A

Surviving the Web
A Journey into Web Session Security

Stefano Calzavara, Università Ca' Foscari, Venezia, Italy
Riccardo Focardi, Università Ca' Foscari, Venezia, Italy
Marco Squarcina, Università Ca' Foscari, Venezia, Italy
Mauro Tempesta, Università Ca' Foscari, Venezia, Italy

In this paper we survey the most common attacks against web sessions, i.e., attacks which target honest web browser users establishing an authenticated session with a trusted web application. We then review existing security solutions which prevent or mitigate the different attacks, by evaluating them along four different axes: protection, usability, compatibility and ease of deployment. We also assess several defensive solutions which aim at providing robust safeguards against multiple attacks. Based on this survey, we identify five guidelines that, to different extents, have been taken into account by the designers of the different proposals we reviewed. We believe that these guidelines can be helpful for the development of innovative solutions approaching web security in a more systematic and comprehensive way.

CCS Concepts: •**Security and privacy** → **Browser security; Web application security;** *Web protocol security;*

General Terms: Security.

Additional Key Words and Phrases: Web sessions, HTTP cookies, web attacks, web defenses.

1. INTRODUCTION

The Web is the primary access point to on-line data and applications. It is extremely complex and variegate, as it integrates a multitude of dynamic contents by different parties to deliver the greatest possible user experience. This heterogeneity makes it very hard to effectively enforce security, since putting in place novel security mechanisms typically prevents existing websites from working correctly or negatively affects the user experience, which is generally regarded as unacceptable, given the massive user base of the Web. However, this continuous quest for usability and backward compatibility had a subtle effect on web security research: designers of new defensive mechanisms have been extremely cautious and the large majority of their proposals consists of very local patches against very specific attacks. This piecemeal evolution hindered a deep understanding of many subtle vulnerabilities and problems, as testified by the proliferation of different threat models against which different proposals have been evaluated, occasionally with quite diverse underlying assumptions. It is easy to get lost among the multitude of proposed solutions and almost impossible to understand the relative benefits and drawbacks of each single proposal without a full picture of the existing literature.

In this paper we take the delicate task of performing a systematic overview of a large class of common attacks targeting the current Web and the corresponding security solutions proposed so far. We focus on attacks against *web sessions*, i.e., attacks which target honest web browser users establishing an authenticated session with a trusted web application. This kind of attacks exploits the intrinsic complexity of the Web by tampering, e.g., with dynamic contents, client-side storage or cross-domain links, so as to corrupt the browser activity and/or network communication. Our choice is motivated by the fact that attacks against web sessions cover a very relevant subset of serious web security incidents [OWASP 2013] and many different defenses, operating at different levels, have been proposed to prevent these attacks.

We consider typical attacks against web sessions and we systematise them based on: (i) their attacker model and (ii) the security properties they break. This first classi-

fication is useful to understand precisely which intended security properties of a web session can be violated by a certain attack and how. We then survey existing security solutions and mechanisms that prevent or mitigate the different attacks and we evaluate each proposal with respect to the security guarantees it provides. When security is guaranteed only under certain assumptions, we make these assumptions explicit. For each security solution, we also evaluate its impact on both *compatibility* and *usability*, as well as its *ease of deployment*. These are important criteria to judge the practicality of a certain solution and they are useful to understand to which extent each solution, in its current state, may be amenable for a large-scale adoption on the Web. Since there are several proposals in the literature which aim at providing robust safeguards against multiple attacks, we also provide an overview of them in a separate section. For each of these proposals, we discuss which attacks it prevents with respect to the attacker model considered in its original design and we assess its adequacy according to the criteria described above.

Finally, we synthesize from our survey a list of five guidelines that, to different extents, have been taken into account by the designers of the different solutions. We observe that none of the existing proposals follows all the guidelines and we argue that this is due to the high complexity of the Web and the intrinsic difficulty in securing it. We believe that these guidelines can be helpful for the development of innovative solutions approaching web security in a more systematic and comprehensive way.

1.1. Scope of the Survey

Web security is complex and web sessions can be attacked at many different layers. To clarify the scope of the present work, it is thus important to discuss some assumptions we make and their import on security:

(1) *perfect cryptography*: at the network layer, web sessions can be harmed by network sniffing or man-in-the-middle attacks. Web traffic can be protected using the HTTPS protocol, which wraps the traffic within a SSL/TLS encrypted channel. We do not consider attacks to cryptographic protocols. In particular, we assume that the attacker cannot break cryptography to disclose, modify or inject the contents sent to a trusted web application over an encrypted channel. However, we do not assume that HTTPS is always configured correctly by web developers, since this is quite a delicate task, which deserves to be discussed in the present survey;
(2) *the web browser is not compromised by the attacker*: web applications often rely on the available protection mechanisms offered by standard web browsers, like the same-origin policy or the `HttpOnly` cookie attribute. We assume that all these defenses behave as intended and the attacker does not make advantage of browser exploits, otherwise even secure web applications would fail to be protected;
(3) *trusted web applications may be affected by content injection vulnerabilities*: this is a conservative assumption, since history teaches us that it is almost impossible to guarantee that a web application does not suffer from this kind of threats. We focus on content injection vulnerabilities which ultimately target the web browser, like cross-site scripting attacks (XSS). Content injections affecting the backend of the web application, like SQL injections, are not covered.

1.2. Structure of the Survey

Section 2 provides some background on the main building blocks of the Web. Section 3 presents the attacks. Section 4 classifies attack-specific solutions with respect to their security guarantees, their level of usability, compatibility and ease of deployment. Section 5 carries out a similar analysis for defenses against multiple attacks. Section 6 presents five guidelines for future web security solutions. Section 7 concludes.

2. BACKGROUND

We provide a brief overview of the basic building blocks of the web ecosystem and their corresponding security cornerstones.

2.1. Languages for the Web

Documents on the Web are provided as *web pages*, hypertext files connected to other documents via hyperlinks. Web pages embody several languages affecting different aspects of the documents. The *Hyper Text Markup Language* (HTML) [W3C 2014c] or a comparable markup language (e.g., XHTML) defines the structure of the page and the elements it includes, while *Cascading Style Sheets* (CSS) [W3C 2014a] are used to add style information to web pages (e.g., fonts, colors, position of elements).

JavaScript [ECMA 2011] is a programming language which allows the development of rich, interactive web applications. JavaScript programs are included either directly in the web page (*inline* scripts) or as external resources, and can dynamically update the contents in the user browser by altering the *Document Object Model* (DOM) [W3C 1998; 2000; 2004], a tree-like representation of the web page. Page updates are typically driven by user interaction or by asynchronous communications with a remote web server based on *Ajax* requests (via the XMLHttpRequest API).

2.2. Locating Web Resources

Web pages and the contents included therein are hosted on *web servers* and identified by a *Uniform Resource Locator* (URL). A URL specifies both the location of a resource and a mechanism for retrieving it. A typical URL includes: (1) a *protocol*, defining how the resource should be accessed; (2) a *host*, identifying the web server hosting the resource; and (3) a *path*, localizing the resource at the web server.

Hosts belong to *domains*, identifying an administrative realm on the Web, typically controlled by a specific company or organization. Domain names are organised hierarchically: sub-domain names can be defined from a domain name by prepending it a string, separated by a period. For example, host `www.google.com` belongs to domain `google.com` which is a sub-domain of the top-level domain `com`.

2.3. Hyper Text Transfer Protocol (HTTP)

Web contents are requested and served using the *Hyper Text Transfer Protocol* (HTTP), a text-based request-response protocol based on the client-server paradigm. The client (browser) initiates the communication by sending an HTTP request for a resource hosted on the server; the server, in turn, provides an HTTP response containing the completion status information of the request and its result. HTTP defines *methods* to indicate the action to be performed on the identified resource, the most important ones being GET and POST. GET requests should only retrieve data and have no other import, while server-side side-effects should only be triggered by POST requests, though web developers do not always comply with this convention. Both GET and POST requests may include custom parameters, which can be processed by the web server.

HTTP is a *stateless* protocol, i.e., it treats each request as independent from all the other ones. Some applications, however, need to remember information about previous requests, for instance to track whether a user has already authenticated and grant him access to his personal page. HTTP *cookies* are the most widespread mechanism employed on the Web to maintain state information about the requesting clients [Barth 2011a]. Roughly, a cookie is a key-value pair, which is set by the server into the client and automatically attached by it to all subsequent requests to the server. Cookies can be set via the `Set-Cookie` header of HTTP or by using JavaScript. Cookies may also have *attributes* which restrict the way the browser handles them (see Section 2.4.4).

2.4. Security Cornerstones and Subtleties

2.4.1. HTTPS. Since all the HTTP traffic flows in the clear, the HTTP protocol does not guarantee several desirable security properties, such as the confidentiality and the integrity of the communication, and the authenticity of the involved parties. To protect the exchanged data, the *HTTP Secure* (HTTPS) protocol [Rescorla 2000] wraps plain HTTP traffic within a SSL/TLS encrypted channel. A web server may authenticate itself at the client by using public key certificates; when the client is unable to verify the authenticity of a certificate, a warning message is displayed and the user can decide whether to proceed with the communication or not.

2.4.2. Mixed Content Websites. A *mixed content* page is a web page that is received over HTTPS, but loads some of its contents over HTTP. The browser distinguishes two types of contents depending on their capabilities on the including page: *passive contents* like images, audio tracks or videos cannot modify other portions of the page, while *active contents* like scripts, frames or stylesheets have access to (parts of) the DOM and may be exploited to alter the page. While the inclusion of passive contents delivered over HTTP into HTTPS pages is allowed by the browser, active mixed contents are blocked by default [W3C 2015b].

2.4.3. Same-Origin Policy. The *same-origin policy* (SOP) [Mozilla 2015] is a standard security policy implemented by all major web browsers: it enforces a strict separation between contents provided by unrelated sites, which is crucial to ensure their confidentiality and integrity. The SOP allows scripts running in a first web page to access data in a second web page only if the two pages have the same *origin*. An origin is defined as the combination of a protocol, a host and a port number [Barth 2011b]. The SOP applies to many operations in the browser, most notably DOM manipulations and cookie accesses. However, some operations are not subject to same-origin checks, e.g., cross-site inclusion of scripts and submission of forms are allowed, thus leaving space to potential attacks.

2.4.4. Cookies. Cookies use a separate definition of origin, since cookies set for a given domain are normally shared across all the ports and protocols on that domain. By default, cookies set by a page are only attached by the browser to requests sent to the same domain of the page. However, a page may also set cookies for a parent domain by specifying it using the `Domain` cookie attribute, as long as the parent domain does not occur in a list of public suffixes[1]: these cookies are shared between the parent domain and all its sub-domains, and we refer to them as *domain cookies*.

Cookies come with two security mechanisms: the `Secure` attribute identifies cookies which must only be sent over HTTPS, while the `HttpOnly` attribute marks cookies which cannot be accessed via non-HTTP APIs, e.g., via JavaScript. Perhaps surprisingly, the `Secure` attribute does not provide integrity guarantees, since secure cookies can be overwritten over HTTP [Barth 2011a].

3. ATTACKING WEB SESSIONS

A *web session* is a semi-permanent information exchange between a browser and a web server, involving multiple requests and responses. As anticipated, stateful sessions on the Web are typically bound to a *cookie* stored in the user browser. When the user authenticates to a website by providing some valid credentials, e.g., a username-password pair, a fresh cookie is generated by the server and sent back to the browser. Further requests originating from the browser automatically include the cookie as a

[1] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

proof of being part of the session established upon password-based authentication. This common authentication scheme is depicted in Figure 1.

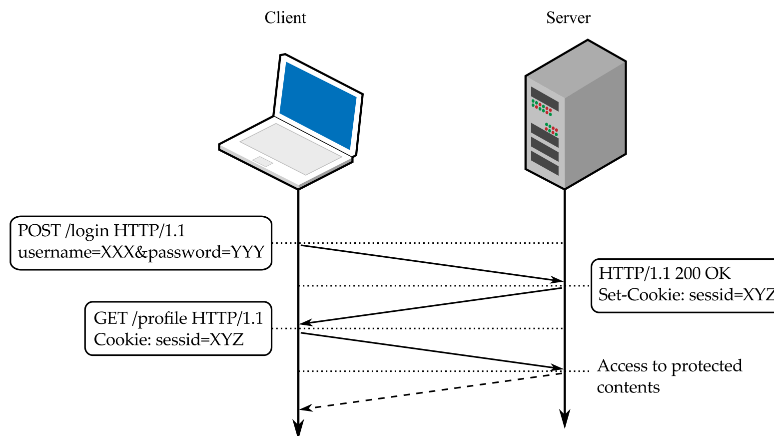

Fig. 1. Cookie-based User Authentication.

Since the cookie essentially plays the role of the password in all the subsequent requests to the web server, it is enough to discover its value to hijack the session and fully impersonate the user, with no need to compromise the low level network connection or the server. We call *authentication cookie* any cookie which identifies a web session.

3.1. Security Properties

We consider two standard security properties formulated in the setting of web sessions. They represent typical targets of web session attacks:

- *Confidentiality*: data transmitted inside a session should not be disclosed to unauthorized users;
- *Integrity*: data transmitted inside a session should not be modified or forged by unauthorized users.

Interestingly, the above properties are not independent and a violation of one might lead to the violation of the other. For example, compromising session confidentiality might reveal authentication cookies, which would allow the attacker to perform arbitrary actions on behalf of the user, thus breaking session integrity. Integrity violations, instead, might cause the disclosure of confidential information, e.g., when sensitive data is leaked via a malicious script injected in a web page by an attacker.

3.2. Threat Model

We focus on two main families of attackers: *web attackers* and *network attackers*. A web attacker controls at least one web server that responds to any HTTP(S) requests sent to it with arbitrary malicious contents chosen by the attacker. We assume that a web attacker can obtain trusted HTTPS certificates for all the web servers under his control and is able to exploit content injection vulnerabilities on trusted websites. A slightly more powerful variation of the web attacker, known as the *related-domain attacker*, can also host malicious web pages on a domain sharing a “sufficiently long” suffix with the domain of the target website [Bortz et al. 2011]. This means in particular that the attacker can set (domain) cookies for the target website [Barth 2011a]. These

cookies are indistinguishable from other cookies set by the target website and are automatically sent to the latter by the browser. Hereafter, we explicitly distinguish a related-domain attacker from a standard web attacker only when the specific setting is relevant to carry out an attack.

Network attackers extend the capabilities of traditional web attackers with the ability of inspecting, forging and corrupting all the HTTP traffic sent on the network, as well as the HTTPS traffic which does not make use of certificates signed by a trusted certification authority. It is common practice in web security to distinguish between *passive* and *active* network attackers, with the first ones lacking the ability of forging or corrupting the unprotected network traffic. From now on, when generically speaking about network attackers, we implicitly refer to active network attackers.

3.3. Web Attacks

3.3.1. Content Injection. This wide class of attacks allows a web attacker to inject harmful contents into trusted web applications. Content injections can be mounted in many different ways, but they are always enabled by an improper or missing sanitization of some attacker-controlled input in the web application, either at the client side or at the server side. These attacks are traditionally assimilated to Cross-Site Scripting (XSS), i.e., injections of malicious JavaScript code; however, the lack of a proper sanitization may also affect HTML contents (markup injection) or even CSS rules [Zalewski 2011; Heiderich et al. 2012].

To exemplify how an XSS works, consider a website `vuln.com` hosting a simple search engine. Queries are performed via a GET request including a search parameter which is displayed in the result page headline "`Search results for foo:`", where `foo` is the value of the search parameter. An attacker can then attempt to inject contents into `vuln.com` just by providing to the user a link including a script as the search term. If the search page does not properly sanitize such an input, the script will be included in the headline of the results page and it will run on behalf of `vuln.com`, thus allowing the attacker to sidestep SOP: for instance, the injected script will be entitled to read the authentication cookies set by `vuln.com`.

XSS attacks are usually classified as either *reflected* or *stored*, depending on the persistence of the threat. Reflected XSS attacks correspond to cases like the one above, where part of the input supplied by the request is "reflected" into the response without proper sanitization. Stored XSS attacks, instead, are those where the injected script is permanently saved on the target server, e.g., in a message appearing on a discussion board. The malicious script is then automatically executed by any browser which visits the attacked page.

Security properties: since content injections allow an attacker to sidestep SOP, which is the baseline security policy of standard web browsers, they can have catastrophic consequences on both the confidentiality and the integrity of a web session. Specifically, they can be used to steal sensitive data from trusted websites, such as authentication cookies and user credentials, and to actively corrupt the page contents, so as to undermine the integrity of a web session.

3.3.2. Cross-Site Request Forgery (CSRF). A CSRF is an instance of the "confused deputy" problem [Hardy 1988] in the context of web browsing. In a CSRF, the attacker forces the user browser into sending HTTP(S) requests to a website where the user has already established an authenticated session: it is enough for the attacker to include HTML elements pointing to the vulnerable website in his own web pages. When rendering or accessing these HTML elements, the browser will send HTTP(S) requests to the target website and these requests will automatically include the authentication cookies of the user. From the target website perspective, these forged requests are

indistinguishable from legitimate ones and thus they can be abused to trigger a dangerous side-effect, e.g., to force a bank transfer from the user account to the attacker account. Notably, the attacker can forge these malicious requests without any user intervention, e.g., by including in a page under his control some `img` tags or a hidden HTML form submitted via JavaScript.

Security properties: a CSRF attack allows the attacker to inject an authenticated message into a session with a trusted website, hence it constitutes a threat to session integrity. It is less known that CSRFs may also be employed to break confidentiality by sending cross-site requests that return sensitive user data bound to the user session. Normally, SOP (Section 2.4.3) prevents a website from reading responses returned by a different site, but websites may explicitly allow cross-site accesses using the Cross-Origin Request Sharing (CORS) standard [W3C 2014b] or mechanisms like JSON with Padding (JSONP) [Ippolito 2015] which can be abused to break session confidentiality. For instance, a CSRF attack leaking the stored files has been reported on the cloud service SpiderOak [Bansal et al. 2013].

3.3.3. Login CSRF. A peculiar instance of CSRF, known as login CSRF, is a subtle attack first described by Barth *et al.*, where the victim is forced to interact with the target website within the attacker session [Barth et al. 2008]. Specifically, the attacker uses his own credentials to silently log in the user browser at the target website, for instance by forcing it into submitting an invisible login form. The outcome of the attack is that the user browser is forced into an *attacker* session: if the user is not careful, she might be tricked into storing sensitive information, like her credit card number, into the attacker account.

Security properties: though this attack does not compromise existing sessions, it fools the browser into establishing a new attacker-controlled (low integrity) session with a trusted website. Login CSRFs may enable confidentiality violations in specific application scenarios, like in the credit card example given above.

3.3.4. Cookie Forcing. A web attacker exploiting a code injection vulnerability may directly impose his own authentication cookies in the victim browser, thus forcing it into the attacker session and achieving the same results of a successful login CSRF, though exploiting a different attack vector. Related-domain attackers are in a privileged position for these attacks, since they can set cookies for the target website from a related-domain host.

Security properties: see login CSRF (Section 3.3.3).

3.3.5. Session Fixation. A session fixation attack allows an attacker to impersonate a user by imposing in the user browser a known session identifier, which is not refreshed upon successful authentication with the vulnerable website. Typically, the attacker first contacts the target site and gets a valid cookie which is then set (e.g., via an XSS attack on the site) into the user browser *before* the initial password-based authentication step is performed. If the website does not generate a fresh cookie upon authentication, the user session will be identified by a cookie known to the attacker. Related-domain attackers have easy access to these attacks, since they can set cookies on behalf of the victim website.

Security properties: by letting the attacker fully impersonate the user at the target website, session fixation harms both the confidentiality and the integrity of the user session, just as if the authentication cookies were disclosed to the attacker.

3.4. Network Attacks

Though network attacks are arguably more difficult to carry out on the Web than standard web attacks, they typically have a tremendous impact on both the confidentiality

and the integrity of the user session. Since the HTTP traffic is transmitted in clear, a network attacker, either passive or active, can eavesdrop sensitive information and compromise the confidentiality of HTTP sessions. Websites which are served on HTTP or on a mixture of HTTPS and HTTP are prone to expose non-secure cookies or user credentials to a network attacker: in these cases, the attacker will be able to fully impersonate the victim at the target website. An active network attacker can also mount man-in-the-middle attacks via e.g., ARP spoofing, DNS cache poisoning or by setting up a fake wi-fi access point. By interposing himself between the victim and the server, this attacker can arbitrarily modify HTTP requests and responses exchanged by the involved parties, thus breaking the confidentiality and the integrity of the session. Also, active network attackers can compromise the integrity of cookies [Barth 2011a].

A notable example of network attack is *SSL stripping* [Marlinspike 2009], which is aimed at preventing 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 protocol upgrade is done through HTTP redirect messages, links or HTML forms targets. By corrupting the first server response, an active attacker may force the session in clear by replacing all the HTTPS references with their HTTP version and then forward the traffic received by the user to the real web server, possibly over HTTPS. The same operation will then be performed for each request/response in the session, hence the web application will work seamlessly, but the communication will be entirely under the control of the attacker. This attack is particularly subtle, since the user might fail to notice the missing usage of HTTPS, which is only notified by some components of the browser user interface (e.g., a padlock icon).

4. PROTECTING WEB SESSIONS

4.1. Evaluation Criteria

We evaluate existing defenses along four different axes:

(1) *protection*: we assess the effectiveness of the proposed defense against the conventional threat model of the attack, e.g., the web attacker for CSRF. If the proposal does not prevent the attack in the most general case, we discuss under which assumptions it may still be effective;
(2) *usability*: we evaluate whether the proposed mechanism affects the end-user experience, for instance by impacting on the perceived performances of the browser or by involving the user into security decisions;
(3) *compatibility*: we discuss how well the defense integrates into the web ecosystem with respect to the current standards, the expected functionalities of websites, and the performances provided by modern network infrastructures. For example, solutions that prevent some websites from working correctly are not compatible with the existing Web. On the other hand, a minor extension to a standard protocol which does not break backward compatibility, such as the addition of new HTTP headers that can be ignored by recipients not supporting them, is acceptable;
(4) *ease of deployment*: we consider how practical would be a large-scale deployment of the defensive solution by evaluating the overall effort required by web developers and system administrators for its adoption. If they have to pay an unacceptably high cost, the solution will likely never be deployed on a large scale.

We deem a negative impact on server-side performances as a compatibility problem rather than a usability problem when the overall response time can be kept constant by increasing the computational resources of the server, thus keeping the user experience unaffected. To provide a concise yet meaningful evaluation of the different proposals,

usability, compatibility and ease of deployment are assigned a score from a three-levels scale: Low, Medium and High. Table I provides the intuition underlying these scores.

Table I. Evaluation Criteria

	Usability	*Compatibility*	*Ease of Deployment*
Low	Users must take several security decisions	The correct functioning of some websites is precluded	Applications need to be heavily rewritten, complex security policies must be deployed
Medium	Perceivable slowdown of performances that affects the client	Moderate increase of the server workload	Moderate server-side modifications, small declarative policies have to be written
High	The user experience is not affected in any way	The defense fits the web ecosystem, no impact on server workload	The protection can be enabled just by installing an additional component or by minimal server-side modifications

We exclude from our survey several solutions which would require major changes to the current Web, such as new communication protocols or authentication mechanisms replacing cookies and passwords [Johns et al. 2012; Hallgren et al. 2013; Singh et al. 2012; Dacosta et al. 2012].

4.2. Content Injection: Mitigation Techniques

Given the critical impact of content injection attacks, there exist many proposals which focus on them. In this section we discuss those solutions which do not necessarily prevent a content injection, but rather mitigate its malicious effects, e.g., by thwarting the leakage of sensitive data.

4.2.1. HttpOnly Cookies. `HttpOnly` cookies have been introduced in 2002 with the release of Internet Explorer 6 SP1 to prevent the theft of authentication cookies via content injection attacks. Available on all major browsers, this simple yet effective mechanism limits the scope of cookies to HTTP(S) requests, making them unavailable to malicious JavaScript injected in a trusted page.

The protection offered by the `HttpOnly` attribute is only limited to the theft of authentication cookies. The presence of the attribute is transparent to users, hence it has no usability import. Also, the attribute perfectly fits the web ecosystem in terms of compatibility with legacy web browsers, since unknown cookie attributes are ignored. Finally, the solution is easy to deploy, assuming there is no need of accessing authentication cookies via JavaScript for generic reasons [Zhou and Evans 2010].

4.2.2. SessionShield and Zan. SessionShield [Nikiforakis et al. 2011] 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 [Tang et al. 2011], a browser-based defense which (among other things) automatically applies the `HttpOnly` attribute to the authentication cookies detected through the usage of a heuristic. As previously discussed, `HttpOnly` cookies cannot be accessed by JavaScript and will only be attached to outgoing HTTP(S) requests.

The protection offered by SessionShield and Zan is limited to the improper exfiltration of authentication cookies. These defenses do not prompt the user with security decisions, neither slow down perceivably the processing of web pages, hence they are

fine from a usability point of view. However, the underlying heuristic for detecting authentication cookies poses some compatibility concerns, since it may break websites when a cookie is incorrectly identified as an authentication cookie and made unavailable to legitimate scripts that need to access it. Both SessionShield and Zan are very easy to deploy, given their purely client-side nature.

4.2.3. Request Filtering Approaches. Noxes is one of the first developed client-side defenses against XSS attacks [Kirda et al. 2006]. It is implemented as a web proxy installed on the user machine, aimed at preserving the confidentiality of sensitive data in web pages, such as authentication cookies and session IDs. Instead of blocking malicious script execution, Noxes analyzes the pages fetched by the user in order to allow or deny outgoing connections on a whitelist basis: only local references and static links embedded into a page are automatically considered safe with respect to XSS attacks. For all the other links, Noxes resorts to user interaction to take security decisions which can be saved either temporarily or permanently. Following Noxes, Vogt *et al.* introduce a modified version of Firefox [Vogt et al. 2007] where they combine dynamic taint tracking and lightweight static analysis techniques to track the flow of a set of sensitive data sources (e.g., cookies, document URLs) within the scripts included in a page. When the value of a tainted variable is about to be sent to a third-party domain, the user is required to authorize or deny the communication.

The protection offered by these approaches is not limited to authentication cookies, but it prevents the exfiltration of arbitrary sensitive data manipulated by web pages. According to the authors, the solutions are not affected by performance problems, however Noxes still suffers from usability issues, as it requires too much user interaction given the high number of dynamic links in modern web pages [Nikiforakis et al. 2011]. The modified Firefox in [Vogt et al. 2007] attempts to lower the number of security questions with respect to Noxes, but still many third-party domains such as `.google-analytics.com` should be manually whitelisted to avoid recurring alert prompts. On the other hand, due to the fine-grained control over the filtering rules, both mechanisms are deemed compatible, assuming that the user takes the correct security decisions. Both solutions are easy to deploy, since no server-side modification is required and users simply need to install an application on their machines.

4.2.4. Critical Evaluation. The exfiltration of sensitive data is a typical goal of content injection attacks. Preventing authentication cookie stealing is simple nowadays, given that the `HttpOnly` attribute is well supported by all modern browsers, and several languages and web frameworks allow the automatic enabling of the attribute for all the authentication cookies [OWASP 2014]. Conversely, solutions aimed at providing wider coverage against general data leakage attacks never gained popularity, mainly due to their impact on the user experience.

4.3. Content Injection: Prevention Techniques

While the proposals discussed in the previous section are designed to block leakages of sensitive data, the defenses presented in this section attempt to prevent the execution of malicious contents injected into web pages.

4.3.1. Client-side Filtering. XSS filters like IE XSS Filter [Ross 2008] and WebKit XSSAuditor [Bates et al. 2010] are useful to prevent reflected XSS attacks. Before interpreting the JavaScript code in a received page, these client-side filters check whether potentially dangerous payloads, like `<script>` tags, included in the HTTP request are also found within the response body: if a match is detected, the payload is typically stripped from the rendered page without asking for user intervention. The NoScript extension for Firefox [Maone 2004] applies an even stricter policy, since it directly pre-

vents script execution, thus blocking both stored and reflected XSS attacks. This policy can be relaxed on selected domains, where only XSS filtering mechanisms are applied.

XSS filtering proved to be quite effective in practice, despite not being always able to prevent all the attacks. A typical example is a web application which takes a `base64` encoded string via a GET variable and includes the decoded result in the generated page: an attacker may easily bypass the XSS filter by supplying the `base64` encoding of a malicious JavaScript which will, in turn, be decoded by the server and included in the response body. Additionally, XSS filters have also been exploited to introduce new flaws in otherwise secure websites, e.g., by disabling legitimate scripts found in the original pages [Nava and Lindsay 2009; Johns et al. 2014].

The filtering approach against reflected XSS attacks showed no negative impact on the user experience and a good compatibility with modern web applications. Indeed, IE XSS Filter and WebKit XSSAuditor have been included in major browsers. The additional security features offered by NoScript however come at a cost on usability, since the user is involved in the process of dynamically populating the whitelist of the extension whenever a blocked script is required to preserve the functionality of the website. Nevertheless, it is possible to relax the behaviour of NoScript to improve the user experience, by configuring the extension so that it only applies filtering against reflected XSS attacks.

4.3.2. Server-side Filtering. An alternative to the in-browser filtering approach is to perform attack detection on the server-side. Xu *et al.* present a method based on fine-grained taint tracking analysis [Xu et al. 2006] which improves an earlier solution named CSSE [Pietraszek and Berghe 2005]. This approach is designed to prevent a variety of attacks including content injections. The idea is to apply a source-to-source transformation of server-side C programs to track the flow of potentially malicious input data and enforce taint-enhanced security policies. By marking every byte of the user input as tainted, reflected XSS attacks can be prevented by policies that forbid the presence of tainted dangerous HTML tag patterns inside the web application output.

The protection offered by this approach and its ease of deployment crucially depend on the enforced security policy. A simple policy preventing user-provided `<script>` tags from appearing in the web page is trivial to write, but ineffective against more sophisticated attacks. However, writing a more comprehensive set of rules while maintaining the full functionalities of websites is considered a challenging task [Louw and Venkatakrishnan 2009]. The existence of ready-to-use policies would make it easier to apply the security mechanism. Still, server modifications are required to enable support for the protection mechanism on the script language engine, which brings a significant performance overhead on CPU intensive applications, reported to be between 50% and 100%. This partially hinders both compatibility and ease of deployment.

4.3.3. XSS-Guard. The idea of server-side source-to-source program transformation is also employed in XSS-Guard [Bisht and Venkatakrishnan 2008], a solution for Java applications aimed at distinguishing malicious scripts reflected into web pages from legitimate ones. For each incoming request, the rewritten application generates two pages: the first includes the original user input, while the second is produced using input strings not including harmful characters (e.g., sequences of `A`'s). The application checks the equivalence of the scripts contained in the two pages by string matching or, in case of failure, by comparing their syntactic structure. Additional or modified scripts found within the real page are considered malicious and stripped from the page returned to the user.

The protection offered by XSS-Guard is good, but limited to reflected XSS attacks. Moreover, since the script detection procedure is borrowed from the Firefox browser, some quirks specific to other browsers may allow to escape the mechanism. However,

XSS-Guard is usable, since the browsing experience is not affected by its server-side adoption. The performance overhead caused by the double page generation ranges from 5% to 24%, thus increasing the server workload: this gives rise to some concerns about compatibility. On the other hand, enabling the solution on existing Java programs is simple, since no manual code changes are required and web developers only need to automatically translate their applications.

4.3.4. BEEP. Browser-Enforced Embedded Policies (BEEP) [Jim et al. 2007] is a hybrid client-server approach which hinges on the assumption that web developers have a precise understanding of which scripts should be trusted for execution. Websites provide a filtering policy to the browser in order to allow the execution of trusted scripts only, thus blocking any malicious scripts injected in the page. The policy is embedded in web pages through a specific JavaScript function which is invoked by a specially-modified browser every time a script is found during the parsing phase. This function accepts as parameters the code and the DOM element of the script and returns a boolean value which determines whether the execution is allowed or not.

The proposed mechanism exhibits some security defects, as shown in [Athanasopoulos et al. 2009]. For instance, an attacker may reuse whitelisted scripts in an unanticipated way to alter the behaviour of the application. Regarding usability, the adoption of this solution may cause some slowdowns at the client-side when accessing websites which heavily rely on inline JavaScript contents. Compatibility however is preserved, since browsers not compliant with BEEP will still render pages correctly without the additional protection. The deployment of BEEP is not straightforward, since the effort required to modify existing web applications to implement the security mechanism depends on the complexity of the desired policy.

4.3.5. Blueprint. Blueprint [Louw and Venkatakrishnan 2009] tackles the problem of denying malicious script execution by relieving the browser from parsing untrusted contents: indeed, the authors argue 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 annotate the parts of the web application code which include a block of user-provided content in the page. For each block, the server builds a parse tree of the user input, stripped of all the dynamic contents (e.g., JavaScript, Flash). The sanitized tree is encoded as a `base64` string and included in the page within an invisible `<code>` block. This `base64` data is then processed by a client-side JavaScript which is in charge of reconstructing the DOM of the corresponding portion of the page.

Despite providing strong protection against stored and reflected XSS attacks, Blueprint suffers from performance issues which impact on both usability and compatibility [Weinberger et al. 2011]. Specifically, the server workload is increased by a 35%-55% due to the parse tree generation, while the page rendering time is significantly affected by the amount of user contents to be dynamically processed by the browser. Also, Blueprint requires a considerable deployment effort, since the web developer must manually identify and update all the code portions of web applications that write out the user input.

4.3.6. Noncespaces. Along the same line of research, Noncespaces [Gundy and Chen 2012] is a hybrid approach that allows web clients to distinguish between trusted and untrusted contents to prevent content injection attacks. This solution provides a policy mechanism which enables web developers to declare granular constraints on elements and attributes according to their trust class. All the (X)HTML tags and attributes are associated to a specific trust class by automatically enriching their names with a random string, generated by the web application, that is unknown to the attacker. In case of XHTML documents, the random string is applied as a namespace prefix (`<r617:h1`

`r617:id=''Title''>Title</r617:h1>`), while in the HTML counterpart the prefix is simply concatenated (`<r617h1 r617id=''Title''>Title</r617h1>`). The server sends the URL of the policy and the mapping between trust classes and random strings via custom HTTP headers. A proxy installed on the user machine validates the page according to the policy and returns an empty page to the browser in case of violations, i.e., if the page contains a tag or attribute with a random string which is invalid or bound to an incorrect trust class.

The solution is an improvement over BEEP in preventing stored and reflected XSS. Since random prefixes are not disclosed to the attacker, Noncespaces is not affected by the exploits introduced in [Athanasopoulos et al. 2009]. Additionally, the mechanism allows web developers to permit the inclusion of user-provided HTML code in a controlled way, thus offering protection also against markup injections. Although the impact on server-side performance is negligible, the policy validation phase performed by the proxy on the client-side introduces a noticeable overhead which may range from 32% to 80%, thus potentially affecting usability. Furthermore, though Noncespaces can be safely adopted on XHTML websites, it is affected by compatibility problems on HTML pages, due to the labelling process which disrupts the names of tags and attributes, and thus the page rendering, on unmodified browsers. Web developers are required to write security policies and revise web applications to support Noncespace, hence the ease of deployment depends on the granularity of the enforced policy.

4.3.7. DSI. In parallel with the development of Noncespaces, Nadji *et al.* proposed a similar solution based on the concept of document structure integrity (DSI) [Nadji et al. 2009]. The approach relies on server-side taint-tracking to mark nodes generated by user-inserted data, so that the client is able to recognize and isolate them during the parsing phase to prevent unintended modifications to the document structure. Untrusted data is delimited by special markers, i.e., sequences of randomly chosen Unicode whitespace characters. These markers are shipped to the browser in the `<head>` section of the requested page along with a simple policy which specifies the allowed HTML tags within untrusted blocks. The policy enforcement is performed by a modified browser supporting the security mechanism which is also able to track dynamic updates to the document structure.

This solution shares with Noncespaces a similar degree of protection. Nevertheless, from a performance standpoint, the defense introduces only a limited overhead on the client-side, since the policies are simpler with respect to Noncespaces and the enforcement mechanism is integrated in the browser instead of relying on an external proxy. As a result, the user experience is not affected. Compatibility is preserved, given that the labelling mechanism does not prevent unmodified browsers from rendering correctly DSI-enabled web applications. Finally, even the deployment is simplified, since no changes to the applications are required and the policy language is more coarse grained than the one proposed in Noncespaces.

4.3.8. Content Security Policy. The aforementioned proposals share the idea of defining a client-side security policy [Weinberger et al. 2011]. The same principle is embraced by the Content Security Policy (CSP) [W3C 2012], a web security policy standardized by the W3C and adopted by all major browsers. CSP is deployed via an additional HTTP response header and allows the specification of the trusted origins from which the browser is permitted to fetch the resources included in the page. The control mechanism is fairly granular, allowing one to distinguish between different types of resources, such as JavaScript, CSS and XHR targets. By default, 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`). However, these constraints can be disabled by using the `'unsafe-inline'` and the `'unsafe-eval'` rules. With the

introduction of CSP Level 2 [W3C 2015a], it is now possible to selectively white-list inline resources without allowing indiscriminate content execution. Permitted resources can be identified in the policy either by their hashes or by random nonces included in the web page as attributes of their enclosing tags.

When properly configured, CSP provides an effective defense against XSS attacks. Still, general content injection attacks, such as markup code injections, are not prevented. CSP policies are written by web developers and transparent to users, so their design supports usability. Compatibility and deployment cost are better evaluated together for CSP. On the one hand, it is easy to write a very lax policy which allows the execution of inline scripts and preserves the functionality of web applications by putting only mild restrictions on cross-origin communication: this ensures compatibility. On the other hand, an effective policy for legacy applications can be difficult to deploy, since inline scripts and styles should be removed or manually white-listed, and trusted origins for content inclusion should be carefully identified [Weinberger et al. 2011]. As of now, the deployment of CSP is not particularly significant or effective [Weissbacher et al. 2014; Calzavara et al. 2016]. That said, the standardization of CSP by the W3C suggests that the defense mechanism is not too hard to deploy on many websites, at least to get some limited protection.

4.3.9. Critical Evaluation. Content injection is one of the most widespread threats to the security of web sessions [OWASP 2013]. Indeed, modern web applications include contents from a variety of sources, burdening the task of identifying malicious contents. Few proposals attempt to provide a comprehensive defense against content injection and the majority of the most popular solutions are only effective against reflected XSS or have very limited scope. Indeed, among the surveyed solutions, client-side XSS filters and `HttpOnly` cookies are by far the most widespread protection mechanisms, implemented by the majority of the web browsers. Under the current state of the art, achieving protection against stored injections while preserving the application functionality requires the intervention of web developers.

Although several of the discussed approaches were only proposed in research papers and never embraced by the industry, some of them contributed to the development of existing web standards. For instance, the hash-based whitelisting approach of inline scripts supported by CSP has been originally proposed as an example policy in the BEEP paper [Jim et al. 2007]. More research is needed to provide more general defenses against a complex problem like content injection.

4.4. Cross-Site Request Forgery and Login CSRF

We now discuss security solutions which are designed to protect against CSRF and login CSRF. We treat these two attacks together, since security solutions which are designed to protect against one of the attacks are typically also effective against the other. In fact, both CSRF and login CSRF exploit cross-site requests which trigger dangerous side-effects on a trusted web application.

4.4.1. Purely Client-side Solutions. Several browser extensions and client-side proxies have been proposed to counter CSRF attacks, including RequestRodeo [Johns and Winter 2006], CsFire [Ryck et al. 2010; Ryck et al. 2011] and BEAP [Mao et al. 2009]. All of these solutions share the same idea of stripping authentication cookies from potentially malicious cross-site requests sent by the browser. The main difference between these proposals concerns the way cross-site requests are deemed malicious: different, more or less accurate heuristics have been put forward for the task.

These solutions are designed to protect against web attackers who host on their web servers pages that include links to a victim website, in the attempt of fooling the browser into sending malicious authenticated requests towards the victim website.

Unfortunately, this protection becomes ineffective if a web attacker is able to exploit a content injection vulnerability on the target website, since it may force the browser into sending authenticated requests originating from a *same-site* position.

A very nice advantage of these client-side defenses is their usability and ease of deployment: the user can just install the extension/proxy on her machine and she will be automatically protected from CSRF attacks. On the other hand, compatibility 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 latest release of CsFire [Ryck et al. 2011], but a large-scale evaluation on the real Web has unveiled that even this approach may sometimes break useful functionalities of standard web browsing: for instance, it breaks legitimate accesses to Flickr or Yahoo via the OpenID single sign-on protocol [Czeskis et al. 2013].

4.4.2. Allowed Referrer Lists (ARLs). ARLs have been proposed as a client/server solution against CSRF attacks [Czeskis et al. 2013]. Roughly, an ARL is just a whitelist that specifies which origins are entitled to send authenticated request to a given website. The whitelist is compiled by web developers willing to secure their websites, while the policy enforcement is done by the browser. If no ARL is specified for a website, the browser behaviour is unchanged when accessing it, i.e., any origin is authorized to send authenticated requests to the website.

ARLs are effective against web attackers, provided that no content injection vulnerability affects any of the whitelisted pages. Their design supports usability, since their enforcement is lightweight and transparent to browser users. Moreover, compatibility is ensured by the enforcement of security restrictions only on websites which explicitly opt-in to the protection mechanism. The ease of deployment of ARLs is acceptable in most cases. Users must adopt a security-enhanced web browser, but ARLs do not require major changes to the existing ones: the authors implemented ARLs 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 only limited efforts: for instance, e-commerce websites may include in their ARL only the desired e-payment provider, e.g., Paypal. However, notice that a correct ARL for Paypal may be large and rather dynamic, since it should enlist all the websites relying on Paypal for payment facilities.

4.4.3. Tokenization. Tokenization is a popular server-side countermeasure against CSRF attacks [Barth et al. 2008]. The idea is that all the requests that might change the state of the web application should include a secret token randomly generated by the server for each session and, possibly, each request: incoming requests that do not include the correct token are rejected. The inclusion of the token is transparently done by the browser during the legitimate use of the website, e.g., every security-sensitive HTML form in the web application is extended to provide the token as a hidden parameter. It is crucial that tokens are bound to a specific session. Otherwise, an attacker could legitimately acquire a valid token for his own session and transplant it into the user browser, to fool the web application into accepting malicious authenticated requests as part of the user session.

Tokenization is robust against web attackers only if we assume they cannot perform content injection attacks. In fact, a content injection vulnerability might give access to all the secret tokens, given that they are included in the DOM of the web page. The usage of secret tokens is completely transparent to the end-user, so there are no usability concerns. However, tokenization may be hard to deploy for web developers. The manual insertion of secret tokens is tedious and typically hard to get right. Some web development frameworks offer automatic support for tokenization, but this

is 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 many different languages [Czeskis et al. 2013].

4.4.4. NoForge. NoForge [Jovanovic et al. 2006] is a server-side proxy sitting between the web server and the web applications to protect. It implements the tokenization approach against CSRF on all requests, without requiring any change to the web application code. NoForge parses the HTTP(S) responses sent by the web server and automatically extends each hyperlink and form contained in them with a secret token bound to the user session; incoming requests are then delivered to the web server only if they contain a valid token.

The protection and the usability offered by NoForge are equivalent to what can be achieved by implementing tokenization at the server side. The adoption of a proxy for the tokenization task significantly simplifies the deployment of the defensive solution, but it has a negative impact on compatibility, since HTML links and forms which are dynamically generated at the client side will not be rewritten to include the secret token. As a result, any request sent by clicking on these links or by submitting these forms will be rejected by NoForge, thus breaking the web application. The authors of NoForge are aware of this problem and state that it can be solved by manually writing scripts which extend links and forms generated at the client side with the appropriate token [Jovanovic et al. 2006]. However, if this need is pervasive, the benefits on deployment offered by NoForge can be easily voided. For this reason we argue that the design of NoForge is not compatible with the modern Web.

4.4.5. Origin Checking. Origin checking is a popular alternative to tokenization [Barth et al. 2008]. Modern web browsers implement the `Origin` header, identifying the security context (origin) that caused the browser to send an HTTP(S) request. For instance, if a link to `http://b.com` is clicked on a page downloaded from `http://a.com`, the corresponding HTTP request will include `http://a.com` in the `Origin` header. Web developers may inspect this header 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. Server-side origin checking is entirely transparent to the end-user and has no impact on the navigation experience, so it may not hinder usability. This solution is simpler to deploy than tokenization, since it can be implemented by using a web application firewall like ModSecurity[2]. Unfortunately, the `Origin` header is not attached to all the cross-origin requests: for instance, the initial proposal of the header was limited to POST requests [Barth et al. 2008] and current web browser implementations still do not ensure that the header is always populated [Barth 2011b]. Web developers should be fully aware of this limitation and ensure that all the state-changing operations in their applications are triggered by requests bearing the `Origin` header. In practice, this may be hard to ensure for legacy web applications [Czeskis et al. 2013].

4.4.6. Critical Evaluation. Effectively preventing CSRFs and login CSRFs is surprisingly hard. Even though the root cause of the security problem is well-understood for these attacks, it is challenging to come up with a solution which is at the same time usable, compatible and easy to deploy. At the time of writing, Allowed Referrer Lists (ARLs) represent the most promising defensive solution against CSRFs and login CSRFs. They are transparent to end-users, respectful towards legacy technology and do not require changes to web application code. Unfortunately, ARLs are not im-

[2]https://www.modsecurity.org/

plemented in major web browsers, so in practice tokenization and origin checking are the most widespread solutions nowadays. These approaches however may be hard to deploy on legacy web applications.

4.5. Cookie Forcing and Session Fixation

We collect together the defenses proposed against cookie forcing and session fixation. In fact, both the attacks rely on the attacker capability to corrupt the integrity of the authentication cookies set by a trusted website.

4.5.1. Serene. The Serene browser extension offers automatic protection against session fixation attacks [Ryck et al. 2012]. It inspects each outgoing 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(S) headers, it is stripped from the outgoing request, hence cookies which have been fixated or forced by a malicious script cannot be used to authenticate the client. The key observation behind this design is that existing websites set their authentication cookies using HTTP(S) headers in the very large majority of cases.

The solution is designed to be robust against web attackers, since they can only set a cookie for the website by exploiting a markup/script injection vulnerability. Conversely, Serene is not effective against related-domain attackers who might use their sites to legitimately set cookies for the whole domain using HTTP headers. The main advantages of Serene are its usability and ease of deployment: users only need to install Serene in their browser and it will provide automatic protection against session fixation for any website, though the false negatives produced by the heuristic for authentication cookies detection may still leave room for attacks. The compatibility of Serene crucially depends on its heuristic: false positives may negatively affect the functionality of websites, 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 [Calzavara et al. 2014].

4.5.2. Origin Cookies. Origin cookies have been proposed to fix some known integrity issues affecting cookies [Bortz et al. 2011]. We have already discussed that standard HTTP cookies do not provide strong integrity guarantees against related-domain attackers and active network attackers. The observation here is that these attackers exploit the relaxation of the same-origin policy applied to cookies (see Section 2.4.4). Origin cookies, instead, are bound to an exact web origin. 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 can be set by websites simply by adding the `Origin` attribute to standard cookies. Origin cookies are sent by the browser inside a new custom header `Origin-Cookie`, thus letting websites distinguish origin cookies from normal ones.

Since origin cookies are isolated between origins, the additional powers of related-domain attackers and active network attackers in setting or overwriting cookies are no longer a problem. The use of origin cookies is transparent to users and their design supports backward compatibility, since origin cookies are treated as standard cookies by legacy browsers (unknown cookie attributes are ignored). Origin cookies are easy to deploy on websites entirely hosted on a single domain and only served over a single protocol: for such a website, it would be enough to add the `Origin` attribute to all its cookies. On the other hand, if a web application needs to share cookies between different protocols or related domains, then the web developer is forced to implement a protocol to link together different sessions built on distinct origin cookies. This may be a non-trivial task to carry out for existing websites.

4.5.3. Authentication Cookies Renewal. The simplest and most effective defense against session fixation is implemented at the server side, by ensuring that the authentication cookies identifying the user session are refreshed when the level of privilege changes, i.e., when the user provides her password to the web server and performs a login [Johns et al. 2011]. If this is done, no cookie fixed by an attacker before the first authentication step may be used to identify the user session. Notice that this countermeasure does not prevent cookie forcing, since the attacker can first authenticate at the website using a standard web browser and then directly force his own cookies into the user browser.

Renewing authentication cookies upon password-based authentication is a recommended security practice and it is straightforward to implement for new web applications. However, retrofitting a legacy 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 or via a server-side proxy, to enforce the renewal of the authentication cookies whenever an incoming HTTP(S) request is identified as a login attempt [Johns et al. 2011]. Clearly, these server-side solutions must ensure that login attempts are accurately detected to preserve compatibility: this is the case, for instance, when the name of the POST parameter bound to the user password is known.

4.5.4. Critical Evaluation. Session fixation is a dangerous attack, but it is relatively easy to prevent. Renewing the authentication cookies upon user authentication is the most popular, effective and widespread solution against these attacks. The only potential issue with this approach is implementing a comprehensive protection for legacy web applications [Johns et al. 2011]. Cookie forcing, instead, is much harder to defend against. The integrity problems of cookies are well-known to security experts, but no real countermeasure against them has been implemented in major web browsers for the sake of backward compatibility. A recent interesting paper by Zheng *et al.* discusses this problem in more detail [Zheng et al. 2015].

4.6. Network Attacks

4.6.1. HTTPS with Secure Cookies. Though it is obvious that websites concerned about network attackers should make use of HTTPS, there are some points worth discussing. For instance, 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 confidentiality and integrity. As a matter of fact, many websites are still only partially deployed over HTTPS, either to increase performance or because only a part of their contents needs to be secured. However, cookies set by a 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 over HTTP, it is still possible to protect the confidentiality of the authentication cookies by setting the `Secure` attribute, which instructs the browser to send these cookies only over HTTPS connections. Even if a website is fully deployed over HTTPS, the `Secure` attribute should be set on its authentication cookies, otherwise a network attacker could still force their leakage in clear by injecting non-existing HTTP links to the website in unrelated web pages [Jackson and Barth 2008].

Activating HTTPS support on a server requires little technical efforts, but needs a signed public key certificate: while the majority of HTTPS-enabled websites employ certificates signed by recognized certification authorities, a non-negligible percentage uses certificates that are self-signed or signed by CAs whose root certificate is not included in major web browsers [Fahl et al. 2014]. Unless explicitly included in the OS or in the browser keychain, these certificates trigger a warning when the browser attempts to validate them, similarly to what happens when a network attacker acts as a

man-in-the-middle and provides a fake certificate: in such a case, a user that proceeds ignoring the warning may be exposed to the active attacker, as if the communication was performed over an insecure channel. The adoption of `Secure` cookies is straightforward whenever the entire website is deployed over HTTPS, since it is enough to add the `Secure` attribute to all the cookies set by the website. For mixed contents websites, `Secure` cookies cannot be used to authenticate the user on the HTTP portion of the site, hence they may be hard to deploy, requiring a change to the cookie scheme.

4.6.2. HProxy. HProxy is a client-side solution which protects against SSL stripping by analyzing the browsing history in order to produce a profile for each website visited by the user [Nikiforakis et al. 2010]. HProxy inspects all the responses received by the user browser and compares them against the corresponding profiles: divergences from the expected behaviour are evaluated through a strict set of rules to decide whether the response should be accepted or rejected.

HProxy is effective only on already-visited websites and the offered protection crucially depends on the completeness of the detection ruleset. From a usability perspective, the browsing experience may be affected by the adoption of the proposed defense mechanism, as it introduces an average overhead of 50% on the overall page load time. The main concern however is about compatibility, since it depends on the ability of HProxy to tell apart legitimate modifications in the web page across consecutive loads from malicious changes performed by the attacker. False positives in this process may break the functionality of benign websites. HProxy is easy to deploy, since the user only needs to install the software on her machine and configure the browser proxy settings to use it.

4.6.3. HTTP Strict Transport Security. HSTS is a security policy implemented in all modern web browsers, which allows a web server to force a client to subsequently communicate only over a secure channel [Hodges et al. 2012]. The policy can be delivered solely over HTTPS using a custom header, where it is possible to specify whether the policy should be enforced also for requests sent to sub-domains (e.g., to protect cookies shared with them) and its lifetime. When the browser performs a request to a HSTS host, its behaviour is modified so that every HTTP reference is upgraded to the HTTPS protocol before being accessed; TLS errors (e.g., self-signed certificates) terminate the communication session and the embedding of mixed contents pages (see Section 2.4.2) are forbidden.

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 clearly the approach cannot cover the entire Web. Additionally, a recently introduced attack against HSTS [Selvi 2014] exploits a Network Time Protocol weakness found on major operating systems that allows to modify the current time via a man-in-the-middle attack, thus making HSTS policies expire. Usability and compatibility are both high, since users are not involved in security decisions and the HTTP(S) header for HSTS is ignored by browsers not supporting the mechanism. The ease of deployment is high, given that web developers can enable the additional HTTP(S) header with little effort by modifying the web server configuration.

4.6.4. HTTPS Everywhere. This extension for Firefox, Chrome and Opera [EFF 2011] 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 from the website is needed: the hard-coded ruleset is populated by security experts and volunteers.

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 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 behaviour; these problems can be rectified by refining the ruleset. The solution is very easy to deploy: the user is only required to install the extension to enforce the usage of HTTPS on supported websites.

4.6.5. Critical Evaluation. HTTPS is pivotal in defending against network attacks: all the assessed solutions try to promote insecure connections to encrypted ones or force web developers to deploy the whole application on HTTPS. Mechanisms exposing compatibility problems are unlikely to be widely adopted, as in the case of HProxy due to its heuristic approach. All the other defenses, instead, are popular standards or enjoy a large user base. Academic solutions proved to be crucial for the development of web standards: HSTS is a revised version of ForceHTTPS [Jackson and Barth 2008] in which a custom cookie was used in place of an HTTP header to enable the protection mechanism.

4.7. Summary

We summarize in Table II all the defenses discussed so far. We denote with ★ those solutions whose ease of deployment depends on the policy complexity. When the adoption of a security mechanism is much harder on legacy web applications with respect to newly developed or modern ones, we annotate the score with †.

5. 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. In the literature we also find proposals providing a more comprehensive solution to a range of different threats. These proposals are significantly more complex than those in the previous section, hence it is much harder to provide a schematic overview of their merits and current limitations.

5.1. Origin-Bound Certificates

Origin-Bound Certificates (OBC) [Dietz et al. 2012] have been proposed as an extension to the TLS protocol that binds authentication tokens to trusted encrypted channels. The idea is to generate, on the client side, a different certificate for every web origin upon connection. This certificate is sent to the server and used to cryptographically bind authentication cookies to the channel established between the browser and that specific origin. The browser relies on the same certificate when arranging a TLS connection with a previously visited origin. The protection mechanism implemented by OBC is effective at preventing the usage of authentication cookies outside of the intended channel: for instance, a cookie leaked via a content injection vulnerability cannot be reused by an attacker to identify himself as the victim on the vulnerable website, since the victim certificate is not disclosed. Similarly, session fixation attacks are defeated by OBC, given that the cookie value associated to the attacker channel cannot be used within the victim TLS connection.

The presence of OBC is completely transparent to the user and the impact on performances is negligible after certificate generation, so the usability of the solution is high. Compatibility is not at harm, since the browser and the server must explicitly agree on the use of OBC during the TLS handshake. One problem is represented by domain cookies, i.e., cookies accessed by multiple origins: to overcome this issue, the authors suggested a *legacy mode* of OBC in which the client generates certificates bound to the whole domain instead of a single origin. Being an extension to the TLS protocol, OBC

Table II. Analysis of Proposed Defenses

	Defense	*Type*	*Usability*	*Compatibility*	*Ease of Deployment*
Content injection mitigation	HttpOnly cookies	hybrid	H	H	H
	SessionShield/Zan	client	H	L	H
	Requests filtering	client	L	H	H
Content injection prevention	Client-side XSS filters	client	H	H	H
	Server-side filtering	server	H	M	L/M★
	XSS-Guard	server	H	M	H
	BEEP	hybrid	M	H	L/M★
	Blueprint	hybrid	M	M	L
	Noncespaces	hybrid	M	L	L/M★
	DSI	hybrid	H	H	M
	CSP	hybrid	H	H	L/M★
CSRF Login CSRF	Client-side defenses	client	H	L	H
	Allowed referrer lists	hybrid	H	H	L/M★
	Tokenization	server	H	H	L/H†
	NoForge	server	H	L	H
	Origin checking	server	H	H	L/H†
Cookie forcing Session fixation	Serene	client	H	L	H
	Origin cookies	hybrid	H	H	M/H†
Session fixation	Auth. cookies renewal	server	H	H	M/H†
Network attacks	HTTPS w. secure cookies	hybrid	H	H	M/H†
	HProxy	client	M	L	H
	HSTS	hybrid	H	H	H
	HTTPS Everywhere	client	M	H	H

requires changes to both parties involved in the encrypted channel initiation. The authors successfully implemented the described mechanism on the open-source browser Chromium and on OpenSSL by altering approximately 1900 and 320 lines of code, respectively. However, web developers are not required to adapt their applications to use OBC, which has a beneficial impact on ease of deployment.

5.2. Browser-based Information Flow Control

Browser-based information flow control is a promising approach to uniformly prevent a wide class of attacks against web sessions. FlowFox [Groef et al. 2012] was the first web browser implementing a full-fledged information flow control framework for confidentiality policies on JavaScript code. Later work on the same research line includes JSFlow [Hedin et al. 2014], COWL [Stefan et al. 2014] and an extension of Chromium with information flow control [Bauer et al. 2015], which we refer to as ChromiumIFC. These solutions explore different points of the design space:

— FlowFox is based on *secure multi-execution*, a dynamic approach performing multiple runs of a given program (script) under a special policy for input/output operations ensuring non-interference [Devriese and Piessens 2010]. To exemplify, assume the existence of two security levels Public and Secret, then the program is executed twice (once per level) under the following regime: (1) outputs marked Public/Secret are only done in the execution at level Public/Secret; and (2) inputs at level Public are fed to both the executions, while inputs at level Secret are only fed to the execu-

tion at level Secret (a default value for the input is provided to the Public execution). This ensures by construction that Private inputs do not affect Public outputs;
— JSFlow is based on a dynamic *type system* for JavaScript. JavaScript values are extended with a security label representing their confidentiality level and labels are updated to reflect the computational effects of the monitored scripts. Labels are then dynamically checked to ensure that computations preserve non-interference;
— COWL performs a *compartmentalization* of scripts and assigns security labels at the granularity of compartments encapsulating contents from a single origin. It enforces coarse-grained policies on communication across compartments and towards remote origins via label checking;
— ChromiumIFC implements a lightweight dynamic *taint tracking* technique to constrain information flows within the browser and prevent the leakage of secret information. In contrast to previous proposals, this solution is not limited to JavaScript, but it spans all the most relevant browser components.

The different design choices taken by the reviewed solutions have a clear import on our evaluation factors. In terms of protection, enforcing information flow control on scripts is already enough to prevent many web threats. For instance, assuming an appropriate security policy, web attackers cannot leak authentication cookies using XSS [Groef et al. 2012] or run CSRF attacks based on JavaScript [Khan et al. 2014]. This is true also in presence of stored XSS attacks, provided that information flow control is performed on the injected scripts. However, there are attack vectors which go beyond scripts, e.g., a web attacker can carry out a CSRF by injecting markup elements. Preventing these attacks requires a more extensive monitoring of the web browser, as the one proposed by ChromiumIFC.

To the best of our knowledge, there has been no thorough usability study for any of the cited solutions. It is thus unclear if and to which extent users need to be involved in security decisions upon normal browsing. However, degradation of performances caused by information flow tracking may hinder the user experience and negatively affect usability. For instance, the performances of FlowFox are estimated to be around 20% worse than those of a standard web browser, even assuming only policies with two security levels [Groef et al. 2012]. Better performances can be achieved by using simpler enforcement mechanisms and by lowering the granularity of enforcement, for instance the authors of COWL performed a very promising performance evaluation of their proposal [Stefan et al. 2014];

Compatibility and ease of deployment are better evaluated together, since there is a delicate balance between the two in this area, due to the flexibility of information flow policies. On the one hand, inaccurate information flow policies can break existing websites upon security enforcement, thus affecting compatibility. On the other hand, accurate information flow policies may be large and hard to get right, thus hindering deployment. We think that a set of default information flow policies may already be enough to stop or mitigate a wide class of attacks against web sessions launched by malicious scripts: for instance, cookies could be automatically marked as private for the domain which set them. Indeed, a preliminary experiment with FlowFox on the top 500 sites of Alexa shows that compatibility is preserved for a very simple policy which marks as sensitive any access to the cookie jar [Groef et al. 2012]. Reaping the biggest benefits out of information flow control, however, necessarily requires some efforts by web developers.

5.3. Security Policies for JavaScript

Besides information flow control, in the literature there are several frameworks for enforcing general security policies on untrusted JavaScript code [Meyerovich and

Livshits 2010; Yu et al. 2007; Louw et al. 2013; Phung et al. 2009; Van Acker et al. 2011]. We just provide a brief overview on them here and we refer the interested reader to a recent survey by Bielova [Bielova 2013] for additional details. The core idea behind all these proposals is to implement a runtime monitor that intercepts the API calls made by JavaScript programs and checks whether the sequence of such calls complies with an underlying security policy. This kind of policies has proved helpful for protecting access to authentication cookies, thus limiting the dangers posed by XSS, and for restricting cross-domain communication attempts by untrusted code, which helps at preventing CSRF attacks. We believe that other useful policies for protecting web sessions can be encoded in these rather general frameworks, though the authors of the original papers do not discuss them in detail. Since all these proposals assume that JavaScript code is untrusted, they are effective even in presence of stored XSS attacks, provided that the injected scripts are subject to policy enforcement.

As expected, security policies for JavaScript share many of the strengths and weaknesses of browser-based information flow control in terms of protection, usability and compatibility. Ease of deployment, instead, deserves a more careful discussion, since it fundamentally depends on the complexity of the underlying policy language. For instance, in [Meyerovich and Livshits 2010] security policies are expressed in terms of JavaScript code, while the framework in [Yu et al. 2007] is based on *edit automata*, a particular kind of state machine with a formal semantics. Choosing the right policy language may significantly improve the ease of deployment, though we believe that meaningful security policies require some efforts by web developers. There is some preliminary evidence that useful policies can be automatically synthesized by static analysis or runtime training: the idea is to monitor normal JavaScript behaviour and to deem as suspicious all the unexpected script behaviours [Meyerovich and Livshits 2010]. However, we believe more research is needed to draw a fair conclusion on how difficult it is to deploy these mechanisms in practice.

5.4. 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 server-side proxy [Guha et al. 2009]. The static analysis is employed by web developers to construct the control flow graph of the Ajax application to protect, while the proxy dynamically monitors browser requests to prevent violations to the expected control flow of the web application. The solution also implements defenses against *mimicry attacks*, in which the attacker complies with legitimate access patterns in his malicious attempts. This is done by making each session (and thus each graph) slightly different than the other ones by placing unpredictable, dummy requests in selected points of the control flow. The JavaScript code of the web application is then automatically modified to trigger these requests, which instead cannot be predicted by the attacker.

The approach is deemed useful to mitigate the threats posed by content injection and to prevent 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 stored XSS attacks exploiting Ajax requests, whenever they are mounted after the construction of the control flow graph.

The solution offers high usability, since it is transparent to users and the runtime overhead introduced by the proxy is minimal. According to the authors, the adoption of a context-sensitive static analysis for JavaScript makes the construction of the control flow graph very precise, which is crucial to preserve the functionality of the web appli-

cation and ensure compatibility. The authors claim that the solution is easy to deploy, since the construction of the control flow graph is totally automatic and the adoption of a proxy does not require changes to the web application code.

5.5. Escudo

Escudo [Jayaraman et al. 2010] is an alternative protection model for web browsers, extending the standard same-origin policy to rectify several of its known shortcomings. By noticing a strong similarity between the browser and an operating system, the authors of Escudo argue for the adoption of a protection mechanism based on hierarchical rings, whereby different elements of the DOM are placed in rings with decreasing privileges; the definition of the number of rings and the ring assignment for the DOM elements is done by web developers. Developers can also assign protection rings to their cookies, while the internal browser state containing, e.g., the history, is set by default in ring 0. Access to objects in a given ring is only allowed to subjects being in the same or lower rings.

Escudo is designed to prevent XSS and CSRF attacks. Untrusted web contents should be assigned to the least privileged ring, so that scripts crafted by exploiting a reflected XSS vulnerability would do no harm. Similarly, requests from untrusted web pages should be put in a low privilege ring without access to authentication credentials, thus preventing CSRF attacks. Notice, 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 network attacks.

Escudo does not require user interventions for security enforcement and it only leads to a slight overhead on page rendering (around 5%). This makes the solution potentially usable. However, deploying ring assignments for Escudo looks challenging. The authors evaluated 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 of Escudo may be complicated, especially if a comprehensive protection 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 compatibility issue will arise.

5.6. CookiExt

CookiExt [Bugliesi et al. 2015] is a Google Chrome extension protecting the confidentiality of authentication cookies against both web and network attacks. The extension adopts a heuristic to detect authentication cookies in incoming 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 which have been extended with the `Secure` attribute. In order to preserve compatibility, the extension implements a fallback mechanism which removes the `Secure` attribute automatically assigned to authentication cookies in case the server does not support HTTPS for some of the web pages. The design of CookiExt has been formally validated by proving that a browser with CookiExt satisfies noninterference with respect to the value of the authentication cookies. In particular, it is shown that what an attacker can observe of the CookiExt browser behaviour is unaffected by the value of authentication cookies. CookiExt does not protect against CSRF and session fixation: it just ensures the confidentiality of the authentication cookies.

CookiExt does not require any user interaction and features a lightweight implementation, which guarantees a high level of usability. Preliminary experiments performed by the authors show good compatibility results on existing websites from Alexa, since only minor annoyances due to security enforcement have been found; however, a large-scale evaluation of the extension is still missing. Being implemented as a browser extension, CookiExt is very easy to deploy.

5.7. SessInt

SessInt [Bugliesi et al. 2014] is an extension for Google Chrome providing a purely client-side countermeasure against the most common attacks targeting web sessions. The extension prevents the abuse of authenticated requests and protects authentication credentials. It enforces web session integrity by combining access control and taint tracking mechanisms in the browser. The security policy applied by SessInt has been verified against a formal threat model including both web and network attackers. As a distinguishing feature with respect to other client-side solutions, SessInt is able to stop CSRF attacks even when they are launched by exploiting reflected XSS vulnerabilities. On the other hand, no protection is given against stored XSS.

The protection provided by SessInt is fully automatic: its security policy is uniformly applied to every website and no interaction with the web server or the end-user is required. Also, the performance overhead introduced by the security checks of SessInt is negligible and no user interaction is needed. However, the protection offered by SessInt comes at a cost on compatibility: the current prototype of the extension breaks several useful web scenarios, including single sign-on protocols and e-payment systems. The implementation as a browser extension makes SessInt very easy to deploy.

5.8. Same Origin Mutual Approval

SOMA [Oda et al. 2008] is a research proposal describing a simple yet powerful policy for content inclusion and remote communication on the Web. SOMA enforces that a web page from a domain d_1 can include contents from an origin o hosted on domain d_2 only if both the following checks succeed: (1) d_1 has listed o as an allowed source of remote contents and (2) d_2 has listed d_1 as an allowed destination for content inclusion. SOMA is designed to offer protection against web attackers: web developers can effectively prevent 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.

The protection offered by SOMA does not involve user intervention and the performances of the solution look satisfactory, especially on cached page loads, where only an extra 5% of network latency is introduced. This ensures that SOMA can be a usable solution. Moreover, if a SOMA policy correctly includes all the references to the necessary web resources, no compatibility issues will occur. Writing correct policies looks feasible in practice, since similar specifications are also used by popular web standards like CSP. The deployment of SOMA would not be trivial, but 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 expected be relatively small in practice; most importantly, no change to the web application code is required.

5.9. App Isolation

App Isolation [Chen et al. 2011] is a defense mechanism aimed at offering, within a single browser, the protection granted by the usage of different browsers for navigating websites at separate levels of trust. If one “sensitive” browser is only used to navigate trusted websites, while another “non-sensitive” browser is only used to access

potentially malicious web pages, many of the threats posed by the latter are voided by the absence of shared state between the two browsers. For instance, CSRF attacks would fail, since they would be launched from an attacker-controlled web page in the non-sensitive browser, but the authentication cookies for all trusted web applications would only be available in the sensitive browser. Enforcing this kind of guarantees within a single browser requires two ingredients: (1) a strong *state isolation* among web applications and (2) an *entry point restriction*, preventing the access to sensitive web applications from maliciously crafted URLs. Indeed, in the example above, protection would be voided if the link mounting the CSRF attack was opened in the sensitive browser. This design is effective at preventing reflected XSS attacks, session fixation and CSRF. However, stored XSS attacks against trusted websites will bypass the protection offered by App Isolation, since the injected code would be directly delivered from a trusted position.

The usability of App Isolation looks promising, since the protection is applied automatically and the only downside is a slight increase in the loading time of the websites, due to the additional round-trip needed to fetch the list of allowed entry points. The compatibility of the solution is ensured by the fact that supporting browsers only enforce protection when explicitly requested by the web application. Web developers, however, should compile a list of entry points defining the allowed landing pages of their web applications. 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 sites, e.g., newspapers websites, where users may want to jump directly to any URL featuring an article. The ease of deployment thus crucially depends on the nature of the web application to protect.

5.10. Summary

We summarize our observations about the described solutions in Table III. Again, we denote with ★ the solutions where the ease of deployment is affected by the policy complexity. Additionally, we use a dash symbol whenever we do not have any definite evidence about a specific aspect of our investigation based on the existing literature. Most notably, we leave empty the Usability and Compatibility entries for browser-based information flow control and JavaScript security policies, since they depend too much on the specific implementation choices and the policies to enforce. More research is needed to understand these important aspects.

Table III. Defenses Against Multiple Attacks

Defense	*Type*	*Content injection*	*CSRF Login CSRF*	*Session fixation Cookie forcing*	*Network attacks*	*Usability*	*Compatibility*	*Ease of Deployment*
OBC	hybrid	✓	✗	✓	✓	H	H	M
Browser IFC	hybrid	✓	✓	✓	✗	-	-	L/M★
JS Policies	hybrid	✓	✓	✓	✗	-	-	L/M★
Ajax IDS	server	✓	✓	✓	✗	H	H	H
Escudo	hybrid	✓	✓	-	✗	H	H	L/M★
CookiExt	client	✓	✗	✗	✓	H	M	H
SessInt	client	✓	✓	✓	✓	H	L	H
SOMA	hybrid	✓	✓	✗	✗	H	H	M
App Isolation	hybrid	✓	✓	✓	✗	H	H	L/M★

6. PERSPECTIVE

Having examined different proposals, we now identify five guidelines for the designers of novel web security mechanisms. This is a synthesis of sound principles and insights which have, to different extents, been taken into account by all the designers of the proposals we surveyed.

6.1. Transparency

We call *transparency* the combination of high usability and full compatibility: we think this is the most important ingredient to ensure a large scale deployment of any defensive solution for the Web, given its massive user base and its heterogeneity. It is well-known that security often comes at the cost of usability and that usability defects ultimately weaken security, since users resort to deactivating or otherwise sidestepping the available protection mechanisms [Theofanos and Pfleeger 2011]. The Web is extremely variegate and surprisingly fragile even to small changes: web developers who do not desire to adopt new defensive technologies should be able to do so, without any observable change to the semantics of their web applications when these are accessed by security-enhanced web browsers; dually, users who are not willing to update their web browsers should be able to seamlessly navigate websites which implement cutting-edge security mechanisms not supported by their browsers.

All the security decisions must be ultimately taken by web developers. On the one hand, users are not willing or do not have the expertise to be involved in security decisions. On the other hand, it is extremely difficult for browser vendors to come up with "one size fits all" solutions which do not break any website. Motivated web developers, instead, can be fully aware of their web application semantics, thoroughly test new proposals and configure them to support compatibility.

Examples: Hybrid client/server solutions like ARLs (Section 4.4.2), CSP (Section 4.3.8) and SOMA (Section 5.8) are prime examples of proposals which ensure transparency, since they do not change the semantics of web applications not adopting them. Conversely, purely client-side defenses like Serene (Section 4.5.1) and SessInt (Section 5.7) typically present some compatibility issues, since they lack enough contextual information to be always precise in their security decisions: this makes them less amenable for a large-scale deployment.

6.2. Security by Design

Supporting the current Web and legacy web applications is essential, but developers of new websites should be provided with tools which allow them to realize applications which are secure *by design*. Our feeling is that striving for backward compatibility often hinders the creation of tools which could actually improve the development process of new web applications. Indeed, backward compatibility is often identified with problem-specific patches to known issues, which developers of existing websites can easily plug into their implementation to retrofit it. The result is that developing secure web applications using the current technologies is a painstaking task, which involves actions at too many different levels. Developers should be provided with tools and methodologies which allow them to take security into account from the first phases of the development process. This necessarily means deviating from the low-level solutions advocated by many current technologies, to rather focus on more high-level security aspects of the web application, including the definition of principals and their trust relations, the identification of sensitive information, etc.

Examples: Proposals which are secure by design include the non-interference policies advocated by FlowFox (Section 5.2) and several frameworks for enforcing arbitrary security policies on untrusted JavaScript code (Section 5.3). Popular examples of solu-

tions which are not secure by design include the usage of secret tokens against CSRF attacks (Section 4.4.3): indeed, not every token generation scheme is robust [Barth et al. 2008] and ensuring the confidentiality of the tokens may be hard, even though this is crucial for the effectiveness of the solution.

6.3. Ease of Adoption

Server-side solutions should require a limited effort to be understood and adopted by web developers. For instance, the usage of frameworks which automatically implement recommended security practices, often neglected by web developers, can significantly simplify the development of new secure applications. For client-side solutions, it is important that they work out of the box when they are installed in the user browser: proposals which are not fully automatic are going to be ignored or misused. Any defensive solution which involves both the client and the server is subject to both the previous observations. Since it is unrealistic that a single protection mechanism is able to accommodate all the security needs, it is crucial to design the defensive solution so that it gracefully interacts with existing proposals which address orthogonal issues and which may already be adopted by web developers.

Examples: Many client-side defenses are easy to adopt, since they are deployed as browser extensions which automatically provide additional protection: this is the case of tools like CsFire (Section 4.4.1) and CookiExt (Section 5.6). Server-side or hybrid client/server solutions are often harder to adopt, for different reasons: some proposals like Escudo (Section 5.5) are too fine-grained and thus require a huge configuration effort, while others like FlowFox (Section 5.2) may be hard for web developers to understand. Good examples of hybrid client/server solutions which promise an easy adoption, since they speak the same language of web developers, include SOMA (Section 5.8) and HSTS (Section 4.6.3). Origin checking is often straightforward to implement as a server-side defense against CSRF attacks (Section 4.4.5).

6.4. Declarative Nature

To support a large-scale deployment, new defensive solutions should be *declarative* in nature: web developers should be given access to an appropriate policy specification language, but the enforcement of the policy should not be their concern. Security checks should not be intermingled with the web application logic: ideally, no code change should be implemented in the web application to make it more secure and a thorough understanding of the web application code should not be necessary to come up with reasonable security policies. This is dictated by very practical needs: existing web applications are huge and complex, are often written in different programming languages and web developers may not have full control over them.

Examples: Whitelist-based defenses like ARLs (Section 4.4.2) and SOMA (Section 5.8) are declarative in nature, while the tokenization (Section 4.4.3) is not declarative at all, since it is a low-level solution and it may be hard to adopt on legacy web applications.

6.5. Formal Specification and Verification

Formal models and tools have been recently applied to the specification and the verification of new proposals for web session security [Bohannon and Pierce 2010; Akhawe et al. 2010; Fett et al. 2014; Bugliesi et al. 2014]. While a formal specification may be of no use for web developers, it assists security researchers in understanding the details of the proposed solution. Starting from a formal specification, web security designers can be driven by the enforcement of a clear *semantic* security property, e.g., non-interference [Groef et al. 2012] or session integrity [Bugliesi et al. 2014], rather

than by the desire of providing ad-hoc solutions to the plethora of low-level attacks which currently target the Web.

This is not merely a theoretical exercise, but it has clear practical benefits. First, it allows a comprehensive identification of *all* the attack vectors which may be used to violate the intended security property, thus making it harder that subtle attacks are left undetected during the design process. Second, it forces security experts to focus on a rigorous threat model and to precisely state all the assumptions underlying their proposals: this helps making a critical comparison of different solutions and simplifies their possible integration. Third, more speculatively, targeting a property rather than a mechanism allows to get a much better understanding of the security problem, thus fostering the deployment of security mechanisms which are both more complete and easier to use for web developers.

Examples: To the best of our knowledge, only very few of the proposals we surveyed are backed up by a solid formal verification. Some notable examples include CookiExt (Section 5.6), SessInt (Section 5.7), FlowFox (Section 5.2) and CsFire (Section 4.4.1).

6.6. Discussion

Retrospectively looking at the solutions we reviewed, we identify a number of carefully crafted proposals which comply with several of the guidelines we presented. Perhaps surprisingly, however, we also observe that *none* of the proposals complies with all the guidelines. We argue that this is not inherent to the nature of the guidelines, but rather the simple consequence of web security being hard: indeed, many different problems at very different levels must be taken into account when targeting the largest distributed system in the world.

The challenges of the web platform. Nowadays, there is a huge number of different web standards and technologies, and most of them are scattered across different RFCs. This makes it hard to get a comprehensive picture of the web platform and, conversely, makes it extremely easy to underestimate the impact of novel defense mechanisms on the web ecosystem. Moreover, the sheer size of the Web makes it difficult to assume typical use case scenarios, since large-scale evaluations often reveal surprises and contest largely accepted assumptions [Richards et al. 2011; Nikiforakis et al. 2012; Calzavara et al. 2014].

Particular care is needed when designing web security solutions, given the massive user base of the Web, whose popularity heavily affects what security researchers and engineers may actually propose to improve its security. Indeed, one may argue that the compatibility and the usability of a web defense mechanism may even be more important than the protection it offers. This may be hard to accept, since it partially limits the design space for well-thought solutions tackling the root cause of a security issue. However, the quest for usability and compatibility is inherently part of the web security problem and it should never be underestimated.

The architecture of an effective solution. Purely client-side solutions are likely to break compatibility, since the security policy they apply should be acceptable for every website, but "one size fits all" solutions do not work in a heterogeneous environment like the Web. The best way to ensure that a client-side defense preserves compatibility is to adopt a whitelist-based approach, so as to avoid that the defensive mechanism is forced to guess the right security decision. However, the protection offered by a whitelist is inherently limited to a known set of websites.

Similarly, purely server-side approaches have their limitations. Most of the server-side solutions we surveyed are hard to adopt and not declarative at all. When this is not the case, like in NoForge (Section 4.4.4), compatibility is at risk. Indeed, just

as client-side solutions are not aware of the web application semantics, server-side approaches have very little knowledge of the client-side code running in the browser.

Based on our survey and analysis, we confirm that hybrid client/server designs hold great promise in being the most effective solution for future proposals [Weinberger et al. 2011]. We observe that it is relatively easy to come up with hybrid solutions which are compliant with the first four guidelines: SOMA (Section 5.8), HSTS (Section 4.6.3) and ARLs (Section 4.4.2) are good examples.

A note on formal verification. It may be tempting to think that proposals which comply with the first four guidelines are already good enough, since their formal verification can be performed a posteriori. However, this is not entirely true: solutions which are not designed with formal verification in mind are often over-engineered and very difficult to prove correct, since it is not obvious what they are actually trying to enforce. For many solutions, we just know that they prevent some attacks, but it is unclear whether other attacks are feasible under the same threat model and there is no assurance that a sufficiently strong security property can be actually proved for them.

We thus recommend to take formal verification into account from the first phases of the design process. A very recent survey discusses why and how formal methods can be fruitfully applied to web security and highlights open research directions [Bugliesi et al. 2017].

Open problems and new research directions. We have observed that, at the moment, there exist no solution complying with the five guidelines above and that solutions complying with the first four guidelines still miss a formal treatment. One interesting line of research would be to try to formally state the security properties provided by those solutions under various threat models. As we discussed, proving formal properties of existing mechanisms is not trivial (and sometimes not even feasible) and requires, in the first place, to come up with a precise statement of the security goals. SOMA (Section 5.8), HSTS (Section 4.6.3) and ARLs (Section 4.4.2) are certainly good candidates for this formal analysis.

However, having a single solution covering the five guidelines would be far from providing a universal solution for web session security. We have seen that most of the proposals target specific problems and attacks. The definition of a general framework for studying, comparing, and composing web security mechanisms might help understanding in which extent different solutions compose and what would be the resulting security guarantee. Modular reasoning looks particularly important in this respect, since the web platform includes many different components and end-to-end security guarantees require all of them to behave correctly. This would go in the direction of securing web sessions in general, instead of just preventing classes of attacks.

For what concerns new solutions, we believe that they should be supported by a formal specification and a clear statement of the security goals and of the threat model. The development of new, well-founded solutions would certainly benefit from the investigation and formal analysis of existing, practical solutions. However, new solutions should try to tackle web session security at a higher level of abstraction, independently of the specific attacks. They should be designed with all of the above guidelines in mind which, in turns, suggests a hybrid approach. The formal model would clarify what are the critical components to control and what (declarative) server side information is necessary to implement a transparent, secure by design and easy to adopt solution.

7. CONCLUSION

We took a retrospective look at different attacks against web sessions and we surveyed the most popular solutions against them. For each solution, we discussed its security guarantees against different attacker models, its impact on usability and compatibility,

and its ease of deployment. We then synthesized five guidelines for the development of new web security solutions, based on the lesson learned from previous experiences. We believe that these guidelines can help web security experts in proposing novel solutions which are both more effective and amenable for a large-scale adoption.

REFERENCES

Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*. 290–304.

Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos. 2009. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the 2009 IEEE Web 2.0 Security and Privacy Workshop*.

Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2013. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Proceedings of the 2nd International Conference on Principles of Security and Trust, POST 2013*. 126–146.

Adam Barth. 2011a. HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265. (2011).

Adam Barth. 2011b. The web origin concept. http://tools.ietf.org/html/rfc6454. (2011).

Adam Barth, Collin Jackson, and John C. Mitchell. 2008. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 75–88.

Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular Expressions Considered Harmful in Client-side XSS Filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*. 91–100.

Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015*.

Nataliia Bielova. 2013. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming* 82, 8 (2013), 243–262.

Prithvi Bisht and V. N. Venkatakrishnan. 2008. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2008*. 23–43.

Aaron Bohannon and Benjamin C. Pierce. 2010. Featherweight Firefox: Formalizing the Core of a Web Browser. In *USENIX Conference on Web Application Development, WebApps 2010*.

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

Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. 2017. Formal methods for web security. *J. Log. Algebr. Meth. Program.* (2017). To appear.

Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. 2015. CookiExt: Patching the Browser Against Session Hijacking Attacks. *Journal of Computer Security* 23, 4 (2015), 509–537.

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

Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*. To appear.

Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. 2014. Quite a Mess in My Cookie Jar!: Leveraging Machine Learning to Protect Web Authentication. In *Proceedings of the 23rd International World Wide Web Conference, WWW 2014*. 189–200.

Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. 2011. App isolation: Get the Security of Multiple Browsers with Just One. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 227–238.

Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. 2013. Lightweight Server Support for Browser-Based CSRF Protection. In *Proceedings of the 22nd International World Wide Web Conference, WWW 2013*. 273–284.

Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. 2012. One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens. *ACM Transactions on Internet Technology* 12, 1 (2012), 1–24.

Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 109–124.

Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. 2012. Origin-Bound Certificates: a Fresh Approach to Strong Client Authentication for the Web. In *Proceedings of the 21th USENIX Security Symposium, USENIX 2012*. 317–331.

ECMA. 2011. ECMAScript Language Specification. http://www.ecma-international.org/ecma-262/5.1/. (2011).

EFF. 2011. HTTPS Everywhere. https://www.eff.org/https-everywhere. (2011).

Sascha Fahl, Yasemin Acar, Henning Perl, and Matthew Smith. 2014. 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, ASIA CCS 2014*. 507–512.

Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System. In *Proceedings of the 35th IEEE Symposium on Security and Privacy, S&P 2014*. 673–688.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 748–759.

Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. 2009. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009*. 561–570.

Matthew Van Gundy and Hao Chen. 2012. Noncespaces: Using Randomization to Defeat Cross-site Scripting Attacks. *Computers & Security* 31, 4 (2012), 612–628.

Per A. Hallgren, Daniel T. Mauritzson, and Andrei Sabelfeld. 2013. 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*. 71–82.

Norman Hardy. 1988. The Confused Deputy (or why capabilities might have been invented). *Operating Systems Review* 22, 4 (1988), 36–38.

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *Proceedings of the 29th Symposium On Applied Computing, SAC 2014*. 1663–1671.

Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 760–771.

Jeff Hodges, Collin Jackson, and Adam Barth. 2012. HTTP Strict Transport Security (HSTS). http://tools.ietf.org/html/rfc6797. (2012).

Bob Ippolito. 2015. JSONP. http://json-p.org/. (2015).

Collin Jackson and Adam Barth. 2008. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*. 525–534.

Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. 2010. ESCUDO: A Fine-Grained Protection Model for Web Browsers. In *Proceedings of the 2010 International Conference on Distributed Computing Systems, ICDCS 2010*. 231–240.

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

Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. 2011. Reliable Protection Against Session Fixation Attacks. In *Proceedings of the 26th ACM Symposium on Applied Computing, SAC 2011*. 1531–1537.

Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. 2012. BetterAuth: Web Authentication Revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012*. 169–178.

Martin Johns, Ben Stock, and Sebastian Lekies. 2014. A Tale of the Weaknesses of Current Client-Side XSS Filtering. In *Blackhat USA 2014*.

Martin Johns and Justus Winter. 2006. RequestRodeo: Client Side Protection against Session Riding. *Proceedings of the OWASP Europe 2006 Conference* (2006), 5–17.

Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, SecureComm 2006*. 1–10.

Wilayat Khan, Stefano Calzavara, Michele Bugliesi, Willem De Groef, and Frank Piessens. 2014. Client Side Web Session Integrity as a Non-interference Property. In *Proceedings of the 10th International Conference on Information Systems Security, ICISS 2014*. 89–108.

Engin Kirda, Christopher Krügel, Giovanni Vigna, and Nenad Jovanovic. 2006. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006*. 330–337.

Mike Ter Louw, Phu H. Phung, Rohini Krishnamurti, and Venkat N. Venkatakrishnan. 2013. SafeScript: JavaScript Transformation for Policy Enforcement. In *Proceedings of the 18th Nordic Conference on Secure IT Systems, NordSec 2013*. 67–83.

Mike Ter Louw and V. N. Venkatakrishnan. 2009. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy, S&P 2009*. 331–346.

Ziqing Mao, Ninghui Li, and Ian Molloy. 2009. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In *Proceedings of the 13th International Conference on Financial Cryptography and Data Security, FC 2009*. 238–255.

Giorgio Maone. 2004. The NoScript Firefox Extension. http://noscript.net/. (2004).

Moxie Marlinspike. 2009. New Tricks for Defeating SSL in Practice.. In *BlackHat DC 2009*.

Leo A. Meyerovich and V. Benjamin Livshits. 2010. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 481–496.

Mozilla. 2015. Same-origin policy. http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. (2015).

Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*.

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

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 736–747.

Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. 2011. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011*. 87–100.

Nick Nikiforakis, Yves Younan, and Wouter Joosen. 2010. HProxy: Client-Side Detection of SSL Stripping Attacks. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2010*. 200–218.

Terri Oda, Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. 2008. SOMA: Mutual Approval for Included Content in Web Pages. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 89–98.

OWASP. 2013. Top 10 Security Threats. https://www.owasp.org/index.php/Top_10_2013-Top_10. (2013).

OWASP. 2014. HttpOnly. https://www.owasp.org/index.php/HttpOnly. (2014).

Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight Self-protecting JavaScript. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2009*. 47–60.

Tadeusz Pietraszek and Chris Vanden Berghe. 2005. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, RAID 2005*. 124–145.

Eric Rescorla. 2000. HTTP Over TLS. https://tools.ietf.org/html/rfc2818. (2000).

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. 52–78.

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

Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. 2010. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Proceedings of Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010*. 18–34.

Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. 2011. Automatic and Precise Client-Side Protection against CSRF Attacks. In *Proceedings of the 16th European Symposium on Research in Computer Security, ESORICS 2011*. 100–116.

Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2012. Serene: Self-Reliant Client-Side Protection against Session Fixation. In *Proceedings of the 2012 Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012*. 59–72.

Jose Selvi. 2014. Bypassing HTTP Strict Transport Security. In *BlackHat DC 2014*.

Kapil Singh, Helen J. Wang, Alexander Moshchuk, Collin Jackson, and Wenke Lee. 2012. Practical End-to-End Web Content Integrity. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012*. 659–668.

Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, David Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*. 131–146.

Shuo Tang, Nathan Dautenhahn, and Samuel T. King. 2011. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 615–626.

Mary Frances Theofanos and Shari Lawrence Pfleeger. 2011. Guest Editors' Introduction: Shouldn't All Security Be Usable? *IEEE Security & Privacy* 9, 2 (2011), 12–17.

Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2011. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011*. 307–316.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Network and Distributed System Security Symposium, NDSS 2007*.

W3C. 1998. Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1. (1998).

W3C. 2000. Document Object Model (DOM) Level 2 Core Specification. http://www.w3.org/TR/DOM-Level-2-Core. (2000).

W3C. 2004. Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/DOM-Level-3-Core. (2004).

W3C. 2012. Content Security Policy. http://www.w3.org/TR/CSP/. (2012).

W3C. 2014a. Cascading Style Sheets. http://www.w3.org/Style/CSS/. (2014).

W3C. 2014b. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors. (2014).

W3C. 2014c. HTML5: a vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/. (2014).

W3C. 2015a. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/. (2015).

W3C. 2015b. Mixed content. http://www.w3.org/TR/2015/CR-mixed-content-20151008/. (2015).

Joel Weinberger, Adam Barth, and Dawn Song. 2011. Towards Client-side HTML Security Policies. In *6th USENIX Workshop on Hot Topics in Security, HotSec 2011*.

Michael Weissbacher, Tobias Lauinger, and William K. Robertson. 2014. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2014*. 212–233.

Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium, USENIX 2006*. 121–136.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*. 237–249.

Michal Zalewski. 2011. Postcards From the Post-XSS World. (2011). http://lcamtuf.coredump.cx/postxss/.

Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. 2015. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the 24th USENIX Security Symposium, USENIX 2015*. 707–721.

Yuchen Zhou and David Evans. 2010. Why Aren't HTTP-only Cookies More Widely Deployed?. In *Web 2.0 Security and Privacy Workshop, W2SP 2010*.