Università
Ca'Foscari
Venezia

PH.D PROGRAMMEE

IN COMPUTER SCIENCE

CYCLE XXX

PH.D THESIS

Analysis and Prevention of Security Threats in Web and Cryptographic Applications

SSD: INF/01

COORDINATOR OF THE PH.D PROGRAMME
Prof. Riccardo FOCARDI

SUPERVISOR
Prof. Riccardo FOCARDI

CANDIDATE
Marco SQUARCINA
Matricola 814359

Abstract

In recent years we have faced a multitude of security flaws posing a serious threat to the whole society, ranging from individuals to national critical infrastructures. For this reason, it is of crucial importance to effectively enforce security on real systems, by identifying flaws and putting in place novel security mechanisms and techniques. Along this path, we provide practical contributions on Web security and cryptographic APIs.

We first review the field of Web session security by surveying the most common attacks against web sessions. Existing security solutions are evaluated along four different axes: protection, usability, compatibility and ease of deployment. We also identify a few guidelines that can be helpful for the development of innovative solutions approaching web security in a more systematic and comprehensive way. Additionally, we propose a new browser-side security enforcement technique for Web protocols. The core idea is to extend the browser with a monitor which, given the protocol specification, enforces the required confidentiality and integrity properties, as well as the intended protocol flow.

For what concerns the security of cryptographic APIs, we investigate an effective method to monitor existing cryptographic systems in order to detect, and possibly prevent, the leakage of sensitive cryptographic keys. Key security is stated formally and it is proved that the method is sound, complete and efficient under the assumption that a key fingerprint is given for each sensitive key. We also provide a thoughtful analysis of Java keystores, storage facilities to manage and securely store keys in Java applications. We devise a precise threat model and distill a set of security properties. We report on unpublished attacks and weaknesses in implementations that do not adhere to state-of-the-art cryptographic standards and discuss the fixes on popular Java libraries released after our responsible disclosure.

Acknowledgements

First of all I would like to express my deepest gratitude to my supervisor prof. Riccardo Focardi for the trust placed in me in these years and for his friendly guidance. I would also like to thank prof. Alessandro Armando and prof. Frank Piessens who accepted to devote their time to reading and reviewing this work.

I am grateful to a number of people I had the pleasure to collaborate with during the past few years, in particular dr. Stefano Calzavara, dr. Graham Steel, prof. Flaminia Luccio, prof. Matteo Maffei, Mauro Tempesta, Francesco Palmarini and Clara Schneidwind. I thank Mauro and Francesco again as part of the c00kies@venice hacking team, along with Claudio Bozzato, Francesco Cagnin, Marco Gasparini, Andrea Possemato, Francesco Benvenuto, Lorenzo Veronese and Andrea Baesso. They always supported me when needed and put their trust in me, thanks a lot. Together we reached some outstanding achievements and I am so proud of all of them.

Thanks to all the persons who believe in me (most of the time more than I do!), my family, prof. Camil Demetrescu and Emilio Coppa from the CyberChallenge.IT program, my extremely committed students of the Security course at Ca' Foscari and all the friends who are thousands of kilometers away, still close. Finally, I would like to thank my wife Paola who has always been ready to follow me in past and future journeys.

So long, and thanks for all the fish.

Contents

Dedicated to Paola

Preface

The work presented in this thesis is based on some research papers written during my Ph.D. studies in Computer Science at Università Ca' Foscari, Venezia from September 2014 to December 2017.

Chapter 1 is a joint work with Stefano Calzavara, Riccardo Focardi and Mauro Tempesta published in April 2017 in the *ACM Computing Surveys* (*CSUR*) journal [44]. Chapter 2 is based on an unpublished paper made in collaboration with Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Mauro Tempesta and Clara Schneidwind.

Chapter 3 is the outcome of a joint work with my supervisor Riccardo Focardi that I presented in August 2017 at the *30th IEEE Computer Security Foundations Symposium* (*CSF 2017*) [72]. Chapter 4 is the result of research program started during my visit at Cryptosense in Paris. The collaboration with Riccardo Focardi, Francesco Palmarini, Graham Steel and Mauro Tempesta lead to a paper that will be presented at the *Network and Distributed System Security Symposium* (*NDSS 2018*).

Introduction

In recent years we have faced a multitude of security flaws posing a serious threat to the whole society, ranging from individuals to national critical infrastructures. In this scenario, the Web plays a pivotal role since it 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 paired with a endless quest for usability and backward compatibility makes it very hard to effectively enforce security. As a consequence, the large majority of defensive mechanisms only address very specific problems because developers and researchers have been cautious at not breaking the functionalities of existing websites. Due to this situation, it is extremely difficult to understand the benefits and drawbacks of each single proposal without a full picture of the literature.

In this thesis we provide a systematization of knowledge in the context of the security of Web sessions by assessing a large class of common attacks and existing defenses. We evaluate each security solutions and mechanisms with respect to the security guarantees it provides, its impact on both *compatibility* and *usability*, as well as its *ease of deployment*. We also synthesize some guidelines that we believe can be helpful for the development of innovative solutions approaching web security in a more systematic and comprehensive way.

Furthermore, we propose an innovative approach to strengthen the security guarantees of Web protocols in general, without focusing on a single vulnerability. The key idea is to extend browsers with a security monitor that is able to enforce the compliance of browser behaviours with respect to an ideal Web protocol specification. We carefully designed the security monitor to interact gracefully with existing websites, so that the website functionality is preserved unless it critically deviates from the intended protocol specification.

Another fundamental technology for IT security is Cryptography. Even if there are well established standards for cryptographic operations, cryptography is complex and variegate and typically requires to combine in non trivial ways different algorithms and mechanisms. Moreover, cryptography is intrinsically related to the secure management of cryptographic keys which need to be protected and securely stored by applications. Leaking cryptographic keys, in fact, vanishes any advantage of cryptography, allowing attackers to break message confidentiality and integrity, to authenticate as legitimate users or impersonate legitimate services.

In this thesis we address the problem of leaking cryptographic keys in two different contexts. First, we report on the problem of run-time detection of cryptographic API attacks aimed at leaking sensitive keys. We propose an effective method to monitor existing cryptographic systems in order to detect and prevent the leakage of sensitive cryptographic keys.

Key security is stated formally and it is proved that the method is sound, complete and efficient. We also present a proof-of-concept tool for PKCS#11 that is able to detect, on a significant fragment of the API, all key management attacks from the literature. Secondly, we provide a thoughtful analysis of Java keystores, storage facilities to manage and securely store keys in Java applications. We devise a precise threat model and distill a set of security properties. We report on unpublished attacks and weaknesses in implementations that do not adhere to state-of-the-art cryptographic standards and discuss the fixes on popular Java libraries released after our responsible disclosure.

Structure of the Thesis

Due to the diversity of the contributions discussed, the thesis is divided into two distinct parts to gather together the works in the fields of Web security and cryptographic API, respectively. The following is an overview of the contents of each chapter.

- Chapter 1 surveys the most common attacks and security mechanisms on web sessions. Existing security solutions which prevent or mitigate the different attacks are evaluated along four different axes: protection, usability, compatibility and ease of deployment. Additionally, several defensive solutions which aim at providing robust safeguards against multiple attacks are assessed. Five guidelines for the design of security solutions are then distilled from the reviewed existing mechanisms;

- Chapter 2 introduces a novel browser-side security enforcement technique for web protocols. We first devise the challenges for the security of web protocols and then we illustrate the core idea of our approach on OAuth 2.0, the de-facto standard for cross-site authorization in the Web. The analysis is based both on a review of well-known attacks reported in the literature and an extensive experimental evaluation in the wild;

- Chapter 3 proposes an effective method to monitor existing cryptographic systems in order to detect, and possibly prevent, the leakage of sensitive cryptographic keys. Key security is formally defined and proofs that the method is sound, complete and efficient are provided. We discuss practical implementations and report on a proof-of-concept log analysis tool for PKCS#11 we developed;

- Chapter 4 examines Java keystores, the standard storage facilities to securely store cryptographic keys in Java applications. We consider seven keystore implementations from Oracle JDK and Bouncy Castle, a widespread cryptographic library. We describe, in detail, how the various keystores enforce confidentiality and integrity of the stored keys through password-based cryptography and we report on several attacks on implementations that do not adhere to state-of-the-art cryptographic standards.

Summary of Contributions

For what concerns the field of Web security, we contribute in Chapter 1 by performing a systematic overview of a large class of common attacks targeting Web sessions and the corresponding security solutions. We also believe that the five guidelines can be helpful for the development of innovative solutions approaching Web security in a more systematic and comprehensive way. In Chapter 2 we identify the fundamental challenges for the security of web protocols in terms of enforcing *confidentiality* and *integrity* of message components, as well as the intended *protocol flow*. We discuss concrete examples of their security import in the context of OAuth 2.0. We propose the Web Protocol Security Enforcer (WPSE for short), a browser-side security monitor designed to tackle the challenges we identified. We formalize the behaviour of WPSE in terms of a finite state automaton and we develop a prototype implementation of the security monitor as a Google Chrome extension, which we make publicly available. We rigorously analyse the design of WPSE against OAuth 2.0 by discussing how the security monitor prevents a number of attacks previously reported in the literature [160, 13, 68]. We prove that browser-side security monitoring of web protocols is both useful and feasible by experimentally assessing the effectiveness of WPSE against 90 websites using OAuth 2.0 to implement single sign-on at major identity providers.

In the context of cryptographic APIs and applications, Chapter 3 contributes by modeling the problem of run-time detection of cryptographic API attacks, also in a distributed setting. We provide a sound and complete characterization of attacks based on the monitoring of a subset of API calls and we we prove that the problem of finding attacks cannot be decided without a key fingerprinting abstract mechanism. We show that key fingerprinting enables a sound, complete and efficient run-time analysis. We discuss practical implementations and we develop a proof-of-concept log analysis tool for PKCS#11, the RSA standard interface for cryptographic tokens [139, 144], that is able to detect all the key-management attacks reported in [60, 71]. In Chapter 4 we define a general threat model for password-protected keystores and we distill a set of significant security properties and consequent rules that any secure keystore should adhere to. We perform a thoughtful analysis of seven keystores and report undocumented details about their cryptographic implementations. We show critical unpublished attacks and weaknesses in the analyzed keystores. We empirically estimate the speed-up due to bad cryptographic implementations with respect to the most resistant keystore and to NIST recommendations.

Part I

Web Security

Chapter 1

Surviving the Web: A Journey into Web Session Security

1.1 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 chapter 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 [134] 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 classification 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.

1.1.1 Scope of the Work

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.

1.1.2 Structure of the Chapter

Section 1.2 provides some background on the main building blocks of the Web. Section 1.3 presents the attacks. Section 1.4 classifies attack-specific solutions with respect to their security guarantees, their level of usability, compatibility and ease of deployment. Section 1.5 carries out a similar analysis for defenses against multiple attacks. Section 1.6 presents five guidelines for future web security solutions. Section 1.7 concludes.

1.2 Background

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

1.2.1 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) [177] or a comparable markup language (e.g., XHTML) defines the structure of the page and the elements it includes, while *Cascading Style Sheets* (CSS) [170] are used to add style information to web pages (e.g., fonts, colors, position of elements).

JavaScript [65] 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) [174, 175, 176], 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).

1.2.2 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, the host `www.google.com` belongs to the domain `google.com` which is a sub-domain of the top-level domain `com`.

1.2.3 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 her access to her

personal page. HTTP *cookies* are the most widespread mechanism employed on the Web to maintain state information about the requesting clients [18]. 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 1.2.4).

1.2.4 Security Cornerstones and Subtleties

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 [141] 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.

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 [178].

Same-Origin Policy

The *same-origin policy* (SOP) [124] 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. 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 [19]. 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.

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 [18].

1.3 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 proof of being part of the session established upon password-based authentication. This common authentication scheme is depicted in Figure 1.1.

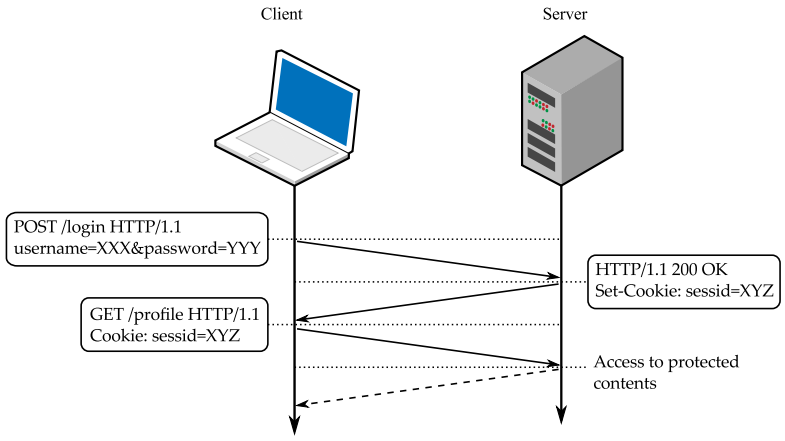


FIGURE 1.1: Cookie-based User Authentication.

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.

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

1.3.1 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.

1.3.2 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 [36]. This means in particular that the attacker can set (domain) cookies for the target website [18]. 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.

1.3.3 Web Attacks

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 [190, 87].

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.

Cross-Site Request Forgery (CSRF)

A CSRF is an instance of the "confused deputy" problem [84] 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 1.2.4) 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 [173] or mechanisms like JSON with Padding (JSONP) [89] 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 [14].

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 [20]. 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.

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 1.3.3).

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.

1.3.4 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 [18].

A notable example of network attack is *SSL stripping* [118], 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 can 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).

1.4 Protecting Web Sessions

1.4.1 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 1.1 provides the intuition underlying these scores.

	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

TABLE 1.1: Evaluation Criteria

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 [98, 82, 155, 59].

1.4.2 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.

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 [194].

SessionShield and Zan

SessionShield [128] 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 [161], 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.

Request Filtering Approaches

Noxes is one of the first developed client-side defenses against XSS attacks [105]. 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. Inspired by Noxes, Vogt *et al.* introduce a modified version of Firefox [168] 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 [128]. The modified Firefox in [168] 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.

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 [133]. Conversely, solutions aimed at providing wider coverage against general data leakage attacks never gained popularity, mainly due to their impact on the user experience.

1.4.3 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.

Client-side Filtering

XSS filters like IE XSS Filter [143] and WebKit XSSAuditor [21] 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 [117] applies an even stricter policy, since it directly prevents 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 [126, 96].

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.

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 [185] which improves an earlier solution named CSSE [138]. 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 [112]. 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.

XSS-Guard

The idea of server-side source-to-source program transformation is also employed in XSS-Guard [30], 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.

BEEP

Browser-Enforced Embedded Policies (BEEP) [95] 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 [12]. 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.

Blueprint

Blueprint [112] 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). This 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 [182]. 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.

Noncespaces

Along the same line of research, Noncespaces [80] 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 [12]. 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.

DSI

In parallel with the development of Noncespaces, Nadji *et al.* proposed a similar solution based on the concept of document structure integrity (DSI) [125]. 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.

Content Security Policy

The aforementioned proposals share the idea of defining a client-side security policy [182]. The same principle is embraced by the Content Security Policy (CSP) [171], 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 [172], 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 [182]. As of now, the deployment of CSP is not particularly significant or effective [184, 41]. 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.

Critical Evaluation

Content injection is one of the most widespread threats to the security of web sessions [134]. 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 [95]. More research is needed to provide more general defenses against a complex problem like content injection.

1.4.4 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.

Purely Client-side Solutions

Several browser extensions and client-side proxies have been proposed to counter CSRF attacks, including RequestRodeo [97], CsFire [148, 147] and BEAP [116]. 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 [147], 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 [58].

Allowed Referrer Lists (ARLs)

ARLs have been proposed as a client/server solution against CSRF attacks [58]. Roughly, an ARL is just a whitelist that specifies which origins are entitled to send authenticated requests 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.

Tokenization

Tokenization is a popular server-side countermeasure against CSRF attacks [20]. 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-dependent and may not be powerful enough for sophisticated web applications developed using many different languages [58].

NoForge

NoForge [100] 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 [100]. 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.

Origin Checking

Origin checking is a popular alternative to tokenization [20]. 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

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

was limited to POST requests [20] and current web browser implementations still do not ensure that the header is always populated [19]. 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 [58].

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 implemented 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.

1.4.5 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.

Serene

The Serene browser extension offers automatic protection against session fixation attacks [149]. 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 [43].

Origin Cookies

Origin cookies have been proposed to fix some known integrity issues affecting cookies [36]. 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 1.2.4). 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.

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 [99]. 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 [99]. 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.

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 [99]. 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 [192].

1.4.6 Network Attacks

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 performances 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 [90].

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 [67]. 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.

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 [127]. 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.

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 [88]. 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 (see Section 1.2.4) is 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 [154] 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.

HTTPS Everywhere

This extension for Firefox, Chrome and Opera [66] 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.

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 [90] in which a custom cookie was used in place of an HTTP header to enable the protection mechanism.

Summary

We summarize in Table 1.2 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 †.

1.5 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.

Origin-Bound Certificates

Origin-Bound Certificates (OBC) [62] have been proposed as an extension to the TLS protocol that binds authentication tokens to trusted encrypted channels. The idea is to generate, on

	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

TABLE 1.2: Analysis of Proposed Defenses

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 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.

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 [76] 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 [86], COWL [158] and an extension of Chromium with information flow control [22], 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 [61]. 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 execution 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 [76] or run CSRF attacks based on JavaScript [104]. 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 [76]. 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 [158];

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 [76]. Reaping the biggest benefits out of information flow control, however, necessarily requires some efforts by web developers.

Security Policies for JavaScript

Besides information flow control, in the literature there are several frameworks for enforcing general security policies on untrusted JavaScript code [119, 189, 113, 137, 165]. We just provide a brief overview on them here and we refer the interested reader to a recent survey by Bielova [28] 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 [119] security policies are expressed in terms of JavaScript code, while the framework in [189] 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 [119]. However, we believe more research is needed to draw a fair conclusion on how difficult it is to deploy these mechanisms in practice.

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 [79]. 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 application 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.

Escudo

Escudo [92] 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.

CookiExt

CookiExt [38] 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 non-interference 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 the 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.

SessInt

SessInt [39] 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.

Same Origin Mutual Approval

SOMA [130] 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.

App Isolation

App Isolation [47] 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.

Summary

We summarize our observations about the described solutions in Table 1.3. 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.

		Attacks				*Evaluation*		
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*

TABLE 1.3: Defenses Against Multiple Attacks

1.6 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.

1.6.1 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 [162]. 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 1.4.4), CSP (Section 1.4.3) and SOMA (Section 1.5) 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 1.4.5) and SessInt (Section 1.5) 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.

1.6.2 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 1.5) and several frameworks for enforcing arbitrary security policies on untrusted JavaScript code (Section 1.5). Popular examples of solutions which are not secure by design include the usage of secret tokens against CSRF attacks (Section 1.4.4): indeed, not every token generation scheme is robust [20] and ensuring the confidentiality of the tokens may be hard, even though this is crucial for the effectiveness of the solution.

1.6.3 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 1.4.4) and CookiExt (Section 1.5). Server-side or hybrid client/server solutions are often harder to adopt, for different reasons: some proposals like Escudo (Section 1.5) are too fine-grained and thus require a huge configuration effort, while others like FlowFox (Section 1.5) 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 1.5) and HSTS (Section 1.4.6). Origin checking is often straightforward to implement as a server-side defense against CSRF attacks (Section 1.4.4).

1.6.4 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 1.4.4) and SOMA (Section 1.5) are declarative in nature, while the tokenization (Section 1.4.4) is not declarative at all, since it is a low-level solution and it may be hard to adopt on legacy web applications.

1.6.5 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 [32, 5, 69, 39]. 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 [76] or session integrity [39], 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 1.5), SessInt (Section 1.5), FlowFox (Section 1.5) and CsFire (Section 1.4.4).

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 [142, 129, 43].

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 1.4.4), 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 [182]. We observe that it is relatively easy to come up with hybrid solutions which are compliant with the first four guidelines: SOMA (Section 1.5), HSTS (Section 1.4.6) and ARLs (Section 1.4.4) 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 [37].

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 1.5), HSTS (Section 1.4.6) and ARLs (Section 1.4.4) 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.

1.7 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.

Chapter 2

WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

2.1 Introduction

Web protocols are security protocols deployed on top of HTTP and HTTPS, most notably to implement authentication and authorization at remote servers. Popular examples of web protocols include OAuth 2.0[1], OpenID Connect[2] and Shibboleth[3], which are routinely used by millions of users to access security-sensitive functionalities on their personal accounts.

Unfortunately, designing and implementing web protocols is a particular error-prone task even for security experts, as witnessed by the large number of vulnerabilities recently presented in the literature [160, 13, 14, 193, 108, 109, 186, 179]. The main reason for this is that web protocols involve communication with a web browser, which does not strictly follow the protocol specification, but reacts asynchronously to any input it receives, producing messages which may have an import on protocol security. Reactiveness is particularly dangerous because the browser is agnostic to the web protocol semantics: it does not know when a protocol starts, nor when it ends, and is unaware of the order in which messages should be processed as well as the confidentiality and integrity guarantees desired for a protocol run. For example, in the context of OAuth 2.0, Bansal *et al.* [13] discussed *token redirection attacks* enabled by the presence of open redirectors, while Fett *et al.* [68] presented *state leak attacks* enabled by the communication of the Referer header; these attacks are not apparent from the protocol specification alone, but come from the subtleties of the browser behaviour.

Major service providers try to aid software developers to correctly integrate web protocols in their websites by means of JavaScript APIs; however, web developers are not forced to use them, can still use them incorrectly [180], and the APIs themselves do not necessarily implement the best security practices [160]. This unfortunate situation led to the proliferation of attacks against web protocols even at popular services.

In this chapter, we propose a fundamental paradigm shift to strengthen the security guarantees of web protocols. The key idea we put forward is to extend browsers with a security monitor which is able to enforce the compliance of browser behaviours with respect to an ideal web protocol specification. This approach brings two main practical benefits:

1. protocol specifications can be written and verified once, possibly as a community effort, and then uniformly enforced at a number of different websites by the browser;

2. the security import of incorrect server-side web protocol implementations is significantly mitigated, since the browser is aware of the intended protocol flow and any deviation from it is detected at runtime.

Remarkably, though changing the behaviour of web browsers is always delicate for backward compatibility, the security monitor we propose is carefully designed to interact gracefully with existing websites, so that the website functionality is preserved unless it critically deviates from the intended protocol specification. Moreover, a large set of the monitor functionalities can be implemented as a browser extension and deployed without requiring

[1] https://oauth.net/2/
[2] http://openid.net/connect/
[3] https://shibboleth.net/

changes to web browsers, thereby offering immediate protection to Internet users and promising a significant practical impact.

2.1.1 Contributions

In this chapter, we make the following contributions:

1. we identify three fundamental challenges for the security of web protocols, that is, enforcing *confidentiality* and *integrity* of message components, as well as the intended *protocol flow*. We discuss concrete examples of their security import in the context of the popular authorization protocol OAuth 2.0;

2. we propose the Web Protocol Security Enforcer (WPSE for short), a browser-side security monitor designed to tackle the challenges we identified. We formalize the behaviour of WPSE in terms of a finite state automaton and we develop a prototype implementation of the security monitor as a Google Chrome extension, which we make publicly available;

3. we rigorously analyse the design of WPSE against OAuth 2.0 by discussing how the security monitor prevents a number of attacks previously reported in the literature [160, 13, 68];

4. we experimentally assess the effectiveness of WPSE by testing it against 90 websites using OAuth 2.0 to implement single sign-on at major identity providers. In our analysis, we identified security flaws in 55 websites (61.1%), including new critical vulnerabilities caused by tracking libraries such as Facebook Pixel and Google AdSense, all of which fixable by WPSE and we discovered that WPSE works flawlessly on 83 websites (92.2%), with the remaining ones deviating from the protocol specification (and at least one of them introducing a critical vulnerability because of that), which proves that the browser-side security monitoring of web protocols is both useful and feasible.

2.2 Security Challenges in Web Protocols

Web protocols come with various security challenges which can often be attributed to the presence of the web browser that acts as a non-standard protocol participant. In the following, we discuss the different security challenges for web protocols using the OAuth 2.0 authorization protocol as case study.

2.2.1 Background: OAuth 2.0

We will illustrate our approach on OAuth 2.0, the de-facto standard web authorization protocol, which we briefly overview below. OAuth 2.0 [83] is a web protocol that enables resource owners to grant third parties controlled access to resources hosted at remote servers. In practice, the OAuth 2.0 protocol is also used for authenticating the resource owner to third parties

by giving them access to the resource owner's identity stored at an identity provider. This functionality is also known as single sign-on (SSO). Using standard terminology, we refer to the third-party application as *relying party* or *RP* for short, and to the website storing the resources (including the identity) as *identity provider* or *IdP* for short[4].

OAuth 2.0 comes with four different flows (also called *grant types*): *authorization code* mode, *implicit* mode, *resource owner password credentials* mode and *client side credentials* mode. In the following, we will focus on the authorization code mode and the implicit mode, as they are the commonly used by websites in practice.

The high-level structure of OAuth 2.0 under these two flows is similar:

1. the user U sends *RP* a request which requires to access a remote resource at *IdP*;
2. *RP* sends *IdP* an authorization request, asking a number of permissions;
3. U authenticates at *IdP* and grants the requested permissions (if she did not already);
4. *IdP* provides an authorization credential to *RP*;
5. *RP* uses the authorization credential to obtain access to the user's resource at the *IdP*.

We now summarize how the two flows work. For a more detailed protocol description, we refer the reader to [68].

Authorization Code Mode

The authorization code mode is intended for *RP*s whose main functionality is carried out at the server-side. The high-level protocol flow is depicted in Figure 2.1 and described in the following. For the sake of readability we introduce a simplified version of the protocol omitting implementation details. We give a more detailed version of parts of the protocol in Section 2.4.

① U sends a request to *RP* for accessing a remote resource. The request specifies the *IdP* that holds the resource. In the case of SSO, this step determines which *IdP* should be used.

② *RP* redirects U to the login endpoint of *IdP*. This request contains the *RP*'s identity at *IdP*, the URI that *IdP* should redirect to after successful login and an optional state parameter for CSRF protection that should be bound to U's state;

③ *IdP* answers to the authorization request with a login form and the user provides her credentials for *IdP*;

④ *IdP* redirects to the URI of *RP* specified at step ②, including the previously received state parameter and an authorization code;

[4]The OAuth 2.0 specification distinguishes between *resource servers* and *authorization servers* instead of considering one identity provider that stores the user's identity as well as its resources [83], but it is common practice to unify resource and authorization servers as one party [68], [160], [109].

⑤ *RP* makes a request to *IdP* with the authorization code, including its identity, the redirect URI and optionally a shared secret with the *IdP*;

⑥ *IdP* answers with an access token to *RP*;

⑦ *RP* makes a request for the user's resource to *IdP*, including the access token;

⑧ *IdP* answers *RP* with the user's resource at *IdP*.

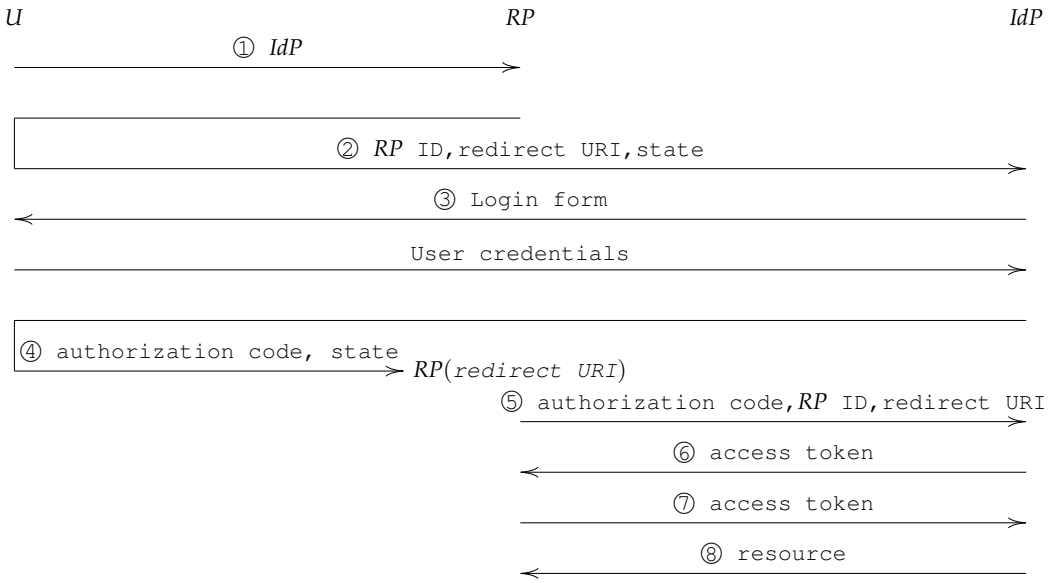


FIGURE 2.1: OAuth 2.0 (authorization code mode)

Implicit Mode

The implicit mode is intended for browser-side *RP* applications. Instead of first granting an authorization code to *RP*, the access token is directly provided by *IdP* in a fashion that allows the browser-side *RP* application to access it.

Steps ①-③ and ⑦-⑧ are the same as for the authorization code mode. Steps ④-⑥ are replaced by the following:

④ *IdP* redirects to the URI of *RP* specified at step ②, including the previously received state parameter and an access token in the fragment identifier;

⑤ *RP* answers with a script which extracts the state parameter and the access token from the fragment;

⑥ the script sends a request to *RP* including the extracted information.

2.2.2 Challenge #1: Protocol Flow

Protocols are specified in terms of a number of sequential message exchanges which honest participants are expected to follow, but the browser is not forced to comply with the intended protocol flow.

Example in OAuth 2.0

The use of the state parameter is recommended to prevent attacks leveraging this idiosyncrasy. If *RP* does not provide the state parameter in its authorization request to *IdP* at step ②, then an attack is possible that logs in an honest user with an attacker's account (in the case of OAuth 2.0 being used for SSO). This attack is also known as *session swapping* [160].

We give a short overview on this attack in the authorization code mode: an attacker A initiates social login at *RP* with an identity provider *IdP* and performs steps ① to ③ of the protocol. In step ④, A learns an authorization code and creates an exploit page that embeds the authorization code in a way that it is sent to *RP* upon viewing the page. When an honest user browses this page, the *RP* completes the login, establishing a session with the attacker in the user's browser. A similar attack works in the implicit mode.

2.2.3 Challenge #2: Secrecy of Messages

The security of protocols typically relies on the confidentiality of cryptographic keys and credentials, but the browser is not aware of which data must be kept secret for protocol security.

Example in OAuth 2.0

In the case of OAuth 2.0, the secrecy of the authorization credentials (namely the authorization codes and the access tokens) is crucial for meeting the protocol's security requirements, as the knowledge of those credentials allows an attacker to access the user's resources. The secrecy of the state parameter is also important to ensure session integrity.

An example of an unintended leakage of these secrets is the so called *state leak attack* described in [68]. In this attack, an honest user agent U is expected to complete steps ① to ④ of the authorization code mode with *IdP* at *RP*. If the page loaded at the redirect URI in step ③ contains a link to a malicious website, clicking this link causes the state parameter and the authorization code (that are part of the currently loaded page's URL) to be leaked in the Referer header of the outgoing request.

The learned authorization code can potentially be used to obtain a valid access token for U at *IdP*. In addition, the leaked state parameter enables the session swapping attack previously described.

2.2.4 Challenge #3: Integrity of Messages

Protocol participants are typically expected to perform a number of runtime checks to prove the integrity of the messages they receive and ensure the integrity of the messages they send, but the browser cannot perform these checks unless they are explicitly carried out in a JavaScript implementation of the web protocol.

Example in OAuth 2.0

An example attack that exploits this weakness is the *naïve RP session integrity* attack presented in [68]. This attack relies on the assumption that *RP* supports SSO with different identity providers and, in order to distinguish which identity provider was used, relies on different redirect URIs for each of them. In this case, an attacker can confuse the *RP* about which identity provider is currently used and login the attacker at an honest user's browser. To this end, the attacker starts a social login session at *RP* with an honest identity provider *HIdP* to obtain an authorization code for her account at *HIdP* – steps ① to ④. If an honest user starts a social login session at *RP* with a malicious identity provider *AIdP*, then in step ④, *AIdP* is expected to redirect the user to *AIdP*s redirect URI at *RP*. If instead *AIdP* redirects to the redirect URI of *HIdP*, but includes the authorization code from the attacker session, then *RP* mistakenly assumes that the user intended to login with *HIdP*. Consequently, *RP* completes the social login – steps ⑤ to ⑧ – with *HIdP* providing the attacker's authorization code to it which will make *HIdP* authenticate the user as the attacker.

2.3 WPSE: Design and Implementation

The Web Protocol Enforcer (WPSE) is the first browser-side security monitor designed to address the security challenges of web protocols. We discuss its design and semantics and present a proof-of-concept implementation.

2.3.1 Protocol Specification

For the sake of simplicity, we illustrate WPSE on a toy example involving a simple login protocol. More realistic and interesting examples are discussed in the next sections.

Protocol Flow

The login protocol involves three participants: the browser B, an authentication service A hosted at `accounts.example.com` and a website E hosted at `www.example.com`. The protocol works as follows: first, B sends a POST request to A submitting the user's credentials in the form of username and password. If the credentials are valid, the corresponding response from A sets a `Secure` cookie `s` containing a session identifier and redirects B to the successful login page at E. This page sets a cookie `p` containing the user's preferences for the website and redirects B to the user's profile at E, which is dynamically generated based on the session information bound to `s`. When the credentials are wrong, instead, no cookie is set and B is redirected to an error page at A, from where the user can try to login again. A protocol run which successfully authenticates the user is shown in Figure 2.2.

WPSE describes web protocols in terms of the HTTP(S) exchanges observed by the web browser, following the so-called *browser relayed messages* methodology first introduced by Wang *et al.* [179]. The specification of the protocol flow defines the syntactic structure and the expected order of the HTTP(S) messages, supporting the possibility of selecting different execution branches when a particular protocol message is sent or received by the browser.

FIGURE 2.2: Basic login protocol (successful login) involving a website (E), a browser (B) and an authentication service (A)

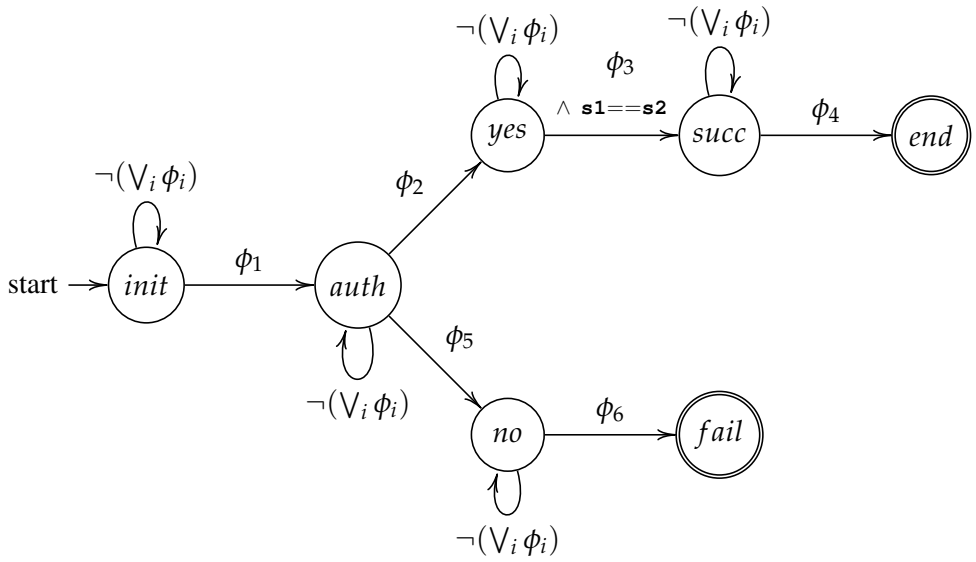
WPSE takes as input a protocol specification that allows to express this structure. For the sake of presentation, we give the protocol specification in terms of a finite state automaton depicted in Figure 2.3. Intuitively, each state in the automaton represents one stage of the protocol execution in the browser. By receiving an HTTP(S) response or submitting an HTTP(S) request that is expected by the protocol in the current stage, the automaton steps to the next state until reaching a final state (states *auth* and *fail* in the example) indicating that the protocol run was completed. The branching of the protocol is expressed by adding non-determinism to the automaton as done in state *auth* that allows for receiving a success and a failure response to the user's login trial.

In order to specify the protocol messages expected in each stage, the outgoing edges of the automaton's corresponding state are labeled by *message patterns* describing the shape of valid messages at this point of execution. Message patterns can either be of the form $\texttt{Req}(m)$ indicating that the browser sends out a request with a message of shape m or of the form $\texttt{Resp}(m)$ indicating that a response of shape m is received. In both cases, m is a regular expression defining the shape of the message. Note that we represent HTTP(S) requests simplified as $e\langle a\rangle$ where e is the recipient endpoint and a is the comma separated list of arguments. Accordingly, we use the notation $e(h)$ for HTTP(S) responses with e being the sender's endpoint and h being the list of headers. The message patterns should be considered as *guards* of the transition that evaluate to true for such requests and responses that match the pattern including the regular expression. Guards can be composed using standard logical connectives.

In addition to providing the outgoing edges for progressing in the protocol execution, each state of the automaton also allows for pausing the protocol execution in the presence of requests and responses that are unrelated to the protocol. Messages are considered unrelated to the protocol, if they are not of the shape of any valid message in the protocol execution. In the automaton, this is expressed by introducing a self-loop for each state with the disjunction of all negated patterns for valid protocol messages as a guard. We consider the negation of a pattern just to invert the pattern's semantics.

Note that the automata constructed for protocols expose a tree structure. In particular, this structure induces a partial ordering on automaton states. In the following, we additionally assume a fixed total ordering on automaton states that respects the partial ordering induced

by the tree structure.



ϕ_1 = *Req*(https://*A*/login.php<user:*,pwd:*>)
ϕ_2 = *Resp*(https://*A*/login.php(Set-Cookie:s=**s1{***S***}:=(**.***),
Location:https://*E*/success.php))
ϕ_3 = *Req*(https://*E*/success.php<Cookie: s=**s2:=(**.*?**)**>)
ϕ_4 = *Resp*(https://*E*/success.php(Set-Cookie:p=.*,
Location:https://*E*/profile.php))
ϕ_5 = *Resp*(https://*A*/login.php(Location:https://*A*/fail.php))
ϕ_6 = *Req*(https://*A*/fail.php<>)
S = https://www.example.com, https://accounts.example.com

FIGURE 2.3: Finite state automaton for the basic login protocol

Binders

In order to incorporate integrity and secrecy policies into the syntax of the automaton, we allow for binding parts of message patterns in the guards of the automaton to identifiers. To this end, the syntax **x:=(***exp***)** is used for binding identifier **x** to the (part of the) message matching the regular subexpression *exp*. This binding notation can be arbitrarily inserted inside the regular expressions of the message patterns. In order to clarify the scope of the binder, the subsequent subexpression *exp* is expected to be enclosed in a *capturing group*[5] (denoted by parentheses). In each state, all identifiers that were introduced in patterns of non self-loop edges on a path to a preceding state, can be considered to be bound to a value, where the notion of a preceding state is well defined by the partial order on the automaton states.

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

For instance, the binder in message pattern Φ_3 of the automaton in Figure 2.3 binds identifier **s2** to the value of the cookie `s` sent in the request to the successful login page at E in state *succ*. This binding remains in place for all successor states of *succ*. Notice that binders occurring in negated patterns are not considered. Otherwise, identifiers introduced in self-loops could not be ensured to be bound in a later state as the automaton does not enforce that a self-loop edge is taken.

Secrecy Policy

The secrecy policy defines which parts of the HTTP(S) responses included in the protocol specification must be kept confidential among a set of web origins. These message components must be isolated from browser accesses, e.g., computations performed by JavaScript, and only be attached to HTTP(S) requests sent to the web origins which are intended to learn them.

In the case of the basic login protocol, the session cookie `s` should only be disclosed to A and E to prevent session hijacking by an attacker learning the cookie value. We express secrecy policies in the automaton representation by extending the notation of binders by an optional *secrecy set*. The secrecy set specifies the origins that the bound value is allowed to be shared with. In Figure 2.3, the binder in pattern Φ_2 binds the value of cookie `s` (that is set by the response from A) to identifier, noted **s1{**S**}:=**, and defines the secrecy set S of the cookie value to only contain `https://www.example.com` and `https://accounts.example.com`. Note that previously introduced identifiers can occur in the definition of the secrecy set

Integrity Policy

The integrity policy defines runtime checks over the HTTP(S) messages. These checks allow for the comparison of incoming messages with the messages received during the preceding protocol execution. If any of the integrity check fails, the corresponding message is not processed and the protocol run is aborted.

In the case of the basic login protocol, we would like to ensure that the session cookie `s` received after submitting the authentication credentials is the same cookie which is sent to the successful login page. This is useful to prevent session hijacking attacks, where the attacker forces the user's browser into a malicious session.

To express integrity constraints in the automaton, we enrich the guards of the edges to include comparisons ranging over the identifiers introduced by preceding messages. The integrity constraint of the basic login protocol is consequently described by adding the comparison **s1** $==$ **s2** to the guard Φ_3. This additional guard requires that the value bound to **s1** in pattern Φ_2 equals the value that will be bound to **s2** in pattern Φ_3 when a request matching Φ_3 is received in state *yes*.

2.3.2 Security Enforcement

Given a protocol specification, WPSE generates a security monitor which ensures compliance with respect to the intended protocol flow, while enforcing the desired secrecy and integrity policies. We discuss next the details of the construction.

Protocol Flow

Given a protocol specification, WPSE generates a finite state automaton as previously discussed including one state for each HTTP(S) request and response in the protocol, plus one initial state. The set of the final states of the automaton includes all the states corresponding to the last HTTP(S) request or response of one of the protocol branches.

Due to the self loops at all but the final states, protocol-unrelated messages are always allowed by WPSE. This is important for website functionality, because the input/output behavior of browsers on realistic websites is complex and hard to fully determine when writing a protocol specification. Also, the same protocol may be run on different websites, which need to fetch very different resources as part of their protocol-unrelated functionalities, and we would like to ensure that the same protocol specification can be enforced uniformly on all these websites.

We achieve determinism on the finite state automaton by forcing the choice of the transition to the next automaton state according to the total ordering when more than one transition is enabled.

Monitor Semantics

If we assume that no secrecy or integrity policy is put in place, the semantics of the monitor is straightforward. In this case, the guards of all edges only consist of (potentially logically composed) message patterns. We say that an HTTP(S) request *satisfies* a pattern $\texttt{Req}\,(e\langle a\rangle)$ when it is directed at e and satisfies all the syntactic constraints of a; an analogous notion holds for HTTP(S) responses and patterns of the form $\texttt{Resp}\,(e(h))$. The monitor just listens for all the HTTP(S) requests and responses passing through the browser, trying to walk the automaton from the initial state to a final state by checking them against the patterns of the outgoing transitions of the current state. If an HTTP(S) request or response does not satisfy any of the patterns of the outgoing transitions of the current state, it is blocked and the monitor is reset to the initial state, so that the protocol run is aborted.

We complete the description of the monitor semantics by extending it to the security policy components. We first discuss the *secrecy policy*: confidential message components are stripped from HTTP(S) responses and substituted by random placeholders. If the monitor encounters an HTTP(S) response including a binder with a secrecy set, the value bound to the capturing box subsequent to the binder is stripped off. When the monitor detects an HTTP(S) request including an identifier, it replaces the latter with the corresponding original value, but only if the HTTP(S) request is directed to one of the web origins which is entitled to learn it (as specified in the secrecy set of the corresponding binder). A similar idea has been explored by Stock and Johns to strengthen the security of password managers [159]. Since the

substitution of confidential message components with placeholders changes the content of the messages, thus introducing potential deviations with respect to the labels of the constructed automaton, the monitor processes HTTP(S) responses before stripping confidential values and HTTP(S) requests after replacing the placeholders with the original values. This way, the input/output behavior exposed by the monitor is the same as the one of the ideal protocol specification.

As to the *integrity policy*, the desired integrity checks are directly encoded in the syntax of the guards of the automaton, so that a transition is only enabled when it satisfies the expected integrity checks. Since these checks may depend on previously received messages, like in the example of Figure 2.3, the monitor keeps track of the values bound to the identifiers introduced in the protocol specification when walking the automaton.

Multiple Protocols

There are a couple of delicate points to address when multiple protocol specifications $P_1, \ldots, P_n$ must be honored by WPSE at the same time:

1. if two different protocols P_i and P_j share messages with the same structure, there might be situations where the monitor does not know which of the two protocols is being run, yet a message may be allowed by P_i and disallowed by P_j or vice-versa;

2. if the monitor is trying to enforce a given protocol P_i, it must not allow for a message which may be part of another protocol P_j, otherwise it would be trivial to sidestep the security policy of P_i by first making the browser process the first message of P_j.

Both problems are solved by replacing the protocol specifications $P_1, \ldots, P_n$ with a single specification P with n branches, one for each P_i. Using this construction, any ambiguity on which protocol specification should be enforced is solved by the determinism of the finite state automaton. Moreover, the self loops of the automaton will only match the messages which are not part of any of the n protocol specifications, thereby preventing unintended protocol interleavings. Notice that the semantics of the security monitor depends on the order of the automaton's states, due to the way we enforce determinism: if P_i starts with a request to u including two parameters a and b, while P_j starts with a request to u including only the parameter a, then P_i' first (distinct) state should be positioned before P_j's first (distinct) state in the ordering. Otherwise, P_i would never be executed as the initial message for P_i will be matched by the pattern for P_j's initial message and as a consequence the monitor will always enforce P_j.

2.3.3 Implementation

We have developed a proof-of-concept implementation of WPSE as an extension for Google Chrome[6]. The automaton defining the protocol is for this purpose expressed as an XML specification.

[6]Available at `https://sites.google.com/site/wpseproject/`

The extension intercepts all the HTTP(S) messages that are sent or received by the browser using the webRequest API and processes them as discussed above. Our prototype implements a single security monitor which is shared among all the tabs in the browser. As a consequence, only one web protocol at a time can be run simultaneously by the open pages. Since web protocols typically require some user intervention to be initiated and they last only for a few HTTP(S) exchanges, we do not expect this choice to significantly affect the browsing experience of the typical user.

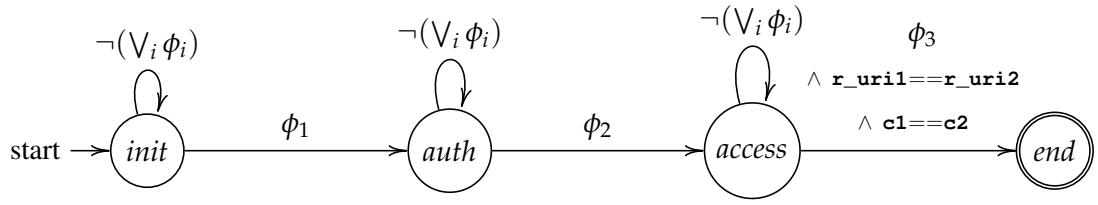
2.4 Fortifying OAuth 2.0 with WPSE

OAuth 2.0 is a popular real-world protocol whose security has been studied in depth [13, 68, 160] and is thus an excellent benchmark for WPSE. We show how the protocol and a possible security policy are encoded in our tool and we discuss which attacks from the literature are prevented.

2.4.1 Protocol Specification

Table 2.4 presents the automaton specification of one of the OAuth 2.0 implementations available at Google. Specifically, it models the authorization code mode of the protocol without the state parameter, which is recommended but optional at Google. The automaton models the message exchanges: the first request to Google starts the protocol run, by specifying that the authorization code mode is desired and by sending the redirect URI of the relying party. The corresponding response redirects the browser using a Location header. The last message in the specification is the request triggered by the redirect, which must include the authorization code provided by Google. The regular expression on the code parameter ensures that the security monitor is only activated when the value of the parameter is at least 40 characters long, which we observed to be true for Google. This makes the security enforcement more precise and it is helpful because the structure of the last message of the protocol needs to satisfy only weak syntactic constraints.

The binders introduced in the patterns of the protocol messages and responses are used to refer to relevant message components. By the appropriate embedding into the regular expressions, the binders identify the redirect URI (**`r_uri1`** for the one specified by the first request and **`r_uri2`** for the one finally used for redirection), the origin of the relying party (**`r_orig`**, extracted from the redirect URI) and the authorization code (**`c1`** for the code received in the response and **`c2`** for the one sent in the request to *IdP*) respectively. The secrecy policy requires that the authorization code is kept secret between Google and the relying party, while the integrity policy ensures that the last message of the protocol is directed at the redirect URI specified in the first message of the protocol. Despite its simplicity, this policy is strong enough to stop a number of dangerous attacks. In order to illustrate how the protocol is specified in the extension, in Table 2.1 we also provide the XML specification corresponding to the automaton.



ϕ_1 = *Req* (https://accounts.google.com/oauth2/(?:.*?/)?auth
<response_type:code,
redirect_uri:^**r_uri1:=(r_orig:=(**https?://.*?/**).*?)**(?:\?|$)>)
ϕ_2 = *Resp* (https://accounts.google.com/oauth2/(?:.*?/)?auth
(Location:[?&]code=**c1{**S**}:=(**.*?**)**(?:&|$))
ϕ_3 = *Req* (**r_uri2:=(***)<code:**c2:=(**[^\s]{40,}>**)**)
S = https://accounts.google.com, **r_orig**

FIGURE 2.4: Finite state automaton for the OAuth 2.0 protocol

2.4.2 Prevented Attacks

We reviewed attacks on OAuth 2.0 presented in the literature, analysing whether or not they are prevented by our extension. We focus on the attacks presented in [13], [68] and [160] and picked those that apply to the OAuth 2.0 flows presented in this work.

Table 2.2 gives an overview on the attacks from the literature that WPSE is able to prevent. We group them into three categories according to the type of violation of the browser-side protocol and security property specification that they expose.

Protocol Flow Deviations

This category covers attacks that involve the user's browser only in some parts of the protocol flow. Some attacks, e.g., some versions of CSRF and session swapping rely on completing a social login in the user's browser that was not initiated before. As this is a clear deviation from a protocol flow, WPSE stops these attacks right away. We explain the way WPSE works in these cases by taking the example of session swapping. As discussed in Section 2.2.2, in the session swapping attack, the attacker tricks the user into sending a request containing the attacker's authorization credential (e.g., the authorization code) to *RP* (step ④ of the protocol flow). As the state parameter is not present, *RP* cannot check that this request was not preceded by a corresponding social login request by the user. The monitor will directly stop this attack as its initial state expects the authorization request to *IdP* (step ② of the protocol) to be carried out by the browser, as specified in the guard Φ_1 of the automaton in Figure 2.4. As the attack's initial request principally matches the shape of a valid request of the OAuth 2.0 protocol (namely the request specified in pattern Φ_3), the automaton underlying the monitor does not provide an outgoing edge for the request initiated by the attacker. Consequently the monitor blocks the request and thereby stops the attack.

Secrecy Violations

This category covers attacks where sensitive information is unintentionally leaked to uneligible parties via browser requests. Sources of these attacks can either be open redirectors at *RP* endpoints or leakage via the Referer header or by redirection.

The attacks in this category (summarized in Table 2.2) have in common that requests with sensitive information are sent by the browser to parties that are not trusted to obtain this information. Either these parties are untrusted third parties that should not be involved in the protocol flow at all (as it is the case for the State Leak Attack and Unauthorized Login by Authentication Code Redirection) or protocol parties that are not trusted with a specific secret (as in the 307 Redirect Attack).

WPSE can prevent this class of attacks as the secrecy policy allows one to specify for each secret the origins it is allowed to be shared with. We illustrate how the monitor would prevent these attacks using the example of the State Leak Attack discussed in [68].

As described in section 2.2.3, in the State Leak Attack, the state parameter and the authorization code are leaked to the attacker in the Referer of a request to a malicious page which is initiated after the *IdP* redirection to the *RP* (step ④ of the protocol).

The secrecy policy for the authorization code is specified by giving the secrecy set for the binder of **`c1`** in pattern Φ_2. It requires the authorization code (**`c1`**) that is received in step ② of the OAuth 2.0 protocol only to be shared between Google and the *RP* (whose origin is bound to **`r_orig`**). The monitor consequently strips off the value of the authorization code before passing the request to the browser. When, after step ④ of the protocol, a request is sent out to the attacker, the monitor first checks whether to replace the placeholder of the authorization code (that is part of the outgoing request's Referer header) with the original value. As the request is directed to an attacker origin (which should be different from the ones specified in the secrecy set), the placeholder is not replaced. Still, the request (containing a random value instead of the state parameter) is sent out to the attacker because it does not match any of the requests defined in the protocol specification and is therefore irrelevant to protocol flow checks.

Integrity Violations

This category contains attacks that maintain the general protocol flow (from the browser's perspective), but involve server-side parties that are not those requested by the user. Examples for these attacks are given in Table 2.2. In all listed cases, *RP* confuses the *IdP* issued by the user for social login with another party. As the information that identifies the correct *IdP* (e.g., the redirect URI) goes through the browser, it is possible to prevent the browser actions that confuse the *RP* by performing browser-side integrity checks. One example is the enforcement of the usage of the correct redirect URI by the *IdP* that prevents the Naïve RP Session Integrity Attack presented in Section 2.2.4. We use this example to illustrate how WPSE prevents the attack by enforcing browser-side integrity checks.

The guard on the edge between the states *access* and *end* carries the integrity constraint that is violated by the attack. In step ② of the OAuth 2.0 protocol, the redirect URI is received

as parameter. This request corresponds to the one specified in pattern Φ_1 of the automation and consequently the redirect URI will be bound to the identifier **`r_uri1`**. In the case of the Naïve RP Session Integrity Attack, this redirect URI will be the one that *RP* associates with the malicious *AIdP*. As in the attack, in protocol step ④, the browser gets redirected to the redirect URI of the honest *HIdP* (edge between *access* and *end*), **`r_uri2`** is bound to the redirect URI for *HIdP*.

The monitor will stop the protocol execution when encountering this request to the wrong URI given that it does not pass the integrity check, despite of matching the shape of a valid request in the protocol flow. Consequently, the corresponding state in the automaton does not provide a matching outgoing edge and the monitor is reset.

2.4.3 Out-of-Scope Attacks

Even though WPSE is able to prevent a wide range of attacks, it needs to be emphasized that some attack classes are out of scope for browser-side security monitoring. Table 2.3 gives an overview of known attacks on OAuth 2.0 that cannot be stopped by WPSE.

Cross-Site Request Forgery

CSRF attacks can only be prevented if they cause a deviation from the protocol flow observable on the browser-side. CSRF attacks that e.g., exploit vulnerabilities at the *RP*s login page or are for other reasons able to forge the initial social login request, cannot be stopped as the browser-side protocol flow stays unchanged. An example for this is the Social Login CSRF through IdP Login CSRF attack [13]. In this attack, the attacker exploits a CSRF vulnerability on *IdP*s login form. When the user browses a malicious website, the attacker redirects the user to the *IdP*s login page with the attacker's credentials and silently logs in the attacker in the user's browser. When later on a social login at *RP* with *IdP* is performed, the user will be logged in as the attacker at *RP* as well. This attack can not be prevented by WPSE as from the browser's perspective, the protocol flow is not disrupted. From the browser side, it can not distinguished whether the initial login request was performed intentionally or not.

Network Attacks

Another class of attacks that can not be prevented are network attacks as the plain version of the IdP-Mix-Up attack [68]. In this case, the attack is caused by a network attacker that manipulates HTTP requests and responses. As this happens after the requests left the browser and before the responses enter the browser, no violation of any policy can be observed on the browser side. In the IdP Mix-Up attack, the user's SSO credential for an honest *IdP* is leaked to a malicious *IdP* by *RP*. The attack works in the authorization code mode and in the implicit mode and assumes the presence of a network attacker and that *RP* saves the user's choice about which *IdP* to use for login in a session after protocol step ①. Additionally, it is assumed that *RP* gives the same redirect URI to all *IdP*s it is registered with. The attack proceeds as follows: when $\mathcal{U}$ initiates social login at *RP* with *HIdP* (step ① of the OAuth 2.0 protocol), the outgoing request is manipulated to contain a malicious identity provider *AIdP*.

As a consequence, *RP* redirects to *AIdP*. The network attacker manipulates this response again to redirect to *HIdP* instead (step ②). The user registers with *HIdP* and finally gets redirected to *RP* with the state parameter and the authorization code in step ④. But as *RP* saved the user's intention to identify with *AIdP* after step ①, it requests *AIdP* for an access token in step ⑤ and thereby leaks the state parameter and authorization code to malicious *AIdP*. As these secrets are leaked by a request originating from *RP* no browser-side secrecy policy can prevent this leakage. In addition, on the browser side no protocol deviation can be observed as the requests are manipulated after passing the browser and the responses before entering the browser, respectively.

Server-Side Attacks

Also attacks caused by missing server-side checks or a poor quality of the involved credentials cannot be caught by WPSE. Exemplary for these issues, we present an instance of the Impersonation Attack described in [160]. This attack applies to the implicit mode of OAuth 2.0 and assumes that the *RP* uses public or guessable information retrieved from the *IdP*, such as the public id of the user on the *IdP*, to perform authentication. In addition, it is assumed that *RP* does not check whether the authorization request (step ② of the protocol) and the request containing the authentication credential (step ⑥) originated from the same browser. In this case, it is possible that the attacker uses her own browser to send a forged credential for the user to the *RP* and like this logs in as the user. Since this attack is cause by a flawed server-side logic and is carried out without the user's browser, there is no way for a browser-side monitor to prevent it.

2.5 Discussion

A number of points of the design and the implementation of WPSE are worth discussing more in detail.

2.5.1 Protocol Flow

WPSE provides a significant improvement in security over standard web browsers, but the protection it offers is not for free, because it requires the specification of both a protocol flow and a security policy. We think that it is possible to develop automated techniques to reconstruct the intended protocol flow from observable browser behaviours, while synthesizing the security policy looks more difficult. Manually finding the best security policy for a protocol may require significant expertise, but even simple policies can be useful to prevent a number of dangerous attacks, as demonstrated in Section 2.4.

The specification style of the protocol flow supported by WPSE is simple, because it only allows sequential composition of messages and branching. As a result, our finite state automata are significantly simpler than the request graphs proposed by Guha *et al.* [79] to represent legitimate browser behaviors (from the server perspective). For instance, our finite state automata do not include loops and interleaving of messages, because it seems that these

features are not extensively used in web protocols. Like standard security protocols, web protocols are typically specified in terms of a fixed number of sequential messages, which are appropriately supported by the specification language we chose.

2.5.2 Secrecy Enforcement

The implementation of the secrecy policies of WPSE is secure, but restrictive. Since WPSE substitutes confidential values with random placeholders, only the latter are exposed to browser-side scripts. Shielding secret values from script accesses is crucial to prevent confidentiality breaches via untrusted scripts or XSS, but it might also break the website functionality if a trusted script needs to compute over a secret value exchanged in the protocol. The current design of WPSE only supports a limited use of secrets by browser-side scripts, i.e., scripts can only forward secrets unchanged to the web origins entitled to learn them. We have shown that this is enough to support existing protocols like OAuth 2.0, but other protocols may require more flexibility.

Dynamic information flow control deals with the problem of letting programs compute over secret values while avoiding confidentiality breaches and it has been applied in the context of web browsers [76, 85, 27, 140, 22]. We believe that dynamic information flow control can be fruitfully combined with WPSE to support more flexible secrecy policies. This integration can also be useful to provide confidentiality guarantees for values which are generated at the browser-side and sent in HTTP(S) requests, rather than received in HTTP(S) responses. We leave the study of the integration of dynamic information flow control into WPSE to future work.

2.5.3 Extension APIs

The current prototype of WPSE suffers from some limitations due to the Google Chrome extension APIs. In particular, the body of HTTP messages cannot be modified by extensions, hence the secrecy policy cannot be implemented correctly when secret values are embedded in the page contents or the corresponding placeholders are sent as POST parameters. Currently, we only protect secret values contained in the HTTP headers of a response (e.g., cookies or parameters included in the URL of a Location header) and we only substitute the corresponding placeholders when they are communicated via HTTP headers or as URL parameters. Of course, this is not a limitation of our general approach, but rather one of the extension APIs, which can be solved by implementing the security monitor directly in the browser. Despite these limitations, we were able to test the current prototype of WPSE on a number of real-world websites with very promising results, as described in Section 2.6.

2.6 Experimental Evaluation

Our technique aims to provide a practical benefit to the security of websites in a real-world setting. To this end, we experimentally assessed the effectiveness of WPSE by testing it against websites using OAuth 2.0 to implement SSO at high-profile *IdP*s.

2.6.1 Experimental Setup

We developed a crawler to automatically identify existing OAuth 2.0 implementations in the wild. Our analysis is not meant to provide a comprehensive coverage of the deployment of OAuth 2.0 on the Web, but just to identify a few popular identity providers and their relying parties to carry out a first experimental evaluation of WPSE.

We first started from a comprehensive list of OAuth 2.0 identity providers[7] and we collected for each of them the list of the HTTP(S) endpoints used in their implementation of the protocol. Inspired by [3], our crawler looks for login pages on websites to find syntactic occurrences of these endpoints: after accessing a homepage, the crawler finds a list of (at most) 10 links which may point to a login page, using a simple heuristic. It also retrieves, using the Bing search engine, the 5 most popular pages of the website. For all these pages, the crawler checks for the presence of the OAuth 2.0 endpoints in the HTML code and in the 5 topmost scripts included by them. By running our crawler on the Alexa 100k top websites, we found that Facebook (1,666 websites), Google (1,071 websites) and VK (403 websites) are the most popular identity providers in the wild.

We then developed a faithful XML representation of the OAuth 2.0 implementations available at the selected identity providers. There is obviously a large overlap between these specifications, though slight differences are present in practice. For instance, the use of the `response_type` parameter is mandatory at Google, but can be omitted at Facebook and VK to default to the authorization code mode. For the sake of simplicity, we decided to model the most common use case, i.e., we assume that the user has an ongoing session with the identity provider and that authorization to access the user's resources on the provider has been previously granted to the relying party. For each identity provider we devised a specification that supports the OAuth 2.0 authorization code and implicit modes, with and without the optional state parameter, leading to 4 possible execution paths. Finally, we created a dataset of 90 websites by sampling 30 relying parties for each identity provider, covering both the authorization code mode and the implicit mode of OAuth 2.0. In the following we report on the results of testing our extension against these websites from both a security and a compatibility point of view.

2.6.2 Security Analysis

We devised an automated technique to check whether WPSE can stop real-world attacks. Since we did not want to attack the websites, we focused on two classes of vulnerabilities which are easy to detect just by navigating the websites when using WPSE. The first class of vulnerabilities enables confidentiality violations: it is found when one of the placeholders generated by WPSE to enforce its secrecy policies is sent to an unintended web origin. The second class of vulnerabilities, instead, is related to the use of the state parameter: if the state parameter is unused or set to a predictable static value, then a session swapping attack becomes possible (see Section 2.2.2). We can detect these cases by checking which protocol specification is enforced by WPSE and by making the state parameter secret, so that all the

[7] `https://en.wikipedia.org/wiki/List_of_OAuth_providers`

values bound to it are collected when they are substituted by the placeholders to enforce the secrecy policy.

We observed that our extension prevented the leakage of sensitive data on 4 different relying parties. Interestingly, we found that the security violation exposed by the tool are in all cases due to the presence of tracking or advertisements libraries such as Facebook Pixel[8], Google AdSense,[9] Heap,[10] and others.

This has been observed, for instance, on ticktick.com, a website offering collaborative task management tools. The leakage is enabled on two conditions: i) the website allows its users to perform a login via Google using the implicit mode; and ii) the Facebook tracking library is embedded in the page which serves as redirect URI. Under these settings, right after step ④ of the protocol (see Section 2.2.1), the tracking library sends a request to `https://www.facebook.com/tr/` with the full URL of the current page, which includes the access token issued by Google. We argue that this is a critical vulnerability, given that leaking the access token to an unauthorized party allows unintended access to sensitive data owned by the users of the affected website. We promptly reported the issue to Facebook, which acknowledged it, but did not comment on eventual fixes. We are in the process of disclosing the vulnerabilities with the other affected websites and major tracking library vendors.

For what concerns the second class of vulnerabilities, 55 out of 90 websites have been found affected by the lack or misuse of the state parameter. More in detail, we manually classified 41 websites that do not support it, while the remaining 14 websites miss the security benefit of the state parameter by using a predictable or constant string as a value. We claim that such disheartening situation is mainly caused by the identity providers not setting this important parameter as mandatory. In fact, the state parameter is listed as recommended by Google and optional by VK. On the other hand, Facebook sets the state parameter as mandatory, but our experiments showed that it fails to fulfill the requirement in practice. Additionally, it would be advisable to clearly point out in the OAuth 2.0 documentation of each provider the security implications of the parameter. For instance, according to the Google documentation[11], the state parameter can be used "for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery": we believe that this description is too vague and opens the door to misunderstandings.

2.6.3 Compatibility Analysis

To detect whether WPSE negatively affects the web browser functionality, we performed a basic navigation session on the websites in our dataset. This interaction includes an access to their homepage, the identification of the SSO page, the execution of the OAuth 2.0 protocol, and a brief navigation of the private area of the website. We were able to navigate 81 websites

[8] `https://www.facebook.com/business/a/facebook-pixel`
[9] `https://www.google.com/adsense`
[10] `https://heapanalytics.com/`
[11] `https://developers.google.com/identity/protocols/OAuth2WebServer`

flawlessly, but we also found 9 websites where we did not manage to successfully complete the protocol run.

In all the cases, the reason for the compatibility issues was the same, i.e., the presence of an HTTP(S) request with a parameter called `code` after the execution of the protocol run. This message has the same syntactic structure as the last request sent as part of the authorization code mode of OAuth 2.0 and is detected as an attack when our security monitor moves back to its initial state at the end of the protocol run, because the message is indistinguishable from a session swapping attempt (see Section 2.2.2). We manually investigated all these cases: 2 of them were related to the use of the Gigya social login provider, which offers a unified access interface to many identity providers including Facebook and Google; the other 7, instead, were due to a second exchange of the authorization code at the end of the protocol run. We were able to solve the first issue by writing an XML specification for Gigya (limited to Facebook and Google), while the other cases openly deviate from the OAuth 2.0 specification, where the authorization code is only supposed to be sent to the redirect URI and delivered to the relying party from there. These custom practices are hard to explain and to support and, unsurprisingly, may introduce security flaws. In fact, one of the websites deviating from the OAuth 2.0 specification suffers from a serious security issue, because the authorization code is first communicated to the website over HTTP before being sent over HTTPS, thus becoming exposed to network attackers. We responsibly disclosed this security issue to the website owners.

In the end, all the compatibility issues we found boil down to the fact that a web protocol message has an extremely weak syntactic structure, which may end up matching a custom message used by websites as part of their functionality. We think that most of these issues can be robustly solved by using more explicit message formats for standardized web protocols like OAuth 2.0: explicitness is indeed a prudent engineering practice for traditional security protocols [2]. Having structured message formats could be extremely helpful for a precise browser-side fortification of web protocols which minimizes compatibility issues.

2.7 Related Work

Web Protocol Analysis

Though traditional protocol verification tools cannot readily produce reliable security proofs for web protocols because they do not fully cover browser behaviours, they have been successfully applied to attack finding. For instance, Armando *et al.* analyzed both the SAML protocol and a variant of the protocol implemented by Google using the SATMC model-checker [11]. Their analysis exposed an attack against the authentication goals of the Google implementation. Follow-up work by the same group used a more accurate model to find an authentication flaw also in the original SAML specification [10].

The first paper to highlight the differences between web protocols and traditional cryptographic protocols in terms of security analysis is due to Gross *et al.* [77]. The paper presented a model of web browsers based on a formalism reminiscent of input/output automata, which

supports a simple graphical representation similar to UML state diagrams. The paper applied the browser model to analyze the security of password-based authentication, a central ingredient of most browser-based protocols. The model was later used to formally assess the security of the WSFPI protocol [78]. More recently, other formal models have been proposed to analyze web protocols. Akhawe *et al.* used the Alloy framework to develop a core model of the web infrastructure, geared towards attack finding [5]. The paper studied the security of the WebAuth authentication protocol among other case studies, finding a login CSRF attack against it. The WebSpi library for ProVerif by Bansal *et al.* has been successfully applied to find attacks against existing web protocols, including OAuth 2.0 [13] and cloud storage protocols [14]. Fett, Kuesters and Schmitz developed the most comprehensive model of the web infrastructure available to date and fruitfully applied it to the analysis of a number of web protocols, including BrowserID [69], SPRESSO [70] and OAuth 2.0 [68].

It is worth noticing that protocol analysis techniques are useful to verify the security of protocol specifications, but they assume websites are correctly implemented and do not depart from the protocol specification. This sets this research line apart from run-time enforcement techniques like the one introduced in the present chapter.

Web Protocol Implementations

Many security researchers performed empirical security assessments of existing web protocol implementations, finding dangerous attacks in the wild. Protocols which deserved attention by the research community include SAML [156], OAuth 2.0 [160, 109] and OpenID Connect [108].

Given the prevalence and significance of attacks against real-world websites, many automated tools for vulnerability finding in web protocol implementations have also been proposed by security researchers [179, 193, 186, 115]. None of these works, however, presented a technique to protect users confronted with vulnerable websites in their browsers.

Security Automata

The use of finite state automata for security enforcement is certainly not new. The pioneering work in the area is due to Schneider [152], which first introduced a formalization of security automata and studied their expressive power in terms of a class of enforceable policies. Security automata can only stop a program execution when a policy violation is detected; later work by Ligatti *et al.* extended the class of security automata to also include edit automata, which can suppress and insert individual program actions [111]. Edit automata have been applied to the web security setting by Yu *et al.*, who used them to express security policies for JavaScript code [189]. The focus of their paper, however, is not on web protocols and is only limited to JavaScript, because input/output operations which are not JavaScript-initiated are not exposed to their security monitor.

Guha *et al.* also used finite state automata to encode web security policies [79]. Their approach is based on three steps: first, they apply a static analysis for JavaScript to construct the control flow graph of an Ajax application to protect and then they use it to synthesize

a request graph, which summarizes the expected input/output behavior of the application. Finally, they use the request graph to instruct a server-side proxy, which performs a dynamic monitoring of browser requests to prevent observable violations to the expected control flow. The security enforcement can thus be seen as the computation of a finite state automaton built from the request graph. Their technique, however, is only limited to Ajax applications and operates at the server side, rather than at the browser side.

Browser-Side Defenses

The present work positions itself in the popular research line of extending web browsers with stronger security policies. To the best of our knowledge, this is the first work which explicitly focuses on web protocols, but a number of other proposals on browser-side security are worth mentioning. Enforcing information flow policies in web browsers is a hot topic nowadays and a few fairly sophisticated proposals have been published as of now [76, 85, 27, 140, 22]. Information flow control can be used to provide confidentiality and integrity guarantees for browser-controlled data, but it cannot be directly used to detect deviations from expected web protocol executions, which instead are naturally captured by security automata. Combining our approach with browser-based information flow control can improve its practicality, because a more precise information flow tracking would certainly help a more permissive security enforcement.

A number of browser changes and extensions have been proposed to improve web session security, both from the industry and the academia. Widely deployed industrial proposals include Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS). Notable proposals from the academia include Allowed Referrer Lists [58], SessionShield [128], Zan [161], CSFire [147], Serene [149], CookiExt [38], SessInt [39] and Michrome [42]. Moreover, JavaScript security policies are a very popular research line in their own right: we refer to the survey by Bielova [28] for a good overview of existing techniques. None of these works, however, tackles web protocols.

2.8 Conclusion

We presented WPSE, the first browser-side security monitor designed to address the security challenges of web protocols, and we showed that the security policies enforceable by WPSE are expressive enough to prevent a number of real-world attacks against the OAuth 2.0 authorization protocol. Our analysis is based both on a review of well-known attacks reported in the literature and an extensive experimental analysis in the wild, which exposed several undocumented security vulnerabilities fixable by WPSE in existing OAuth 2.0 implementations. We also showed that WPSE works flawlessly on the large majority of the websites we tested, so we conclude that the browser-side security monitoring of web protocols is both useful for security and feasible in practice.

There are many avenues for future work. First, we would like to enrich our analysis of the OAuth 2.0 protocol to identify sufficient conditions to establish a formal security proof

for the protocol even if best security practices are not properly implemented by web developers, but the protocol is run in a web browser extended with WPSE; this would integrate our present investigation into the rich research line on web protocol analysis [5, 13, 68]. Then, we observe that our current assessment of WPSE in the wild only covers two specific classes of vulnerabilities, which can be discovered just by navigating the tested websites: extending the analysis to cover active attacks (in an ethical manner) is an interesting direction to get a better picture of the current state of the OAuth 2.0 deployment. Finally, we plan to identify automated techniques to synthesize protocol specifications for WPSE starting from observable browser behaviours in order to make it easier to adopt our security monitor in additional real-world web protocols like OpenID Connect and Shibboleth.

```
<Specification name="google-explicit-nostate">
  <Protocol>
    <Request method="GET" desc="req_init">
      <Endpoint>
        <Regexp>
          https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth
        </Regexp>
      </Endpoint>
      <Parameter name="response_type"> code </Parameter>
      <Parameter name="redirect_uri" id="req_init_redirect_uri" />
    </Request>
    <Response desc="resp_init">
      <Endpoint>
        <Regexp>
          https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth
        </Regexp>
      </Endpoint>
      <Header name="Location" id="resp_init_location" />
    </Response>
    <Request method="GET" desc="req_code">
      <Endpoint id="req_code_endpoint"/>
      <Parameter name="code" id="req_code_code">
        <Regexp> [^\s]{40,} </Regexp>
      </Parameter>
    </Request>
  </Protocol>
  <Identifiers>
    <Definition id="redirect_uri">
      <Source> ${req_init_redirect_uri} </Source>
      <Regexp> ^(https?://.*?)(?:\?|$) </Regexp>
    </Definition>
    <Definition id="client_origin">
      <Source> ${req_init_redirect_uri} </Source>
      <Regexp> ^(https?://.*?/).* </Regexp>
    </Definition>
    <Definition id="resp_init_location_code">
      <Source> ${resp_init_location} </Source>
      <Regexp> [?&amp;]code=(.*?)(?:&amp;|$) </Regexp>
    </Definition>
  </Identifiers>
  <Policy>
    <Secrecy>
      <!-- the auth code contained in the Location header must be kept
           secret -->
      <Target> ${resp_init_location_code} </Target>
      <Origin> ${client_origin} </Origin>
      <Origin> https://accounts.google.com/ </Origin>
    </Secrecy>
    <Integrity>
      <!-- the last message must be sent to the redirect URI initially
           specified -->
      <Target> ${req_code_endpoint} </Target>
      <Matches> ${redirect_uri} </Matches>
    </Integrity>
  </Policy>
</Specification>
```

TABLE 2.1: XML specification for the Google implementation of OAuth 2.0 (no state parameter)

Protocol Flow Deviations	Secrecy Violations	Integrity Violations
• Session swapping [160], Social Login CSRF on stateless clients [13]* • Force-Login Attack [160], Automation Login Attack [13] • Unauthorized Login by Authentication Code Redirection [13] • Resource Theft by Access Token Redirection [13] • IdP Mix-Up Attack (web attacker version) [68] • Cross Social Network Request Forgery (web attacker version) [13]	• 307 Redirect Attack [68] • State Leak Attack [68] • Unauthorized Login by Authentication Code Redirection [13] • Resource Theft by Access Token Redirection [13]	• Naïve Session Integrity Attack [68]

* the attack can be prevented only if the authorization request is forged

TABLE 2.2: Attacks from the literature prevented by WPSE

CSRF Attacks	Network Attacks	Others
• Force-Login Attack [160], Automation Login Attack [13] • Social Login CSRF through IdP Login CSRF [13]	• IdP Mix-Up Attack (standard) [68] • Cross Social Network Request Forgery (standard) [13]	• Impersonation [160]

TABLE 2.3: Attacks from the literature not prevented by WPSE

Part II

Cryptographic APIs

Chapter 3

Run-time Attack Detection in Cryptographic APIs

3.1 Introduction

Cryptography is one of the dominant technologies to provide security in various settings and cryptographic hardware and services are becoming more and more pervasive in everyday applications. The interfaces to cryptographic devices and services are implemented as *Security APIs* that allow untrusted code to access resources in a secure way. These APIs provide various functionalities such as: the creation or deletion of keys; the encryption, decryption, signing and verification of data under some key; the import and export of *sensitive* keys, i.e., keys that should never be revealed as plaintext outside smartcards and hardware security modules [139, 144].

Cryptographic APIs have been found vulnerable to many attacks that compromise sensitive cryptographic keys (see, e.g., [6, 33, 35, 48]). Some attacks are related to the key wrapping operation: for example, attacks on the IBM CCA interface are due to the improper way of binding a cryptographic key to its usage rules through the XOR function [33], and attacks on security tokens can be mounted by assigning particular sets of attributes to the keys, and by performing particular sequences of (legal) API calls [35]. Other attacks, e.g., the ones on PIN processing APIs, are based on formats used for message encryption [50], or on the lack of integrity of user data [46].

In the literature we find many proposals for preventing or mitigating such attacks, but they typically require to modify the API or to configure it in a way that might break existing applications (see, e.g. [35, 45, 53, 60, 73, 107]). This makes it hard to adopt such proposals, especially because security APIs are often used in highly sensitive settings, such as financial and critical infrastructures, where systems are rarely modified and legacy applications are very common. Notice that, in these settings, the leakage of a cryptographic key might have very serious consequences such as decrypting confidential data, breaking integrity or forging digitally signed documents and transactions. It is thus of ultimate importance to introduce mechanisms that can detect or prevent attacks and that can be also deployed in practice.

In this chapter we explore a different approach. Instead of trying to fix the API or developing a new secure one, we propose an effective method that can be used to monitor existing systems in order to detect, and possibly prevent, the leakage of sensitive cryptographic keys. The method collects logs for various devices and is able to detect, offline, any leakage of sensitive keys. For example, by tracking keys we may discover that a sensitive key is being wrapped under an untrusted one, that might be known to the attacker; whenever a sensitive key is leaked in the clear, as in the so-called *wrap/decrypt* attack [49], we are able to identify the problem through a special *key fingerprinting* functionality, that allows for an efficient offline log analysis without affecting in any way the cryptographic application.

3.1.1 Challenges

Devising a run-time analysis of cryptographic APIs presents many challenges. First of all, it needs to track the usage of any sensitive key, without exposing its value. Cryptographic APIs store cryptographic keys securely and give access to them through handles so that it is not necessary to know key values to perform operations. Monitoring keys that are referred

through handles can be tricky. In particular, when a key is leaked it is not possible to discover this immediately, since the key value is not available.

The analysis must be very accurate: false positives might result in unnecessary key updates of all keys that are believed to be leaked, while false negatives would miss actual key leakages, with possible serious consequences. In this respect, it is of ultimate importance that any proposed monitoring method is supported by a formal proof of soundness and completeness with respect to a class of attacks.

The analysis should work on distributed executions across many devices or applications. Cryptographic keys are often shared among devices and services and API level attacks can thus be effectively run in a distributed fashion, with the aim of bypassing any local monitoring. Thus, it is important that the analysis is able to collect logs from various sources and check them consistently, in order to find attacks that might leak a key of one cryptographic service through another one, in a different physical location.

Finally, the analysis should be efficient and should be able to scale on fairly big logs. Ideally, the monitor should continuously collect distributed logs and perform the analysis in real-time. Once the analysis has been proved accurate and suitably tested, the monitor might run "in the middle" of the API calls, and could be able to spot attacks on the fly and prevent them, by blocking the call right before the key is leaked.

3.1.2 Contributions

We contribute to the state of the art in various respects: (i) we model the problem of run-time detection of cryptographic API attacks. Our model captures distributed attacks, i.e., attacks performed by executing API calls on different devices and services; (ii) we provide a sound and complete characterization of attacks based on the monitoring of a subset of API calls; (iii) we prove that the problem of finding attacks cannot be decided in general because, intuitively, it is not possible to distinguish sensitive keys from non-sensitive ones just by their values; (iv) we propose a key fingerprinting abstract mechanism and a run-time analysis that is sound, complete and efficient; key fingerprinting is only used for logging purposes in order to make the analysis feasible and accurate and it does not affect the cryptographic applications invoking the API; (v) we discuss practical implementations and we develop a proof-of-concept log analysis tool for PKCS#11, the RSA standard interface for cryptographic tokens [139]. The tool is able to detect, on a significant fragment of the API, all key-management attacks reported in [60, 71].

3.1.3 Related Work

The first paper that has applied general analysis tools to the analysis of security APIs is [188], but no formal statement of the security guarantees provided by the analysis was done. The first automated analysis of PKCS#11 with a formal statement of the underlying assumptions has been presented in [60]. In [35], the model of [60] has been generalized and provided with a reverse-engineering tool that automatically refines the model depending on the actual behavior of the device. When new attacks were found, they were tested directly on the device

to get rid of possible spurious attacks determined by the model abstraction. The automated tool of [35] has successfully found attacks that leak the value of sensitive keys on real devices. In [4, 45], type-based techniques have been used to statically analyze the security of cryptographic API specification. Computational security guarantees of cryptographic APIs have been studied in [106, 107, 151].

All of above works aim at analyzing a given API specification or configuration, looking for attack sequences or proving the absence of attacks. None of them perform a run-time analysis of API invocation sequences.

Caml Crush [24] is a PKCS#11 Filtering Proxy that can be configured to prevent dangerous PKCS#11 commands and mechanisms. Caml Crush performs a run-time analysis, but there are important differences with respect to our proposal: (i) Caml Crush modifies the API behavior by imposing restrictions that prevent attacks. For example it prevents keys to be assigned conflicting roles so that attacks such as Clulow's wrap/decrypt are prevented. Our approach is different: we do not impose any restriction on how keys are configured and used, and in fact we do not even consider key attributes, since our method is independent of the specific API. We just monitor the API calls that can be responsible of leaking a sensitive key. While Caml Crush can break applications that do not adhere to the imposed policy, even when no key leakage happens, our approach is more accurate and only stops the calls that are responsible of key leakages greatly reducing the number of false positives; (ii) Caml Crush does not track keys on multiple devices and is thus unable to prevent the distributed attacks we discuss in Section 3.3, unless the wrapping API is modified. More precisely, in Caml Crush it is possible to enable an ad hoc modification of the wrapping API that tracks key attributes but makes the API incompatible with the standard PKCS#11 one: keys wrapped under Caml Crush modified wrapping API cannot be unwrapped on standard devices; (iii) our method can work offline by simply analyzing logs, while Caml Crush requires an online component to actively monitor API calls. Together with (i), we believe that this is a fundamental feature that might facilitate the adoption of the method, since it would never interfere with (and possibly break) applications, a crucial requirement in critical settings (e.g., banks); (iv) Camel Crush is tailored to PKCS#11 while our approach is more general in principle. Indeed, the method we propose does not rely on any specific feature of PKCS#11 such as key attributes. It only requires the specification of which keys are sensitive (i.e., not accessible in the clear), a basic security property useful in any key management API (e.g., Microsoft CAPI and CNG, Java JCA).

The model presented in this work is based on the one in [60] but there are important differences: we remove from the model any detail that is specific of PKCS#11, in order to model generic cryptographic APIs. We define a notion of local and distributed secure execution that formalizes when a specific execution is secure with respect to a set of sensitive keys. In particular, our executions are not regulated by any key policy or attributes as in [60]. Finally, the focus of [60] is the discovery of sequences of API calls that might leak a key. Here, instead, we study how to detect attacks on given (distributed) sequences of API calls by log inspection, i.e., without knowledge of the actual key values.

3.1.4 Structure of the Chapter

In Section 3.2 we present the core formal model, which is based on [60]; Section 3.3 introduces our notion of secure distributed execution that we characterize in terms of the analysis of a subset of the API calls; Section 3.4 presents our method for the run-time analysis based on key fingerprinting and we prove that it has linear complexity with respect to the length of logs and the number of sensitive keys; in Section 3.5 we report on a prototype implementation in PKCS#11, where we implement key fingerprinting through standard API calls. We show that the tool can effectively detect and prevent known attacks on a significant fragment of PKCS#11 key management; Section 3.6 draws some concluding remarks.

3.2 Core Model

The core of our model is a variation of the one introduced by Delaune, Kremer and Steel (DKS) [60], in which anything specific to PKCS#11 has been removed, and with labels referring to API calls on the transitions. The latter will be required to formalize secure executions in Section 3.3.

The attacker is assumed to be able to call commands of the API in any order providing any known value. Data and keys are modeled as terms and the rules of the API and the abilities of an attacker are written as rules that, given some terms, produce new ones. Cryptography is modeled symbolically: the intruder is assumed not to be able to break cryptography by brute-force or cryptanalysis, i.e., (s)he can only read an encrypted message if (s)he knows the correct key, along the standard Dolev-Yao approach [64]. Since the attacker is at the API level, we do not distinguish between malicious or legitimate users.

3.2.1 Syntax

As in DKS, we assume a given *signature* Σ, i.e., a finite set of *function symbols*, with an arity function $ar : \Sigma \to \mathbb{N}$, a (possibly infinite) set of *names* $\mathcal{N}$ and a (possibly infinite) set of *variables* $\mathcal{X}$. Names represent keys, data values, nonces, etc. Function symbols model cryptographic primitives. We also denote with Σ_{api} the set of API function symbols and extend the arity function to this set in the expected way. The set of *plain terms* $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$ is defined by the following grammar

$$\begin{array}{rcll} \mathsf{t} & := & \mathsf{x} & \mathsf{x} \in \mathcal{X} \\ & | & \mathsf{n} & \mathsf{n} \in \mathcal{N} \\ & | & \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_j) & \mathsf{f} \in \Sigma \text{ and } ar(\mathsf{f}) = j \end{array}$$

The set $\mathcal{PT}(\Sigma, \mathcal{N}, \emptyset)$, also referred to as $\mathcal{PT}(\Sigma, \mathcal{N})$, is called the set of *ground terms*. We use $vars(\mathtt{t})$ and $names(\mathtt{t})$ for the set of variables and names that occur in the term $\mathtt{t}$ and extend the notations to set of terms.

We simplify the DKS model by removing the set of literals from each rule. As a result, user's capabilities are not restricted by the attributes assigned to key handles. Additionally, we make explicit the API function call used to fire a rule by including the function as a label.

$$
\begin{array}{rcl}
 & \xrightarrow[\text{new } n,k]{\mathsf{KeyGen}} & \mathsf{h}(n,\ k) \\
 & \xrightarrow[\text{new } n,s]{\mathsf{KeyPairGen}} & \mathsf{h}(n,\ \mathsf{priv}(s)),\ \mathsf{pub}(s) \\
\mathsf{h}(x_1,\ y_1),\ \mathsf{h}(x_2,\ y_2) & \xrightarrow{\mathsf{Wrap}_{ss}} & \mathsf{senc}(y_2,\ y_1) \\
\mathsf{h}(x_1,\ \mathsf{priv}(z)),\ \mathsf{h}(x_2,\ y_2) & \xrightarrow{\mathsf{Wrap}_{sa}} & \mathsf{aenc}(y_2,\ \mathsf{pub}(z)) \\
\mathsf{h}(x_1,\ y_1),\ \mathsf{h}(x_2,\ \mathsf{priv}(z)) & \xrightarrow{\mathsf{Wrap}_{as}} & \mathsf{senc}(\mathsf{priv}(z),\ y_1) \\
\mathsf{h}(x,\ y_2),\mathsf{senc}(y_1,\ y_2) & \xrightarrow[\text{new } n_1]{\mathsf{Unwrap}_{ss}} & \mathsf{h}(n_1,\ y_1) \\
\mathsf{h}(x,\ \mathsf{priv}(z)),\mathsf{aenc}(y_1,\ \mathsf{pub}(z)) & \xrightarrow[\text{new } n_1]{\mathsf{Unwrap}_{sa}} & \mathsf{h}(n_1,\ y_1) \\
\mathsf{h}(x,\ y_2),\mathsf{senc}(\mathsf{priv}(z),\ y_2) & \xrightarrow[\text{new } n_1]{\mathsf{Unwrap}_{as}} & \mathsf{h}(n_1,\ \mathsf{priv}(z)) \\
\mathsf{h}(x_1,\ y_1),y_2 & \xrightarrow{\mathsf{Encrypt}_s} & \mathsf{senc}(y_2,\ y_1) \\
\mathsf{h}(x_1,\ y_1),\mathsf{senc}(y_2,\ y_1) & \xrightarrow{\mathsf{Decrypt}_s} & y_2 \\
\mathsf{h}(x_1,\ \mathsf{priv}(z)),y_1 & \xrightarrow{\mathsf{Encrypt}_a} & \mathsf{aenc}(y_1,\ \mathsf{pub}(z)) \\
\mathsf{h}(x_1,\ \mathsf{priv}(z)),\mathsf{aenc}(y_2,\ \mathsf{pub}(z)) & \xrightarrow{\mathsf{Decrypt}_a} & y_2
\end{array}
$$

TABLE 3.1: API rules

The description of the system is given as a finite set of rules $\mathcal{R}$ of the form

$$T \xrightarrow[\text{new } \tilde{n}]{f} T'$$

where $T, T' \subseteq \mathcal{PT}$ are sets of plain terms, $\tilde{n} \subseteq \mathcal{N}$ is a set of names and $f \in \Sigma_{api}$ is an API function symbol. When $\tilde{n} = \emptyset$, we omit new $\tilde{n}$ from the rule.

Intuitively, the rule can be fired when all the terms in T are in the user knowledge and the API function f is invoked. The effect of the rule is that the user knowledge is augmented with terms in T'. The new $\tilde{n}$ means that all the names in $\tilde{n}$ need to be replaced by fresh names in T'. This models nonce or key generation: if the rule is executed several times, the effects are different as different names will be used each time.

We consider the signature $\Sigma = \{\mathsf{senc}, \mathsf{aenc}, \mathsf{pub}, \mathsf{priv}, \mathsf{h}\}$, as in DKS. The function symbols senc and aenc of arity 2 represent symmetric and asymmetric encryption, whereas pub and priv of arity 1 are constructors to obtain public and private keys, respectively. The symbol h allows to model key handles.

Example 1 (Ciphertext, Keys and Handles)**.** *We show a few examples of ciphertext, key and handle terms. Term* $\mathsf{senc}(k_2,\ k_1)$ *represents key* k_2 *encrypted under symmetric key* k_1*. Private key* $\mathsf{priv}(s)$ *and public key* $\mathsf{pub}(s)$ *represent a keypair generated from a common seed* s*. Finally* $\mathsf{h}(n,\ k)$ *is a handle referring to key* k*. Nonce* n *is used to make it possible to have*

multiple handles, e.g., $\mathsf{h}(\mathsf{n}, \mathsf{k})$ *and* $\mathsf{h}(\mathsf{n}', \mathsf{k})$*, for the same key* k*. Notice that from a handle it is not possible to recover the value of the key.*

We consider the following set of API functions

$$\begin{aligned}\Sigma_{api} = \{&\mathsf{KeyGen}, \mathsf{KeyPairGen},\\ &\mathsf{Wrap}_{\mathsf{ss}}, \mathsf{Wrap}_{\mathsf{sa}}, \mathsf{Wrap}_{\mathsf{as}},\\ &\mathsf{Unwrap}_{\mathsf{ss}}, \mathsf{Unwrap}_{\mathsf{sa}}, \mathsf{Unwrap}_{\mathsf{as}},\\ &\mathsf{Encrypt}_{\mathsf{s}}, \mathsf{Encrypt}_{\mathsf{a}},\\ &\mathsf{Decrypt}_{\mathsf{s}}, \mathsf{Decrypt}_{\mathsf{a}}\}\end{aligned}$$

Intuitively, KeyGen, KeyPairGen are nullary functions for generating symmetric keys and key pairs, respectively; $\mathsf{Wrap}_{\mathsf{ss}}, \mathsf{Wrap}_{\mathsf{sa}}, \mathsf{Wrap}_{\mathsf{as}}$ and $\mathsf{Unwrap}_{\mathsf{ss}}, \mathsf{Unwrap}_{\mathsf{sa}}, \mathsf{Unwrap}_{\mathsf{as}}$ are used to respectively wrap and unwrap keys under other keys. We model the common cases of wrapping a symmetric key under a symmetric and an asymmetric one plus the case of wrapping an asymmetric key under a symmetric one. Wrap operations take two key handles as arguments while unwrap operations take a handle and a ciphertext and generate a new handle in the device, pointing to the unwrapped key. Finally, $\mathsf{Encrypt}_{\mathsf{s}}, \mathsf{Encrypt}_{\mathsf{a}}$ and $\mathsf{Decrypt}_{\mathsf{s}}, \mathsf{Decrypt}_{\mathsf{a}}$ perform symmetric and asymmetric encryption and decryption. They respectively take as arguments a plaintext/ciphertext and the handle of the encryption/decryption key. The set of rules of our model are listed in Table 3.1.

Example 2 (Wrap API)**.** *As an example, consider the rule*

$$\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1),\ \mathsf{h}(\mathsf{x}_2, \mathsf{y}_2) \xrightarrow{\mathsf{Wrap}_{\mathsf{ss}}} \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$$

used to wrap a symmetric key with another symmetric key. We have that $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ *and* $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$ *are handles for keys* y_1 *and* y_2*, respectively, while* $\mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$ *is the symmetric encryption of* y_2 *under* y_1*. The rule states that the key* y_2 *can be wrapped, i.e., encrypted, with* y_1 *when the API function* $\mathsf{Wrap}_{\mathsf{ss}}$ *is fired and both handles for keys* $\mathsf{y}_1, \mathsf{y}_2$ *are known. The wrapped key is then added to the set of known terms.*

3.2.2 Semantics

We enrich the semantics of DKS with labels, as they will be required for the run-time analysis. The semantics is thus defined in terms of a labeled transition system $(Q, A, \twoheadrightarrow, q_0)$. Q defines the set of possible states, where each state $q \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ is the set of ground terms in the user's knowledge. A is the set of actions such as

$$\begin{aligned}A = \{\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \mid\ &\mathsf{f} \in \Sigma_{api},\ n = ar(\mathsf{f}),\\ &\forall\, i \in [1, n] : \mathsf{t}_i \in \mathcal{PT}(\Sigma, \mathcal{N})\}\end{aligned}$$

Given a rule $a \in A$, we write $args(a) \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ for the set $\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$ of arguments of a. The initial state $q_0 \in Q$ represents the initial knowledge of the user. The transition

$$\begin{aligned} y_1, y_2 &\xrightarrow{DY} \mathsf{senc}(y_2, y_1) \\ \mathsf{senc}(y_2, y_1), y_1 &\xrightarrow{DY} y_2 \\ \mathsf{senc}(y_2, y_i) &\xrightarrow{DY} y_2 \end{aligned}$$

$$\begin{aligned} y_1, y_2 &\xrightarrow{DY} \mathsf{aenc}(y_2, y_1) \\ \mathsf{aenc}(y_2, \mathsf{pub}(z)), \mathsf{priv}(z) &\xrightarrow{DY} y_2 \\ \mathsf{aenc}(y_2, \mathsf{pub}(z_i)) &\xrightarrow{DY} y_2 \end{aligned}$$

TABLE 3.2: Dolev-Yao *SK*-rules, with insecure keys $k_i, \mathsf{priv}(s_i) \notin SK$ ranged over by $y_i, \mathsf{priv}(z_i)$

relation $\twoheadrightarrow \subseteq Q \times A \times Q$ is defined as follows. We have that $q \xrightarrow{a} q'$ if

$$R := T \xrightarrow{f} T'$$

is a fresh renaming w.r.t. $names(q)$ of a rule in $\mathcal{R}$ and there exists a grounding substitution θ for R such that $T\theta \subseteq q$ and given $a = f'(t_1, \ldots, t_n)$ we have that $f' = f$, $ar(f) = n$ and $args(a) = T\theta$. Then $q' = q \cup T'\theta$.

Given a LTS $P = (Q, A, \twoheadrightarrow, q_0)$, an execution is a sequence of transitions

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$$

that we abbreviate as $q_0 \xrightarrow{\alpha}{}^{*} q_n$, with $\alpha = a_1, a_2, \ldots, a_n$.

Example 3 (Wrap and Decrypt Attack)**.** *Consider, for example, the execution representing a wrap/decrypt attack in which the value of a key* k_2 *is exposed by wrapping* k_2 *with* k_1 *and then by decrypting the wrapped data with* k_1*. Given an initial state* $q_0 = \{\mathsf{h}(n_1, k_1), \mathsf{h}(n_2, k_2)\}$*, we have that*

$$\begin{aligned} q_0 &\xrightarrow{\mathsf{Wrap_{ss}}(\mathsf{h}(n_1,k_1),\mathsf{h}(n_2,k_2))} q_1 & q_1 &= q_0 \cup \{\mathsf{senc}(k_2, k_1)\} \\ q_1 &\xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(n_1,k_1),\mathsf{senc}(k_2,k_1))} q_2 & q_2 &= q_1 \cup \{k_2\} \end{aligned}$$

That we write $q_0 \xrightarrow{\alpha}{}^{*} q_2$*, with*

$$\begin{aligned} \alpha = \mathsf{Wrap_{ss}}(&\mathsf{h}(n_1, k_1), \mathsf{h}(n_2, k_2)), \\ &\mathsf{Decrypt_s}(\mathsf{h}(n_1, k_1), \mathsf{senc}(k_2, k_1)) \end{aligned}$$

α defines the sequence of actions performed by the attacker to access the value of k_2 *by reaching the state* q_2*.*

3.3 Secure Executions

Our notion of secure execution is parametric with respect to a set of secure key, which might be different for different executions. Intuitively, we want to let each administrator to specify, locally, the set of sensitive keys that need to be monitored. Thus, what is sensitive is not established globally but we need to be able to compose local executions and recover a partial view of what is sensitive, so to capture distributed attacks, i.e., attacks in which one key might be leaked on a different device in order to bypass local monitoring.

3.3.1 Secure Local Executions

We let SK denote a set of sensitive keys that we want to monitor in a certain local execution. Secure keys are either symmetric keys k or private asymmetric keys $\mathsf{priv}(\mathsf{s})$. In the following we let K range over k and $\mathsf{priv}(\mathsf{s})$. We only consider executions starting from a state q_0 in which all of the secure keys are safely stored in the device. They should not be publicly known or encrypted under an insecure key. Formally:

Definition 1 (SK-Secure Initial State)**.** *Let SK be a set of sensitive keys. An initial state q_0 is secure if any secure key K does not appear in q_0 in the forms k, $\mathsf{senc}(\mathsf{k}, \mathsf{k_i})$ and $\mathsf{aenc}(\mathsf{k}, \mathsf{pub}(\mathsf{s_i}))$, with $\mathsf{k_i}, \mathsf{priv}(\mathsf{s_i}) \notin SK$.*

From now on we will only consider executions with secure initial states.

We consider a Dolev-Yao attacker parametrized by SK that can perform encryption and decryption operations using known keys (as usual) plus any insecure key $\mathsf{k_i}, \mathsf{priv}(\mathsf{s_i}) \notin SK$, ranged over by $\mathsf{y_i}, \mathsf{priv}(\mathsf{z_i})$. Attacker is formalized by the rules in Table 3.2. From now on we assume that executions can include attacker's actions.

An execution is secure if and only if it does not leak any of its secure key:

Definition 2 (SK-Secure Execution)**.** *Let $\sigma = q_0 \xrightarrow{\alpha}\!\!\!\!\rightarrow^* q_n$ be an execution. Then, σ is SK-*secure *if and only if $SK \cap q_n = \emptyset$.*

Notice that freshly generated keys may or may not be included in SK. There might be cases in which an administrator wants to monitor new keys, and other situations in which new keys are just session keys that are destroyed when the session is closed, and does not need to be monitored. Both these situations can be modeled by including or not new keys in the set SK.

A SK-secure execution is also secure with respect to any subset of SK, i.e., with respect to strictly less sensitive keys. This is proved formally in the following lemma:

Lemma 1. *Let σ be SK-secure. Then σ is SK'-secure for each $SK' \subseteq SK$.*

Proof. Trivial since $SK' \cap q_n \subseteq SK \cap q_n = \emptyset$. □

We now prove that in any insecure execution there is at least either a wrap operation of a secure key under an insecure one, or a decrypt operation of a secure key encrypted under another secure key.

Proposition 1. *Let $\sigma = q_0 \xrightarrow{\alpha}\!\!\!\!\rightarrow^* q_n$ be an execution. Then, σ is SK-secure if and only if none of the following is in σ:*

1. $\mathsf{Wrap_{ss}}(\mathsf{h}(\mathsf{n_i}, \mathsf{k_i}), \mathsf{h}(\mathsf{n_s}, \mathsf{k_s}))$;
2. $\mathsf{Wrap_{sa}}(\mathsf{h}(\mathsf{n_i}, \mathsf{priv}(\mathsf{s_i})), \mathsf{h}(\mathsf{n_s}, \mathsf{k_s}))$;
3. $\mathsf{Wrap_{as}}(\mathsf{h}(\mathsf{n_i}, \mathsf{k_i}), \mathsf{h}(\mathsf{n_s}, \mathsf{priv}(\mathsf{s_s})))$;
4. $\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n_s}, \mathsf{k_s}), \mathsf{senc}(\mathsf{k'_s}, \mathsf{k_s}))$;
5. $\mathsf{Decrypt_a}(\mathsf{h}(\mathsf{n_s}, \mathsf{priv}(\mathsf{s_s})), \mathsf{aenc}(\mathsf{k_s}, \mathsf{pub}(\mathsf{s_s})))$;
6. $\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n_s}, \mathsf{k_s}), \mathsf{senc}(\mathsf{priv}(\mathsf{s_s}), \mathsf{k_s}))$.

where $\mathsf{k_i}, \mathsf{priv}(\mathsf{s_i}) \notin SK$ *and* $\mathsf{k_s}, \mathsf{k'_s}, \mathsf{priv}(\mathsf{s_s}) \in SK$.

Proof. ($\Rightarrow$) We have to prove that if σ is SK-secure then none of the above API calls happens in σ. We in fact prove that if one of the calls is in σ then σ is not SK-secure. It is enough to observe that wrapping a secure key under an insecure one allows the attacker to decrypt it (cf. Table 3.2), and decrypting a secure key clearly reveals it as plaintext. In both cases σ is not SK-secure, from which the thesis follows.

($\Leftarrow$) We have to prove that if none of the above API calls happens in σ then σ is SK-secure. We proceed by contradiction: assume that $\sigma = q_0 \xrightarrow{\alpha}\!\!\!\!\rightarrow^* q_n$ is not SK-secure. Then, by Definition 2, $SK \cap q_n \neq \emptyset$. We let $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} = SK \cap q_n$. By Definition 1 we know that $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} \cap q_0 = \emptyset$. We thus consider the shortest prefix of σ: $q_0 \xrightarrow{\alpha}\!\!\!\!\rightarrow^* q_{k-1} \xrightarrow{a_k}\!\!\!\!\rightarrow q_k$ such that $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} \cap q_{k-1} = \emptyset$ and $\exists i \in 1, \ldots, m \,.\, \mathsf{K}_i \in q_k$. Intuitively, q_k is the first state that contains a sensitive key in the clear. Recall that K_i is either k or $\mathsf{priv}(\mathsf{k})$. We now consider all the possible API calls that might have returned K_i in the clear.

From Table 3.1 we only have $\mathsf{Decrypt_s}$ and $\mathsf{Decrypt_a}$. Consider $\mathsf{Decrypt_s}$: it requires: $\mathsf{h}(\mathsf{n'}, \mathsf{k'}), \mathsf{senc}(\mathsf{K_i}, \mathsf{k'}) \in q_{k-1}$. Now if $\mathsf{k'} \in SK$ we are in case 4 or 6, while if $\mathsf{k'} \notin SK$ there must have been a previous API call corresponding to case 1 or 3. In fact, since the term $\mathsf{senc}(\mathsf{K_i}, \mathsf{k'})$ cannot be in q_0 (cf. Definition 1) and cannot come from a previous application of DY rules (cf. Table 3.2) because we have assumed that K_i is being leaked now, the only other way to obtain it is by invocation of the Wrap_* API. A similar reasoning applies to $\mathsf{Decrypt_a}$ for cases 5 and 2. We thus get a contradiction.

From Table 3.2 the key can only come from the decryption of either $\mathsf{senc}(\mathsf{K_i}, \mathsf{k'})$ or $\mathsf{aenc}(\mathsf{K_i}, \mathsf{pub}(\mathsf{s}))$. Consider $\mathsf{senc}(\mathsf{K_i}, \mathsf{k'})$: since it cannot be $\mathsf{k'} \in SK$ (because $\mathsf{k'}$ would be known by the attacker contradicting the fact that no sensitive key is in q_{k-1}) we necessarily have that $\mathsf{k'} \notin SK$. As before we have that there must have been a previous API call corresponding to case 1 or 3. Following a similar reasoning, for $\mathsf{aenc}(\mathsf{K_i}, \mathsf{pub}(\mathsf{s}))$ we can conclude that there must have been a previous API call corresponding to case 2, which gives a contradiction. □

The above proposition gives a precise characterization of insecure executions and has important implications: first of all, it is enough to monitor wrap and decrypt API calls and

rise an alert any time one of the above cases occur. When one of the above cases occur we are guaranteed that it is going to be an attack. Moreover, no attack will be missed since any attack requires one of the above API calls. Finally, the proposition shows that the complexity of Dolev-Yao reasoning disappears since it is enough to just focus on single API calls. This is very convenient in order to apply the theory to the analysis of real logs.

3.3.2 Secure Distributed Executions

Even if an execution is secure locally, with respect to its set SK of sensitive keys, it might be the case that the execution is leaking keys coming from other devices, that are not monitored in SK. It could also be possible that keys in SK are sent to other devices and are leaked remotely. In order to find these distributed attacks we define a notion of security with respect to a set of executions, each with a set of local sensitive keys. Intuitively, we require that each execution is secure with respect to the union of the set of sensitive keys.

Since executions might contain freshly generated keys, from now on we will always consider appropriate alpha-conversion of executions so that freshly generated keys never collide. So, if a freshly generated key is in a local SK it won't appear in any other local SK from a different execution.

Definition 3 (Secure Distributed Executions)**.** *Let $\mathcal{S}$ be a set of distinct executions starting from their own initial states with their respective sets of sensitive keys $\{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$. Let $SK = \bigcup_{i=1,\ldots,n} SK_i$. We say that $\mathcal{S}$ is secure iff $\sigma_1, \ldots, \sigma_n$ are SK-secure.*

It is quite immediate to see that if a set of executions is secure, then each execution is locally secure. In fact, secure distributed executions require security with respect to a bigger set of keys. Interestingly, the other implication does not hold: it might be the case that secure, local executions become insecure when taken together. This confirms the intuition that there exist distributed attacks that cannot be detected locally. Thus, collecting local executions from different devices might reveal attacks that cannot be detected locally. We illustrate through a simple example.

Example 4 (Distributed Wrap-and-Decrypt Attack)**.** *Consider the following two executions σ and σ':*

$$\sigma = q_0 \xrightarrow{\mathsf{Wrap_{ss}}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\mathsf{h}(\mathsf{n}_2,\mathsf{k}_2))} q_0 \cup \{\mathsf{senc}(\mathsf{k}_2,\, \mathsf{k}_1)\}$$

$$\sigma' = q'_0 \xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\mathsf{senc}(\mathsf{k}_2,\mathsf{k}_1))} q'_0 \cup \{\mathsf{k}_2\}$$

Intuitively, σ and σ' represent two executions on different devices. In σ a sensitive key k_2 is wrapped under another sensitive key k_1, which is the standard key for security exporting a sensitive key. We have, in particular, that σ is $\{\mathsf{k}_1, \mathsf{k}_2\}$-secure. In σ' we suppose to have a device with just k_1 key, meaning that the administrator is only monitoring that single sensitive key.

An attacker might decrypt the ciphertext obtained from σ on the first device using the second device. This is what happens in σ': $\mathsf{senc}(\mathsf{k}_2,\, \mathsf{k}_1)$ is decrypted under k_1. The local

administrator cannot notice the leakage of k_2 *if such a key is unknown locally. In particular, we have that* σ' *is* $\{\mathsf{k}_1\}$*-secure.*

In summary, we have two executions that are secure with respect to the local knowledge of sensitive keys. However, it is clear that the two executions represent a distributed wrap-and-decrypt attack in which a sensitive key from the first device (k_2*) is leaked on the second device. This attack is captured by putting together the two executions: we let* $\mathcal{S} = \{(\{\mathsf{k}_1, \mathsf{k}_2\}, \sigma), (\{\mathsf{k}_1\}, \sigma')\}$ *and we obtain that* $\mathcal{S}$ *is not secure since* σ' *is not secure with respect to* $\{\mathsf{k}_1, \mathsf{k}_2\}$*. This can be seen by observing that* $\{\mathsf{k}_1, \mathsf{k}_2\} \cap (q'_0 \cup \{\mathsf{k}_2\}) \supseteq \{\mathsf{k}_2\} \neq \emptyset$*.*

The relation between local and distributed security is proved formally in the following proposition:

Proposition 2. *Let* $\mathcal{S} = \{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$*. Then:*

- $\mathcal{S}$ *secure* $\Rightarrow$ σ_i SK_i*-secure for each* $i = 1, \ldots, n$*;*
- $\mathcal{S}$ *secure* $\not\Leftarrow$ σ_i SK_i*-secure for each* $i = 1, \ldots, n$*.*

Proof. ($\Rightarrow$) By definition, $\mathcal{S}$ secure means that the distinct executions $\sigma_1, \ldots, \sigma_n$ are SK-secure, with $SK = \bigcup_{i=1,\ldots,n} SK_i$. Since $SK_i \subseteq SK$, by Lemma 1 we directly obtain that σ_i is SK_i-secure for each $i = 1, \ldots, n$.

($\not\Leftarrow$) The implication does not hold because of the existence of distributed attacks coming from locally secure executions, as shown in Example 4. □

3.4 Analysis

We present a way to analyze executions offline. This allows for monitoring devices without necessarily being online, i.e., in between the application and the security hardware. We believe this is important to make the proposal realistic. In fact, in our experience, it would be hard to add an online element in the chain of a critical application based on secure hardware. The offline analysis can detect leakage of keys so that administrators can take suitable actions. Of course, if the solution works offline it is also possible to place it actively in the middle of the API calls, taking decision in real time, and preventing key leakage.

Logs can be taken locally and analyzed directly but, as we have shown in Example 4, attacks might happen across multiple devices, so it is crucial to consider the possibility of collecting local logs to look for distributed attacks.

It is important to notice that in order to monitor the above calls we need a way to distinguish secure keys from insecure ones and we need to track wrapped secure keys. We will discuss how this can be achieved in the next section. There are two important aspects to consider, in order to make the analysis effective and implementable: (i) the information that is tracked in the logs should not be too complex and should not grow too much, in order for the analysis to scale in space and time; (ii) the analysis should not require the whole execution logs in order to detect attacks, i.e., it should detect attacks even when logs represent partial executions.

It is important to point out that our model of distributed execution already detects attacks even when relevant API calls are missing, i.e., even when logs are partial. Thus, the requirement (ii) is implicit in the model we consider. As a consequence, any log analysis will necessarily have to fulfill (ii) in order to detect all the attacks. We illustrate this crucial point through an example:

Example 5 (Partial Executions)**.** *Consider a variant of Example 4 in which σ does not contain the wrap operation used by the attacker to mount the distributed wrap-and-decrypt attack. Execution σ could contain other API calls but, for simplicity, we just take it empty:*

$$\sigma = q_0$$
$$\sigma' = q_0' \xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\mathsf{senc}(\mathsf{k}_2,\mathsf{k}_1))} q_1' \qquad\qquad q_1' = q_0' \cup \{\mathsf{k}_2\}$$

The point here is that σ' is an attack to k_2 but there is no information in the logs about ciphertext $\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1)$.

The attack is nevertheless captured by the model. As before, in the first device we assume to have two sensitive keys, and we trivially have that σ is $\{\mathsf{k}_1, \mathsf{k}_2\}$-secure. However, $\{(\{\mathsf{k}_1, \mathsf{k}_2\}, \sigma), (\{\mathsf{k}_1\}, \sigma')\}$ is not secure since σ' is not secure with respect to $\{\mathsf{k}_1, \mathsf{k}_2\}$. Intuitively, since the set of sensitive keys is composed of all the sensitive keys from the various devices, attacks on a remote device will be naturally captured by the model that simply checks for the leakage of sensitive (possible remote) keys.

3.4.1 The Log Analysis Problem

We now state precisely the problem of log analysis. It is important to observe that, for obvious reasons, we cannot log the actual values of sensitive keys. The obvious replacement for key values are handles but this will introduce a major challenge: how to detect the leakage of a key value without knowing it.

Definition 4 (Log Analysis Problem)**.** *Let $\mathcal{S}$ be the distributed execution $\{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$.* Log analysis *is the problem of deciding whether or not $\mathcal{S}$ is secure given the following inputs:*

- *The executions $\bar{\sigma} = \sigma_1, \ldots, \sigma_n$, that we call logs;*
- *The handles H referring to sensitive keys occurring in $\sigma_1, \ldots, \sigma_n$ that belongs to $SK = \bigcup_{i=1,\ldots,n} SK_i$, i.e.,*

$$H = \{\mathsf{h}(\mathsf{n}, \mathsf{k}) \mid \mathsf{h}(\mathsf{n}, \mathsf{k}) \text{ occurs in } \bar{\sigma} \text{ and } \mathsf{k} \in SK\}$$

and under the following assumptions:

1. *terms can only be compared by syntactic equality;*
2. *offline encryption and decryption operations are possible only when the corresponding key is known (in a standard Dolev-Yao fashion).*

We now show that the log analysis problem is not solvable in general, because of the impossibility of linking a key value to its handle(s). Notice that this result holds because of the assumption that log analysis is done offline. If we have the possibility of interacting with the devices then keys could be distinguished by performing operations with them.

Proposition 3 (Unsolvability of Log Analysis Problem)**.** *The log analysis problem cannot be solved for all possible $\mathcal{S}$'s.*

Proof. We consider an instance of Example 5 in which $q_0 = \{\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)\}$ and $q_0' = \{\mathsf{h}(\mathsf{n}_1', \mathsf{k}_1), \mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1)\}$. Intuitively, the initial states q_0 and q_0' only contain the key handles and q_0' additionally contains the ciphertext that will be decrypted to leak k_2. The input to the log analysis problem is thus σ, σ' and $H = \{\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{h}(\mathsf{n}_1', \mathsf{k}_1)\}$. The final state q_1' additionally contains the key value k_2. Now, there is clearly no way to link k_2 to the key handles in H, since we have assumed that terms can only be compared when they are identical (up to standard Dolev-Yao operations). Key k_2 could be used to encrypt other terms but since there is no ciphertext encrypted under k_2 the produced terms would never match any existing term. Intuitively, k_2 is leaked but it is not possible to detect offline whether or not it is a sensitive key pointed by one of the handles in H. □

Algorithm 1 Log Analysis using Key Fingerprinting.

```
1:  procedure LogAnalysis(σ̄, H)
2:      FSK = [ ]                          ▷ Initialize the list of fingerprints of sensitive keys as empty
3:      for (a, ret) ∈ σ̄ do                ▷ Collect all the fingerprints of sensitive keys
4:          if a == KeyFprint(h) and h ∈ H then   ▷ If the API call is KeyFprint in sensitive handle
5:              FSK ← FSK + [ret]          ▷ The actual fingerprint ret is added to FSK
6:          end if
7:      end for
8:      for (a, ret) ∈ σ̄ do                ▷ Search for insecure wrap and decrypt
9:          if a == Wrap*(h1, h2) and h1 ∉ H and h2 ∈ H then   ▷ Insecure wrap of sensitive key
10:             return a                   ▷ The insecure wrap is returned: S is insecure
11:         end if
12:         if a == Decrypt*(h, t) and h ∈ H and kf(ret) ∈ FSK then   ▷ Decrypt of a sensitive key
13:             return a                   ▷ The insecure decrypt is returned: S is insecure
14:         end if
15:     end for
16:     return None                        ▷ No attack found: S is secure
17: end procedure
```

3.4.2 Log Analysis with Key Fingerprinting

In order to be able to solve the log analysis problem we need to log additional information that can be used offline to track keys. We consider an abstract key fingerprinting function

whose value will be logged together with handles and that will allow to link a key value to a handle.

Definition 5 (Key Fingerprinting)**.** *A key fingerprinting is a deterministic one-way function. Formally, we note it as a special term* $\mathsf{kf}(\mathsf{k})$ *and we assume that* $\mathsf{kf}(\mathsf{k})$ *can be computed by anyone who knows* k*, while* k *cannot be computed from* $\mathsf{kf}(\mathsf{k})$*.*

We add a corresponding API call that allows for obtaining key fingerprints from their handles, and a corresponding Dolev-Yao rule for offline computation:

$$\begin{array}{rcl} \mathsf{h}(\mathsf{x},\,\mathsf{y}) & \xrightarrow{\mathsf{KeyFprint}} & \mathsf{kf}(\mathsf{y}) \\ \mathsf{y} & \xrightarrow{\mathsf{DY}} & \mathsf{kf}(\mathsf{y}) \end{array}$$

We can prove that key fingerprinting does not introduce new attacks. In particular, by adding the above rules we obtain the same characterization of Proposition 1.

Proposition 4. *Let* $\sigma = q_0 \xrightarrow{\alpha}\!\!\!\!\rightarrow^{*} q_n$ *be an execution possibly containing key fingerprint API calls and direct (Dolev-Yao) fingerprinting computations. Then, Proposition 1 holds.*

Proof. Proof is the same as the one of Proposition 1. In fact, fingerprinting does not add any new way of leaking a key in the clear, nor it can produce cryptographic terms. As a consequence, adding fingerprinting does not add any new case to the proof of Proposition 1. □

Notice that the above proposition holds in our symbolic model because there is no notion of cost for the attack and, in general, we do not take into account cryptanalytic issues. It is important to observe that just using a standard one-way cryptographic hash to implement kf would provide the attacker a faster way to bruteforce cryptographic keys since hash functions are usually much faster than encryption algorithms. We will discuss possible implementations later on.

With key fingerprinting we can solve the log analysis problem efficiently. It is enough to assume that each sensitive key is fingerprinted in each local log. Intuitively, whenever we have a decrypt operation we test the leaked key against all the available fingerprints.

Our solution is coded as Algorithm 1: the algorithm first computes the set of fingerprints for sensitive keys (FSK) by looking for the actual calls to $\mathsf{KeyFprint}(\mathsf{h})$ where $\mathsf{h} \in H$ is a handle to sensitive keys.[1] This is done by the for loop from line 3 to line 7. The notation $(a, ret) \in \bar{\sigma}$ means that we loop over all possible API calls and a ranges over the actual call while ret ranges over the returned value. In fact, when we find a handle that belongs to H we add the returned value, i.e., the fingerprint, to the set FSK.

Then, the algorithm looks for attacks, in terms of the characterization of Proposition 4. In particular it searches for any $\mathsf{Wrap}_{*}(\mathsf{h}_1, \mathsf{h}_2)$ API call in which a sensitive key referred by h_2 is wrapped under a non-sensitive key referred by h_1. When this happens, the algorithm terminates and returns the call responsible for the attack. Similarly the algorithm looks for

[1] Notice that, for the sake of readability, we abbreviate handles as h.

any $\mathsf{Decrypt}_*(\mathsf{h},\mathsf{t})$ call that returns a key *ret* whose fingerprint $\mathsf{kf}(ret)$ belongs to the ones computed in the first phase (FSK). Again, when this situation is spotted, the responsible call is returned as a witness of the attack. If none of the above is found in all the executions the algorithm returns "None" to indicate that no attack has been found and that $\mathcal{S}$ is in fact secure.

Theorem 1 (Log Analysis through Key Fingerprinting)**.** *Let* $\mathcal{S} = \{(SK_1,\sigma_1),\ldots,(SK_n,\sigma_n)\}$ *and* $SK = \bigcup_{i=1,\ldots,n} SK_i$*, such that for each* $\mathsf{K} \in SK$ *we have that* $\mathsf{kf}(\mathsf{K})$ *occurs in* $\bar{\sigma}$*. Then, the log analysis problem can be solved in* $O(|\bar{\sigma}| + |H|)$ *steps.*

Proof. The algorithm correctly computes the set FSK of all fingerprints of sensitive keys because of the assumption that those fingerprints are all in the logs. Then, the correctness of solving log analysis directly derives from the characterization of Proposition 4. Both loops take, in the worst case, $|\bar{\sigma}|$ iterations while lookup in sets H and FSK can be done in constant time building appropriate hashtables, from which we get linear complexity. □

Notice that, since Algorithm 1 only inspects a subset of the API calls, it is enough to just log those calls. This would greatly reduce the size of $|\bar{\sigma}|$ and, consequently, the execution time of the analysis.

3.4.3 Practical Considerations

Our approach requires the specification of which keys are considered sensitive. This decision must be definitely taken by the administrator, who is supposed to know what are the important cryptographic keys. It is worth noticing that key management APIs usually have a way to specify what keys should be regarded as sensitive, i.e., not accessible in the clear, so it is reasonable to assume that this property is going to be specified in some way.

Theorem 1 proves that log analysis can be solved efficiently when fingerprints for sensitive keys are available. Thus, in order to implement the proposed analysis, it is necessary that the relative fingerprint API calls are performed, in each local log. For long-term keys, it is reasonable to assume that key fingerprints will stabilize over time and could be reliably shared after an initial startup phase. For new keys that are freshly generated during the execution we can imagine that the logging system is instrumented so to ask for the key fingerprint of each new key, i.e., every time a KeyGen or KeyPairGen is invoked. We believe this is a mild, realistic assumption that does not significantly impact on the applicability of the methods. In fact, recall that without fingerprinting we know that log analysis is not even solvable (cf. Proposition 3). Additionally, a practical way to prevent that logs grow indefinitely is to delete part of them when there is a consensus that the knowledge on sensitive keys is synchronized, i.e., that no key occurring in the deleted logs will be discovered to be sensitive in the future.

Another consideration regards fingerprints. One problem with fingerprints that is not captured by our symbolic model is the possible exploitation of fingerprints in cryptanalytic attacks. Using cryptographic hash functions, for example, would speed up key bruteforcing. Moreover, using a fixed function for all devices allows for precomputing fingerprints which,

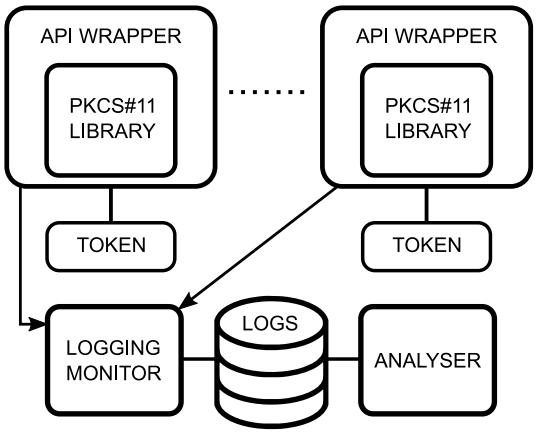


FIGURE 3.1: Diagram of our log analysis system for PKCS#11

in turns, would reduce the key space to bruteforce. A reasonable alternative would be to digitally sign the fingerprint using a dedicated private key, different for each device. This would (i) slow down bruteforcing and (ii) prevent fingerprint precomputation. It is out of the scope of this work to prove the security of practical implementations of key fingerprint and we leave this as a future work.

3.5 Prototype Implementation

To show the feasibility of our approach we discuss the implementation of a proof-of-concept log analysis tool for PKCS#11. The tool[2] is able to identify all the key-management attacks found in [60, 71] involving symmetric encryption operations. We plan to support the detection of attacks using asymmetric keys on PKCS#11 as a future work.

Our solution consists of three components, as outlined in Figure 3.1: (i) a software layer that wraps the existing PKCS#11 library interface. The wrapper allows the instrumentation of selected API calls to record the operations executed by the underlying library. It also computes key fingerprints to solve the log analysis problem; (ii) a logging facility to store the logs of each session in a central repository; (iii) the analyzer that parses the logs generated by the first two components and applies Algorithm 1 to discover attacks aimed at leaking the value of secure keys.

3.5.1 Fingerprint Computation

Before giving full details of the system, we introduce our fingerprint approach for PKCS#11. Fingerprint computation is indeed a challenging problem, given that it is not possible to access the value of a sensitive key directly. Additionally, since our solution does not extend the existing API with an ad-hoc fingerprint function, the fingerprint must be produced by the device reusing existing PKCS#11 functions.

Of course, a simple way to perform fingerprint computation would be to use the `C_DigestKey` API call. Given a key handle, this function returns a digest of the key value

[2] `https://github.com/secgroup/p11d`

using a cryptographic hash function. According to the considerations made in Section 3.4, the use of a hash function would weaken the security of the system by decreasing the cost of a bruteforce attack to recover the key value. For this reason, we devised a different approach that allows to compute key fingerprints using keys as intended.

The functions allowed to be executed under a key are determined by the set of its attributes. For instance, a key with the `CKA_ENCRYPT` attribute enabled is allowed to perform encryption calls via `C_Encrypt`. Given a key k, we exploit the capabilities of each key to compute multiple fingerprints depending on the allowed operations. We devise three possible fingerprints for key k:

- if `CKA_ENCRYPT` is enabled, we pick a random value r and we encrypt it with k. We say that the *encryption fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_E$, is the pair $(\mathsf{r}, \mathsf{enc\_data})$ where enc_data denotes r encrypted under k using the `C_Encrypt` function with a compatible mechanism;

- similarly, if `CKA_DECRYPT` is enabled, we let dec_data be a random value r decrypted with k using the `C_Decrypt` function with a compatible mechanism. The *decryption fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_D$, is the pair $(\mathsf{r}, \mathsf{dec\_data})$;

- if k is a wrapping key, i.e., the `CKA_WRAP` attribute is enabled, we state that the *wrap fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_W$, is wrap_data, where wrap_data is the result of wrapping the key with itself via a call to `C_WrapKey`.

If all the operations required to produce a fingerprint are forbidden, i.e., the attributes are disabled, we temporarily alter the `C_Encrypt` attribute to generate a valid encryption fingerprint.

The proposed fingerprint approach for PKCS#11 allows to precisely identify a key by performing an offline computation of the fingerprint, once the plain text value of the key is known. Moreover, the results of the operations are unique for each key, in practice, since the probability of obtaining the same result given two different keys is negligible. With respect to the practical considerations mentioned in the previous section regarding fingerprints, we claim that this solution does not speed up key bruteforcing and thus it does not decrease the security of fingerprinted keys. If one of the functions used to compute the fingerprint is allowed for a given key, then guessing the key from the encryption (decryption) of a random value would not be faster than doing the same with a value chosen by the attacker for which (s)he might have precomputed offline encryptions (decryptions) with a large number of keys.

3.5.2 Working Principles

We implement a wrapper of the full PKCS#11 API. However, it follows from Proposition 1 and Proposition 4 that monitoring only a small subset of PKCS#11 functions, i.e., `C_WrapKey` and `C_Decrypt`, is enough to detect all possible attacks. We also instrument the `C_GetAttributeValue` function that allows to directly read the value of a non-sensitive key. We do not assume the set of secure keys SK to be static during each execution, thus we need to track functions that allow the creation of new sensitive objects such

as C_GenerateKey. For convenience, we also instrument C_Login to list long-term keys stored in the device and we initialize *SK* with sensitive keys found among them, even if in general we want to let each administrator to specify, locally, the set of sensitive keys that need to be monitored. For all the remaining functions, the wrapper transparently performs the corresponding call.

For each new key found during a C_Login call at the start of each session, or generated using C_GenerateKey, the wrapper computes the fingerprints and sends them to the logging facility along with the list of publicly-readable attributes and the object handle assigned to the key. Recall that handles are not guaranteed to be fixed for the lifetime of an object, still they allow to access the same object for the entire session duration [139]. The C_WrapKey function is instrumented to track the handles of the wrapping key and the wrapped key. C_Decrypt tracks the handle of the decryption key and the value of the decrypted data. Similarly, the C_GetAttributeValue tracks the handle of the actual key and the accessed value.

The logging component allows multiple sessions to be tracked in a centralized repository at the same time. Each log file produced during this step represents a single execution within a session. Since the C_Login function is called every time a new session is initialized by PKCS#11 applications, the first entry of every log file contains the list of long-term keys found in the device. As an example, Table 3.3 provides the textual representation of a possible log entry produced by a call to the login function. In this case only one key is found in the device. The listed key is sensitive and the handle that points to it in this session has value 0x00. The key has the encrypt and decrypt attributes enabled, thus fingerprints are computed by encrypting and decrypting random values according to the method described before.

```
["C_Login", [
  {"extractable": 0x00, "decrypt": 0x01,
   "sensitive": 0x01, "encrypt": 0x01,
   "wrap": 0x00, "unwrap": 0x00,
   "label": 0x4d7950726563696f7573,
   "keytype": 0x1300000000000000,
   "handle": 0x01,
   "fingerprint": {
     "decrypt": [
       0x96ccb41274a8adbf, 0x0c2e551a66cb4d86
     ],
     "encrypt": [
       0xd387b0b818a52d2a, 0xea1b3c934ed860f5
     ]
   }
]]
```

TABLE 3.3: Log entry for the C_Login call

The analyzer then parses the collected logs and applies Algorithm 1. For each sensitive key found in the logs as a result of either a C_Login or C_GenerateKey, the component updates the set H with its handle paired with the identifier of the current session. In parallel, the analyzer stores the fingerprints of this key in *FSK*. When the set H and

FSK are initialized, the program iterates over all the logged `C_WrapKey`, `C_Decrypt` and `C_GetAttributeValue` calls looking for attacks: i) insecure wraps of secure keys are easily detected by checking if the pairs in the form $(\mathsf{session}, \mathsf{h})$ of the wrapping key and wrapped key handles belong to H. If a secure key is wrapped under an insecure one, the operation is marked as an attack and the analysis terminates; ii) decryptions of secure keys are identified in two steps. The analyzer first checks if the handle used in the decryption operation points to a secure key. In this case, it tests the decrypted data *ret* found in the log entry of `C_Decrypt` against all the fingerprints in *FSK*, by simulating calls to `C_Decrypt` and `C_Encrypt`. To perform the comparison with a wrap fingerprint, the application encrypts *ret* under itself and checks if the result matches the wrapped data in the fingerprint. Otherwise, if the fingerprint is in the form $(\mathsf{r}, \mathsf{data})$, depending on the type of the fingerprint, the tool executes an encryption or a decryption of the random value r under the key *ret* and compares the result with data. If no match is found after iterating the process over all the fingerprints in *FKS*, the operation is considered safe, otherwise it is marked as an attack and the analysis stops; iii) direct accesses to the value of a non-sensitive key via a `C_GetAttributeValue` are threat of practical importance if the attacker manages somehow to alter the `CKA_SENSITIVE` attribute of a secure key. These attacks are easily detected by the analyzer by testing the value returned by the API call against all the fingerprints in *FSK*, as in the previous case.

3.5.3 Experimental Tests

We now show how our solution is effective against a range of key-management attacks, also in a distributed setting. All the attacks reported in this section, as well as others from the literature, can be simulated using our tool and a software token provided by openCryptoki[3]. Unless stated otherwise, in the following examples we denote by h_i the handle pointing to a key k_i.

Example 6 (Wrap and Decrypt Attack)**.** *We discuss how the wrap/decrypt attack outlined in Example 3 is detected. In this simulation, the attacker calls* `C_GenerateKey` *to generate a non-sensitive key* k_2 *with the* `CKA_WRAP` *and* `CKA_DECRYPT` *attributes enabled. Using this key, (s)he leaks the value of the long-term sensitive key* k_1 *by wrapping* k_1 *under* k_2 *and decrypting the result again with* k_2*. As pointed out in Table 3.4, the attack is detected by our tool on the* `C_WrapKey` *operation since the attacker is wrapping a secure key pointed by* h_1 *with an insecure one pointed by* h_2*.*

Example 7 (Re-import Attack)**.** *In our implementation of the re-import attack, the attacker executes* `C_GenerateKey` *to generate a key* k_2 *with the* `CKA_UNWRAP` *attribute set. (S)he then unwraps a random value* r *with this key to create a new key* k_3 *in the device with the* `CKA_WRAP` *attribute set. Notice that* k_3 *is the decryption of* r *under* k_2*. The value* r *is unwrapped again using* k_2 *to re-import* k_3*, this time with the* `CKA_DECRYPT` *attribute set. We let* h_3 *and* h_4 *be the handles returned by the first and the second unwrap, respectively. Now, to leak the sensitive key* k_1*, the attacker wraps* k_1 *under* k_3 *pointed by* h_3 *and decrypts*

[3] `https://github.com/opencryptoki/opencryptoki`

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Wrap-Decrypt attempt
[*] Generate a key k2 for wrapping and
    decrypting
[*] Wrap the sensitive key k1 with the key k2
[*] Decrypt the wrapped key k1 with the key k2
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-1000.log
    The sensitive key h1 has been wrapped with
    the insecure key h2
```

TABLE 3.4: Wrap/Decrypt attack detection

the wrapped key with k_3 *pointed by* h_4*. As shown in Table 3.5, our tool detects the attack on the* `C_WrapKey` *operation since* h_1 *points to a secure key, while* h_3 *does not.*

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Re-import attempt
[*] Generate a key k2 for unwrapping
[*] Unwrap a random bytestream with k2 to
    import a new key k3 pointed by h3 that
    can wrap
[*] Unwrap a random bytestream with k2 to
    import a new key k3 pointed by h4 that
    can decrypt
[*] Wrap the sensitive key k1 with h3
[*] Decrypt the wrapped key k1 with h4
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-2000.log
    The sensitive key h1 has been wrapped with
    the insecure key h3
```

TABLE 3.5: Re-import attack detection

Example 8 (Wrap and Unwrap Attack)**.** *We assume the existence of a long-term sensitive key* k_1 *in the device. The key has the attributes* `CKA_WRAP` *and* `CKA_UNWRAP` *enabled. The attack consists in wrapping* k_1 *with itself and reimporting the key as a non-sensitive one under a new handle* h_2*. Then, by using the* `C_GetAttributeValue` *on* h_2*, the attacker can directly read the value of* k_1*. Our tool is able to detect the attack by testing the plain value of* k_1 *against the fingerprints in FSK. The attack trace and the log analysis performed by the tool are provided in Table 3.6.*

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Wrap-Unwrap attempt
[*] Wrap k1 with k1
[*] Re-import k1 as non-sensitive
[*] Recovering k1 value: "a1a2a3a4a5a6a7a8"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-3000.log
    The plaintext value of a sensitive key
    has been directly read
```

TABLE 3.6: Wrap/Unwrap attack detection

Example 9 (Distributed Wrap and Decrypt Attack)**.** *The last attack we discuss is wrap/decrypt in the distributed setting, as presented in Example 4. We assume two long-term sensitive keys* k_1 *and* k_2 *in the first device. We also assume* k_2 *to be found in a second device. The key* k_2 *has, at least, the attribute* `CKA_WRAP` *enabled in the first device and the attribute* `CKA_DECRYPT` *enabled in the second one. The attacker connects to the first device and wraps* k_1 *under* k_2 *and (s)he saves the wrapped data. Then, (s)he connects to the second device and decrypts the wrapped data with* k_2 *to access the value of* k_1*. When both* k_1 *and* k_2 *are secure keys, the* `C_WrapKey` *operation is not detected by our tool as an attack since we are wrapping a secure key with another secure key. Nevertheless, the* `C_Decrypt` *call returns the value of the secure key* k_1 *and thus allows our tool to match* k_1 *against fingerprints in FSK, revealing that an attack occurred. See Table 3.7 for the detailed execution and the attack detection analysis.*

3.6 Conclusion

Attacks on cryptographic APIs are notoriously hard to detect and fix. Even simple key management operations may be subject to API level vulnerabilities that leak cryptographic keys in the clear. For example, an attacker can wrap a secure key under another secure key and then ask the device to decrypt the ciphertext, obtaining the former key in the clear. In the literature we find many proposals for preventing or mitigating this kind of attacks but they typically require to modify the API or to configure it in a way that might break existing applications. This makes it very hard to adopt these proposals for critical applications and infrastructures, where systems are rarely modified and legacy applications are very common. At the same time, in these critical settings, the leakage of a cryptographic key can cause serious consequences.

In this work we have investigated a new method to analyze cryptographic API logs. Log analysis is interesting because it has a very low impact on existing systems and is frequently used in industrial systems, financial applications and critical infrastructures. Log analysis of cryptographic APIs is challenging since keys are never supposed to be leaked in the clear,

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 2 key(s)
[A] Distributed Wrap-Decrypt attempt (1)
[*] Wrap the sensitive key k1 with the
    sensitive key k2
[*] Wrapped data: "d6c22bb28cd93ec0"

$ LD_PRELOAD=./p11d.so ./attack 1 12345
[I] Found 1 key(s)
[A] Distributed Wrap-Decrypt attempt (2)
[*] Decrypt wrapped data "d6c22bb28cd93ec0"
    with the key k2
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-4001.log
    The plaintext value of a sensitive key
    has been leaked after decryption with
    key h1
```

TABLE 3.7: Distributed Wrap/Decrypt attack detection

meaning that tracking different keys might become hard, especially if we want to analyze logs offline without interacting with the cryptographic devices.

More specifically, we have extended an existing model for security API analysis in order to model API logs. We have given a formal definition of secure execution that scales to a distribute setting, in which logs from services and devices that are placed in different physical locations, can be collected and searched for distributed attack sequences. We have shown examples of distributed attacks that cannot be detected locally and we have proved that the problem of detecting these attacks offline is unsolvable, because of the impossibility of tracking keys. We have shown that by adding a simple API for key fingerprinting, log analysis becomes feasible and efficient. We actually proved that security can be characterized in term of absence of particular combination of parameters in a subset of the API calls, i.e., Wrap and Decrypt.

Finally, we have implemented a tool for PKCS#11 APIs that simulates key fingerprinting through the available cryptographic operations for a given key, and can detect all documented attacks on PKCS#11 that directly leak a key in the clear. The tool constitutes a proof-of-concept that the method is effective and that can be implemented even without a dedicated key fingerprinting API. It is worth noticing, that adding a key fingerprinting API for logging purposes would not affect existing applications. Compared to previous works, our approach does not require existing API functions to be modified, therefore legacy applications do not need to be updated. Instead, we propose to add a new fingerprinting function that is solely used by the monitoring solution to provide more informative logs. In this respect, extending existing devices with this new mechanism seems a realistic possibility and we would encourage producer to consider this idea for next generation devices.

As a future work, we intend to extend our tool to cover a more extensive fragment of PKCS#11 and we want to experiment with candidate key fingerprinting APIs on software emulators of PKCS#11. We also intend to characterize other cryptographic APIs by studying formally which rules are considered problematic and should be tracked in the logs. Intuitively, the problematic rules are the ones that either directly leak a key in the clear, or generate a term containing a sensitive key that can be deconstructed by the DY attacker, as when wrapping a sensitive key under a nonsensitive one. Lastly, we plan to cover cryptanalytic attacks related to weak cryptographic mechanisms and side channels.

Chapter 4

A Security Evaluation of Java Keystores

4.1 Introduction

Cryptography is a fundamental technology for IT security. Even if there are well established standards for cryptographic operations, cryptography is complex and variegated, typically requiring a non-trivial combination of different algorithms and mechanisms. Moreover, cryptography is intrinsically related to the secure management of cryptographic keys which need to be protected and securely stored by applications. Leaking cryptographic keys, in fact, diminishes any advantage of cryptography, allowing attackers to break message confidentiality and integrity, to authenticate as legitimate users or impersonate legitimate services. Quoting [153], "key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system".

In the recent years we have faced a multitude of flaws related to cryptography (*e.g.*, [25, 15, 57, 56]). Some of these are due to the intrinsic complexity of cryptography, that makes it hard to design applications that adopt secure combinations of mechanisms and algorithms. For example, in padding oracle attacks, the usage of some (standard) padding for the plaintext combined with a specific algorithm or mechanism makes it possible for an attacker to break a ciphertext in a matter of minutes or hours [167, 31, 15]. Most of the time this is not a developer fault as, unfortunately, there are well-known flawed mechanisms that are still enabled in cryptographic libraries. In other cases, the attacks are due to flaws in protocols or applications. The infamous Heartbleed bug allowed an attacker to get access to server private keys through a simple over-read vulnerability. Once the private key was leaked, the attacker could decrypt encrypted traffic or directly impersonate the attacked server [57].

Thus, breaking cryptography is not merely a matter of breaking a cryptographic algorithm: the attack surface is quite large and the complexity of low-level details requires abstractions. Crypto APIs offer a form of abstraction to developers that allows to make use of cryptography in a modular and implementation-independent way. The Java platform, for example, provides a very elegant abstraction of cryptographic operations that makes it possible, in many cases, to replace a cryptographic mechanism or its implementation with a different one without modifying the application code.

Crypto APIs, however, do not usually provide security independently of the low-level implementation: default mechanisms are often the weakest ones, thus developers have to face the delicate task of choosing the best mechanism available for their needs. For example, in the Java Cryptography Architecture (JCA), ECB is the default mode of operation for block ciphers [91] and PKCS#1 v1.5 is the default padding scheme for RSA [93], which is well know to be subject to padding oracle attacks [31]. Additionally, crypto APIs that promise to provide security for cryptographic keys have often failed to do so: in PKCS#11, the standard API to cryptographic tokens, it is possible to wrap a sensitive key under another key and then just ask the device to decrypt it, obtaining the value of the sensitive key in the clear [49], and violating the requirement that "sensitive keys cannot be revealed in plaintext off the token" [144].

In this thesis we analyze in detail the security of key management in the Java ecosystem and, in particular, of Java keystores. Password-protected keystores are, in fact, the standard

way to securely manage and store cryptographic keys in Java: once the user (or the application) provides the correct password, the keys in the keystore become available and can be used to perform cryptographic operations, such as encryption and digital signature. The `KeyStore` Java class abstracts away from the actual keystore implementation, which can be either in the form of an encrypted file or based on secure hardware. As discussed above, this abstraction is very important for writing code that is independent of the implementation but developers are still required to select among the various keystore *types* offered by Java. Unfortunately, the information in the keystore documentation is not enough to make a reasoned and informed choice among the many alternatives. More specifically, given that the Java Keystore API does not provide control over the cryptographic mechanisms and parameters employed by each keystore, it is crucial to assess the security provided by the different implementations, which motivated us to perform the detailed analysis reported in this chapter. In fact, our work is the first one studying the security of keystores for general purpose Java applications.

We have estimated the adoption rate and analyzed the implementation details of seven different Java keystores offered by the Oracle JDK and by Bouncy Castle, a widespread cryptographic library. Keystores are used by hundreds of commercial applications and open-source projects, as assessed by scraping the GitHub code hosting service including leading web applications servers and frameworks, *e.g.*, Tomcat [8], Spring [157], Oracle Weblogic [181]. Additionally, keystores have been found to be widespread among security-critical custom Java software for large finance, government and healthcare companies audited by the authors.

The security of keystores is achieved by performing a cryptographic operation C under a key which is derived from a password through a function F called Key Derivation Function (KDF). The aim of the cryptographic operation C is to guarantee confidentiality and/or integrity of the stored keys. For example, a Password-Based Encryption (PBE) scheme is used to protect key confidentiality: in this case C is typically a symmetric cipher, so that keys are encrypted using the provided password before being stored in the keystore. In order to retrieve and use that key, the keystore implementation will perform the following steps: (a) obtain the password from the user; (b) derive the encryption key from the password using F; (c) decrypt the particular keystore entry through C, and retrieve the actual key material. Notice that different passwords can be used to protect different keys and/or to achieve integrity. To prevent attacks, it is highly recommended that C and F are implemented using standard, state-of-the-art cryptographic techniques [122, 146].

Interestingly, we have found that the analyzed keystores use very diverse implementations for C and F and in several cases they do not adhere to standards or use obsolete and ad-hoc mechanisms. We show that, most of the time, keystores using weak or custom implementations for the key derivation function F open the way to password brute-forcing. We have empirically measured the speed-up that the attacker achieves when these flawed keystores are used and we show that, in some cases, brute-forcing is three orders of magnitude faster with respect to the keystores based on standard mechanisms. We even found keystores using the deprecated cipher RC2 that enables an attacker to brute-force the 40-bit long cryptographic

key in a matter of hours using a standard desktop computer.

Our analysis has also pointed out problems related to availability and malicious code execution, which are caused by *type-flaws* in the keystore, *i.e.*, bugs in which an object of a certain type is interpreted as one of a different type. In particular, by directly tampering with the keystore file, an attacker could trigger denial of service (DoS) attacks or even arbitrary code execution. Interestingly, we also found that the use of standard key derivation functions can sometimes enable DoS attacks. These functions are parametrized by the number of internal iterations, used to slow down brute-forcing, which is stored in the keystore file. If the number of iterations is set to a very big integer, the key derivation function will hang, blocking the whole application.

Unless stated otherwise, our findings refer to Oracle JDK 8u144 and Bouncy Castle 1.57, the two latest releases at the time of the first submission of this work in August 2017.

4.1.1 Contributions

Our contributions can be summarized as follows:

- (i) we define a general threat model for password-protected keystores and we distill a set of significant security properties and consequent rules that any secure keystore should adhere to;
- (ii) we perform a thoughtful analysis of seven keystores, we report undocumented details about their cryptographic implementations and we classify keystores based on our proposed properties and rules;
- (iii) we report on unpublished attacks and weaknesses in the analyzed keystores. For each attack we point out the corresponding violations of our proposed properties and rules and we provide a precise attacker model;
- (iv) we empirically estimate the speed-up due to bad cryptographic implementations and we show that, in some cases, this allows to decrease the guessing time of three orders of magnitude with respect to the most resistant keystore, and four orders of magnitude with respect to NIST recommendations; interestingly, the attack on Oracle JKS keystore that we present in this chapter, and we previously mentioned in a blog post [55], has been recently integrated into the Hashcat password recovery tool;
- (v) we discuss the advancements on the security of Oracle and Bouncy Castle keystore implementations following our responsible disclosure. The Oracle Security Team acknowledged the reported issues by assigning two CVE IDs [120, 121] and released partial fixes in the October 2017 Critical Patch Update [54]. Other fixes are expected to be released in January 2018 [132]. Bouncy Castle developers patched some of the reported vulnerabilities in version 1.58. As of November 2017, remaining issues are being addressed in the development repository.

4.1.2 Structure of the Chapter

We discuss related work in Section 4.2; in Section 4.3 we define the security properties of interest, the rules for the design of secure keystores and the threat model; in Section 4.4 we report on our analysis of seven Java keystores; in Section 4.5 we describe unpublished attacks on the analyzed keystores; in Section 4.6 we make an empirical comparison of the password cracking speed among the keystores; in Section 4.7 we discuss the improvements implemented by Oracle and Bouncy Castle following our responsible disclosure; finally, in Section 4.8 we draw some concluding remarks.

4.2 Related Work

Cooijmans *et al.* [51] have studied various key storage solutions in Android, either provided as an operating system service or through the Bouncy Castle cryptographic library. The threat model is very much tailored to the Android operating system and radically different from the one we consider in this chapter. Offline brute-forcing, for example, is only discussed marginally in the paper. Interestingly, authors show that under a root attacker (*i.e.*, an attacker with root access to the device), the Bouncy Castle software implementation is, in some respect, more secure than the Android OS service using TrustZone's capabilities, because of the possibility to protect the keystore with a user-supplied password. Differently from our work, the focus of the paper is not on the keystore design and the adopted cryptographic mechanisms.

Sabt *et al.* [150] have recently found a forgery attack in the Android KeyStore service, an Android process that offers a keystore service to applications and is out of the scope of our work. However, similarly to our results, the adopted encryption scheme is shown to be weak and not compliant to the recommended standards, enabling a forgery attack that make apps use insecure cryptographic keys, voiding any benefit of cryptography.

Li *et al.* [110] have analyzed the security of web password managers. Even if the setting is different, there are some interesting similarities with keystores. In both settings a password is used to protect sensitive credentials, passwords in one case and keys in the other. So the underlying cryptographic techniques are similar. However the kind of vulnerabilities found in the paper are not related to cryptographic issues. Gasti *et al.* [74] have studied the format of password manager databases. There is some similarity with our work for what concerns the threat model, *e.g.*, by considering an attacker that can tamper with the password database. However, the setting is different and the paper does not account for cryptographic weaknesses and brute-forcing attacks.

Many papers have studied password resistance to guessing, e.g., [102, 40, 183, 191]. While this is certainly a very important subject, our work takes a complementary perspective: we analyze whether Java keystores provide a sufficient resistance to brute-forcing, compared to existing standards and recommendations. Of course, using a tremendously weak password would make it possible for the attacker to guess it, independently of the keystore implementation. Similarly, if the password is very long and with high entropy, the guess

would be infeasible anyway. However, when a password is reasonably strong, the actual implementation makes a difference: brute-force is prevented only when key derivation is done accordingly to recommendations.

Kelsey *et al.* introduced the notion of *key stretching*, a mechanism to increase the time of brute-forcing for low entropy keys [103]. The basic idea is that key derivation should iterate the core derivation function l times so to multiply the computational cost of brute-forcing by l and make it equivalent to the cost of brute-forcing a password with additional $log_2 l$ bits. Intuitively, through this strategy, brute-forcing each password requires the same time as brute-forcing l passwords. Combined with standard random salting (to prevent precomputation of keys), key stretching effectively slows down brute-forcing, and prevents guessing the password even when its complexity is not very high. This idea is at the base of modern, state-of-the-art key derivation functions. In [1, 187, 23], this mechanism has been formalized and analyzed, providing formal evidence of its correctness. Standard key derivation functions are all based on key stretching and salting to slow down brute-forcing [122, 146]. In our work we advocate the use of these standard mechanisms for keystores security.

4.3 Security Properties and Threat Model

In this section, we identify a set of fundamental security properties that should be guaranteed by any keystore (Section 4.3.1). We then distill rules that should be followed when designing a keystore in order to achieve the desired security properties (Section 4.3.2). Finally, we introduce the threat model covering a set of diverse attacker capabilities that enable realistic attack scenarios (Section 4.3.3).

4.3.1 Security Properties

We consider standard security properties such as confidentiality and integrity of keys and keystore entries. Breaking confidentiality of sensitive keys allows an attacker to intercept all the encrypted traffic or to impersonate the user. Breaking integrity has similar severe consequences as it might allow an attacker to import fake CA certificates and old expired keys. Additionally, since the access to a keystore is mediated by a software library or an application, we also consider the effect that a keystore has on the execution environment. Thus, we require the following properties:

P1 Confidentiality of encrypted entries

P2 Integrity of keystore entries

P3 System integrity

Property **P1** states that the value of an encrypted entry should be revealed only to authorized users, who know the correct decryption password. According to **P2**, keystore entries should be modified, created or removed only by authorized users, who know the correct integrity password, usually called *store password*. Property **P3** demands that the usage of a keystore

should always be tolerated by the environment, *i.e.*, interacting with a keystore, even when provided by an untrusted party, should not pose a threat to the system, cause misbehaviours or hang the application due to an unsustainable performance hit.

A keystore file should be secured similarly to a password file: the sensitive content should not be disclosed even when the file is leaked to an attacker. In fact, it is often the case that keystores are shared in order to provide the necessary key material to various corporate services and applications. Thus, in our threat model we will always assume that the attacker has read access to the keystore file (*cf.* Section 4.3.3). For this reason we require that the above properties hold even in the presence of offline attacks. The attacker might, in fact, brute-force the passwords that are used to enforce confidentiality and integrity and, consequently, break the respective properties.

4.3.2 Design Rules

We now identify a set of core rules that should be embraced by the keystore design in order to provide the security guarantees of Section 4.3.1:

R1 Use standard, state-of-the-art cryptography

R2 Choose strong, future-proof cryptographic parameters, while maintaining acceptable performance

R3 Enforce a typed keystore format

Rule **R1** dictates the use of modern and verified algorithms to achieve the desired keystore properties. It is well-known that the design of custom cryptography is a complex task even for experts, whereas standard algorithms have been carefully analyzed and withstood years of cracking attempts by the cryptographic community [16]. In this context, the National Institute of Standards and Technology (NIST) plays a prominent role in the standardization of cryptographic algorithms and their intended usage [17], engaging the cryptographic community to update standards according to cryptographic advances. For instance, NIST declared SHA1 unacceptable to use for digital signatures beginning in 2014, and more recently, urged all users of Triple-DES to migrate to AES for encryption as soon as possible [164] after the findings published in [26]. The KDF function recommended by NIST [163] is PBKDF2, as defined in the PKCS#115 standard, which supersedes the legacy PBKDF1. Another standard KDF function is defined in PKCS#1112, although it has been deprecated for confidentiality purposes in favour of PBKDF2.

Key derivation functions combine the password with a randomly generated salt and iteratively apply a pseudorandom function (*e.g.*, a hash function) to produce a cryptographic key. The salt allows the generation of a large set of keys corresponding to each password [187], while the high number of iterations is introduced to hinder brute-force attacks by significantly increasing computational times. Rule **R2** reflects the need of choosing parameters to keep pace with the state-of-the-art in cryptographic research and the advances in computational capabilities. The latest NIST draft on Digital Identity Guidelines [75] sets the minimum KDF iteration count to 10,000 and the salt size to 32 bits. However, such lower bounds on

the KDF should be significantly raised for critical keys according to [163] which suggests to set the number of iterations as high as can be tolerated by the environment, while maintaining acceptable performance. For instance, Apple iOS derives the decryption key for the device from the user password using a KDF with an iteration count calculated by taking into account the computational capabilities of the hardware and the impact on the user experience [9].

Finally, rule **R3** states that the keystore format must provide strong typing for keystore content, such that cryptographic objects are stored and read unambiguously. Despite some criticism over the years [81], the PKCS#1112 standard embraces this principle providing precise types for storing many cryptography objects. Additionally, given that keystore files are supposed to be accessed and modified by different parties, applications parsing the keystore format must be designed to be robust against malicious crafted content.

Interestingly, not following even one of the aforementioned rules may lead to a violation of confidentiality and integrity of the keystore entries. For instance, initializing a secure KDF with a constant or empty salt, which violates only **R2**, would allow an attacker to precompute the set of possible derived keys and take advantage of *rainbow tables* [131] to speed up the brute-force of the password. On the other hand, a KDF with strong parameters is useless once paired with a weak cipher, since it is easier to retrieve the encryption key rather than brute-forcing the password. In this case only **R1** is violated.

Additionally, disrespecting Rule **R3** may have serious consequences on system integrity (breaking property **P3**), which range from applications crashing due to parsing errors while loading a malicious keystore to more severe scenarios where the host is compromised. An attacker exploiting type-flaw bugs could indirectly gain access to the protected entries of a keystore violating the confidentiality and integrity guarantees. System integrity can additionally be infringed by violating Rule **R2** with an inadequate parameter choice, *e.g.*, an unreasonably high iteration count value might hang the application, slow down the system or prevent the access to cryptographic objects stored in a keystore file due to an excessive computational load. In Section 4.5 we show how noncompliance to these rules translate into concrete attacks.

4.3.3 Threat Model

In our standard attacker model we always assume that the attacker has read access to the keystore file, either authorized or by means of a data leakage. We also assume that the attacker is able to perform offline brute-force attacks using a powerful system of her choice.

We now present a list of interesting attacker settings, that are relevant with respect to the security properties defined in Section 4.3.1:

S1 Write access to the keystore

S2 Integrity password is known

S3 Confidentiality password of an entry is known

S4 Access to previous legitimate versions of the keystore file

Setting **S1** may occur when the file is shared over a network filesystem, *e.g.*, in banks and large organizations. Since keystores include mechanisms for password-based integrity checks, it might be the case that they are shared with both read and write permissions, to enable application that possess the appropriate credentials (*i.e.*, the integrity password) to modify them. We also consider the case **S2** in which the attacker possesses the integrity password. The password might have been leaked or discovered through a successful brute-force attack. The attacker might also know the password as an insider, *i.e.*, when she belongs to the organization who owns the keystore. Setting **S3** refers to a scenario in which the attacker knows the password used to encrypt a sensitive object. Similarly to the previous case, the password might have been accessed either in a malicious or in honest way. For example, the password of the key used to sign the `apk` of an Android application [7] could be shared among the developers of the team.

In our experience, there exists a strong correlation between **S2** and **S3**. Indeed, several products and frameworks use the same password both for confidentiality and for integrity, *e.g.*, Apache Tomcat for TLS keys and IBM WebSphere for LTPA authentication. Additionally, the standard utility for Java keystores management (`keytool`) supports this practice when creating a key: the tool invites the user to just press the `RETURN` key to reuse the store password for encrypting the entry.

To summarize, our standard attacker model combined with **S1**-**S3** covers both reading and writing capabilities of the attacker on the keystore files together with the possibility of passwords leakage. On top of these settings, we consider the peculiar case **S4** that may occur when the attacker has access to backup copies of the keystore or when the file is shared over platforms supporting version control such as *Dropbox*, *ownCloud* or *Seafile*.

4.4 Analysis of Java Keystores

The Java platform exposes a comprehensive API for cryptography through a *provider*-based framework called Java Cryptography Architecture (JCA). A provider consists of a set of classes that implement cryptographic services and algorithms, including keystores. In this section, we analyze the most common Java software keystores implemented in the Oracle JDK and in a widespread cryptographic library called Bouncy Castle that ships with a provider compatible with the JCA. In particular, since the documentation was not sufficient to assess the design and cryptographic strength of the keystores, we performed a comprehensive review of the source code exposing, for the first time, implementation details such as on-disk file structure and encoding, standard and proprietary cryptographic mechanisms, default and hard-coded parameters.

For reader convenience, we provide a brief summary of the cryptographic mechanisms and acronyms used in this section: Password-Based Encryption (PBE) is an encryption scheme in which the cryptographic key is derived from a password through a Key Derivation Function (KDF); a Message Authentication Code (MAC) authenticates data through a secret key and HMAC is a standard construction for MAC which is based on cryptographic hash

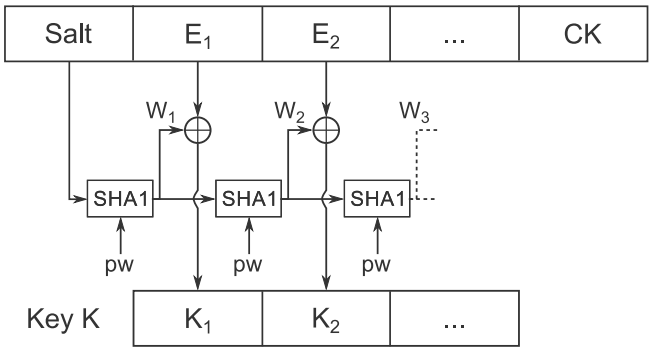


FIGURE 4.1: Decryption in the custom stream cipher used by JKS.

functions; Cipher Block Chaining (CBC) and Counter with CBC-MAC (CCM) are two standard modes of operation for block ciphers, the latter is designed to provide both authenticity and confidentiality.

4.4.1 Oracle Keystores

The Oracle JDK offers three keystore implementations, namely JKS, JCEKS and PKCS12, which are respectively made available through the providers SUN, SunJCE and SunJSSE [52]. While JKS and JCEKS rely on proprietary algorithms to enforce both the confidentiality and the integrity of the saved entries, PKCS12 relies on open standard format and algorithms as defined in [145].

JKS

Java KeyStore (JKS) is the first official implementation of a keystore that appeared in Java since the release of JDK 1.2. To the time, it is still the *default* keystore in Java 8 when no explicit choice is made. It supports encrypted private key entries and public key certificates stored in the clear. The file format consists of a header containing the magic file number, the keystore version and the number of entries, which is followed by the list of entries. The last part of the file is a digest used to check the integrity of the keystore. Each entry contains the type of the object (key or certificate) and the label, followed by the cryptographic data.

Private keys are encrypted using a custom stream cipher designed by Sun, as reported in the OpenJDK source code. In order to encrypt data, a keystream W is generated in 20-bytes blocks with W_0 being a random salt and $W_i = SHA1(password||W_{i-1})$. The encrypted key E is computed as the XOR of the private key K with the keystream W, hence K and E share the same length. The ciphertext is then prepended with the salt and appended with the checksum $CK = SHA1(password||K)$. The block diagram for decryption is shown in Figure 4.1.

The integrity of the keystore is achieved through a custom hash-based mechanism: JKS computes the SHA1 hash of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and the keystore content. The result is then checked against the 20 bytes digest at the end of the keystore file.

JCEKS

Java Cryptography Extension KeyStore (JCEKS) has been introduced after the release of JDK 1.2 in the external Java Cryptography Extension (JCE) package and merged later into the standard JDK distribution from version 1.4. According to the Java documentation, it is an alternate proprietary keystore format to JKS "that uses much stronger encryption in the form of Password-Based Encryption with Triple-DES" [91]. Besides the improved PBE mechanism, it allows for storing also symmetric keys.

The file format is almost the same of JKS with a different magic number in the file header and support for the symmetric key type. The integrity mechanism is also borrowed from JKS.

JCEKS stores certificates as plaintext, while the PBE used to encrypt private keys, inspired by PBES1 [122], is based on 20 MD5 iterations and a 64 bits salt. Given that Triple-DES is used to perform the encryption step, the key derivation process must be adapted to produce cipher parameters of the adequate size. In particular, JCEKS splits the salt in two halves and applies the key derivation process for each of them. The first 192 bits of the combined 256 bits result are used as the Triple-DES key, while the remaining 64 bits are the initialization vector.

PKCS12

The PKCS12 keystore supports both private keys and certificates, with support for secret keys added in Java 8. Starting from Java 9, Oracle replaced JKS with PKCS12 as the default keystore type [94].

The keystore file is encoded as an ASN.1 structure according to the specification given in [145]. It contains the version number of the keystore, the list of keys and the certificates. The last part of the keystore contains an HMAC (together with the parameters for its computation) used to check the integrity of the entire keystore by means of a password.

The key derivation process, used for both confidentiality and integrity, is implemented as described in the PKCS#1112 standard [145] using SHA1 as hashing function, 1024 iterations and a 160 bit salt. Private keys and secret keys (when supported) are encrypted using Triple-DES in CBC mode. Certificates are encrypted as well in a single encrypted blob, using the RC2 cipher in CBC mode with a 40-bit key. While each key can be encrypted with a different password, all the certificates are encrypted reusing the store password.

4.4.2 Bouncy Castle Keystores

Bouncy Castle is a widely used open-source crypto API. As of 2014, it provides the base implementation for the crypto library used in the Android operating system [51]. It supports four different keystore types via the BC provider: BKS, UBER, BCPKCS12 and the new FIPS-compliant BCFKS. Similarly to the Oracle keystores, all the BC keystores rely on passwords to enforce confidentiality over the entries and to verify the integrity of the keystore file.

BKS

The Bouncy Castle Keystore (BKS) allows to store public/private keys, symmetric keys and certificates. The BKS keystore relies on a custom file structure to store the entries. The file contains the version number of the BKS keystore, the list of stored cryptographic entries and an HMAC, along with its parameters, computed over the entries as integrity check.

Only symmetric and private keys can be encrypted in BKS, with Triple-DES in CBC mode. The key derivation schema is taken from PKCS#1112 v1.0, using SHA1 as hashing function, a random number of iterations between 1024 and 2047 which is stored for each entry and a 160 bit salt.

The integrity of the keystore is provided by an HMAC using the same key derivation scheme used for encryption and applied to the integrity password. For backward compatibility, the current version of BKS still allows to load objects encrypted under a buggy PBE mechanism used in previous versions of the keystore[1]. If the key is recovered using an old mechanisms, it is immediately re-encrypted with the newer PBE scheme.

UBER

UBER shares most of its codebase with BKS, thus it supports the same types of entries and PBE. Additionally, it provides an extra layer of encryption for the entire keystore file, which means that all metadata around the keys and certificates are encrypted as well. The PBE mechanism used for encrypting the file is Twofish in CBC mode with a key size of 256 bits. The KDF is PKCS#1112 v1.0 with SHA1 using a 160 bits salt and a random number of iterations in the range 1024 and 2047.

The integrity of the keystore is checked after successful decryption using the store password. The plaintext consists of the keystore entries followed by their SHA1 checksum. UBER recomputes the hash of the keystore and compares it with the stored digest.

BCFKS

BCFKS is a new FIPS-compliant [166] keystore introduced in the version 1.56 of Bouncy Castle[2] offering similar features to UBER. This keystore provides support for secret keys in addition to asymmetric keys and certificates.

The entire keystore contents is encrypted using AES in CCM mode with a 256 bits key, so to provide protection against introspection. After the encrypted blob, the file contains a block with a HMAC-SHA512 computed over the encrypted contents to ensure the keystore integrity. The store password is used to derive the two keys for encryption and integrity.

All key derivation operations use PBKDF2 with HMAC-SHA512 as pseudorandom function, 512 bits of salt and 1024 iterations. Each key entry is separately encrypted with a different password using the same algorithm for the keystore confidentiality, while this possibility is not offered for certificates.

[1] `https://github.com/bcgit/bc-java/blob/master/prov/src/main/java/org/bouncycastle/jce/provider/BrokenPBE.java`

[2] `https://github.com/bcgit/bc-java/commit/80fd6825`

BCPKCS12

The BCPKCS12 keystore aims to provide a PKCS#1112-compatible implementation. It shares the same algorithms and default parameters for key derivation, cryptographic schemes and file structure of the Oracle JDK version detailed in Section 4.4.1. Compared to Oracle, the Bouncy Castle implementation lacks support for symmetric keys and the possibility to protect keys with different passwords, since all the entries and certificates are encrypted under the store password. The BC provider also offers a variant of the PKCS#1112 keystore that allows to encrypt certificates using the same PBE of private keys, that is Triple-DES in CBC mode.

4.4.3 Keystores Adoption

We have analyzed 300 Java projects supporting keystores that are hosted on Github to estimate the usage of the implementations examined in this work. Applications range from amateur software to well-established libraries developed by Google, Apache and Eclipse.

We searched for occurrences of known patterns used to instantiate keystores in the code of each project. We have found that JKS is the most widespread keystore with over 70% of the applications supporting it. PKCS12 is used in 32% of the analyzed repositories, while JCEKS adoption is close to 10%. The Bouncy Castle keystores UBER and BCPKCS12 are used only in 3% of the projects, while BKS can be found in about 6% of the examined software. Finally, since BCFKS is a recent addition to the Bouncy Castle library, none of the repositories is supporting it.

4.4.4 Summary

In Tables 4.1 and 4.2 we summarize the features and the algorithms (rows) offered by the keystore implementations (columns) analyzed in this section. Table 4.1 does not contain the row "Store Encryption" since none of the JDK keystores provides protection against introspection.

To exemplify, by reading Table 4.1 we understand that the JCEKS keystore of the SunJCE provider relies on a custom PBE mechanism based on MD5 using only 20 iterations to derive the Triple-DES key for the encryption of keys. The ✓ mark shows that the keystore supports secret keys, while ✗ denotes that certificates cannot be encrypted.

4.5 Attacks

In the previous section, we have shown that the analyzed keystores use very diverse key derivation functions and cryptographic mechanisms and, in several cases, they do not adhere to standards or use obsolete and ad-hoc mechanisms. We now discuss how this weakens the overall security of the keystore and enables or facilitates attacks. In particular, we show that keystores using weak or ad-hoc implementations for password-based encryption or integrity checks open the way to password brute-forcing. During the in-depth analysis of keystores,

		JKS	JCEKS	PKCS12
Provider		Sun	SunJCE	SunJSSE
Support for secret keys		✗	✓	✓*
Keys PBE	KDF	Custom (SHA1)	Custom (MD5)	PKCS12 (SHA1)
	Salt	160b	64b	160b
	Iterations	-	20	1024
	Cipher	Stream cipher	3DES (CBC)	3DES (CBC)
	Key size	-	192b	192b
Certificates PBE	KDF	✗	✗	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Cipher			RC2 (CBC)
	Key size			40b
Store Integrity	KDF	SHA1 with password	SHA1 with password	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Mechanism			HMAC (SHA1)

* since JDK 1.8

TABLE 4.1: Summary of Oracle JDK keystores (Oracle JDK 8u144 and below).

		BKS	UBER	BCFKS	BCPKCS12
Provider		Bouncy Castle	Bouncy Castle	Bouncy Castle	Bouncy Castle
Support for secret keys		✓	✓	✓	✗
Keys PBE	KDF	PKCS12 (SHA1)	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b	160b	512b	160b
	Iterations	1024–2047	1024–2047	1024	1024
	Cipher	3DES (CBC)	3DES (CBC)	AES (CCM)	3DES (CBC)
	Key size	192b	192b	256b	192b
Certificates PBE	KDF	✗	✗	✗	PKCS12 (SHA1)
	Salt				160b
	Iterations				1024
	Cipher				RC2 / 3DES (CBC)
	Key size				40b / 192b
Store Encryption	KDF	✗	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	✗
	Salt		160b	512b	
	Iterations		1024–2047	1024	
	Cipher		Twofish (CBC)	AES (CCM)	
	Key size		256b	256b	
Store Integrity	KDF	PKCS12 (SHA1)	SHA1 after decrypt	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b		512b	160b
	Iterations	1024–2047		1024	1024
	Mechanism	HMAC (SHA1)		HMAC (SHA512)	HMAC (SHA1)

TABLE 4.2: Summary of Bouncy Castle keystores (bc 1.57 and below).

Algorithm 2 JKS 1-block Crack

```
1:  procedure JKS_1BlockCrack(Salt, E_{1..n}, CK)
2:      known_plaintext ← 0x30 || length(E)
3:      test_bytes ← known_plaintext ⊕ E_1
4:      for password in passwords do
5:          W_1 ← SHA1(password || Salt)
6:          if W_1 = test_bytes then
7:              K ← Decrypt(Salt, E, password)
8:              checksum ← SHA1(password || K)
9:              if CK = checksum then
10:                 return password
11:             end if
12:         end if
13:     end for
14: end procedure
```

we have also found security flaws that can be exploited in practice to mount denial of service and code execution attacks.

Attacks in this section are organized according to the security properties violated, as defined in Section 4.3.1. For each attack we provide a detailed description discussing the attacker settings and the rules that are not followed by the keystore implementation (*cf.* Section 4.3.2). We conclude with some general security considerations that are not specific to any particular attack.

Table 4.3 provides a high-level overview of the properties which are guaranteed by the analyzed keystores with respect to the attacks presented in this section. We consider versions of Oracle JDK and Bouncy Castle before and after disclosing our findings to the developers. Specifically, we refer to JDK 8u144 and 8u152 for Oracle, while version 1.57 of Bouncy Castle is compared against the development repository as of November 28, 2017.[3] We use the symbol → to point out improvements in newer versions. Details of the changes are listed in Section 4.7. The ✓✓ symbol denotes that a property is satisfied by the keystore under any attacker setting and the implementation adhere to the relevant design rules listed in Section 4.3.2. We use ✓ when no clear attack can be mounted but design rules are not completely satisfied, *e.g.* a legacy cipher like Triple-DES is used. The ✗ symbol indicates that the property is broken under the standard attacker model. When a property is broken only under a specific setting **Sx**, we report it in the table as ✗_{Sx}. If a more powerful attack is enabled by additional settings, we clarify in the footnotes.

As an example, consider the system integrity property (**P3**) in the JCEKS keystore: up to JDK 8u144 included, write capabilities (**S1**) allow to DoS the application loading the keystore; when integrity and key confidentiality passwords are known (**S2** and **S3**), the attacker can also achieve arbitrary code execution on the system (*cf.* note 3 in the table). The rightmost side of the arrow indicates that JDK 8u152 does not include mitigations against the code execution attack.

[3] https://github.com/bcgit/bc-java/tree/8ed589d

	JKS	JCEKS	PKCS12	BKS	UBER	BCFKS	BCPKCS12
(**P1**) Entries confidentiality	✗	✗ → ✓	✓[1]	✓	✓	✓ → ✓✓	✓[1]
(**P2**) Keystore integrity	✗[2]	✗[2]	✓ → ✓✓	✓	✓	✓ → ✓✓	✓ → ✓✓
(**P3**) System integrity	✓✓	✗^{3}_{S1} → $\text{✗}_{\text{S1-3}}$	✗_{S1} → ✓✓	✗_{S1}	✓✓	✓✓	✗_{S1} → ✓✓

[1] only confidentiality of certificates can be violated
[2] under additional settings **S1** or **S4** it might be possible to use rainbow tables
[3] under additional settings **S2** and **S3** it is possible to achieve arbitrary code execution on JDK $\leq$ 8u152

Legend:
- ✓✓ property is always satisfied
- ✓ no clear attacks but rules not completely satified
- ✗ property is broken in the standard attacker model
- ✗_{Sx} property is broken under a attacker setting **Sx**

TABLE 4.3: Properties guaranteed by keystores with respect to attacks, before and after updates listed in Section 4.7.

4.5.1 Attacks on Entries Confidentiality (P1)

JKS Password Cracking

The custom PBE mechanism described in Section 4.4.1 for the encryption of private keys is extremely weak. The scheme requires only one SHA1 hash and a single XOR operation to decrypt each block of the encrypted entry resulting in a clear violation of rule **R1**. Since there is no mechanism to increase the amount of computation needed to derive the key from the password, also rule **R2** is neglected.

Despite the poor cryptographic scheme, each attempt of a brute-force password recovery attack would require to apply SHA1 several times to derive the whole keystream used to decrypt the private key. As outlined in Figure 4.1, a successful decryption is verified by matching the last block (CK) of the protected entry with the hash of the password concatenated with the decrypted key. For instance, a single password attempt to decrypt a 2048 bit RSA private key entry requires over 60 SHA1 operations.

We found that such password recovery attack can be greatly improved by exploiting the partial knowledge over the plaintext of the key. Indeed, the ASN.1 structure of a key entry enables to efficiently test each password with a single SHA1 operation. In JKS, private keys are serialized as DER-encoded ASN.1 objects, along the PKCS#111 standard [123]. For instance, an encoded RSA key is stored as a sequence of bytes starting with byte `0x30` which represent the ASN.1 type `SEQUENCE` and a number of bytes representing the length of the encoded key. Since the size of the encrypted key is the same as the size of the plaintext, these bytes are known to the attacker. On average, given n bytes of the plaintext it is necessary to continue decryption beyond the first block only for one password every 256^n attempts.

The pseudocode of the attack is provided in Algorithm 2, using the same notation introduced in Section 4.4.1. We assume that the algorithm is initialized with the salt, all the blocks of the encrypted key and the checksum. The XOR operation between the known plaintext and the first encrypted block (line 3) is performed only once for all the possible passwords. As a halt condition, the result is then compared against the digest of the salt concatenated to the tested password (lines 5-6). To further verify the correctness of the password, a standard decrypt is performed.

A comparison between the standard cracking attack and our improved version is depicted in Figure 4.2. From the chart it is possible to see that the cost of the single block attack (referred to as 1-block) is independent from the size of the encrypted entry, while the number of operations required to carry out the standard attack is bound to the size of the DER-encoded key. As an example, for a 4096 bit private RSA key, the 1-block approach is two orders of magnitude faster than the standard one.

Based on our findings, that we previously mentioned in a blog post [55], this attack has been recently integrated into Hashcat 3.6.0[4] achieving a speed of 8 billion password tries/sec with a single NVIDIA GTX 1080 GPU.

[4] https://hashcat.net/forum/thread-6630.html

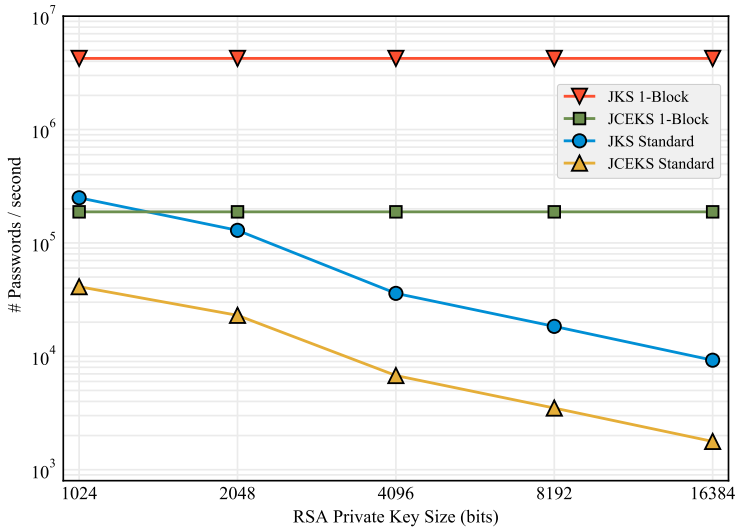


FIGURE 4.2: Performance comparison of password cracking for private RSA keys on JKS and JCEKS using both the standard and the improved 1-block method on a Intel Core i7 6700 CPU.

JCEKS Password Cracking

The PBE mechanism discussed in Section 4.4.1 uses a custom KDF that performs 20 MD5 iterations to derive the encryption key used in the Triple-DES cipher. This value is three orders of magnitude lower than the iteration count suggested in [75], thus violating both rules **R1** and **R2**. Given that keys are DER-encoded as well, it is possible to speed up a brute-force attack using a technique similar to the one discussed for JKS. Figure 4.2 relates the standard cracking speed to the single block version. Notice that the cost of a password-recovery attack is one order of magnitude higher than JKS in both variants due to the MD5 iterations required by the custom KDF of JCEKS.

PKCS#1112 Certificate Key Cracking

Oracle PKCS12 and BCPKCS12 keystores allow for the encryption of certificates. The PBE is based on the KDF defined in the PKCS#1112 standard paired with the legacy RC2 cipher in CBC mode with a 40 bit key, resulting in a clear violation of rule **R1**. Due to the reduced key space, the protection offered by the KDF against offline attacks can be voided by directly brute-forcing the cryptographic key. Our serialized tests, performed using only one core of an Intel Core i7 6700 CPU, show that the brute-force performance is 8,300 $passwords/s$ for password testing (consisting of a KDF and decryption run), while the key cracking speed is 1,400,000 $keys/s$. The worst-case scenario that requires the whole 40-bits key space to be exhausted, requires about 9 days of computation on our system. This time can be reduced to about 1 day by using all eight cores of our processor. We estimate that a modern high-end GPU should be able to perform this task in less than one hour.

Notice, however, that although finding the key so easily makes the encryption of certificates pointless, an attacker cannot use the key value to reduce the complexity of cracking the

integrity password since the random salt used by the KDF makes it infeasible to precompute the mapping from passwords to keys.

4.5.2 Attacks on Keystore Integrity (P2)

JKS/JCEKS Integrity Password Cracking

The store integrity mechanism used by both JKS and JCEKS (*cf.* Section 4.4.1) only relies on the SHA1 hash digest of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and with the keystore data. In contrast with rule **R1**, this technique based on a single application of SHA1 enables to efficiently perform brute-force attacks against the integrity password. Section 4.6 reports on the computational effort required to attack the integrity mechanism for different sizes of the keystore file.

Additionally, since SHA1 is based on the Merkle-Damgård construction, this custom approach is potentially vulnerable to extension attacks [63]. For instance, it may be possible for an attacker with write access to the keystore (**S1**) to remove the original digest at the end of the file, extend the keystore content with a forged entry and recompute a valid hash without knowing the keystore password. Fortunately, this specific attack is prevented in JKS and JCEKS since the file format stores the number of entries in the keystore header.

JKS/JCEKS Integrity Digest Precomputation

The aforementioned construction to ensure the integrity of the keystore suffers from an additional problem. Assume the attacker has access to an empty keystore, for example when an old copy of the keystore file is available under a file versioning storage (**S4**). Alternatively, as special case of **S1**, the attacker may be able to read the file, but the interaction with the keystore is mediated by an application that allows to remove entries without disclosing the store password. This file consists only of a fixed header followed by the SHA1 digest computed using the password, the string "`Mighty Aphrodite`" and the header itself. Given that there is no random salting in the digest computation, it would be possible to mount a very efficient attack to recover the integrity password by exploiting precomputed hash chains, as done in rainbow tables [131].

4.5.3 Attacks on System Integrity (P3)

JCEKS Code Execution

A secret key entry is stored in a JCEKS keystore as a Java object having type `SecretKey`. First, the key object is serialized and wrapped into a `SealedObject` instance in an encrypted form; next, this object is serialized again and saved into the keystore.

When the keystore is loaded, all the serialized Java objects stored as secret key entries are evaluated. An attacker with write capabilities (**S1**) may construct a malicious entry containing a Java object that, when deserialized, allows her to execute arbitrary code in the

application context. Interestingly, the attack is not prevented by the integrity check since keystore integrity is verified only after parsing all the entries.

The vulnerable code can be found in the `engineLoad` method of the class `JceKeyStore` implemented by the SunJCE provider.[5] In particular, the deserialization is performed on lines 837-838 as follows:

```
// read the sealed key
try {
    ois = new ObjectInputStream(dis);
    entry.sealedKey =
        (SealedObject) ois.readObject();
    ...
```

Notice that the cast does not prevent the attack since it is performed after the object evaluation.

To stress the impact of this vulnerability, we provide three different attack scenarios: i) the keystore is accessed by multiple users over a shared storage. An attacker can replace or add a single entry of the keystore embedding the malicious payload, possibly gaining control of multiple hosts; ii) a remote application could allow its users to upload keystores for cryptographic purposes, such as importing certificates or configuring SSL/TLS. A crafted keystore loaded by the attacker may compromise the remote system; iii) an attacker may even forge a malicious keystore and massively spread it like a malware using email attachments or instant messaging platforms. Users with a default application associated to the keystore file extension (*e.g.*, keystore inspection utilities such as KSE [6]) have a high probability of being infected just by double clicking on the received keystore. Interestingly, all the malicious keystores generated during our tests did not raise any alert on antivirus tools completing a successful scan by *virustotal.com*.

We checked the presence of the vulnerability from Java 6 onwards. We were able to achieve arbitrary command execution on the host with JDK $\leq$ 7u21 and JDK $\leq$ 8u20 by forging a payload with the tool `ysoserial`.[7] Newer versions are still affected by the vulnerability, but the JDK classes exploited to achieve code execution have been patched. Since the deserialization occurs within a Java core class, the classpath is restricted to bootstrap and standard library classes. However, by embedding a recursive object graph in a JCEKS entry, an attacker can still hang the deserialization routine consuming CPU indefinitely and thus causing a DoS in the target machine. We were able to mount this attack on any version of the Oracle JDK $\leq$ 8u144.

The implementation choice for storing secret keys in JCEKS is a clear violation of Rule **R3**, since these entities are essentially stored as Java code. The correct approach is to adopt standard formats and encodings, such as the PKCS#118 format used in the PKCS12 keystore.

[5] `http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/5534221c23fc/src/share/classes/com/sun/crypto/provider/JceKeyStore.java`

[6] `http://keystore-explorer.org`

[7] `https://github.com/frohoff/ysoserial`

JCEKS Code Execution After Decryption

When the attacker knows the integrity password and the confidentiality password of a secret key entry (**S2**, **S3**) in addition to **S1**, the previous attack can be further improved to achieve arbitrary command execution even on the latest, at the time of writing, Java 8 release (8u152). This variant of the attack assumes that the application loading the JCEKS keystore makes use of one of the widespread third-party libraries supported by `ysoserial`, such as *Apache Commons Collections* or the *Spring* framework: such libraries have been found [169] to contain vulnerable gadget chains that can be exploited by the malicious payload.

When a `SealedObject` wrapping a secret key is successfully loaded and decrypted, an additional deserialization call is performed over the decrypted content. The `SealedObject` class extends the classpath to allow the deserialization of any class available in the application scope, including third-party libraries. By exploiting this second deserialization step, an attacker may construct more powerful payloads to achieve command execution.

The exploitation scenarios are similar to the ones already discussed in the previous variant of the attack. Additionally, we point out that even an antivirus trained to detect deserialization signatures would not be able to identify the malicious content since the payload is stored in encrypted form in the keystore.

DoS by Integrity Parameters Abuse

Many keystores rely on a keyed MAC function to ensure the integrity of their contents. The parameters of the KDF used to derive the key from the store password are saved inside the file. Thus, an attacker with write capabilities (**S1**) may tamper with the KDF parameters to affect the key derivation phase that is performed before assessing the integrity of the keystore file. In particular, the attacker may set the iteration count to an unreasonably high value in order to perform a DoS attack on applications loading the keystore.

We found that Oracle PKCS12, BKS and BCPKCS12 implementations are affected by this problem. Starting from valid keystore files, we managed to set the iteration count value to $2^{31} - 1$. Loading such keystores required around 15 minutes at full CPU usage on a modern computer. According to [163] the iteration count should not impact too heavily on the user-perceived performance, thus we argue that this is a violation of Rule **R2**.

4.5.4 Bad Design Practices

During our analysis we found that some of the keystores suffered from bad design decisions and implementation issues that, despite not leading to proper attacks, could lead to serious security consequences.

Our review of the Oracle PKCS12 keystore code showed that the KDF parameters are not treated uniformly among MAC, keys and certificates. During a store operation, the Oracle implementation does not preserve the original iteration count and salt size for MAC and certificates that has been found at load time in the input keystore file. Indeed, iteration

count and salt size are silently set to the hard-coded values of 1024 and 20 byte, respectively. Since this keystore format is meant to be interoperable, this practice could have security consequences when dealing with keystores generated by third-party tools. For instance, PKCS12-compatible keystores generated by OpenSSL default to 2048 iterations: writing out such keystore with the Oracle JDK results in halving the cost of a password recovery attack.

The Bouncy Castle BCPKCS12 implementation suffers a similar problem: in addition to MAC and certificate parameters, also the iteration count and the salt size used for private keys are reverted to default values when the keystore is saved to disk. Following our report to the Bouncy Castle developers, this behaviour is currently being addressed in the next release by preserving the original parameters whenever possible.[8]

Lastly, the construction of the integrity mechanism for the UBER keystore could cause an information leakage under specific circumstances. After a successful decryption using the store password, UBER recomputes the hash of the keystore and compares it with the stored digest. This MAC-then-encrypt approach is generally considered a bad idea, since it can lead to attacks if, for example, there is a perceptible difference in behavior (an error message, or execution time) between a decryption that fails because the padding is invalid, or a decryption that fails because the hash is invalid (a so-called padding oracle attack [167]).

4.5.5 Security Considerations

We now provide general considerations on the security of Java keystores. The first one is about using the same password for different purposes. If the integrity password is also used to ensure the confidentiality of encrypted entries, then the complexity of breaking either the integrity or the confidentiality of stored entries turns out to be the one of attacking the weakest mechanism. For instance, we consider a keystore where cracking the integrity password is more efficient than recovering the password used to protect sensitive entries: as shown in Section 4.6, this is the case of PKCS12 and BCPKCS12 keystores. Under this setting, sensitive keys can be leaked more easily by brute-forcing the integrity password.

Although this is considered a bad practice in general [110], all the keystores analyzed permit the use of the same password to protect sensitive entries and to verify the integrity of the keystore. This practice is indeed widespread [74] and, as already stated in Section 4.3.3, prompted by `keytool` itself. Furthermore, our analysis found that the BCPKCS12 keystore forcibly encrypts keys and certificates with the store password. For these reasons, we argue that using the same password for integrity and confidentiality is not a direct threat to the security of stored keys when both mechanisms are resistant to offline attacks and a strong password is used. Still the security implications of this practice should be seriously considered.

The second consideration regards how the integrity of a keystore is assessed. Indeed, a poorly designed application may bypass the integrity check on keystores by providing a null or empty password to the Java `load()` function. All the Oracle keystores analyzed in the previous section and BouncyCastle BKS are affected by this problem. On the other

[8] `https://github.com/bcgit/bc-java/commit/ebe1b25a`

hand, keystores providing protection to entries inspection, such as UBER and BCFKS, cannot be loaded with an empty password since the decryption step would fail. Lastly, BCPKCS12 throws an exception if an attempt of loading a file with an empty password is made. Clearly, if the integrity check is omitted, an attacker can trivially violate Property **P2** by altering, adding or removing any entry saved in the clear. Conversely, the integrity of encrypted sensitive keys is still provided by the decryption mechanism that checks for the correct padding sequence at the end of the plaintext. Since the entries are typically encoded (*e.g.*, in ASN.1), a failure in the parse routine could also indicate a tampered ciphertext.

We also emphasize that the 1-block cracking optimization introduced in 4.5.1 is not limited to JKS and JCEKS. Indeed, by leveraging the structure of saved entries, all the analyzed keystores enable to reduce the cost of the decrypt operation to check the correctness of a password. However, excluding JKS and JCEKS, this technique only provides a negligible speed-up on the remaining keystores given that the KDF is orders of magnitude slower than the decrypt operation.

Finally, we point out that the current design of password-based keystores cannot provide a proper key-revocation mechanism without a trusted third-party component. For instance, it may be the case that a key has been leaked in the clear and subsequently substituted with a fresh one in newer versions of a keystore file. Under settings **S1** and **S4**, an attacker may replace the current version of a keystore with a previously intercepted valid version, thus restoring the exposed key. The integrity mechanism is indeed not sufficient to distinguish among different versions of a keystore protected with the same store password. For this reason, the store password must be updated to a fresh one every time a rollback of the keystore file is not acceptable by the user, which is typically the case of a keystore containing a revoked key.

4.6 Estimating Brute-Force Speed-Up

We have discussed how weak PBEs and integrity checks in keystores can expose passwords to brute-forcing. In this section we make an empirical comparison of the cracking speed to bruteforce both the confidentiality and integrity mechanisms in the analyzed keystores. We also compute the speed-up with respect to BCFKS, as it is the only keystore using a standard and modern KDF, *i.e.*, PBKDF2, which provides the best brute-forcing resistance. Notice, however, that the latest NIST draft on Digital Identity Guidelines [75] sets the minimum KDF iteration count to 10,000 which is one order of magnitude more than what is used in BCFKS (*cf.* Table 4.2). Thus all the speed-up values should be roughly multiplied by 10 if compared with a baseline implementation using PBKDF2 with 10,000 iterations.

It is out of the scope of this work to investigate brute-forcing strategies. Our tests only aim at comparing, among the different keystores, the actual time to perform the key derivation step and the subsequent cryptographic operations, including the check to assess key correctness. Our study is independent of the actual password guessing strategy adopted by the attacker.

Algorithm 3 Confidentiality password cracking benchmark

```
1: procedure BENCHCONFIDENTIALITY(test_duration)
2:     encrypted_entry ← (B_1, ..., B_2000)
3:     passwords ← (pw_1, ..., pw_n)                    ▷ all 10-bytes passwords
4:     salt ← constant
5:     counter ← 0
6:     while ELAPSEDTIME < test_duration do
7:         password ← next(passwords)
8:         key ← KDF_key(password, salt)
9:         iv ← KDF_iv(password, salt)                  ▷ not in JKS, BCFKS
10:        plaintext ← DECRYPTBLOCK(encrypted_entry, key, iv)
11:        VERIFYKEY(plaintext)
12:        counter ← counter + 1
13:    end while
14:    return counter
15: end procedure
```

Algorithm 4 Integrity password cracking benchmark

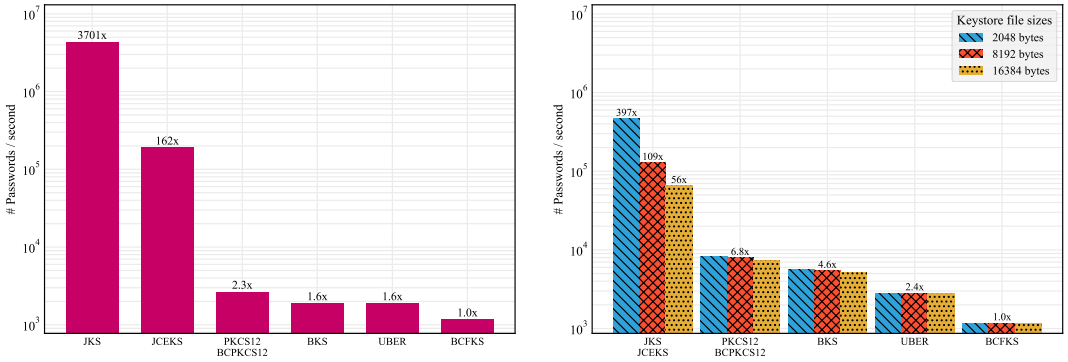
```
1: procedure BENCHINTEGRITY(test_duration)
2:     keystore_content_small ← (B_1, ..., B_2048)
3:     keystore_content_medium ← (B_1, ..., B_8192)
4:     keystore_content_large ← (B_1, ..., B_16384)
5:     passwords ← (pw_1, ..., pw_n)                    ▷ all 10-bytes passwords
6:     salt ← constant
7:     counter_(small,medium,large) ← 0
8:     for all keystore_content, counter do
9:         while ELAPSEDTIME < test_duration do
10:            password ← next(passwords)
11:            key ← KDF_mac(password, salt)            ▷ not in JKS, JCEKS
12:            mac ← MAC(keystore_content, key)
13:            VERIFYMAC(mac)
14:            counter ← counter + 1
15:        end while
16:    end for
17:    return counter_(small,medium,large)
18: end procedure
```

4.6.1 Test Methodology

We developed a compatible C implementation of the key decryption and the integrity check for each keystore type. Each implementation is limited to the minimum steps required to check the correctness of a test password. This procedure is then executed in a timed loop to evaluate the cracking speed. Algorithms 3 and 4 show the pseudocode of our implementations. Note that, in both algorithms, we set the password length to 10 bytes because it is an intermediate value between trivial and infeasible. Similarly, since the iteration count in BKS and UBER is chosen randomly in the range 1024 and 2047, we set it to the intermediate value 1536.

Confidentiality

The confidentiality password brute-forcing loop (Algorithm 3) is divided into three steps: key derivation, decryption and a password correctness check. The last step is included in



(A) Speed comparison of password recovery attack for key encryption (confidentiality).

(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 4.3: Comparison of keystores password cracking speed. Bar labels indicate the speed-up to the strongest BCFKS baseline.

the loop only to account for its computational cost in the results. Both PBES1 (PKCS#115) and PKCS#1112 password-based encryption schemes, used in all keystores but BCFKS, require to run the KDF twice to derive the decryption key and the IV. On the other hand, in BCFKS the initialization vector is not derived from the password but simply stored with the ciphertext. During our tests we set *encrypted_entry* to a fixed size to resemble an on-disk entry containing a 2048 bits RSA key. However, in Section 4.5.1 we have shown how the partial knowledge of the plaintext structure of a JKS key entry can be leveraged to speed-up brute-forcing. This shortcut can be applied to all the analyzed keystores in order to decrypt only the first block of *encrypted_entry*. For this reason, the key size becomes irrelevant while testing for a decryption password.

Integrity

Similarly, the integrity password cracking code (Algorithm 4) is divided into three steps: key derivation, a hash/MAC computation and the password correctness check. The key derivation step is run once to derive the MAC key in all keystores, with the exception of JKS and JCEKS where the password is fed directly to the hash function (*cf.* Section 4.4.1). As described later in this section, the speed of KDF plus MAC calculation can be highly influenced by the keystore size, thus we performed our tests using a *keystore_content* of three different sizes: 2048, 8192 and 16384 bytes.

Test configuration

We relied on standard implementations of the cryptographic algorithms to produce comparable results: the OpenSSL library (version 1.0.2g) provides all the needed hash functions, ciphers and KDFs, with the exception of Twofish where we used an implementation from the author of the cipher.[9] All the tests were performed on a desktop computer running Ubuntu 16.04 and equipped with an Intel Core i7 6700 CPU; source code of our implementations has

[9] `https://www.schneier.com/academic/twofish/download.html`

been compiled with GCC 5.4 using `-O3 -march=native` optimizations. We run each benchmark on a single CPU core because the numeric results can be easily scaled to a highly parallel systems. To collect solid and repeatable results each benchmark has been run for 60 seconds.

4.6.2 Results

The charts in Figure 4.3 show our benchmarks on the cracking speed for confidentiality (Figure 4.3a) and integrity (Figure 4.3b). On the x-axis there are the 7 keystore types: we group together different keystores when the specific mechanism is shared among the implementations, *i.e.*, PKCS12/BCPKCS12 for both confidentiality and integrity and JKS/JCEKS for integrity. On the y-axis we report the number of tested passwords per second doing a serial computation on a single CPU core: note that the scale of this axis is logarithmic. We stress that our results are meant to provide a relative, inter-keystore comparison rather than an absolute performance index. To this end, a label on top of each bar indicates the speed-up relative to the strongest BCFKS baseline. Absolute performance can be greatly improved using both optimized parallel code and more powerful hardware which ranges from dozens of CPU cores or GPUs to programmable devices such as FPGA or custom-designed ASICs [101, 48, 114].

Confidentiality

From the attack described in Section 4.5.1, it follows that cracking the password of an encrypted key contained in JKS - the default Java keystore - is at least three orders of magnitude faster than in BCFKS. Even without a specific attack, recovering the same password from JCEKS is over one hundred times faster due to its low (20) iteration count. By contrast, the higher value (1024 or 1024-2047) used in PKCS12, BKS and UBER translates into a far better offline resistance as outlined in the chart.

Integrity

Similar considerations can be done for the integrity password resistance. Finding this password in all keystores but JKS is equivalent, or even faster than breaking the confidentiality password. Moreover, the performance of these keystores is influenced by the size of the file due to the particular construction of the MAC function (*cf.* Section 4.4.1). The speed gain (w.r.t. confidentiality) visible in PKCS12, BKS and UBER is caused by the missing IV derivation step which, basically, halves the number or KDF iterations. Interestingly, in BCFKS there is no difference between the two scores: since the whole keystore file is encrypted, we can reduce the integrity check to a successful decryption, avoiding the computation overhead of the HMAC on the entire file.

4.7 Disclosure and Security Updates

We have timely disclosed our findings to Oracle and Bouncy Castle developers in May 2017. The Oracle Security Team has acknowledged the reported issues with CVE IDs [120, 121]

and has released most of the fixes in the October 2017 Critical Patch Update (CPU) [54]. In the following list, we summarize the changes already published by Oracle:

- `keytool` suggests to switch to PKCS12 when JKS or JCEKS keystores are used;
- improved KDF strength of the PBE in JCEKS by raising the iteration count to 200,000. Added a ceiling value of 5 millions to prevent parameter abuse;
- in PKCS12 the iteration count has been increased to 50,000 for confidentiality and 100,000 for integrity. The same upper bound as in JCEKS is introduced;
- fixed the first JCEKS deserialization vulnerability described in Section 4.5.3 by checking that the object being deserialized is of the correct type, *i.e.*, `SealedObjectForKeyProtector`, and by imposing a recursion limit to prevent infinite loops.

Additionally, Oracle informed us that a fix for the second JCEKS deserialization vulnerability is planned for release in the January 2018 CPU [132].

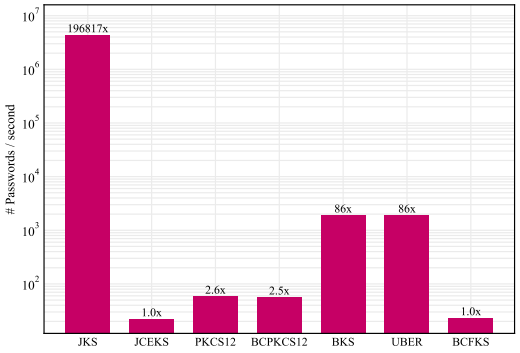
In version 1.58 of the library, Bouncy Castle developers fixed the parameter abuse vulnerability of BCPKCS12 by adding an optional Java system property that imposes an upper bound for the KDF iteration count. Moreover, they have committed in the development repository the following changes that will appear in version 1.59:

- in BCFKS, the iteration count is raised to 51,200 for both confidentiality and integrity;
- in BCPKCS12, the iteration count is increased to 51,200 and 102,400 for confidentiality and integrity, respectively.

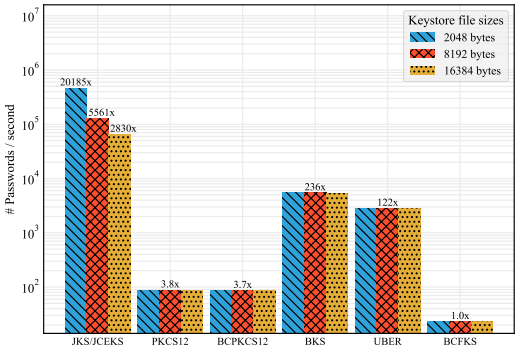
Table 4.3 outlines the improved security guarantess offered by keystore implemenations following the fixes released by Oracle and Bouncy Castle. Additionally, in Figure 4.4 we show the updated results of the brute-force resistance benchmarks to reflect the improved KDF parameters. JCEKS and BCFKS now offer the best resistance to offline brute-force attacks of the confidentiality password. However, JCEKS still provides the weakest integrity mechanism. Thus, if the same password is used both for key encryption and for keystore integrity, then the increased protection level can easily be voided by attacking the latter mechanism. On the other hand, both the confidentiality and the integrity mechanisms have been updated in PKCS12. This keystore, which is now the default in Java 9, offers a much higher security level with respect to the previous release.

4.8 Conclusion

Keystores are the standard way to store and manage cryptographic keys and certificates in Java applications. In the literature there is no in-depth analysis of keystore implementations and the documentation does not provide enough information to evaluate the security level offered by each keystore. Thus, developers cannot make a reasoned and informed choice among the available alternatives.



(A) Speed comparison of password recovery attack for key encryption (confidentiality).



(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 4.4: Revised password cracking benchmarks after library updates.

In this chapter we have thoroughly analyzed seven keystore implementations from the Oracle JDK and the Bouncy Castle library. We have described all the cryptographic mechanisms used to guarantee standard security properties on keystores, including offline attacks. We have pointed out that several implementations adopt non-standard mechanisms and we have shown how this can drastically speed-up the brute-forcing of the keystore passwords. Additionally, we reported new and unpublished attacks and defined a precise threat model under which they may occur. These attacks range from breaking the confidentiality of stored keys to arbitrary code execution on remote systems and denial of service. We also showed how a keystore can be potentially weaponized by an attacker to spread malware.

We have reported the security flaws to Oracle and Bouncy Castle. Most of the issues in the Oracle JDK have been fixed in the October 2017 Critical Patch Update [54] following CVE IDs [120, 121]. Similarly, Bouncy Castle developers committed changes to address several problems discussed in this work.

Following our analysis and succeeding fixes, it appears evident that the security offered by JKS, the default keystore in Java 8 and previous releases, is totally inadequate. Its improved version JCEKS still uses a broken integrity mechanism. For these reasons, we favorably welcome the decision of Oracle to switch to PKCS12 as the default keystore type in the recent Java 9 release. After the previously discussed updates this keystore results quite solid, although certificate protection is bogus and key encryption relies on legacy cryptography.

Alternatives provided by Bouncy Castle have been found to be less susceptible to attacks. Among the analyzed keystores, the updated BCFKS version clearly sets the standard from a security standpoint. Indeed, this keystore relies on modern algorithms, uses adequate cryptographic parameters and provides protection against introspection of keystore contents. Moreover, the development version of Bouncy Castle includes preliminary support for *scrypt* [135, 136] in BCFKS, a *memory-hard* function that requires significant amount of RAM. Considering the steady nature of keystore files, we argue that in addition to approved standard functions, it would be advisable to consider future-proof cryptographic primitives so to be more resistant against parallelized attacks [29, 34].

Conclusion

In this thesis 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. Then we presented WPSE, the first browser-side security monitor designed to address the security challenges of web protocols, and we showed that the security policies enforceable by WPSE are expressive enough to prevent a number of real-world attacks against the OAuth 2.0 authorization protocol. Our analysis is based both on a review of well-known attacks reported in the literature and an extensive experimental analysis in the wild, which exposed several undocumented security vulnerabilities fixable by WPSE in existing OAuth 2.0 implementations. We also showed that WPSE works flawlessly on the large majority of the websites we tested, concluding that the browser-side security monitoring of web protocols is both useful for security and feasible in practice.

For what concerns cryptographic applications, we presented two contributions aimed at preventing the leakage of sensitive keys. The first one solves the problem of leaking a key in the clear by exploiting particular sequences of legal cryptographic API calls. The approach presented is based on the analysis of cryptographic API logs. Log analysis is interesting because it has a very low impact on existing systems and thus it could be deployed on top of industrial systems, financial applications and critical infrastructures without major issues. Additionally, we have thoroughly analyzed seven keystore implementations, cryptographic storage facilities from the Oracle JDK and the Bouncy Castle library. We have assessed all the cryptographic mechanisms used to guarantee standard security properties on keystores and reported on new and unpublished critical attacks. These attacks range from breaking the confidentiality of stored keys to arbitrary code execution on remote systems and denial of service. We also showed how a keystore can be potentially weaponized by an attacker to spread malware. Finally, we discussed the advancements on the security of Oracle and Bouncy Castle keystore implementations following our responsible disclosure.

Bibliography

[1] M. Abadi and B. Warinschi. “Password-Based Encryption Analyzed”. In: *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*. 2005, pp. 664–676.

[2] M. Abadi and R. M. Needham. “Prudent Engineering Practice for Cryptographic Protocols”. In: *IEEE Trans. Software Eng.* 22.1 (1996), pp. 6–15.

[3] S. V. Acker, D. Hausknecht, and A. Sabelfeld. “Measuring login webpage security”. In: *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*. 2017, pp. 1753–1760.

[4] P. Adão, R. Focardi, and F. L. Luccio. “Type-Based Analysis of Generic Key Management APIs”. In: *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. 2013, pp. 97–111.

[5] D. Akhawe et al. “Towards a Formal Foundation of Web Security”. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*. 2010, pp. 290–304.

[6] R. Anderson. “The Correctness of Crypto Transaction Sets”. In: *8th International Workshop on Security Protocols*. Apr. 2000.

[7] *Android Studio User Guide: Sign Your App*. URL: `https://developer.android.com/studio/publish/app-signing.html`.

[8] *Apache Tomcat 7 Documentation: SSL/TLS Configuration*. 2017. URL: `https://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html`.

[9] Apple inc. *iOS Security Guide*. Tech. rep. Mar. 2017. URL: `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[10] A. Armando et al. “An authentication flaw in browser-based Single Sign-On protocols: Impact and remediations”. In: *Computers & Security* 33 (2013), pp. 41–58.

[11] A. Armando et al. “Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps”. In: *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008*. 2008, pp. 1–10.

[12] E. Athanasopoulos, V. Pappas, and E. P. Markatos. “Code-Injection Attacks in Browsers Supporting Policies”. In: *Proceedings of the 2009 IEEE Web 2.0 Security and Privacy Workshop*. 2009.

[13] C. Bansal et al. “Discovering concrete attacks on website authorization by formal analysis”. In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[14] C. Bansal et al. “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*. 2013, pp. 126–146.

[15] R. Bardou et al. “Efficient Padding Oracle Attacks on Cryptographic Hardware”. In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*. 2012, pp. 608–625.

[16] E. Barker. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf. Aug. 2016.

[17] E. Barker and A. Roginsky. *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths (Rev. 1)*. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf. Nov. 2015.

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

[19] A. Barth. *The Web Origin Concept*. http://tools.ietf.org/html/rfc6454. 2011.

[20] A. Barth, C. Jackson, and J. C. Mitchell. “Robust Defenses for Cross-Site Request Forgery”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 2008, pp. 75–88.

[21] D. Bates, A. Barth, and C. Jackson. “Regular Expressions Considered Harmful in Client-side XSS Filters”. In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*. 2010, pp. 91–100.

[22] L. Bauer et al. “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*. 2015.

[23] M. Bellare, T. Ristenpart, and S. Tessaro. “Multi-instance Security and Its Application to Password-Based Cryptography”. In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*. 2012, pp. 312–329.

[24] R. Benadjila, T. Calderon, and M. Daubignard. “Caml Crush: A PKCS#11 Filtering Proxy”. In: *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*. 2014, pp. 173–192. DOI: 10.1007/978-3-319-16763-3_11. URL: http://dx.doi.org/10.1007/978-3-319-16763-3_11.

[25] B. Beurdouche et al. “A Messy State of the Union: Taming the Composite State Machines of TLS”. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P 2015*. 2015, pp. 535–552.

[26] K. Bhargavan and G. Leurent. “On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016*. 2016, pp. 456–467. URL: http://doi.acm.org/10.1145/2976749.2978423.

[27] A. Bichhawat et al. “Information Flow Control in WebKit’s JavaScript Bytecode”. In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014, pp. 159–178.

[28] N. Bielova. “Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser”. In: *Journal of Logic and Algebraic Programming* 82.8 (2013), pp. 243–262.

[29] A. Biryukov, D. Dinu, and D. Khovratovich. “Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications”. In: *Proceedings of the 1st IEEE European Symposium on Security and Privacy, EuroS&P 2016*. 2016.

[30] P. Bisht and V. N. Venkatakrishnan. “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*. 2008, pp. 23–43.

[31] D. Bleichenbacher. “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”. In: *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO ’98*. 1998, pp. 1–12.

[32] A. Bohannon and B. C. Pierce. “Featherweight Firefox: Formalizing the Core of a Web Browser”. In: *USENIX Conference on Web Application Development, WebApps 2010*. 2010.

[33] M. Bond. “Attacks on Cryptoprocessor Transaction Sets”. In: *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES’01)*. Vol. 2162. LNCS. Paris, France: Springer, 2001, pp. 220–234.

[34] D. Boneh, H. Corrigan-Gibbs, and S. Schechter. “Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks”. In: *Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptology and Information Security, ASIACRYPT 2016*. 2016.

[35] M. Bortolozzo et al. “Attacking and Fixing PKCS#11 Security Tokens”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*. 2010, pp. 260–269.

[36] A. Bortz, A. Barth, and A. Czeskis. “Origin Cookies: Session Integrity for Web Applications”. In: *Web 2.0 Security & Privacy Workshop (W2SP 2011)*. 2011.

[37] M. Bugliesi, S. Calzavara, and R. Focardi. “Formal methods for web security”. In: *Journal of Logical and Algebraic Methods in Programming* (2017). To appear.

[38] M. Bugliesi et al. “CookiExt: Patching the browser against session hijacking attacks”. In: *Journal of Computer Security* 23.4 (2015), pp. 509–537.

[39] M. Bugliesi et al. “Provably Sound Browser-Based Enforcement of Web Session Integrity”. In: *Proceedings of the IEEE 27th Computer Security Foundations Symposium, CSF 2014*. 2014, pp. 366–380.

[40] W. E. Burr et al. *Electronic Authentication Guideline: Recommendations of the National Institute of Standards and Technology - Special Publication 800-63-1*. 2012.

[41] S. Calzavara, A. Rabitti, and M. Bugliesi. “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*. 2016, pp. 1365–1375.

[42] S. Calzavara et al. “Micro-policies for Web Session Security”. In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 179–193.

[43] S. Calzavara et al. “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*. 2014, pp. 189–200.

[44] S. Calzavara et al. “Surviving the Web: A Journey into Web Session Security”. In: *ACM Comput. Surv.* 50.1 (Mar. 2017), 13:1–13:34. ISSN: 0360-0300. DOI: 10.1145/3038923. URL: http://doi.acm.org/10.1145/3038923.

[45] M. Centenaro, R. Focardi, and F. L. Luccio. “Type-based analysis of key management in PKCS#11 cryptographic devices”. In: *Journal of Computer Security* 21.6 (2013), pp. 971–1007. DOI: 10.3233/JCS-130479. URL: http://dx.doi.org/10.3233/JCS-130479.

[46] M. Centenaro et al. “Type-Based Analysis of PIN Processing APIs”. In: *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09)*. Ed. by S. L. vol. 5789/2009. 2009, pp. 53–68.

[47] E. Y. Chen et al. “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*. 2011, pp. 227–238.

[48] R. Clayton and M. Bond. “Experience Using a Low-Cost FPGA Design to Crack DES Keys”. In: *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002*. 2002, pp. 579–592.

[49] J. Clulow. “On the Security of PKCS#11”. In: *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*. 2003, pp. 411–425.

[50] J. Clulow. “The Design and Analysis of Cryptographic APIs for Security Devices”. MA thesis. University of Natal, Durban, 2003.

[51] T. Cooijmans, J. de Ruiter, and E. Poll. "Analysis of Secure Key Storage Solutions on Android". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM 2014*. 2014, pp. 11–20.

[52] O. Corporation. *Java Cryptography Architecture, Standard Algorithm Name Documentation for JDK 8*. http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore. 2014.

[53] V. Cortier and G. Steel. "A generic security API for symmetric key management on cryptographic devices". In: *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*. Ed. by M. Backes and P. Ning. Vol. 5789. Lecture Notes in Computer Science. Saint Malo, France: Springer, Sept. 2009, pp. 605–620. DOI: 10.1007/978-3-642-04444-1_37. URL: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CS-esorics09.pdf.

[54] *Critical Patch Updates, Security Alerts and Third Party Bulletin*. 2017. URL: https://www.oracle.com/technetwork/topics/security/alerts-086861.html.

[55] Cryptosense S.A. *Mighty Aphrodite – Dark Secrets of the Java Keystore*. 2016. URL: https://cryptosense.com/mighty-aphrodite-dark-secrets-of-the-java-keystore/.

[56] M. CVE-2012-4929. *CRIME attack*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929. Sept. 2012.

[57] M. CVE-2014-0160. *Heartbleed bug*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Dec. 2013.

[58] A. Czeskis et al. "Lightweight Server Support for Browser-Based CSRF Protection". In: *Proceedings of the 22nd International World Wide Web Conference, WWW 2013*. 2013, pp. 273–284.

[59] I. Dacosta et al. "One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens". In: *ACM Transactions on Internet Technology* 12.1 (2012), pp. 1–24.

[60] S. Delaune, S. Kremer, and G. Steel. "Formal Analysis of PKCS#11 and Proprietary Extensions". In: *Journal of Computer Security* 18.6 (Nov. 2010), pp. 1211–1245. DOI: 10.3233/JCS-2009-0394. URL: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/DKS-jcs09.pdf.

[61] D. Devriese and F. Piessens. "Noninterference through Secure Multi-execution". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 2010, pp. 109–124.

[62] M. Dietz et al. "Origin-Bound Certificates: a Fresh Approach to Strong Client Authentication for the Web". In: *Proceedings of the 21th USENIX Security Symposium, USENIX 2012*. 2012, pp. 317–331.

[63] Y. Dodis, T. Ristenpart, and T. Shrimpton. “Salvaging Merkle-Damgård for Practical Applications”. In: *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2009*. 2009, pp. 371–388.

[64] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions in Information Theory* 2.29 (Mar. 1983), pp. 198–208.

[65] ECMA. *ECMAScript Language Specification Standard, 5.1 Edition*. 2011.

[66] EFF. *HTTPS Everywhere*. `https://www.eff.org/https-everywhere`. Electronic Frontier Foundation, 2011.

[67] S. Fahl et al. “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*. 2014, pp. 507–512.

[68] D. Fett, R. Küsters, and G. Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, pp. 1204–1215.

[69] D. Fett, R. Küsters, and G. Schmitz. “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*. 2014, pp. 673–688.

[70] D. Fett, R. Küsters, and G. Schmitz. “SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015, pp. 1358–1369.

[71] R. Focardi, F. Luccio, and G. Steel. “An Introduction to Security API Analysis”. In: *FOSAD*. 2010, pp. 35–65.

[72] R. Focardi and M. Squarcina. “Run-Time Attack Detection in Cryptographic APIs”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. Aug. 2017, pp. 176–188. DOI: `10.1109/CSF.2017.33`.

[73] S. Fröschle and G. Steel. “Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data”. In: *Revised Selected Papers of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS’09)*. Ed. by P. Degano and L. Viganò. Vol. 5511. Lecture Notes in Computer Science. York, UK: Springer, Aug. 2009, pp. 92–106. DOI: `10.1007/978-3-642-03459-6_7`.

[74] P. Gasti and K. B. Rasmussen. “On the Security of Password Manager Database Formats”. In: *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*. 2012, pp. 770–787.

[75] P. A. Grassi et al. *Digital Identity Guidelines: Authentication and Lifecycle Management*. https://pages.nist.gov/800-63-3/sp800-63b.html#sec5. 2017.

[76] W. D. Groef et al. "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*. 2012, pp. 748–759.

[77] T. Groß, B. Pfitzmann, and A. Sadeghi. "Browser Model for Security Analysis of Browser-Based Protocols". In: *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*. 2005, pp. 489–508.

[78] T. Groß, B. Pfitzmann, and A. Sadeghi. "Proving a WS-federation passive requestor profile with a browser model". In: *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005, Fairfax, VA, USA, November 11, 2005*. 2005, pp. 54–64.

[79] A. Guha, S. Krishnamurthi, and T. Jim. "Using Static Analysis for Ajax Intrusion Detection". In: *Proceedings of the 18th International Conference on World Wide Web, WWW 2009*. 2009, pp. 561–570.

[80] M. V. Gundy and H. Chen. "Noncespaces: Using Randomization to Defeat Cross-site Scripting Attacks". In: *Computers & Security* 31.4 (2012), pp. 612–628.

[81] P. Gutmann. "Lessons Learned in Implementing and Deploying Crypto Software". In: *Proceedings of the 11th USENIX Security Symposium*. 2002, pp. 315–325. ISBN: 1-931971-00-5. URL: http://dl.acm.org/citation.cfm?id=647253.720291.

[82] P. A. Hallgren, D. T. Mauritzson, and A. Sabelfeld. "GlassTube: A Lightweight Approach to Web Application Integrity". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013*. 2013, pp. 71–82.

[83] D. Hardt. *The OAuth 2.0 Authorization Framework*. http://tools.ietf.org/html/rfc6749. 2012.

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

[85] D. Hedin, L. Bello, and A. Sabelfeld. "Information-flow security for JavaScript and its APIs". In: *Journal of Computer Security* 24.2 (2016), pp. 181–234.

[86] D. Hedin et al. "JSFlow: Tracking Information Flow in JavaScript and its APIs". In: *Proceedings of the 29th Symposium on Applied Computing, SAC 2014*. 2014, pp. 1663–1671.

[87] M. Heiderich et al. "Scriptless Attacks: Stealing the Pie Without Touching the Sill". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 2012, pp. 760–771.

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

[89] B. Ippolito. *JSONP*. http://json-p.org/. 2015.

[90] C. Jackson and A. Barth. "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks". In: *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*. 2008, pp. 525–534.

[91] *Java Cryptography Architecture (JCA) Reference Guide*. 2016. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[92] K. Jayaraman et al. "ESCUDO: A Fine-Grained Protection Model for Web Browsers". In: *Proceedings of the 2010 International Conference on Distributed Computing Systems, ICDCS 2010*. 2010, pp. 231–240.

[93] *JDK 7 Security Enhancements*. 2016. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-7.html.

[94] *JDK 9 Early Access Release Notes*. 2017. URL: http://jdk.java.net/9/release-notes.

[95] T. Jim, N. Swamy, and M. Hicks. "Defeating Script Injection Attacks with Browser-enforced Embedded Policies". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*. 2007, pp. 601–610.

[96] M. Johns, B. Stock, and S. Lekies. "A Tale of the Weaknesses of Current Client-Side XSS Filtering". In: *Blackhat USA 2014*. 2014.

[97] M. Johns and J. Winter. "RequestRodeo: Client Side Protection against Session Riding". In: *Proceedings of the OWASP Europe 2006 Conference* (2006), pp. 5–17.

[98] M. Johns et al. "BetterAuth: Web Authentication Revisited". In: *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012*. 2012, pp. 169–178.

[99] M. Johns et al. "Reliable Protection Against Session Fixation Attacks". In: *Proceedings of the 26th ACM Symposium on Applied Computing, SAC 2011*. 2011, pp. 1531–1537.

[100] N. Jovanovic, E. Kirda, and C. Kruegel. "Preventing Cross Site Request Forgery Attacks". In: *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, SecureComm 2006*. 2006, pp. 1–10.

[101] J. P. Kaps and C. Paar. "Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine". In: *Proceedings of the 5th Annual International Workshop in Selected Areas in Cryptography, SAC'98*. 1999, pp. 234–247.

[102] P. G. Kelley et al. "Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms". In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P 2012*. 2012, pp. 523–537.

[103] J. Kelsey et al. “Secure Applications of Low-Entropy Keys”. In: *Proceedings of the 1st International Workshop on Information Security, ISW ’97*. 1997, pp. 121–134.

[104] W. Khan et al. “Client Side Web Session Integrity as a Non-interference Property”. In: *Proceedings of the 10th International Conference on Information Systems Security, ICISS 2014*. 2014, pp. 89–108.

[105] E. Kirda et al. “Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006*. 2006, pp. 330–337.

[106] S. Kremer, R. Künnemann, and G. Steel. “Universally Composable Key-Management”. In: *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. 2013, pp. 327–344.

[107] S. Kremer, G. Steel, and B. Warinschi. “Security for Key Management Interfaces”. In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. 2011, pp. 266–280.

[108] W. Li and C. J. Mitchell. “Analysing the Security of Google’s Implementation of OpenID Connect”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. 2016, pp. 357–376.

[109] W. Li and C. J. Mitchell. “Security Issues in OAuth 2.0 SSO Implementations”. In: *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*. 2014, pp. 529–541.

[110] Z. Li et al. “The Emperor’s New Password Manager: Security Analysis of Web-based Password Managers”. In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 465–479.

[111] J. Ligatti, L. Bauer, and D. Walker. “Edit automata: enforcement mechanisms for run-time security policies”. In: *Int. J. Inf. Sec.* 4.1-2 (2005), pp. 2–16.

[112] M. T. Louw and V. N. Venkatakrishnan. “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*. 2009, pp. 331–346.

[113] M. T. Louw et al. “SafeScript: JavaScript Transformation for Policy Enforcement”. In: *Proceedings of the 18th Nordic Conference on Secure IT Systems, NordSec 2013*. 2013, pp. 67–83.

[114] I. Magaki et al. “ASIC Clouds: Specializing the Datacenter”. In: *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA 2016*. 2016, pp. 178–190.

[115] C. Mainka et al. “SoK: Single Sign-On Security–An Evaluation of OpenID Connect”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. 2017.

[116] Z. Mao, N. Li, and I. Molloy. "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*. 2009, pp. 238–255.

[117] G. Maone. *The NoScript Firefox Extension*. `http://noscript.net/`. 2004.

[118] M. Marlinspike. "New Tricks for Defeating SSL in Practice". In: *BlackHat DC 2009*. 2009.

[119] L. A. Meyerovich and V. B. Livshits. "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*. 2010, pp. 481–496.

[120] MITRE. *CVE-2017-10345*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10345`. Oct. 2017.

[121] MITRE. *CVE-2017-10356*. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10356`. Oct. 2017.

[122] K. Moriarty, B. Kaliski, and A. Rusch. *PKCS#5: Password-Based Cryptography Specification (Version 2.1)*. `https://www.ietf.org/rfc/rfc8018.txt`. Jan. 2017.

[123] K. Moriarty et al. *PKCS#1: RSA Cryptography Specifications (Version 2.2)*. `https://www.ietf.org/rfc/rfc8017.txt`. Nov. 2016.

[124] Mozilla. *Same-Origin Policy*. `http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`. 2015.

[125] Y. Nadji, P. Saxena, and D. Song. "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. 2009.

[126] E. V. Nava and D. Lindsay. "Our Favorite XSS Filters and How to Attack Them". In: *Blackhat USA 2009*. 2009.

[127] N. Nikiforakis, Y. Younan, and W. Joosen. "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*. 2010, pp. 200–218.

[128] N. Nikiforakis et al. "SessionShield: Lightweight Protection against Session Hijacking". In: *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011*. 2011, pp. 87–100.

[129] N. Nikiforakis et al. "You are what you include: large-scale evaluation of remote javascript inclusions". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 2012, pp. 736–747.

[130] T. Oda et al. "SOMA: Mutual Approval for Included Content in Web Pages". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 2008, pp. 89–98.

[131] P. Oechslin. “Making a Faster Cryptanalytic Time-Memory Trade-Off”. In: *Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology, CRYPTO 2003*. 2003, pp. 617–630.

[132] Oracle Corporation. Private communication. Oct. 2017.

[133] OWASP. *HttpOnly*. `https://www.owasp.org/index.php/HttpOnly`. 2014.

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

[135] C. Percival. “Stronger Key Derivation via Sequential Memory-Hard Functions”. In: (May 2009).

[136] C. Percival and S. Josefsson. *The scrypt Password-Based Key Derivation Function*. `https://tools.ietf.org/html/rfc7914`. Aug. 2016.

[137] P. H. Phung, D. Sands, and A. Chudnov. “Lightweight Self-protecting JavaScript”. In: *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2009*. 2009, pp. 47–60.

[138] T. Pietraszek and C. V. Berghe. “Defending Against Injection Attacks Through Context-Sensitive String Evaluation”. In: *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, RAID 2005*. 2005, pp. 124–145.

[139] *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. `http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html`. OASIS Standard. Apr. 2015.

[140] V. Rajani et al. “Information Flow Control for Event Handling and the DOM in Web Browsers”. In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. 2015, pp. 366–379.

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

[142] G. Richards et al. “The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications”. In: *Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP 2011*. 2011, pp. 52–78.

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

[144] RSA laboratories. *PKCS#11 v2.30: Cryptographic Token Interface Standard*. `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm`. Apr. 2009.

[145] RSA laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.0)*. June 1999.

[146] RSA laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.1)*. Oct. 2012.

[147] P. D. Ryck et al. "Automatic and Precise Client-Side Protection against CSRF Attacks". In: *Proceedings of the 16th European Symposium on Research in Computer Security, ESORICS 2011*. 2011, pp. 100–116.

[148] P. D. Ryck et al. "CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests". In: *Proceedings of Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010*. 2010, pp. 18–34.

[149] P. D. Ryck et al. "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*. 2012, pp. 59–72.

[150] M. Sabt and J. Traoré. "Breaking into the KeyStore: A Practical Forgery Attack Against Android KeyStore". In: *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016), Part II*. 2016, pp. 531–548.

[151] G. Scerri and R. Stanley-Oakes. "Analysis of Key Wrapping APIs: Generic Policies, Computational Security". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 281–295.

[152] F. B. Schneider. "Enforceable security policies". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (2000), pp. 30–50.

[153] B. Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[154] J. Selvi. "Bypassing HTTP Strict Transport Security". In: *BlackHat DC 2014*. 2014.

[155] K. Singh et al. "Practical End-to-End Web Content Integrity". In: *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012*. 2012, pp. 659–668.

[156] J. Somorovsky et al. "On Breaking SAML: Be Whoever You Want to Be". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 2012, pp. 397–412.

[157] *Spring Crypto Utils Documentation*. 2017. URL: `http://springcryptoutils.com/keystore.html`.

[158] D. Stefan et al. "Protecting Users by Confining JavaScript with COWL". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*. 2014, pp. 131–146.

[159] B. Stock and M. Johns. "Protecting users against XSS-based password manager abuse". In: *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*. 2014, pp. 183–194.

[160] S. Sun and K. Beznosov. "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 2012, pp. 378–390.

[161] S. Tang, N. Dautenhahn, and S. T. King. "Fortifying web-based applications automatically". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 2011, pp. 615–626.

[162] M. F. Theofanos and S. L. Pfleeger. "Guest Editors' Introduction: Shouldn't All Security Be Usable?" In: *IEEE Security & Privacy* 9.2 (2011), pp. 12–17.

[163] M. S. Turan et al. *Recommendation for Password-Based Key Derivation. Part 1: Storage Applications*. http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf. Dec. 2010.

[164] *Update to Current Use and Deprecation of TDEA*. 2017. URL: https://beta.csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA.

[165] S. Van Acker et al. "WebJail: Least-privilege Integration of Third-party Components in Web Mashups". In: *Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011*. 2011, pp. 307–316.

[166] A. Vassilev. *Annex A: Approved Security Functions for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*. http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf. Apr. 2016.

[167] S. Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Proceedings of the 21st International Conference on the Theory and Applications of Cryptographic Techniques Advances in Cryptology, EUROCRYPT 2002*. 2002, pp. 534–546.

[168] P. Vogt et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Proceedings of the 14th Network and Distributed System Security Symposium, NDSS 2007*. 2007.

[169] *Vulnerability Note VU#576313*. 2015. URL: https://www.kb.cert.org/vuls/id/576313.

[170] W3C. *Cascading Style Sheets*. http://www.w3.org/Style/CSS/. 2014.

[171] W3C. *Content Security Policy*. http://www.w3.org/TR/CSP/. 2012.

[172] W3C. *Content Security Policy Level 2*. https://www.w3.org/TR/CSP2/. 2015.

[173] W3C. *Cross-Origin Resource Sharing*. http://www.w3.org/TR/cors. 2014.

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

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

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

[177] W3C. *HTML5: A Vocabulary and Associated APIs for HTML and XHTML*. http://www.w3.org/TR/html5/. 2014.

[178] W3C. *Mixed Content*. http://www.w3.org/TR/2015/CR-mixed-content-20151008/. 2015.

[179] R. Wang, S. Chen, and X. Wang. "Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". In: *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. 2012, pp. 365–379.

[180] R. Wang et al. "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization". In: *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. 2013, pp. 399–314.

[181] *WebLogic Integration 7.0: Configuring the Keystore*. URL: http://docs.oracle.com/cd/E13214_01/wli/docs70/b2bsecur/keystore.htm.

[182] J. Weinberger, A. Barth, and D. Song. "Towards Client-side HTML Security Policies". In: *6th USENIX Workshop on Hot Topics in Security, HotSec 2011*. 2011.

[183] M. Weir et al. "Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*. 2010, pp. 162–175.

[184] M. Weissbacher, T. Lauinger, and W. K. Robertson. "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*. 2014, pp. 212–233.

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

[186] R. Yang et al. "Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*. 2016, pp. 651–662.

[187] F. F. Yao and Y. L. Yin. "Design and Analysis of Password-Based Key Derivation Functions". In: *IEEE Transactions on Information Theory* 51.9 (2005), pp. 3292–3297.

[188] P. Youn et al. *Robbing the bank with a theorem prover*. Tech. rep. UCAM-CL-TR-644. University of Cambridge, Aug. 2005.

[189] D. Yu et al. "JavaScript Instrumentation for Browser Security". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*. 2007, pp. 237–249.

[190] M. Zalewski. *Postcards From the Post-XSS World*. http://lcamtuf.coredump.cx/postxss/. 2011.

[191] Y. Zhang, F. Monrose, and M. K. Reiter. “The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*. 2010.

[192] X. Zheng et al. “Cookies Lack Integrity: Real-World Implications”. In: *Proceedings of the 24th USENIX Security Symposium, USENIX 2015*. 2015, pp. 707–721.

[193] Y. Zhou and D. Evans. “SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities”. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014, pp. 495–510.

[194] Y. Zhou and D. Evans. “Why Aren’t HTTP-only Cookies More Widely Deployed?” In: *Web 2.0 Security and Privacy Workshop, W2SP 2010*. 2010.