server side, whereas our solution reasons autonomously about which cookies should be made unavailable. Moreover, our policy extension could be seen as a proxy that automatically enforces HttpOnly behavior for cookies that need it.

5.7.3 Deferred loading and one-time cookies

In the deferred loading approach, proposed by Johns et al. [53] it is reasoned that cookies should be kept separate from the page content, to prevent an attacker from stealing a cookie by altering the page. Similarly, one-time cookies [21] are never

made available to JavaScript. Our approach is similar in that it keeps session cookies unavailable to JavaScript. A difference is that our approach identifies session cookies after they have been issued by the web server, and thus does not require cooperation of the server.

5.7.4 SessionShield

The policy extension, wherein the default policy was augmented with session hijacking protection, is closely related to the session hijacking solution proposed by Nikiforakis et al. [77]. In this solution, HTTP session 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: it only prevents non-HTTP channels from accessing session cookies. As was discussed in section 5.3.1, an implementation in the form of a browser extension makes sure that different browsers will not be able to use each other's cookies. This solves the problem that the SessionShield proxy has when it is used by different browsers.

Similarly to our solution, SessionShield does allow untrusted cookies to be accessed via untrusted channels afterwards. This is because SessionShield only intercepts HTTP traffic, and does not notice JavaScript and `<meta>` cookies being set. This allows JavaScript cookies to keep functioning correctly.

As we already noted in the previous chapter, SessionShield does not protect against session fixation. However, it would be relatively easy to extend SessionShield with our policy preventing session fixation. This can be done by blocking session cookies that SessionShield receives from the browser. Indeed, since SessionShield only intercepts HTTP traffic, it keeps in its cookie store only cookies that are set via HTTP. Thus, when a cookie that SessionShield does not know of is received from the browser, it can conclude that this cookie was set over an untrusted channel. All SessionShield has to do then is use its SID detection algorithm (which it shares with our solution) to check whether the cookie is a session cookie. If it is, SessionShield can block the cookie, thus mitigating session fixation attacks.

5.7.5 Noxes and dynamic tainting

The goal of both the Noxes and dynamic tainting solutions is very similar to that of our own solution: to prevent session identifiers from leaking via XSS attacks [59, 111]. Their approach, however, is rather different. Both Noxes and dynamic tainting try to prevent the stealing of cookies by mitigating all attack vectors one by one. Our extended solution takes this one step further by making session cookies completely unavailable to channels other than HTTP. This makes sure that hidden vectors, which were a problem with both Noxes and dynamic tainting, are accounted for.

5.8 Future work

While cross-domain cooking [119] can also be used to perform a successful session fixation attack, it was largely ignored in our solution. Since our Firefox add-on uses

Firefox' top-level domain list to determine whether a cookie is set for a top-level domain, part of this problem is already solved. However, a complete client-side solution to cross-domain cooking would have to consist of prohibiting subdomains from setting cookies for their parent domain. Our logs show that, when loading each of the homepages of AlexaâĂŹs top 100 websites only once, 427 cookies are set for a parent domain. Because of this, it is currently not clear how such a solution can be implemented without breaking lots of websites.

Other session fixation attacks make use of URL parameters and form fields to set the session cookie. These were also not considered in the current implementation. A simple countermeasure could consist of stripping all SIDs from URLs. Unfortunately, this would break all websites that perform their session management exclusively via URL rewriting.

The current solution was only prototyped as an HTTP proxy and implemented as a Firefox extension. Similar extensions could be written for other browsers. Moreover, as was already described in section 5.7.4, SessionShield could easily be extended with our solution. This would yield a proxy with a policy similar to our own, that separates session cookies entirely from the browser.

Chapter 6

Conclusion

In this thesis, we have shown the different methods of session handling that exist on the web today. We discussed how session identifiers can be accessed, and which properties a good session identifier should posess.

Next, we discussed the ways in which session management mechanisms can be exploited by an attacker to make him able to act on a web server as if he was another user. We showed that these attacks are often very complex, and that many different attack vectors are available.

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. Available from: http://portal.acm.org/citation.cfm?id=1367568.

[2] Alexa Internet. Alexa top 1,000,000 sites, 2011. Available from: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[3] Alfresco Software. Alfresco Customer Overview, 2011. Available from: http://www.alfresco.com/customers/.

[4] Paul Arana. Benefits and Vulnerabilities of Wi-Fi Protected Access 2 (WPA2), 2006. Available from: http://cs.gmu.edu/~yhwang1/INFS612/Sample_Projects/Fall_06_GPN_6_Final_Report.pdf.

[5] Michael Barbaro and Tom Zeller Jr. A Face Is Exposed for AOL Searcher No. 4417749, 2006. Available from: https://www.nytimes.com/2006/08/09/technology/09aol.htm.

[6] Scott Barnham. Securing Django with SSL, 2009. Available from: http://www.redrobotstudios.com/blog/2009/02/18/securing-django-with-ssl/.

[7] 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. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.5579&rep=rep1&type=pdf.

[8] 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. Available from: http://portal.acm.org/citation.cfm?doid=1455770.1455782, doi:10.1145/1455770.1455782.

[9] Constantin Bejenaru. Disable session IDs passed via URL. Available from: http://www.frozenminds.com/disable-sessionid.html.

[10] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005. Available from: http://www.ietf.org/rfc/rfc3986.txt.

[11] Prithvi Bisht and V.N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008. Available from: http://www.springerlink.com/index/8561322615176852.pdf.

[12] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *UIST '05 Proceedings of the 18th annual ACM symposium on User interface software and technology*, New York, NY, 2005. ACM. Available from: http://portal.acm.org/citation.cfm?id=1095062, doi:10.1145/1095034.1095062.

[13] Bram Bonné, Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, and Frank Piessens. Client-Side Protection against Session Fixation Attacks. In *W2SP 2011 (rejected)*, Oakland, CA, 2011.

[14] Will Bontrager. Form Submissions Without Submit Buttons, 2005. Available from: http://www.developertutorials.com/tutorials/javascript/javascript-form-submission-no-button-050412-1292/.

[15] Mason Brown, Bob Martin, and Steve Christey. CWE/SANS TOP 25 Most Dangerous Software Errors Version 2.0, 2010. Available from: http://www.sans.org/top25-software-errors/.

[16] Eric Butler. Firesheep, 2010. Available from: http://codebutler.com/firesheep.

[17] CherryPy Team. cherrypy.lib.sessions, 2010. Available from: http://docs.cherrypy.org/dev/refman/lib/sessions.html.

[18] Chromium Developers. API Wish List. Available from: http://www.chromium.org/developers/design-documents/extensions/apiwishlist.

[19] Craig Condit. JSESSIONID considered harmful, 2006. Available from: http://randomcoder.org/articles/jsessionid-considered-harmful.

[20] Ronald Cools and Wim Michiels. *Modellering en simulatie*. Cursusdienst VTK, 2010.

[21] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-Time Cookies: Preventing Session Hijacking Attacks with Disposable Credentials. *Information Security*, 2011. Available from: http://smartech.gatech.edu/handle/1853/37000.

[22] 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. Available from: http://www.springerlink.com/index/m684ml6389715317.pdf.

[23] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems - CHI '06*, number November 2005, page 581, New York, New York, USA, 2006. ACM Press. Available from: http://portal.acm.org/citation.cfm?doid=1124772.1124861, doi:10.1145/1124772.1124861.

[24] 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. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1410997, doi:10.1109/WSE.2004.10013.

[25] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176. Available from: http://www.ietf.org/rfc/rfc4346.txt.

[26] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.494&rep=rep1&type=pdf.

[27] Django Developers. Django-powered sites, 2007. Available from: http://code.djangoproject.com/wiki/DjangoPoweredSites?version=540.

[28] Django Developers. How to use sessions, 2010. Available from: http://docs.djangoproject.com/en/dev/topics/http/sessions/.

[29] Django Developers. source: django/trunk/django/contrib/auth/___init___.py, 2011. Available from: http://code.djangoproject.com/browser/django/trunk/django/contrib/auth/__init__.py.

[30] Django Developers. source: django/trunk/django/contrib/sessions/backends/base.py, 2011. Available from: http://code.djangoproject.com/browser/django/trunk/django/contrib/sessions/backends/base.py.

[31] Drupal Developers. Security Advisory SA-2008-046 - Drupal core - Session fixation, 2008. Available from: http://drupal.org/node/286417.

[32] Drupal Developers. sess_regenerate, 2010. Available from: http://api.drupal.org/api/drupal/includes--session.inc/function/sess_regenerate/6.

[33] Drupal Developers. Success Stories, 2010. Available from: http://drupal.org/success-stories.

[34] Drupal Developers. settings.php, 2011. Available from: http://api.drupal.org/api/drupal/sites--default--settings.php/5/source.

[35] Drupal Developers. user_authenticate_finalize, 2011. Available from: http://api.drupal.org/api/drupal/modules--user--user.module/function/user_authenticate_finalize/6.

[36] D. Eastlake 3rd, J. Schiller, and S. Crocker. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005. Available from: http://www.ietf.org/rfc/rfc4086.txt.

[37] Jeremy Epstein. PHP session id showing in url, 2005. Available from: http://drupal.org/node/25852.

[38] Dan Fairs. Cookieless Django: Sessions and authentication without cookies, 2007. Available from: http://www.stereoplex.com/blog/cookieless-django-sessions-and-authentication-with.

[39] Stephen Farrell. Leaky or Guessable Session Identifiers. *IEEE Internet Computing*, 15(December 2009):88–91, 2011. Available from: http://www.computer.org/portal/web/csdl/doi/10.1109/MIC.2011.12.

[40] 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. Available from: http://www.ietf.org/rfc/rfc2616.txt.

[41] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. Available from: http://www.ietf.org/rfc/rfc2617.txt.

[42] Alexander Frolkin. Secure Login, 2007. Available from: http://drupal.org/project/securelogin.

[43] 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*. USENIX Association, 2001. Available from: http://portal.acm.org/citation.cfm?id=1251346.

[44] Tim Funk. How do I turn off secure cookies for session IDs?, 2004. Available from: http://www.mail-archive.com/tomcat-user@jakarta.apache.org/msg116570.html.

[45] Google Developers. Cookies & Google Analytics, 2011. Available from: https://code.google.com/intl/nl/apis/analytics/docs/concepts/gaConceptsCookies.html.

[46] Robert Graham. SideJacking with Hamster, 2007. Available from: http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html.

[47] Jeff Hodges, Collin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS) (version 2), 2010. Available from: http://tools.ietf.org/html/draft-hodges-strict-transport-sec-02.

[48] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*, chapter 12, 19. Apress, 1 edition, 2008. Available from: http://www.djangobook.com/en/1.0/.

[49] Lin-shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting Browsers from Cross-Origin CSS Attacks. *CCS '10 Proceedings of the 17th ACM conference on Computer and communications security*, 2010. Available from: http://portal.acm.org/citation.cfm?id=1866376, doi:10.1145/1866307.1866376.

[50] Richard Ishida. Declaring character encodings in HTML, 2010. Available from: http://www.w3.org/International/questions/qa-html-encoding-declarations.

[51] Collin Jackson and Adam Barth. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. *Proceedings of the 17th International World Wide Web Conference (WWW2008)*, 2008. Available from: http://portal.acm.org/citation.cfm?id=1367569.

[52] 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. Available from: http://portal.acm.org/citation.cfm?doid=1242572.1242654, doi:10.1145/1242572.1242654.

[53] Martin Johns. SessionSafe: Implementing XSS immune session handling. *Computer Security–ESORICS 2006*, pages 444–460, 2006. Available from: http://www.springerlink.com/index/50L441H2575L2125.pdf.

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

[55] 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. Available from: http://www.informatik.uni-hamburg.de/SVS/papers/2006_owasp_RequestRodeo.pdf.

[56] Paul Johnston. Authentication and Session Management on the Web. *SANS Institute InfoSec Reading Room*, 2004. Available from: www.sans.org/reading_room/whitepapers/webservers/authentication-session-management-web_1545.

[57] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, August 2007. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4198791, doi:10.1109/SECCOMW.2006.359531.

[58] Ian Kaplan. Shannon Entropy, 2002. Available from: http://www.bearcave.com/misl/misl_tech/wavelets/compression/shannon.html.

[59] 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. Available from: http://portal.acm.org/citation.cfm?id=1141357.

[60] Amit Klein. Cross site scripting explained. *Whitepaper, Sanctum Inc.*, 2002. Available from: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Cross+Site+Scripting+Explained#0.

[61] Amit Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. *Whitepaper, Sanctum Inc.*, January 2004. Available from: http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf.

[62] M Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, pages 1–16, 2002. Available from: www.acrossecurity.com/papers/session_fixation.pdfWmxxW5Ow.

[63] Max Kononovich. security issue: login mechanism subject to "session fixation" attacks, 2008. Available from: http://issues.alfresco.com/jira/browse/ALF-465.

[64] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Proposed Standard), February 1997. Obsoleted by RFC 2965. Available from: http://www.ietf.org/rfc/rfc2109.txt.

[65] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965 (Proposed Standard), October 2000. Available from: http://www.ietf.org/rfc/rfc2965.txt.

[66] David M. Kristol. HTTP Cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology (TOIT)*, 1(2):151–198, November 2001. Available from: http://portal.acm.org/citation.cfm?id=502153, doi:10.1145/502152.502153.

[67] James F. Kurose and Keith W. Ross. *Computer networking: a top-down approach*. Pearson/Addison-Wesley, 4th edition, 2008. Available from: http://www.pearsonhighered.com/educator/academic/product/0,1144,0321497708,00.html.

[68] 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. Available from: http://www.springerlink.com/index/f0507054g3r6630h.pdf.

[69] Giorgio Maone. NoScript: Features, 2011. Available from: http://noscript.net/features.

[70] Paul McMahon. Enabling url parameter based sessions in Ruby on Rails, 2010. Available from: http://www.mobalean.com/blog/2010/04/09/enabling-url-parameter-based-sessions-in-ruby-on-rails.

[71] Microsoft Developers. Mitigating Cross-site Scripting With HTTP-only Cookies. Available from: http://msdn.microsoft.com/en-us/library/ms533046.aspx.

[72] Mozilla Developers. XMLHttpRequest allows reading HTTPOnly cookies, 2009. Available from: https://www.mozilla.org/security/announce/2009/mfsa2009-05.html.

[73] Mozilla Developers. nsIEffectiveTLDService, 2010. Available from: https://developer.mozilla.org/en/nsIEffectiveTLDService.

[74] Mozilla Developers. Observer Notifications, 2010. Available from: https://developer.mozilla.org/en/Observer_Notifications.

[75] Mozilla Developers. Storage, 2010. Available from: https://developer.mozilla.org/en/storage.

[76] Neustar. Enterprise Platform: Benchmarking, 2011. Available from: http://www.webmetrics.com/Monitoring-Services/Enterprise-Monitoring/Platform-Features/Benchmarking.

[77] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. *securitee.org*, pages 1–14, 2010. Available from: http://www.securitee.org/files/sshield_essos2011.pdf.

[78] Nick Nikiforakis, Yves Younan, and Wouter Joosen. HProxy: client-side detection of SSL stripping attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 200–218, 2010. Available from: http://www.springerlink.com/index/2h213654j1u56680.pdf.

[79] Obscure. Bypassing JavaScript Filters – the Flash! Attack. *EyeOnSecurity*, 2002. Available from: http://eyeonsecurity.org/papers/flash-xss.htm.

[80] OWASP. Insufficient Session-ID Length, 2009. Available from: https://www.owasp.org/index.php/Insufficient_Session-ID_Length.

[81] Wladimir Palant. Adblock Plus and (a little) more, 2009. Available from: http://adblockplus.org/blog/attention-noscript-users.

[82] Joon S. Park and Ravi Sandhu. Secure cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, 2000. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=865085, doi:10.1109/4236.865085.

[83] F. Permadi. Reading and writing cookie. Available from: http://www.permadi.com/tutorial/flashCookie/index.html.

[84] PHP Developers. Passing the Session ID, 2011. Available from: http://www.php.net/manual/en/session.idpassing.php.

[85] PHP Developers. Runtime Configuration, 2011. Available from: http://www.php.net/manual/en/session.configuration.php.

[86] PHP Developers. session_regenerate_id(), 2011. Available from: http://www.php.net/manual/en/function.session-regenerate-id.php.

[87] PHP Developers. session_set_cookie_params(), 2011. Available from: http://php.net/manual/en/function.session-set-cookie-params.php.

[88] Associated Press. White House opens Web site coding to public, 2009. Available from: http://www.msnbc.msn.com/id/33463174/ns/technology_and_science-internet/.

[89] Ruby on Rails developers. Real applications live in the wild, 2011. Available from: http://rubyonrails.org/applications.

[90] Jesse Ruderman. Signed Scripts in Mozilla, 2011.

[91] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *14th European Symposium on Research in Computer Security (ESORICSâĂŹ09)*, pages 86–103. Springer, 2009. Available from: http://www.springerlink.com/index/d536t14243q50512.pdf.

[92] Michaël Schrank, Bastian Braun, Martin Johns, and Joachim Posegga. Session Fixation âĂŞ the Forgotten Vulnerability? In *ISSE/SICHERHEIT 2010 : Information Security Solutions Europe - Sicherheit, Schutz und Zuverlässigkeit*, pages 341–352, Bonn, 2010. Gesellschaft für Informatik. Available from: http://subs.emis.de/LNI/Proceedings/Proceedings170/P-170.pdf#page=358.

[93] T Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. *Whitepaper, SecureNet GmbH*, 2004. Available from: http://papers/SessionRiding.pdf.

[94] Matt Schwartz. Web application framework, 2010. Available from: http://docforge.com/wiki/Web_application_framework.

[95] Chris Shiflett. Session Fixation, 2004. Available from: http://shiflett.org/articles/session-fixation.

[96] Chris Shiflett. Session Hijacking, 2004. Available from: http://shiflett.org/articles/session-hijacking.

[97] 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. Available from: http://www.computer.org/portal/web/csdl/doi/10.1109/SP.2010.35.

[98] Michael Slater. Using SSL in Rails Applications, 2008. Available from: http://www.buildingwebapps.com/articles/6401-using-ssl-in-rails-applications.

[99] William Stallings. *Network Security Essentials: Applications and Standards*, chapter 5, pages 153–188. Pearson International, 4th edition, 2011.

[100] Adobe Systems. Coding data exchange between JavaScript and Flash Player, 2010. Available from: http://kb2.adobe.com/cps/156/tn_15683.html.

[101] 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. Available from: http://portal.acm.org/citation.cfm?doid=1541822.1541824, doi:10.1145/1541822.1541824.

[102] Mike Ter Louw, Jin Soon Lim, and V.N. Venkatakrishnan. Extensible Web Browser Security. In *4th International Conference on Detection of Intrusions, Malware, and Vulnerability Assesment (DIMVA)*, 2007.

[103] Tomcat Developers. Class Cookie. Available from: http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/Cookie.html.

[104] Tomcat Developers. SSL Configuration HOW-TO. Available from: http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html.

[105] Tomcat Developers. The Valve Component (version 5.5). Available from: http://tomcat.apache.org/tomcat-5.5-doc/config/valve.html.

[106] Tomcat Developers. The Valve Component (version 6). Available from: http://tomcat.apache.org/tomcat-6.0-doc/config/valve.html.

[107] Tomcat Developers. Sites, Applications, and Systems that are Powered By Tomcat, 2011. Available from: http://wiki.apache.org/tomcat/PoweredBy.

[108] Binny V A. Using POST method in XMLHTTPRequest(Ajax), 2010. Available from: http://www.openjs.com/articles/ajax_xmlhttp_using_post.php.

[109] Robert Vamosi. Security Watch: Session fixation, 2006. Available from: http://reviews.cnet.com/4520-3513_7-6627057-1.html.

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

[111] 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. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.2063&rep=rep1&type=pdf.

[112] W3Techs. Usage of server-side programming languages for websites, 2010. Available from: http://w3techs.com/technologies/overview/programming_language/all.

[113] Daniel Wasser. HowTo renew the Session id in Tomcat or JBoss after a login, 2007. Available from: http://www.koelnerwasser.de/?p=11.

[114] Heiko Webers. Ruby On Rails Security Guide, 2008. Available from: http://guides.rubyonrails.org/security.html.

[115] Alma Whitten and J.D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, pages 169–184. Citeseer, 1999. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.6298&rep=rep1&type=pdf.

[116] Wikipedia Authors. Comparison of Web application frameworks, 2011. Available from: https://secure.wikimedia.org/wikipedia/en/wiki/Comparison_of_Web_application_frameworks.

[117] Jeff Williams and Dave Wichers. OWASP top 10. Technical report, OWASP, 2010.

[118] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegel. SWAP: Mitigating XSS attacks using a reverse proxy. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, Vancouver, BC, 2009. IEEE. Available from: http://www.computer.org/portal/web/csdl/doi/10.1109/IWSESS.2009.5068456.

[119] Michal Zalewski. Cross Site Cooking, 2006. Available from: http://www.securiteam.com/securityreviews/5EP0L2KHFG.html.

[120] William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention, 2008. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.147.1445&rep=rep1&type=pdf.

[121] Weilin Zhong. Session Management, 2006. Available from: https://www.owasp.org/index.php/Session_Management.

Fiche masterproef

Student: Bram Bonné

Titel: Improving session security in web applications

Engelse titel: Verhoogde veiligheid van sessies in webapplicaties

UDC: 004.4

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: Computer Science

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

Assessoren: Ir. P. Philippaerts
G. Poulisse

Begeleiders: P. De Ryck
N. Nikiforakis