Fakultät für Informatik und Mathematik

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

Web-based Secure Application Control

Bastian Braun

February, 2015

Eingereicht an der Fakultät für Informatik und Mathematik der Universität Passau

Reviewers: Prof. Dr. Joachim Posegga
Prof. Dr. Frank Piessens

Abstract

The world wide web today serves as a distributed application platform. Its origins, however, go back to a simple delivery network for static hypertexts. The legacy from these days can still be observed in the communication protocol used by increasingly sophisticated clients and applications. This thesis identifies the actual security requirements of modern web applications and shows that HTTP does not fit them: user and application authentication, message integrity and confidentiality, control-flow integrity, and application-to-application authorization. We explore the other protocols in the web stack and work out why they can not fill the gap. Our analysis shows that the underlying problem is the connectionless property of HTTP. However, history shows that a fresh start with web communication is far from realistic. As a consequence, we come up with approaches that contribute to meet the identified requirements.

We first present impersonation attack vectors that begin before the actual user authentication, i.e. when secure web interaction and authentication seem to be unnecessary. Session fixation attacks exploit a responsibility mismatch between the web developer and the used web application framework. We describe and compare three countermeasures on different implementation levels: on the source code level, on the framework level, and on the network level as a reverse proxy.

Then, we explain how the authentication credentials that are transmitted for the user login, i.e. the password, and for session tracking, i.e. the session cookie, can be complemented by browser-stored and user-based secrets respectively. This way, an attacker can not hijack user accounts only by phishing the user's password because an additional browser-based secret is required for login. Also, the class of well-known session hijacking attacks is mitigated because a secret only known by the user must be provided in order to perform critical actions.

In the next step, we explore alternative approaches to static authentication credentials. Our approach implements a trusted UI and a mutually authenticated session using signatures as a means to authenticate requests. This way, it establishes a trusted path between the user and the web application without exchanging reusable authentication credentials. As a downside, this approach requires support on the client side and on the server side in order to provide maximum protection. Another approach avoids client-side support but can not implement a trusted UI and is thus susceptible to phishing and clickjacking attacks.

Our approaches described so far increase the security level of all web communication at all time. This is why we investigate adaptive security policies that fit the actual risk instead of permanently restricting all kinds of communication including non-critical requests. We develop a smart browser extension that detects when the user is authenticated on a website meaning that she can be impersonated because all requests carry

her identity proof. Uncritical communication, however, is released from restrictions to enable all intended web features.

Finally, we focus on attacks targeting a web application's control-flow integrity. We explain them thoroughly, check whether current web application frameworks provide means for protection, and implement two approaches to protect web applications: The first approach is an extension for a web application framework and provides protection based on its configuration by checking all requests for policy conformity. The second approach generates its own policies ad hoc based on the observed web traffic and assuming that regular users only click on links and buttons and fill forms but do not craft requests to protected resources.

Zusammenfassung

Das heutige World Wide Web ist eine verteilte Plattform für Anwendungen aller Art: von einfachen Webseiten über Online Banking, E-Mail, multimediale Unterhaltung bis hin zu intelligenten vernetzten Häusern und Städten. Seine Ursprünge liegen allerdings in einem einfachen Netzwerk zur Übermittlung statischer Inhalte auf der Basis von Hypertexten. Diese Ursprünge lassen sich noch immer im verwendeten Kommunikationsprotokoll HTTP identifizieren. In dieser Arbeit untersuchen wir die Sicherheitsanforderungen moderner Web-Anwendungen und zeigen, dass HTTP diese Anforderungen nicht erfüllen kann. Zu diesen Anforderungen gehören die Authentifikation von Benutzern und Anwendungen, die Integrität und Vertraulichkeit von Nachrichten, Kontrollflussintegrität und die gegenseitige Autorisierung von Anwendungen. Wir untersuchen die Web-Protokolle auf den unteren Netzwerk-Schichten und zeigen, dass auch sie nicht die Sicherheitsanforderungen erfüllen können. Unsere Analyse zeigt, dass das grundlegende Problem in der Verbindungslosigkeit von HTTP zu finden ist. Allerdings hat die Geschichte gezeigt, dass ein Neustart mit einem verbesserten Protokoll keine Option für ein gewachsenes System wie das World Wide Web ist. Aus diesem Grund beschäftigt sich diese Arbeit mit unseren Beiträgen zu sicherer Web-Kommunikation auf der Basis des existierenden verbindungslosen HTTP.

Wir beginnen mit der Beschreibung von Session Fixation-Angriffen, die bereits vor der eigentlichen Anmeldung des Benutzers an der Web-Anwendung beginnen und im Erfolgsfall die temporäre Übernahme des Benutzerkontos erlauben. Wir präsentieren drei Gegenmaßnahmen, die je nach Eingriffsmöglichkeiten in die Web-Anwendung umgesetzt werden können.

Als nächstes gehen wir auf das Problem ein, dass Zugangsdaten im WWW sowohl zwischen den Teilnehmern zu Authentifikationszwecken kommuniziert werden als auch für jeden, der Kenntnis dieser Daten erlangt, wiederverwendbar sind. Unsere Ansätze binden das Benutzerpasswort an ein im Browser gespeichertes Authentifikationsmerkmal und das sog. Session-Cookie an ein Geheimnis, das nur dem Benutzer und der Web-Anwendung bekannt ist. Auf diese Weise kann ein Angreifer weder ein gestohlenes Passwort noch ein Session-Cookie allein zum Zugriff auf das Benutzerkonto verwenden.

Darauffolgend beschreiben wir ein Authentifikationsprotokoll, das vollständig auf die Übermittlung geheimer Zugangsdaten verzichtet. Unser Ansatz implementiert eine vertrauswürdige Benutzeroberfläche und wirkt so gegen die Manipulation derselben in herkömmlichen Browsern.

Während die bisherigen Ansätze die Sicherheit jeglicher Web-Kommunikation erhöhen, widmen wir uns der Frage, inwiefern ein intelligenter Browser den Benutzer – wenn nötig – vor Angriffen bewahren kann und – wenn möglich – eine ungehinderte Kommunikation ermöglichen kann. Damit trägt unser Ansatz zur Akzeptanz von Sicherheitslösungen bei,

die ansonsten regelmäßig als lästige Einschränkungen empfunden werden.

Schließlich legen wir den Fokus auf die Kontrollflussintegrität von Web-Anwendungen. Bösartige Benutzer können den Zustand von Anwendungen durch speziell präparierte Folgen von Anfragen in ihrem Sinne manipulieren. Unsere Ansätze filtern Benutzeranfragen, die von der Anwendung nicht erwartet wurden, und lassen nur solche Anfragen passieren, die von der Anwendung ordnungsgemäß verarbeitet werden können.

Acknowledgments

There have been many people attending me on my way to finish this dissertation. I would like to express my special appreciation to my advisors Joachim Posegga and Frank Piessens for their help, advice, dialogue, and support. I am also particularly thankful to Daniel Schreckling and Martin Johns for being reference persons and sharing their helpful research experience in more than one moment.

In no particular order, I would also like to thank Andrei Sabelfeld, Daniel Hedin, David Sands, Lieven Desmet, Nick Nikiforakis, Steven Van Acker, Philippe De Ryck, Jorge Cuellar, Monica Verma, Jan Wolff, Sebastian Lekies, Walter Tighzert, Dieter Gollmann, Felix Freiling, Thorsten Holz, Konrad Rieck, Jörg Schwenk, Sebastian Schinzel, Christopher Alm, Hannah Lee, Marina Krotofil, Jan Meier, Henrich C. Pöhls, Robert Olotu, Jan Seedorf, Eric Rothstein, Tobias Marktscheffel, Oussama Mahjoub, Arne Bilzhause, Boutheyna Belgacem, Wolfram Gottschlich, Juan David Parra, Hans P. Reiser, Guido Lenk, Marc Maisch, Alexander Seidl, Melanie Volkamer, Markus Karwe, Peng Liu, Juraj Somorovsky, Siglinde Böck, Marita Güngerich, Elisabeth Reither, Erika Langer, Korbinian Pauli, Daniel Hausknecht, Caspar Gries, Benedikt Petschkuhn, Johannes Köstler, Christian v. Pollak, Patrick Gemein, Stefan Kucher, Benedikt Höfling, Michael Schrank, Mathias Wagner, Benedikt Strobl, Christoph Oblinger, Johannes Rückert, Manuel Feifel, Stephan Huber, Tobias Friedl, Wolfgang Frankenberger, Thomas Schreiber, Boris Hemkemeier, and Dirk Wetter.

Most of all, I feel a deep sense of gratitude to my family, my wife, Eyke, and my daughter, Kalea. You are the sunshine that lights up my life!

Contents

List of Tables

List of Figures

1 Introduction

There are two main divisions in the field of web security: secure web applications and secure web communication. The field of secure web applications has been treated thoroughly in the past and is well understood today, hence we focus on the communication between the client and the server in the world wide web as the web has become the dominant network to remotely control applications though it has never been meant to be.

1.1 Motivation

When Tim Berners-Lee proposed the introduction of hypertexts as "a single user-interface to large classes of information (reports, notes, data-bases, computer documentation and on-line help)" [15], he could not foresee that he actually invented the world wide web that eventually evolved from the proposed delivery network for static hypertexts to a fully fledged application platform giving the users access to their bank accounts, email, multimedia entertainment and most recently a user interface to control their smart home devices like the washing machine and home heating.

The web's evolution was driven by upcoming use cases and business models that required new features. On the server side, the most important technical milestone is the introduction of programmable web pages using the common gateway interface (CGI) [111] together with the option to store web data persistently in databases and access that data via respective interfaces. These technologies enabled the dynamic compilation of requested web pages. The respective development on the client side includes the introduction of JavaScript to enable dynamic user interfaces, HTTP cookies to enable the client-side storage of data, and third-party plug-ins like Java, Flash, and Silverlight to enable the execution of browser-independent code in distinct runtime environments.

While the architecture on both sides of the communication channel has changed thoroughly during the past 25 years of web evolution, the communication protocol remained almost the same. Since the standardization of HTTP 0.9 in 1991 [14], only minor changes have been applied to the protocol as all subsequent versions were supposed to be backwards-compatible with the initial version. HTTP version 1.0 [13] introduced methods for requests, e.g. `POST`, `PUT`, `OPTIONS`, `HEAD`, and `DELETE`, to enhance the command options for the browser beyond the previously standardized `GET`. It defines *safe* and *idempotent* methods where `GET` is supposed to be both, meaning that `GET` requests are only supposed to retrieve information but not to change the server's state. `POST` requests, however, are meant to transmit data and trigger its processing possibly leading to state changes. The current HTTP version 1.1 [60] dates back to 1999 and regards

the trend towards web pages that enclose multiple files like images, script files, and style sheets. It improves the performance of page loads by re-using an established TCP connection to transfer those files sequentially. Finally, the draft of HTTP version 2.0 [11] also concentrates on performance improvements. It introduces parallel requests (request multiplexing), improved data compression, and server-side content pushing. Looking back, the protocol's evolution compensates the delay caused by the increased number and size of modern web page elements. New protocol versions do not reflect the changed security requirements introduced by the new web application business models.

Despite the missing adaption of the protocol's security properties, one can argue that HTTP plays an important role in the success story of the world wide web. It started with text-based, simple, and stateless websites and browsers communicating over a simple and connectionless protocol while giving fast and easy access to remotely stored information. The initial web actually had no business model and no threats in mind. The applications and the protocol were comprehensible, thus, lowering the burden for the development of new features without breaking existing functionality. New and upcoming application scenarios were only a small step ahead, fostering a sometimes chaotic series of inventions and the uncoordinated introduction of feature support[1].

The same kind of history also applies to security measures. The introduction of new features is usually accompanied by attacks that misuse the new scope of action. For instance, the invention of JavaScript led to the unauthorized execution of code by cross-site scripting attacks. The implementation of HTTP cookies as authentication credentials together with web pages being compiled from several domains facilitated cross-site request forgery attacks. In consequence, new band-aid solutions came up to mitigate the unintended exploitation of new features be it cookie attributes (secure, HTTPonly) or HTTP headers (CORS, CSP, PKP, HSTS). It is characteristic for the development of the world wide web that the thorough countermeasure against SQL injection, i.e. prepared statements, was actually invented to increase the speed of database queries but not for security reasons. Today, each website must implement the necessary protection instead of being based on a secure protocol.

We identified two particularly interesting aspects concerning the fast and dynamic introduction of new features on the client side and on the server side in combination with the missing adaption of the communication protocol HTTP:

- First, HTTP was never meant to carry authentication or session information as neither was envisaged at the design time of HTTP. Today, however, most web applications offer personalized user accounts and have a need for secure authentication and session management. In the tradition of the simple and stateless protocol, authentication happens in the simplest possible way by transmitting confidential authentication data (password, session cookie) in plain text. We will explore how far more sophisticated authentication protocols can be run over HTTP.

- Second, modern stateful web applications implement intended control flows where the user is supposed to perform one step after the other. The connectionless HTTP,

[1] See, for instance, the development of JavaScript vs JScript, Plug-ins vs ActiveX, HTTP 2.0 vs SPDY.

however, has no control-flow concept, meaning that technically, users can request every action at all application states. We will show the consequences and identify protective measures for stateful web applications.

1.2 Thesis Outline and Contributions

1.2.1 Thesis Overview

We will sketch how a fresh protocol could be to meet the security requirements of modern web applications. The comparison with other protocols in the Internet reveals that a connection-oriented protocol suits the needs of modern web applications best. However, history shows that there is no way for a fresh start using a completely different protocol. Also, the evolvement of the web has been driven by performance improvements and new features – but not increased security. For these reasons, we will then explore approaches for secure authentication and control-flow integrity given the known HTTP protocol.

Section 2 describes the technical background of our work, the attacks in scope, and identifies the attacks' root causes. We explore in Section 3 how other protocols in the Internet meet their security requirements which overlap those of modern web applications. Also, we inspect lower-layer protocols of the web stack in order to identify their protection capabilities.

Section 4 is dedicated to attack vectors on session cookies that are not renewed after authentication. We will argue that session cookies are left unchanged due to a mismatch of responsibilities of the web developer and the web application framework. Due to the connectionless nature of HTTP, an attacker can reuse a cookie he set before user authentication in the victim's browser. After user authentication, this cookie gives access to the user's account. We make the following contributions in Section 4:

- For one, we give a thorough documentation of existing session fixation attack vectors and take steps to assess the attack surface of web applications developed with state-of-the-art web frameworks.
- Second, we provide an approach for transparent, light-weight protection on the framework level. It allows 'patching' web applications without access to the code but just to the underlying framework.
- Furthermore, we developed a proxy-based solution that implements session fixation protection with neither access to the application code nor to the framework.
- In addition, we explain session fixation prevention at development phase and, thus, provide comprehensive protection against session fixation vulnerabilities.

We delve into the reusability of authentication credentials in Section 5. Beside the problem that passwords and session cookies are sent over the wire, they are not bound to the user or her browser, meaning that an attacker can use them to access the user's

account without further ado. We present two approaches that bind browser-stored credentials and user knowledge to prevent the exploitation of stolen credentials. Overall, Section 5 makes the following contributions:

- We identify the root cause for the ongoing threat of phishing attacks in the ineffective countermeasures proposed so far that can be divided into incomplete, cumbersome, and incompatible countermeasures.
- Then, we present our light-weight approach that provides robust security guarantees, even in case that the user's password was successfully stolen. The approach does not require second devices nor does it alter the authentication interaction from the user's point of view.
- Concerning web session attacks, we identify their common root cause in the browser-level authentication implemented by session cookies and the missing user context.
- We present our approach that requires user interaction before security critical actions are performed by the web application. This way, impersonation attacks are rendered ineffective.

While augmented credentials are hard to exploit, we show in Section 6 how an authentication protocol in the web can avoid sending cleartext credentials. After an initial setup phase, the client and the server can exchange authenticated messages over HTTP without the need to enter passwords and without using session tokens. We focus on mobile web applications. However, the approach is well applicable to desktop computers, too. In Section 6, we make the following contributions:

- We analyze how common web authentication attacks, such as phishing or clickjacking, manifest themselves in mobile scenarios and identify a common root cause – the lack of a trusted UI of the browser.
- We propose a novel authorization delegation scheme for mobile web applications that leverages a native companion application. It serves as a trust anchor for the mobile web application's client side through providing the missing trusted UI capabilities.
- We report on a practical implementation of our system as an app for the two currently dominating mobile operating systems, iOS and Android. In this context, we show how the concept can be realized through leveraging the platform-specific facilities for inter-app cooperation.

We explained above that security approaches usually narrow down the scope of new features in order to prevent their misusage. In Section 7, we condense the state-of-the-art protection approaches against attacks on session credentials and relax the imposed restrictions for uncritical cases. In this sense, we comply with the web community's drift towards more features and less restrictions. Section 7 makes the following contributions:

- We analyze existing approaches for secure session tracking and identify five central measures in respect to handling of the session credential.
- Based on this observation, we phrase a permissive policy for the uncritical cases and a restrictive policy for security sensitive situations.
- We describe a model for secure web session tracking, that applies both policies as required and overcomes CSRF, session hijacking, sidejacking, and session fixation attacks.
- We report on an implementation of the approach's client-side part as a Firefox extension, which applies a heuristic to waive the need for server-side support.

Section 8 is dedicated to control-flow integrity. Web applications implementing multi-step workflows are inherently vulnerable to attacks on the application state. In the hypertext scenario envisioned by Tim Berners-Lee, there were no such workflows and, hence, no need for a protocol-level control of request sequences. Today, however, every web application must implement protection. We make the following contributions in Section 8:

- We analyze recent attacks on web applications with respect to user-defined requests and identify their root cause in the missing explicit control-flow definition and enforcement.
- Then, we evaluate the most prevalent web application frameworks in order to assess how far real-world web applications can use existing means to explicitly define and enforce intended control flows. While we find that all tested frameworks allow individual retrofit solutions, only one out of ten provides a dedicated control-flow integrity protection feature.
- Based on this result, we provide our first approach, a control-flow monitor that is applicable to legacy as well as newly developed web applications. It expects a control-flow definition as input and provides guarantees to the web application concerning the sequence of incoming requests and carried parameters.
- Our second approach copes without a hand-crafted policy. It dynamically derives a policy ad hoc on a per-workflow basis by analyzing the web traffic.

Section 9 concludes our findings.

1.2.2 Associated Publications

Parts of and ideas underlying this thesis have been previously published in the following documents:

- Michael Schrank, Bastian Braun, Martin Johns, and Joachim Posegga. Session Fixation – the Forgotten Vulnerability. In *Sicherheit 2010: Sicherheit, Schutz und Zuverlässigkeit*, Lecture Notes in Informatics (LNI), Springer, pages 341 - 352, 2010. [154]

- Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable Protection Against Session Fixation Attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011)*, ACM, pages 1531 - 1537, 2011. [86]

- Bastian Braun, Martin Johns, Johannes Köstler, and Joachim Posegga. PhishSafe: Leveraging Modern JavaScript API's for Transparent and Robust Protection. In *Proceedings of the Fourth ACM Conference on Data and Application Security and Privacy (ACM CODASPY 2014)*, ACM, pages 61 - 72, 2014. [21]

- Bastian Braun, Stefan Kucher, Martin Johns, and Joachim Posegga. A User-level Authentication Scheme to Mitigate Web Session-based Vulnerabilities. In *Proceedings of Trust, Privacy and Security in Digital Business (TrustBus '12)*, Lecture Notes in Computer Science (LNCS), Springer, pages 17 - 29, 2012. [22]

- Bastian Braun, Martin Johns, Johannes Köstler, and Joachim Posegga. A Trusted UI for the Mobile Web. In *Proceedings of the 29th IFIP International Information Security and Privacy Conference (IFIP SEC 2014)*, IFIP Advances in Information and Communication Technology, Springer, pages 127 - 141, 2014. [20]

- Bastian Braun, Korbinian Pauli, Joachim Posegga, and Martin Johns. LogSec: Adaptive Protection for the Wild Wild Web. In *Proceedings of the 2015 ACM Symposium on Applied Computing (SAC 2015)*, ACM, 2015. [23]

- Bastian Braun, Christian v. Pollak, and Joachim Posegga. A Survey on Control-Flow Integrity Means in Web Application Frameworks. In *Proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec 2013)*, Lecture Notes in Computer Science (LNCS), Springer, pages 231 - 246, 2013. [24]

- Bastian Braun, Patrick Gemein, Hans P. Reiser, and Joachim Posegga. Control-Flow Integrity in Web Applications. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*, Lecture Notes in Computer Science (LNCS), Springer, pages 1 - 16, 2013. [18]

- Bastian Braun, Caspar Gries, Benedikt Petschkuhn, and Joachim Posegga. Ghostrail: Ad Hoc Control-flow Integrity for Web Applications. In *Proceedings of the 29th IFIP International Information Security and Privacy Conference (IFIP SEC 2014)*, IFIP Advances in Information and Communication Technology, Springer, pages 264 - 277, 2014. [19]

2 Web Communication and Respective Attacks

We start with explaining the details of authentication in web applications and attacks on authentication and authenticated sessions. We show that these attacks share some common root causes.

2.1 Basics of Web Communication

We want to quickly describe the basics of web communication, i.e. HTTP. The communication between a browser and a web application consists of *HTTP requests* – sent by the browser – and related *HTTP responses* by the web application. For brevity, we will use the terms *request* and *response* in the remainder of this document.

The browser starts the communication with a request towards the web application. The application processes the request, i.e. it stores sent data or assembles requested data and compiles an HTML document to answer the request. This document is finally sent back to the browser in a response. A pair of request and response is called *HTTP round trip*. The browser renders the document, meaning that it displays a graphical representation to the user. The user interacts with the document, for example she enters data into provided forms or clicks an embedded hyperlink. Parts of the interaction can be processed locally in the browser by JavaScript code. Eventually, communication with the web application becomes necessary, e.g. to drop user data or to retrieve new data. This communication starts a new HTTP round trip.

2.2 Authentication

An increasing number of web applications provide personalized content to users, e.g. email, eCommerce, and social network providers. In order to control the access to the user's account, the user and the web application agree on a user ID and a shared secret upon account setup. When the user wants to log into her account, she provides her credentials and the web application compares those with its internal record. The security of this approach relies on the confidentiality of the shared secret.

In the remainder of this document, we will use the terms *user ID* and *username* as well as *shared secret*, *password*, and *PIN* interchangeably.

Technically, in form-based authentication, the user's ID and password are communicated using HTML forms. After the user has entered her credentials, she submits the

form. This causes the web browser to create an HTTP request which carries the values in the form of `GET` or `POST` parameters. In particular, this implies that the password is sent in clear text to the server – though the transmission may be encrypted on a lower layer of the web stack, e.g. SSL or IPsec (see Section 3.2).

2.3 Authentication Tracking in the Web

The user authentication (see Section 2.2) happens within one HTTP round trip. Subsequent HTTP round trips, however, must also contain evidence to originate from the user in order to process them in the correct user context. For this reason, web applications implement *sessions* that allow to chain requests. Sessions are usually implemented using *HTTP cookies* [102]. Cookies have a name, a value, and a set of attributes, among them the domain they stem from and an expiration date. The same-origin policy [150] regulates a cookie's scope: if set by `www.example.com`, the cookie will be appended to all requests targeting the same host, independent of the protocol (`http`, `https`) and the port. However, web applications can relax a cookie's scope, i.e. `*.example.com` in the above case.

Upon user login, the web application issues a cookie that will serve as the *session identifier* until the user logs out. The user's browser stores the cookie and appends it to every request targeting the cookie's domain. The web application accepts every request carrying a user's session cookie to be part of the user's session. This way, the session cookie becomes the user's de-facto credential for the lifetime of the session. Again, as with the password, this credential is communicated in clear text on the HTTP layer. We will use the terms *session identifier*, *session ID*, and *SID* interchangeably in this document.

Basically, there are two alternative options to chain the requests of authenticated users:

- First, web applications can embed hyperlinks in their responses that contain the session ID as an HTTP parameter [60]. A user's interaction will then trigger a request that carries the session ID as a parameter. This approach, however, comes with a number of security issues: the session ID is accessible to JavaScript code from foreign domains, it occurs in log files of web proxies and web applications, it is stored as part of the URL in bookmarks, and a user recommending a URL to a friend also gives access to her session.

- Second, HTTP has a built-in user authentication mechanism that provides an authentication tracking feature [62]. This approach, however, is rarely implemented for two reasons: For one, the initial user authentication happens in a neutral browser pop-up that is not customizable at all. This is not only a design issue but also facilitates spoofing attacks. Second, in the basic authentication protocol, the browser stores user credentials until it is closed, and, for every request towards the authentication domain, it adds the credentials as an HTTP `Authorization` header. Later, the more sophisticated digest authentication protocol came up but

this was too late to stop the upcoming HTML form-based authentication together with cookie-based session tracking.

2.4 Control-flow Integrity in Web Applications

Given the authentication tracking means described above, a web application and a browser can establish a session and implement multi-step workflows. Those workflows consisting of a sequence of user actions, however, pose an inherent threat to web applications if they do not preserve control-flow integrity.

In order to understand control-flow integrity in web applications, it is important to have a notion of connectionless and connection-oriented protocols. We will describe those protocols and their implications in more detail in Section 3. For now, we can consider a protocol as a set of valid messages. A message is valid if it complies with the protocol syntax. The important difference between connectionless and connection-oriented protocols is the following: The set of valid messages is static during a protocol run for connectionless protocols. Connection-oriented protocols, on the contrary, have a variable set of valid messages which means that the validity of a message always depends on the messages exchanged in the past since the establishment of the connection.

HTTP is a connectionless protocol. The logic of current web applications, however, is stateful. This means that a web application may change its internal state upon processing a user request. For example, a social network must update its state to store that a user has a new friend, and the eCommerce application must maintain the user's shopping cart. Some workflows even require multiple steps by the user. For instance, a shopping workflow might first require to put items to the cart, then log in, provide a shipping address and shipping speed, choose a payment option, and finally review the complete order. For every step, the user is supposed to fill some form and press a button.

Putting things together, the set of valid HTTP messages remains static while web applications expect particular, i.e. changing, requests in the next step. So, web applications must ensure that the processing of requests does not lead to an insecure state. Considering the shopping workflow above, a web application must make sure that a malicious user can not request the address of the review page without providing payment details. This is an inherent semantic problem of all stateful web applications. A web application preserves control-flow integrity if every request leads to an application state that is part of the application's intended logic.

2.5 Attacks on Web Applications

In this section, we explain the attacks on web applications that are in scope of this thesis. All described attacks are related to web communication and exploit the connectionless nature of HTTP. Section 3 will analyze in more detail the root causes of the attacks.

2.5.1 Phishing

The term *phishing* subsumes all attacks that aim to obtain the user's password via tricking the user to interact with a web resource that claims to be a legitimate part of the targeted web application but in fact is under the control of the attacker.

Attack Method

The Anti-Phishing Working Group (APWG)[2] states that phishing schemes use "spoofed e-mails purporting to be from legitimate businesses and agencies, designed to lead consumers to counterfeit websites that trick recipients into divulging financial data such as usernames and passwords." [173] Attackers usually send emails or personal instant messages and put pressure on the recipients to perform actions intended by the attacker. For instance, recipients are told that their email quota is reached, their credit card is disabled, or an invoice has not been paid. Usually, to increase the pressure and omit a reconsideration, immediate steps are allegedly necessary. These steps require logging into an account on a website. In this scenario, attackers know the target business. All they need to do is copy the public design of the website and send bulk emails. In order to educate customers, anti-phishing campaigns published rules of conduct. For example, users are advised to not click on links embedded in emails if the link does not include the expected domain of the (seeming) sender. As another rule of thumb, reliable emails contain the recipient's name and maybe other personal information which is supposedly not known to a phisher.

The following attack vectors emerged in the past and illustrate the ongoing arms race between phishers and the anti-phishing community.

Concealing the Target Domain In order to answer the anti-phishing suggestions, phishers took measures to make embedded links look familiar to the user. These measures range from open redirects on the target website, over URLs featuring a target domain prefix and URL shorteners hiding the target, up to malicious relying parties in single sign-on protocols.

Open Redirects First of all, a phisher's chance is considerably higher if the link the victim is supposed to click appears to belong to the expected domain. In that sense, an attacker can succeed if he finds an open redirect function on the target web application. Web applications redirect their users for several reasons: when a requested web page is not found (HTTP 404), users are redirected to a landing page that explains what happened. Webmail providers redirect their customers via 'de-referrers' to avoid that the actual URL of the read email appears as a part of the subsequent request to the foreign domain (in the `Referrer` header). Open redirects do not sanitize their input, i.e., the redirect target and source. Given that the attacker prepared a phishing site for `example.com` that has an open redirect, he can send out emails asking users to click on `https://www.example.com/redirect?target=example-attack.com`. A better masking is possible by URL encoding the target parameter. Finally, the victim

[2] `http://www.apwg.org/`

sees an `https` link to the expected domain and can eventually check the SSL lock on `https://www.example-attack.com` but is attacked, though.

Confusing URLs Second, it is often sufficient to make the URL appear innocent at a first glance. Non-expert users can hardly distinguish between the host, domain, and path elements of a URL. Phishers exploit this weakness crafting links like `https://www.example.com.attacker-domain.com` which seem to contain the expected domain name `example.com`. Similar approaches include typos in the URL, e.g. `https://www.gooogle.com`.

A more sophisticated attack is known as international domain name (IDN) homograph attack [64]. This attack makes use of so-called homographs, characters from non-latin alphabets that are indistinguishable for humans but interpreted by browsers as different symbols.

URL Shorteners The emerging trend towards URL shorteners, that save characters on Twitter and prevent line breaks in emails, makes people familiar with short URLs and redirects to unpredictable URLs. Attackers exploit that people are more used to click on links from `bit.ly`, `tinyurl.com`, `is.gd`, or `goo.gl` than on links containing unknown domains. If the target website looks convincing enough, the user's focus is caught on the content [194].

Malicious Relying Parties Single sign-on (SSO) protocols require the user to log in once with her identity provider to obtain access to all related accounts. If the user first visits a relying party, she is redirected to her identity provider. A malicious relying party can redirect the user to a phishing identity provider to request the user's credentials.

Spear Phishing *Spear phishing* denotes a particular phishing attack vector that targets a set of victims the attacker has information about. While this attack is restricted to those users the attacker could gain knowledge about, it can still hit thousands of users. The first step in a common scenario is a data leak of a company's customer database. In most cases, there is a laxer security policy in place if the database does not contain critical data like passwords, social security numbers, or credit card information. Using the obtained data, however, an attacker can address his victims personally including the name, correct email address, and account number which used to be an indicator of a benign message.

Browser-less Phishing A phisher can circumvent browser-based countermeasures if the user does not use her browser to follow his instructions. As a matter of fact, users regularly experience that colleagues or friends quickly ask for information by email or instant messenger. Phishers convey the notion of this scenario to make their victims reply with the credentials.

2.5.2 Session Hijacking

A successful *session hijacking* attack gives the attacker control over the user's account for the lifetime of the established session. The attacker gains knowledge of the user's

session token and impersonates the user towards the web application. As we pointed out in Section 2.3, the session token is the only authentication credential after the user login. There are basically three options for the attacker to obtain knowledge of the session token:

- *Sidejacking [69]:* First, the attacker can wiretap the communication between the user's machine and the web server. In order to extract the session token from observed web traffic, the attacker first becomes a *man in the middle (MITM)*, e.g. by an ARP spoofing attack [187]. The MITM position gives him access to all web traffic from the victim's machine. This step is not necessary in unprotected wireless networks (*hotspots*) common in public environments like train stations, airports, and hotels. Every peer in the vicinity of a hotspot can eavesdrop all traffic via the air interface.

  Some web applications use SSL to encrypt the communication and protect against MITM attacks. In this case, the attacker can prevent the establishment of a secure connection using SSL stripping [110] if the communication starts unencrypted.

- *Session hijacking via XSS [54]:* Second, the attacker can steal the session token from the user's browser. The attacker first performs a cross-site scripting (XSS) attack to inject JavaScript code that reads the stored cookies (via `document.cookie`) and transmits them to the attacker' site, e.g. via a crafted POST or GET request.

- *Third-party JavaScript inclusions [126]:* Third, the attacker can try and inject malicious code into a JavaScript library that is included by the target domain. This code is treated as same-domain content though loaded from a third-party domain and, hence, has full access to the page content including the stored cookies.

Finally, an attacker can also try to guess the correct session token via a brute-force attack [92]. We ignore this attack because today's web applications use long random strings as session tokens which can hardly be guessed before the session ends [30].

Session hijacking attacks are enabled by the easy transferability of the session token: Cookies and SIDs stored as HTTP parameters are easily accessible to embedded JavaScript code. When stolen, they can be reused by the attacker because they are not bound to the client environment. Neither cookies nor HTTP parameters have been invented with high security requirements in mind but have a high significance today.

We will describe a special variant of a session hijacking attack in the next section.

2.5.3 Session Fixation

Session fixation [99] is a variant of session hijacking (see above). The main difference to the variants discussed above is that session fixation does not rely on SID theft. Instead, the adversary tricks the victim to send an SID that is controlled by the adversary to the server. This can be achieved either by supplying a crafted URL including this SID as a parameter to the victim (in case that the vulnerable web application accepts parameter-based SIDs) or by finding a way to set a copy of this SID cookie to the victim's browser

(more on this attack vector in Section 4.2.1). See Figure 2.1 for a brief example of the attack via a crafted URL. The individual steps of the attack are the following:

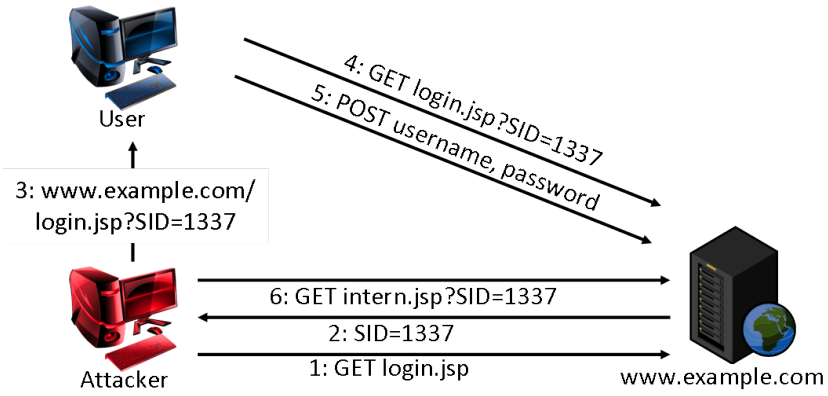

Figure 2.1: An exemplified session fixation attack [99].

- The attacker obtains an SID value from the server (1,2).
- He tricks the victim to issue an HTTP request using this SID during the authentication process (3,4).
- The server receives a request that already contains an SID. Consequently, it uses this SID value for all further interaction with the user and along with the user's authorization state (5).
- Now, the attacker can use the SID to access otherwise restricted resources utilizing the victim's authorization context (6).

Why does Session Fixation Exist?

At first glance, the session fixation attack pattern seems both contrived and unlikely. Why would an application accept a user's SID that was not assigned by the application to this user in the first place? The root problem lies within the following mismatch of responsibilities.

SID management is executed by the utilized programming framework or the application server. All related tasks, such as SID generation, verification, communication, or mapping between SID values to session storage, are handled transparently to the application. The programmer does not need to code any of these tasks manually.

However, the act of authenticating a user and subsequently adding authorization information to her session data is an integral component of the application's logic. Consequently, it is the programmer's duty to implement the corresponding processes. The framework has no knowledge about the utilized authentication scheme or the employed authorization roles.

Finally, in general, modern web application frameworks assign SID values automatically with the very first HTTP response that is sent to a user. This is done to track application usage even before any authentication processes have been executed, e.g., while the user accesses the public part of the application. Often the programmer is not aware of this underlying framework-level SID functionality as he is not responsible for managing the SIDs. He simply relies on the framework-provided automatism.

The combination of the circumstances listed above leads to situations in which applications neglect to reissue SID values after a user's authorization state has changed. This in turn causes session fixation vulnerabilities: As the SID remains unchanged, any initial SID value which was fixed by the attacker stays "valid" and, thus, can be abused.

Why does Client Recognition not Help?

An HTTP session can be considered as the abstraction of a dedicated communication channel between the client and the server. Only the knowledge of the communication channel's name (SID) is needed to access this channel. Hence, knowing the SID enables the adversary to conduct a session hijacking attack.

Additional measures, such as browser recognition (which can be trivially circumvented) or IP binding, have been proposed to address this problem. These measures raise the bar, but they can not finally solve the problem. Instead, they bring new problems under certain circumstances. For instance, IP binding makes a service unusable if accessed from anonymity networks that tend to send packages from changing IP addresses. A mobile device switching from a mobile network to WiFi at home gets a new public address and looses the current session with unsaved data. Network address translation (NAT) is used in mobile, company, and university networks. All requests from the same network appear to come from the same address and thus eliminate IP binding protection. That is why we disregard such additional security measures for the remainder of this document unless any measure plays a decisive role.

Session Fixation and Session Hijacking via XSS

The session fixation attack is similar to the session hijacking attack via XSS in that it needs a vulnerability to prepare the attack. Like the XSS vulnerability allows the attacker to steal the SID, i.e. to gain knowledge of the communication channel's name, the session fixation attack needs a preceding attack to fixate the victim's SID before connection establishment to the web application. So, the attacker sets up a new session and receives an SID. Next, he makes the victim use the same SID. Thus, he transfers the communication channel's end point to the victim. Finally, the attacker can take over the victim's session after the latter authenticates against the application. Thus, the attacker uses the authenticated communication channel addressed by the well-known SID without ever authenticating himself.

The main reason for the session fixation vulnerability lies in the missing renaming of the communication channel after authentication. The SID is not security-critical data before authentication as the user is still unknown and neither the user nor the

channel are trustworthy. However, after the authentication process has been passed successfully, the same communication channel with the same name has become trusted and confidential. The same data must turn from not confidential to confidential as the web application must rely on the opponent's identity being proved by the shared secret channel name. So, only the authorized client is supposed to know the 'ticket' to the established communication channel. This can be easily guaranteed by renewing the SID after every authorization raise, e.g. from an unauthenticated user to an authenticated user but also from an unprivileged user to an administrator.

2.5.4 Clickjacking

A clickjacking attack [144] exploits the fact that the user's perception may vary from the page rendering outcome by the browser. In this attack scenario, the victim is a user that has an account at the target web application. To perform an attack, the attacker prepares a web page that makes the user perform actions on the target web application. Compared to the attacks described so far, the attacker does not learn any credentials of the victim – neither the password nor the session token. Instead, he makes the victim perform actions in the attacker's interest.

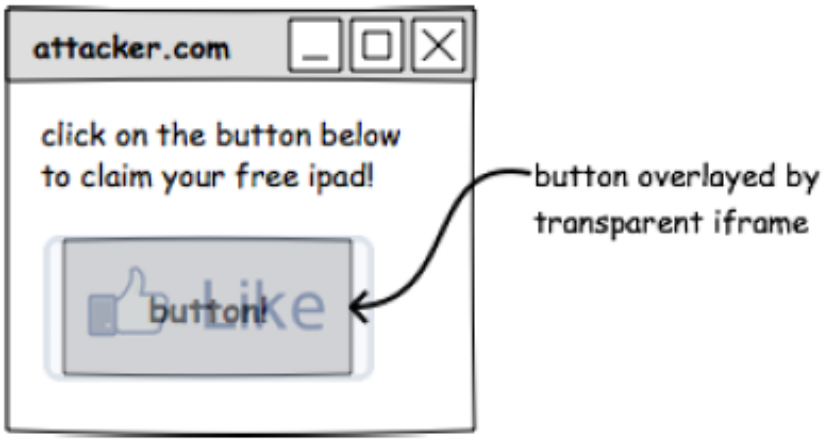

Figure 2.2: Compromising target display integrity via a transparent overlay [100].

Huang et al. identify three kinds of integrity violations where each kind can be used to run a clickjacking attack [81]:

- *Target Display Integrity:* Clickjacking attacks compromising the target display integrity use transparency and overlay elements to deceive the user. There are two options for the attacker: Either he overlays the button of his own web page where the user will probably click with a transparent version of the button on the target web page where the user is supposed to click (see Figure 2.2 for an example). In this scenario, the user only sees the attacker's page but not the button she actually clicks.

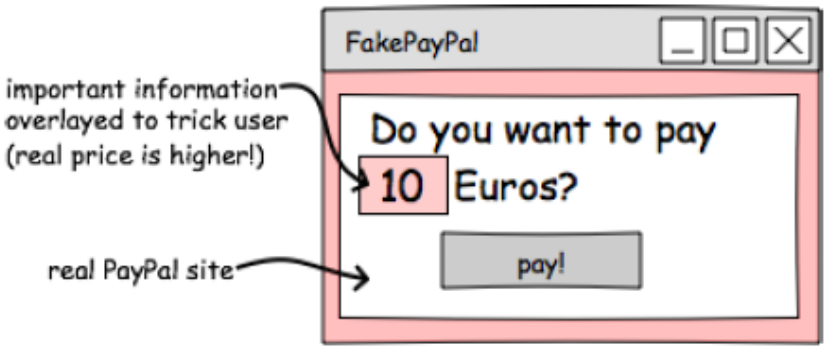

Figure 2.3: Compromising target display integrity via a non-transparent context overlay [100].

Or, the attacker displays the target web page but overlays some parts such that the user can not determine the correct context of the button she clicks though the button is visible (see Figure 2.3 for an example).

- *Pointer Integrity:* Clickjacking attacks compromising pointer integrity exploit the fact that web pages can define many attributes of the user's mouse pointer, including the shape, position, and transparency. An attacker can hide the actual mouse pointer and show a new mouse pointer at a different position (see Figure 2.4 for an example). The browser, however, records the clicks at the actual pointer position. In fact, the user can not know where she clicks just by observing the visible mouse pointer.

Figure 2.4: Compromising pointer integrity [81].

- *Temporal Integrity:* Clickjacking attacks compromising temporal integrity lead the user to believe that she clicks on the link or button of her choice but change the content under the mouse pointer in the very last moment before the click. An attacker needs to estimate when the user will click and exchange the content quickly such that the user can not react. There are two common attack scenarios: For a *double clickjacking* attack, the attacker makes the victim double click several

buttons, e.g. in the course of an online game. Between the first and the second click in some round, he moves a window with the target web page loaded from the background to the foreground (see Figure 2.5 for an example). The button triggering the intended action must be at the same position as the button the user clicked first. The attack is successful if the user's intention to double click leaves no time to react.

Second, a *clickjacking via history navigation* attack first loads the target web page in a browser window and immediately forwards to the attacker's site. The attacker triggers the browser's back navigation function when the user will probably click a button on the attacker's page on the same position as the target button on the last web page. Browsers store the last visited web page in their local cache and load it immediately.

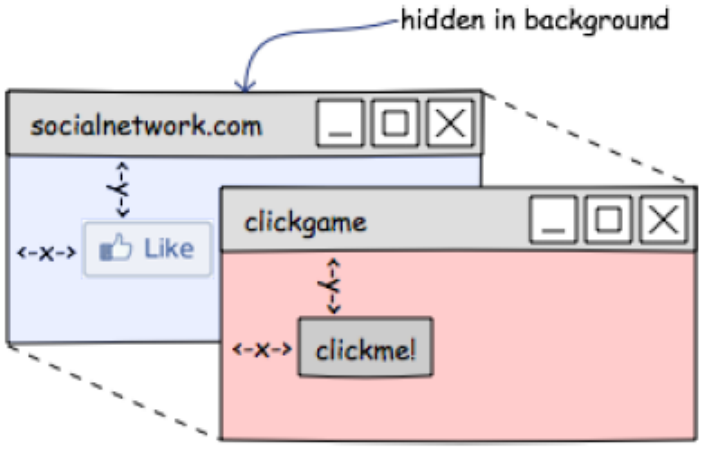

Figure 2.5: Compromising temporal integrity via double clickjacking [100].

In any case, the attacker can make the user perform arbitrary actions as long as these are invokable by mouse clicks.

In summary, clickjacking attacks rely on a mismatch between the user interface as it is perceived by the victim and the actual technical context of the web page elements in the browser. The root cause of clickjacking attacks can be found in the missing trusted path between the user and the web application. Web users facing clickjacking attacks can not know the impact of their click actions. Transferred to the physical world, the user of a coke vending machine will not buy the drink of her choice if the buttons are wrongly tagged.

2.5.5 Cross-site Request Forgery

Just like a clickjacking attacker, a *cross-site request forgery (CSRF)* attacker does not learn the victim's secret credentials, neither the password nor the SID. However, in

contrast to a clickjacking attack, the attacker does not make the user but rather her browser perform actions in his interest. In order to achieve his goal, he inserts a crafted link into some web page that makes the user's browser send a request to the target web application, seemingly on behalf of the user.

For example, the attacker might put `http://www.yourbank.com/transfer.php?from=your_acc&to=my_acc` into an image (`<img>`) tag on any website. Upon visiting this website, the user's browser tries to retrieve a picture and sends the crafted request to the banking website. The attack is successful if the victim is logged into her account at `yourbank.com` at the same time. Her browser will attach the SID cookie to the request and legitimate the money transfer.

The CSRF attack is an instance of the confused deputy problem [74]. A confused deputy is defined as a principal that is innocently fooled into misusing its authorization status. The browser plays the role of the confused deputy in CSRF attack scenarios because it appends the session cookies to cross-domain requests and thus authorizes the attacker-initiated requests on behalf of the victim.

We now leave the field of attacks on the authentication relation between a user and a web application and change over to attacks targeting a web application's request processing.

2.5.6 Exploiting Race Conditions

In this section, we start describing the class of attacks on control-flow integrity. These are not related to authentication issues but rather give the attacker unauthorized access to application resources or allow him to manipulate the application's state in his interest. Race condition exploits belong to the latter category.

In order to exploit race conditions [135] in web applications, attackers can send several crafted requests almost in parallel. Web applications are multi-threaded by design and, so, have an inherent concurrency property when receiving several requests in a short time frame. There is no low-level serialization of requests for performance reasons. The actual application semantics can be changed if the web application does not handle concurrent requests by proper synchronization.

In one real-world example, a web application provided an interface to send a limited number of SMS text messages per day [138]. The web application first checked the current amount of sent messages (*time-of-check*), then delivered the message according to the received request, and finally updated the number of sent messages in the database (*time-of-use*). Attackers were able to send more messages than allowed by the web application by crafting a number of HTTP requests, each containing the receiver and text of the message to be sent. These requests were sent almost in parallel and the multi-threaded web application processed the incoming requests concurrently. This way, the attacker exploited the fact that the messages were sent before the respective database entry was updated, leading to the delivery of all requested messages. The developers' underlying assumption was that users finish one transmission process before sending the next message and do not request one operation of the workflow several times in parallel.

While race conditions are in general known for years, they are a crucial aspect of

control-flow integrity because the expected sequence of steps in a workflow can be manipulated. Instead of proceeding to the next step, the same action is executed repeatedly. This way, the attack leads to a corrupt application state.

2.5.7 HTTP Parameter Manipulation

HTTP requests can contain parameters in addition to the receiving host, path, and resource. As the parameters are sent by the client, the user can control the parameters' values and which parameters are sent to the web application. Manipulations of request parameters belong to the most common attacks on control-flow integrity: An attacker only has to take an embedded URL from a web page, guess the meaning of the parameters, and try values of his interest.

In 2011, the Citigroup faced an attack on their customers' data [177]. The attackers were able to access names, credit card numbers, e-mail addresses and transaction histories. All the attackers had to do was simply counting up the account number in the respective HTTP parameter in the web browser's address bar. By automation, they obtained confidential data of more than 200,000 customers.

Attackers can run more sophisticated attacks given a profound knowledge of the web application: Wang et al. [190] found a bunch of logic flaws in well-known merchant systems and Cashier-as-a-Service (CaaS) services. These flaws allowed them to buy any item for the price of the cheapest item in the store. They started two shopping workflows, one purchasing only a cheap item and the other purchasing an arbitrary number of intended items. After payment of the cheap item, they inserted the `orderID` from the expensive workflow to call an `updateOrderStatus` function, thus marking the expensive order paid.

Technically speaking, web users can call arbitrary functions of the web application with arbitrary parameters. In the Citigroup example, the attackers were not authorized to use the other customers' account numbers as parameters. The CaaS example exploited an unauthorized combination of parameters from different workflows.

File inclusion attacks are a special kind of HTTP parameter manipulation. The successful attacker gains access to protected resources be it static documents or application functions. The attacker exploits the fact that the output of web applications is often generated on-the-fly from static and dynamic content. A malicious user might among other things be able to trick the application into disclosing locally stored, confidential data.

For instance, an application might offer the URL `http://example.com/?view=welcome.html` as a hyperlink. In this case, the `view` parameter holds the name of a file that is supposed to be included in the output. An attacker can change the parameter value to `/etc/passwd`. He succeeds if the application fails to detect the manipulation.

A successful file inclusion attack allows to access all files that are readable to the web server process. Hence, a file inclusion attack is another instance of the confused deputy problem beside CSRF attacks (see Section 2.5.5).

2.5.8 Unsolicited Request Sequences

Attackers can not only modify the requests' parameters but also craft requests to any method of the web application. Besides manipulated HTTP parameters, web applications might face unexpected requests to any method.

For instance, in another given scenario by Wang et al. [190], a malicious shopper is able to add items to his cart between checkout and payment. After starting the checkout workflow which redirects the user to an external CaaS, the amount to pay is fixed. However, he can still add items to the cart in a parallel browser tab. When the user is redirected to the online shop providing a digital payment receipt, the current items from the shopping cart are shipped. He is only charged the value of his cart at checkout time. The recently added items are shipped but not invoiced.

The underlying root causes are the same as those of HTTP parameter manipulation (see above): Users can call any function of the web application at any time with arbitrary parameters. There is no protection on the protocol level with HTTP being a connectionless protocol with a static set of valid messages (see Section 2.4).

Unsolicited request sequences require good knowledge of the web application's public interface. This knowledge, however, can be obtained by studying the application during regular use.

2.5.9 Forceful Browsing

A *forceful browsing* attacker exploits predictable naming schemes in combination with insufficient access control.

For instance, left installation scripts for PHP-based web applications are popular targets for forceful browsing attacks. An administrator uploads such a script to the web server and calls it via her browser in order to install a web application. Attackers can call the installation script and reconfigure the application if the administrator forgets to delete it.

In 2010, a group of attackers gained access to 114,000 user records of iPad owners by requesting a server-side script that was supposed to embed user details into a web page [170]. The attackers could easily guess the naming scheme and access restricted application functions.

Forceful browsing attacks are similar to unsolicited request sequences (see above) by their nature. However, they usually target application features the attacker is generally not authorized to access. On the other side, unsolicited request sequences target function calls covered by the user's access rights but not expected by the application at this point of the workflow.

2.5.10 Compromising Use of the "Back" Button

Current web browsers are fitted with a so-called "Back" button. It is meant to navigate back to the last visited web page. The compromising use of the button is actually

not an attack but a client feature which may have accidental side effects on the web application's state.

Depending on the configuration, the last request either has to be repeated in order to display the page or the content is loaded from the browser's local storage ("cache"). In the context of a workflow, the user takes one step back which in some cases is unwanted and also undetectable by the web application. In fact, the usage of this button usually invokes the last action again rather than rolling back the last changes. Hallé et al. describe related navigation errors [72].

2.5.11 Further Attacks on User Accounts

We want to stress that there are more attacks on user accounts in the web, e.g. based on malware that is installed locally on the victim's machine. These, however, are out of our scope.

3 How Protocols can Meet the Web's Security Requirements

HTTP was invented in order to transport hypertext documents upon request from a server to a client (or even vice versa, see `HTTP PUT`). The client's request carried the precise location of the document, the supported encoding schemes, character sets, and languages. The server could compile the response to the client's request using only the information from the request: It looked up the requested document and optionally transcoded it to make it readable for the client. The stateless HTTP perfectly fitted the needs in the first days of the web being stateless and simple.

Today, however, the web has grown up to a fully fledged, distributed, and stateful application platform. This grown platform has additional requirements: user and application authentication, message integrity and confidentiality, control-flow integrity, and application-to-application authorization. Users need to authenticate towards web applications before accessing their accounts. Also, web applications need to authenticate towards the users to make them provide possibly private or confidential data. The exchanged messages must be integrity and confidentiality protected to prevent injection and spoofing attacks and keep the user's private and confidential data secret. The workflows of modern web applications comprise multiple steps, each changing the application's state. Hence, a web application must enforce the sequence of state-changing requests to prevent state tampering. Finally, modern web applications embed content from third-party web applications. The content is retrieved via the user's browsers and on behalf of the user. Only authorized applications must be able to perform third-party requests.

Of all those requirements, the only feature that can be implemented in HTTP is user authentication [62] but even this is rarely used (see Section 2.3). In the remainder of this section, we will explain the concept of connection-oriented protocols to meet similar security requirements and examine the lower-layer protocols used in the world wide web on their suitability to meet the security requirements.

3.1 Connection-oriented and Connectionless Protocols

We first explain how other protocols on the application layer achieve similar security requirements to today's web and then identify the connection-orientedness as the common basis.

The *Secure Shell (SSH)* protocol is used to establish a connection between two mutually authenticated peers, a client and a server. Both peers run a challenge-response

authentication protocol and exchange session keys during connection setup in order to protect the integrity and confidentiality of exchanged messages.

Also, the *Simple Mail Transfer Protocol (SMTP)* first establishes a connection between the client's mail user agent (MUA) and the mail server (MSA). SMTP is a simple text-based protocol and in that sense similar to HTTP. However, it preserves control-flow integrity by accepting only a precisely defined sequence of client commands. This is possible for two reasons: First, SMTP is a single-purpose protocol only meant to deliver email. So, the application logic is tied to the protocol. Second, the established connection allows the server to chain requests on protocol level and detect violations in the protocol flow immediately. For instance, the client misbehaves if it does not start mail delivery using the `MAIL` command. Web applications, on the contrary, receive a random sequence of `GET` and `POST` requests that can only be linked by the application using HTTP cookies. Hence, unauthorized request sequences do not violate the HTTP protocol specification.

We have seen that an established connection between a client and a server can serve as the basis for integrity and confidentiality protection of messages (as SSH does) and to control the sequence of user actions on the protocol level (as SMTP does). Protocols establishing a connection are called *connection-oriented protocols*:

Definition 1 (Connection-oriented Protocol)**.** *A client-server protocol is connection oriented if the participants first establish a connection. They may negotiate connection parameters that hold until renegotiation or connection closure. In the second phase, the participants exchange the payload. Parameters from the first phase are applied to exchanged messages. Recipients can put messages into correct order. Finally, the connection is finished and no payload can be exchanged unless a new connection is established.*

The connection setup phase allows mutual authentication and the exchange of symmetric keys. Those keys can be used to preserve message integrity and confidentiality in the second phase. The established connection can also preserve control-flow integrity and protect against unauthorized commands from third-party applications: Both properties are granted if the second phase follows a well-defined control flow (like SMTP, see above) and access to the connection on the client side is only permitted for authorized applications. The first phase can be used to agree on an application-specific control flow if necessary. Referring to the definition above, SSH and SMTP are connection-oriented protocols. HTTP, however, is a *connectionless protocol*[3]:

Definition 2 (Connectionless Protocol)**.** *A client-server protocol is connectionless if the client starts sending the payload without a preface. The protocol does not guarantee that messages are received nor the order of reception in case of successful delivery.*

Considering the definitions above, we can derive an important property of connection-oriented protocols: Due to the correct ordering of messages, the protocol can define variant sets of messages for each protocol step. The sequence of received messages adheres

[3]The definitions are based on Tanenbaum's definition of *connection-oriented* and *connectionless service* respectively [168]. Tanenbaum distinguishes between services and protocols. In his definition, services provide an interface to the service users on higher layers. The communication protocol used by the same-layer services on two hosts may be changed without actually changing the service. In our case, however, we focus on the world wide web and consider the protocols on each layer given.

to the protocol if and only if the sequence of sent messages adheres to the protocol. We showed above that SMTP makes use of this property because every email transmission must start with the characteristic `MAIL` command. The established connection ensures that this command is received before the subsequent `RCPT` denoting the recipient of the email.

This property does not hold for connectionless protocols. As messages can arrive in any sequence, each received message must be processed independently of previously received messages. For this reason, connectionless protocols have constant sets of messages and no protocol steps.

Looking at the historic context of HTTP, the choice of a connectionless protocol makes sense: The protocol was meant to enable requests for and the delivery of public and static text documents. There was no need for client or server authentication, nor message integrity or confidentiality, and no control-flow integrity or authorization. Instead, the inventors of the world wide web around Tim Berners-Lee chose the protocol that perfectly fits their needs and remains as simple as possible.

Today, however, it is more difficult and cumbersome to meet the security requirements of the web using a connectionless protocol than using a connection-oriented protocol because every property must be implemented individually by each application. In doing so, malfunctions and incompatibilities are inevitable. In the end, every application provider faces a tradeoff between functionality and security.

In the next section, we shed light on the protocols used in the world wide web on the layers below HTTP to see how far these protocols can support the intended security features.

3.2 Connection-oriented Protocols in the Web Stack

The goal of this section is to find out how far the security requirements of modern web applications can be met on the layers below HTTP. We consider the TCP/IP reference model as the basis for the web [168]. The respective layers and protocols are IP and IPsec on the Internet layer, TCP on the transport layer, and HTTP plus SSL/TLS on the application layer. In the remainder of this thesis, we call these protocols the *web stack*. In our analysis, we focus on the connection-oriented protocols IPsec, TCP, and SSL/TLS. The last section showed that connectionless protocols can hardly achieve the desired properties.

3.2.1 IPsec

While the most common protocol of the network layer, the *Internet Protocol (IP)*, is connectionless, there is a derivative connection-oriented protocol suite named *Internet Protocol security (IPsec)* that provides confidentiality, authentication, and integrity. There are plenty options and parameters for running IPsec. Among others, IPsec can connect two hosts, two networks, or one host with one network. We focus on the abstract

features that are common for all the variations and use the term "peer" to refer to IPsec end points.

IPsec adheres to the three-phase schema we described for connection-oriented protocols: First, the *Internet Key Exchange protocol version 2 (IKEv2)* [93] is used to authenticate the participating peers, establish a secure channel, and finally exchange so-called *Security Associations (SAs)*. Each SA subsumes a security configuration including cryptographic algorithms, keys, key lifetimes, and initialization vectors. There is one SA for each pair of communicating peers and each direction. In the second phase, the peers exchange data and apply an SA for each message. The last phase is connection termination.

IPsec provides security features on a host-to-host level but not on application-to-application or user-to-user levels, i.e. there is no authentication of users or applications. Control-flow integrity and application-to-application authorization are out of scope for IPsec.

3.2.2 TCP

The *Transmission Control Protocol (TCP)* implements the transport layer of the web stack. Its purpose is to make sure that sent messages are finally received, permutations of messages can be detected, and repeated messages can be discarded. So, TCP provides a feature of connection-oriented protocols as a service to the upper layers.

The three phases of TCP are first the connection establishment by a so-called three-way handshake. Then, the payload can be exchanged over the connection. Finally, the client or the server can initiate the connection finalization phase.

TCP introduces port numbers. Given that each port is assigned to only one application, TCP provides address-based application authentication. However, the information contained in a TCP header is not cryptographically protected making it vulnerable to spoofing attacks. There is no user authentication nor message integrity, confidentiality, nor application-to-application authorization. TCP allows the receiving host to put received messages in order. However, there is still no control-flow integrity protection for two reasons: First, a new TCP connection is established for each HTTP round trip, meaning that TCP can only enable the correct defragmentation of single HTTP messages but not control sequences of HTTP round trips. Long-term TCP connections consume considerable resources on the server side and are not an option[4]. Second, TCP makes sure that sent messages are received in order but can not check if the user is authorized to send the message.

[4]We outlined in Section 1.1 that recent HTTP versions allow re-using an established TCP connection. This, however, is meant to transmit all elements of a single web page. The TCP connection is closed when the page is loaded, and a new TCP connection must be established for the next step in the web application's workflow.

3.2.3 SSL/TLS

The *Transport Layer Security* protocol and its predecessor *Secure Socket Layer* provide security features on a sublayer of the application layer. The differences are so marginal that both protocols are used interchangeably. We focus on the more recent TLS and omit mentioning SSL for now.

TLS establishes a connection between two applications, e.g. a browser and a web application. It provides integrity, confidentiality, and authentication for messages exchanged between those applications. Applications must explicitly enable TLS by calling TLS functions instead of TCP functions. Hence, it is not transparent like e.g. IPsec. There are a couple of application-layer protocols being run over a secure TLS link. Beside the Hypertext Transfer Protocol (HTTP) which is most interesting for this work, there are for instance the Simple Mail Transfer Protocol (SMTP), the Post Office Protocol version 3 (POP3), the Internet Message Access Protocol (IMAP), and the Session Initiation Protocol (SIP). Usually, the TLS-enabled applications run on different TCP ports than the non-TLS version of the same application. This fact illustrates that TLS is interwoven with the application layer and not only a protocol on a lower layer.

TLS implements the three phases of connection-oriented protocols the following way: The client starts a TLS session with the server by performing a so-called TLS handshake. During this handshake, client and server first agree on the protocol version and the cipher settings to use. The client can check the server's authenticity based on the presented certificate. Vice versa, client authentication is also possible but without practical relevance. When the connection is established, exchanged messages are integrity and confidentiality protected. Finally, the connection is closed.

TLS is one step forward towards secure web communication. However, TLS can not meet all security requirements of today's web:

- First, only the website that is actually requested can be authenticated. This does not necessarily mean that it is the site the user wanted to access as typographical errors and phishing attacks have shown.
- Second, the user is not authenticated. Client-side TLS authentication is defined but key handling turned out to be too complicated for most users in practice.
- Third, TLS does not maintain a session on application layer. The link between HTTP and TLS is not implemented on session level. This means that attackers can steal an HTTP session token and reuse it in a different TLS session[5].
- Fourth, TLS can not provide security for cross-domain communication. There are legitimate and malicious requests from one web application towards another. The distinction of both is out of scope for TLS, because TLS does not cover authorization issues.

[5]There are approaches against session hijacking attacks that bind the HTTP session to the TLS session [8, 6]. These were published during our research on this topic and have no practical relevance so far.

- Finally, though TLS is connection oriented, an attacker's messages compromising the application's control-flow integrity still comply with TLS.

The last two points, TLS' inability to prevent cross-domain attacks and attacks on control-flow integrity, show that TLS is meant as a universally applicable protocol on a sublayer of the application layer, e.g. together with SMTP, POP3, IMAP, SIP (see above). The root cause is that TLS is agnostic about the syntax of application-layer protocols. Concerning web applications, TLS can not address web documents which is the main purpose of HTTP. So, TLS can not enforce authorization policies. Such policies, however, are necessary to express the allowed sequences of user requests and access rights for third-party web applications.

Practical Issues with SSL/TLS

There are a couple of practical issues with SSL/TLS beside the conceptual issues concerning the security requirements of today's web applications.

SSL Stripping Most web applications serve content both encrypted, via HTTPS, as well as unencrypted, via HTTP. Unfortunately, browsers default to HTTP if the user does not explicitly specify the protocol when she accesses a web page. In consequence, in the majority of all cases, the first HTTP request to a server is sent via plain HTTP. This opens a loophole for a network-based man-in-the-middle attacker – the so-called SSL-stripping attacks [110]. For this first request, an end-to-end SSL/TLS connection has not been established yet. Thus, the attacker can set himself in between the browser and the server and modify the server's responses. This way, even if the server requires HTTPS for certain operations and tries to redirect the browser accordingly, the attacker can simply remove these redirection attempts from the server's responses, before they reach the client. The client is forced to indefinitely communicate unencrypted.

Further Issues with SSL/TLS The recent past has shown, that the current state of SSL/TLS is not fully bullet proof. For one, the security of HTTPS-based communication heavily relies on the security policies and practice of the Certification Authorities (CAs) that issue the root certificates which are included in web browsers by default. However, issues in that domain have been reported repeatedly, e.g., unlimited RA certificates have been issued [75] and the internal systems of several CA's have been compromised [50, 49]. As the CA system and its security is out of reach of the application's developers and operators, the current approach offers severely limited options to mitigate such threats.

3.2.4 Wrap-up

All protocols in the web stack are connection-oriented except the Internet Protocol (IP) and the application-layer HTTP. The protocols in between use connections in order to achieve integrity and confidentiality (IPsec, TLS) as well as reliability (TCP) properties over a connectionless, unprotected and best-effort-based IP connection. Clients and

servers first exchange necessary parameters in a multi-step connection establishment phase. Then, they maintain the agreed configuration and use the secure connection until it is closed. For the whole protocol run, both client and server can easily detect if a message does not comply with the protocol. For instance, the server can immediately cancel communication if a client does not start the TLS handshake with the characteristic `ClientHello` message including supported cipher suites.

To sum up, disregarding the practical security issues of SSL/TLS that have come up in the last years, there are protocols in the web stack to authenticate hosts (IPsec) or applications (SSL/TLS) and provide message integrity and confidentiality. However, user authentication and application authentication on user level, i.e. verifying that the requested application is the intended, is not covered. Finally, addressing web documents and, thus, all kinds of authorization issues lie in the domain of HTTP.

3.3 The Unfeasibility of Connection-oriented HTTP

Regarding the current security requirements of web applications, a connection-oriented HTTP version seems natural. Browsers and web applications regularly establish sessions on HTTP level in order to chain requests. So far, these sessions can not meet the requirements phrased above. By admitting connections on application level, browsers and web applications could negotiate security parameters in a session establishment phase. A multi-step mutual challenge-response authentication on protocol-level could overcome phishing attacks and allow the establishment of session keys. Using session keys, authentication could be preserved until the session ends. Concerning the authorization issues which appear as CSRF attacks and attacks on control-flow integrity, a connection-oriented version of HTTP could exchange workflow definitions in the establishment phase. Any violation of such definitions would become observable immediately. In order to overcome cross-application authorization problems, every third-party web application first needs to establish a connection with the target application, e.g. via the user's browser. This allows to enforce sophisticated and customizable authorization policies.

In practice, however, a connection-oriented variant of HTTP is unrealistic for two reasons: First, replacing the communication infrastructure of the full-grown web requires tremendous efforts and is fault-prone by nature. Second, introducing a new protocol that almost doubles the number of exchanged messages and binds resources for a rather long time is not an option in a web where performance is the first selling point. In this thesis, we present the results of our applied research activities. We show that we can head for secure HTTP sessions using our add-on approaches for legacy connectionless HTTP.

4 Repelling pre-Authentication Attacks

After we explained in the last section that a connection-oriented version of HTTP could meet the security requirements of modern web applications but is not achievable under real-world assumptions, we now start to describe our contributions to web-based secure application control given the well-known connectionless HTTP protocol.

4.1 Motivation

The lack of an established connection means that there is no closed channel between both communication participants. Instead, they establish a virtual channel that can be addressed via an ID (see Section 2.3). Every adversary knowing this ID gains access to the channel and can impersonate the actual account owner.

We will show that it is crucial to issue a fresh ID when the authentication status of the user changes to a higher level, e.g. from an anonymous session to a regular user or from a regular user to an administrator. Leaving the ID unchanged means that the ID's security properties concerning confidentiality and integrity must be upgraded according to the associated privileges . As it is impossible to make possibly leaked information private again, a new ID must be issued.

Contribution

Our contribution is threefold: For one, we give a thorough documentation of existing attack vectors and take steps to assess the attack surface of web applications developed with state-of-the-art web frameworks. Second, we provide an approach for transparent, light-weight protection on the framework level. It allows 'patching' web applications without access to the code but just to the underlying framework. Furthermore, we developed a proxy-based solution that implements session fixation protection with neither access to the application code nor to the framework. In addition, we explain session fixation prevention at the development phase and, thus, provide comprehensive protection against session fixation vulnerabilities.

4.2 Exploiting Session Fixation

We first describe various options for the preconditional session ID fixation in the victim's browser. Then, we check the practical feasibility of one of these attack vectors and assess how vulnerable the most common open source CMS are for session fixation attacks.

4.2.1 Attack Vectors

Given the sources of session fixation vulnerabilities as described in Section 2.5.3, a broad field of attack vectors comes into play. All attacks strive to 'implant' a known SID to the victim's browser. As described in Section 2.3, SIDs are communicated either via HTTP parameters or cookies.

URL Parameter

The easiest way to push an SID to the user's browser is a URL parameter (see Figure 2.1) in situations in which the attacked application accepts such SIDs. The attacker generates a valid SID, e.g. `1337`, of the vulnerable application on address `http://www.example.com`. Then, he sends the URL `http://www.example.com/?SID=1337` to the victim. For example, he could promise that the pictures from his last summer holiday are published there. The victim's browser sends a request to the application making use of the given SID if she clicks on this link. From the application's point of view, she already has a valid session. So, it does not need to deliver a new SID.

As a next step, the victim provides her credentials and logs in to the application. The session has not changed but she is now logged in. The application still identifies the victim by her SID that is sent with every request. However, the attacker also knows this SID and can thus act as the victim in her account. Therefore, he does not even have to interfere in the communication between the victim and the application. The only thing to do is sending a request to the application like `http://www.example.com/account_balance.php?SID=1337`.

Cookies

A slightly more difficult way for the attacker are cookies set by the application. In this scenario, the adversary requests a session cookie from the application. Then, he needs to make his victim accept the same cookie. There are several options for him to reach his goal as we will show. When the victim owns the adversary's session cookie, her browser will provide this cookie to the application. As the adversary can use this cookie, too, he can own his victim's session.

- **Setting cookies via XSS:** The adversary can use XSS to set a cookie in his victim's browser if the web application is vulnerable to this attack. Therefore, he inserts JavaScript code into a web page in the same domain, `.example.com`. When the victim visits this page, the cookie will be set. The adversary can set the expiry date to a date in distant future so that the cookie will not be deleted when the victim restarts her browser. Unlike a cookie stealing attack, the victim does not have to be logged in, thus, the attack also works at the public part of the application. In the past, we saw browser vulnerabilities that even allowed the attacker to set the cookie from a foreign domain [201].
- **Setting cookies via `meta` tags:** Under certain circumstances, a web application might allow user-provided HTML markup but filters user-provided JavaScript. In

such cases, the attacker might be able to inject special `meta` tags. The tag `<meta http-equiv="Set-Cookie" Content="SID=1337"; expires=Monday, 07-Mar-2016 12:00:00 GMT">` sets a cookie which is valid until March 2016.

- **Setting cookies via cross-protocol attacks:** The same-origin policy [150] explicitly includes the port of the URL that was utilized to retrieve JavaScript code as a mandatory component of its origin. In consequence, if a service in a given domain allows the adversary to execute JavaScript, e.g., via XSS, services in that domain which are hosted on different ports are unaffected by this.

  However, HTTP cookies are shared across ports [103]. Thus, cross-protocol attacks come into play: Cross-protocol attacks [179, 3] (see below for further explanations) allow the adversary to create XSS-like situations via exploiting non-HTTP servers, such as SMTP or FTP, that are hosted on the same domain.

  First, the adversary prepares a website with a specially crafted HTML form [179]. The form contains JavaScript code to set the attacker's cookie at his victim's browser. The target of this form is the targeted non-HTTP server[6]. To avoid URL encoding of the content, the adversary chooses the `enctype="multipart/form-data"` parameter. The target server interprets the HTTP request as a valid request of its own protocol, e.g. FTP [39]. However, most of the incoming commands cannot be understood and will be reflected with a respective error message. This message generally contains the erroneous input which is in our case the JavaScript code with the cookie. For compatibility reasons, browsers take all textual input as HTTP even without any valid HTTP header. So, the browser accepts the cookie since it comes from a server in the same domain.

- **Subdomain Cookie Bakery:** JavaScript's same-origin policy prohibits the setting of cookies for other domains. Nevertheless, this only applies to top level domains. Therefore, a possible attack vector is setting a cookie using a vulnerable subdomain, e.g. JavaScript code or a `meta` tag on `vulnerable.example.com` can set a cookie which is subsequently sent by the victims browser for `example.com`. Hence, a vulnerable application on a subdomain may compromise the session security of the whole top level domain.

  Web applications running on so-called DynDNS (dynamic DNS) domains are particularly susceptible to subdomain cookie bakery because all users of the service share the same domain. There is a similar scenario with a number of shared hosting providers.

- **HTTP Response Splitting:** In case the web application is vulnerable to HTTP response splitting, the attacker could use this to send his cookie to the victim's browser [96]. Therefore, he needs a redirected page that includes unfiltered user-provided content and a proxy with a web cache.

[6] NB: Some browser, e.g., Firefox, block HTTP requests to certain well known ports. In such cases the attack requires a susceptible server on a non-blocked port to function.

First, the attacker sends a crafted HTTP request through the proxy to `http://www.example.com/redirect.php`. The application takes information from the attacker's request to generate the target of the redirection, e.g. `/app_by_lang.php?lang=`*attacker's parameter*. This way, he can influence the `Location` header field where the redirection URL is denoted. He can then shape a request that finishes the HTTP response and append a new response. The latter is now fully controlled by himself, i.e. he can use the `Set-Cookie` header and deliver his cookie, e.g. `/redirect.php?lang=en%0d%0a Content-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aSet-Cookie:%20SID=1337%0d%0a Content-Length:%2019%0d%0a%0d%0a<html>Foobar</html>` [96]. The proxy caches the second response if the attacker manages to send a harmless request to `http://www.example.com/` at the right time. The actual answer to his second request will be discarded and considered superfluous. Finally, the proxy serves his specially crafted HTTP response to requests on `http://www.example.com/`.

The attack works similarly if the application takes user-provided data to set a cookie. In this case, the `Set-Cookie` header serves as an entry point instead of the `Location` header.

Figure 4.1: A header injection attack to fixate a session cookie in the victim's browser.

- **HTTP Header Injection:** Finally, the attacker could run an HTTP header injection attack, see Figure 4.1. He prepares a special URL like `http://www.example.com/app_by_lang.php?lang=en%0d%0aSet-Cookie:%20SID=1337%0d%0a` and sends this to his victim. Then, the victim has to click on the link. The web application's response will set the cookie in the victim's browser. When the victim logs in, the attacker can own her account. We tested the feasibility of header injection attacks and give results in Section 4.2.3.

4.2.2 Impact and Discussion

Currently, session fixation is only a second stage attack that usually requires several preconditions.

The attacker needs another vulnerability to provide his cookie to the victim. Alternatively, he must mislead the victim into clicking on his link if the target web application allows URL parameters for session management. We presented different attack vectors in Section 4.2.1. In case the attacker is successful at the first step, he has to make the victim log into her account. The SID is useless as long as it does not belong to a logged-in user. However, the attacker neither knows when the victim logs in nor when she logs out again. He has an unknown window of opportunity. Finally, the target web application has to be vulnerable (see Section 2.5.3 for an analysis of the vulnerability). Actually, it would be essential for a broad attack to provide a unique cookie for each victim. Otherwise, one victim would take the session of another. This is however not always easy to implement for the attacker depending on the attack vector.

When the conditions are met, though, session fixation is a severe attack that allows the attacker to fully impersonate the victim. It is generally not obvious to the victim to be under attack, especially if she is not familiar with session fixation. Most attack scenarios appear to be a software malfunction to the unexperienced user. The victim may even not notice the attack afterwards depending on the actions of the attacker on her behalf.

4.2.3 Practical Experiments

Next, we take steps to assess to which degree session fixation poses a realistic threat. For this purpose, we conduct two series of practical experiments.

First, we test open-source content management systems (CMS) for session fixation issues. This way, we aim to get an estimate if developers are actually aware of the lingering threat of session fixation or if they unknowingly rely on the framework's protection mechanisms.

The result of these tests suggests that a considerable fraction of existing applications indeed are susceptible to session fixation under circumstances that allow the attacker to set cookies on the victim's browser (see Section 4.2.1). Consequently, we examine several popular web application frameworks with respect to their susceptibility to potential header injection vulnerabilities.

Examination of Open-source CMS

We adhere to the following testing methodology in order to assess an application's susceptibility to session fixation (see Figure 4.2): First, we verify that the application indeed issues SIDs before any authentication processes have been undertaken. Then, we test if the application leaves the SID unchanged in case a successful authentication process has happened. If these tests could be answered with 'yes', we conclude that under certain circumstances the application could expose susceptibility to session fixation. The final test probes which attack vectors (see Section 4.2.1) were applicable.

We consider several open-source web applications in their configuration 'out-of-the-box'. The applications implement user management on their own. The results are given in Table 4.1. A plus sign (+) in a column denotes that the application is vulnerable to

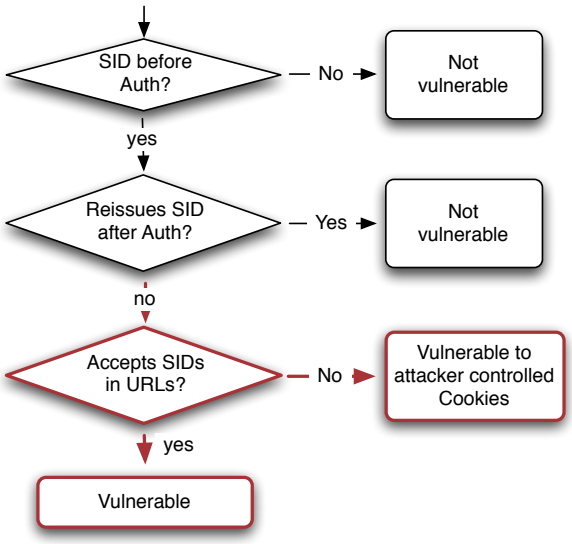

Figure 4.2: Our testing methodology to assess an application's susceptibility to session fixation attacks.

the respective attack vector. An application is vulnerable to *Cookie* if it accepts foisted cookies. The attacker can be successful if he manages to set a cookie at the victim's browser. In case the application allows session tracking via a URL parameter, it is vulnerable to *URL*. Finally, those applications which are vulnerable to *SID* allow the attacker to utilize arbitrary SID values that have not been generated by the application in the first place.

Running Header Injection Attacks

We analyze common web application frameworks to comprehend the feasibility of HTTP header injection attacks. Therefore, we implement web pages which either set a cookie or do forwarding respectively. The web pages accept attacker-controlled input to determine the forwarding target and the cookie value.

- **PHP:** Usually, HTTP forwarding in PHP is implemented by the use of the `header()` function. We implement a page that takes one parameter and inserts this into the `Location` header field. We provide valid input but append the payload to set our cookie. In the initial configuration (PHP 5.3.0, Apache 2.2.11), the cookie is not set but we get a warning message because `header()` may not contain more than a single header line since version 4.4.2 and 5.1.2 and our new line is detected. Then, we downgrade (PHP 5.1.1, Apache 2.0.63) and the attack is successful. As the forwarding works as expected, we are able to set the cookie 'drive-by' without any notice for the victim.

  Cookie setting is done with the `setcookie()` function. We append a line break and a new `Set-Cookie` header. However, the function URL-encodes the cookie

Application	Version	Cookie	URL	SID	Lang
Joomla	1.5	+	-	+	PHP
CMSmadesimple	1.6.6	+	-	+	PHP
PHPFusion	7.00.06	-	-	+	PHP
Redmine	0.9.2	+	-	-	PHP
XWiki	2.0.2.24648	+	-	-	Java
JAMWiki	0.9	+	+	+	Java
Wordpress	2.9.1	-	-	-	PHP
Novaboard	1.1.2	+	-	+	PHP
PHPBB	3.0.6	-	-	-	PHP
SimpleMachinesForum	1.1.11	-	-	-	PHP
Magento Shop	1.3.4.2	+	-	-	PHP
OSCommerce	2.2 RC 2a	+	-	-	PHP

Table 4.1: Results of our tests on session fixation vulnerabilities of open-source CMS. A plus sign (+) in a column denotes that the application is vulnerable to the respective attack vector.

value and thus transforms our `:` and `=` characters to non-interpreted URL codes. We can avoid URL encoding of our input by using `setrawcookie()`. However, it prohibits control characters in the value field.

- **J2EE:** For the Java test scenario, we use Tomcat 6.0.20 as a servlet container. The redirection could not be successfully spoofed. The payload is treated as part of the URL. During our cookie setting approach, we get a `java.lang.IllegalArgumentException` due to the control characters in the payload.

- **CherryPy:** We make use of CherryPy (version 3.1.2) to test the Python functions. The injection to the `Location` header is successful whereas the cookie value turns out to be not exploitable.

- **Perl:** We also implement a Perl (version 5.10.1 with `CGI.pm` version 3.48) script that does nothing but forwarding to a given site. Indeed, we manage to set a cookie at the victim's site, however, for some reason the equal sign between the cookie name and its value is finally coded as a colon and a blank. So, we get a cookie with a given value but without a name. Then, we insert two equal signs and the second one is not recoded. Actually, we are able to set an arbitrary cookie which name ended with a colon.

  The cookie setting scenario is more difficult due to the URL encoding of cookie names and values. We are not able to set our own cookie given that we can influence the value of an unimportant cookie.

- **Ruby on Rails:** For Rails (version 1.9.1), we omit the tests for header injection in forwarding sites as this vulnerability was recently patched [192, 155].

During the cookie task, we face the same situation as in the Perl scenario. Cookies must be declared and cannot be set as ordinary headers. The value that we insert is thus first URL encoded and never interpreted. So, we do not find a way to escape the cookie value context.

4.3 Server-side Measures Against Session Fixation

With 66 % of the examined CMS being vulnerable to at least one attack vector, we list three alternative approaches to counter session fixation. The proposed techniques are designed to fit different situations in respect to the degree of control of the vulnerable application's source code or the application server respectively.

4.3.1 Code-level Countermeasures

As described above, the root cause of session fixation problems is in general a mismatch in the implementation of the session handling, which usually is done on the framework level, and the authentication management, which is realized on the application layer. Consequently, the application's developer has to renew a user's session identifier manually every time this user's authentication state changes (see Figure 4.3) to be secure against session fixation. Note that only the SID is renewed but the stored session data (e.g. a shopping cart) is then tied to the new SID.

```
1 if ($authentication_successful){
2     $_session["authenticated"] = true;
3     session_regenerate_id();
4 }
```

Figure 4.3: Protecting an application on the code level against session fixation attacks, exemplified at PHP [140].

While this requirement can be fulfilled rather straight forward for newly written applications, the same task might prove hard for non-trivial legacy applications, depending on the complexity of the application's authentication management and its degree of encapsulation within the code base.

In addition, assessing if a given application is susceptible to session fixation based on the application's source code alone is also non-trivial. In most cases, a manual test through monitoring and manipulating HTTP communication with the application is easier (see Figure 4.2).

4.3.2 Protection on the Framework Level

We designed a transparent, light-weight solution that takes protective measures on the framework level in order to overcome the divide between the framework's session tracking

and the application's authentication management that is responsible for session fixation vulnerabilities.

Protection Methodology

Our approach functions by mirroring the advised behaviour of Section 4.3.1 within the application framework: Whenever an authentication process has been executed, a new session identifier is generated. However, on the framework level, no knowledge about the internal processes of the application exists. Therefore, the protection mechanism has to deduce that an authentication process has taken place through observations of data that is available on the framework level.

While many characteristics in respect to observable application behavior depend on the utilized combination of application framework and server, we expect one data source to be available universally: the ongoing HTTP communication between the user and the application. Therefore, our solution aims to derive the information regarding authentication processes from this data.

We propose the following methodology: The countermeasure is integrated in the framework's component that is responsible for parsing incoming HTTP requests. These requests are examined whether they contain HTTP parameters that might carry password data. Such parameters can be identified by their name, provided by the application's operator. Whenever such a parameter is detected in an incoming request, the framework-internal functions for the session identifier regeneration are triggered to create a new SID value for the user. This approach renews the session identifier even if the authentication attempt fails. However, this does not pose a problem as the user and the application then share a new valid SID.

Implementation and Evaluation

For our practical experiments, we chose the J2EE application framework [167] as the implementation target. We realized the actual protection mechanism in the form of a J2EE filter. J2EE filters are a properly defined way to add framework components to applications that intercept all incoming and/or outgoing HTTP communication (see Figure 4.4). Our filter implements the functionality as described above: All incoming HTTP requests are examined for HTTP parameters which carry the name of a preconfigured password field. If such a parameter is found, the J2EE session container is triggered to issue a fresh JSESSIONID value to the user's session. So, the session data remains with a new identifier.

Realizing the mechanism in the form of a J2EE filter has several advantages: Foremost, no changes to the application server have to be applied, all necessary components are part of a deployable application. Furthermore, only minor changes to the application's `web.xml` meta file have to be applied to integrate our mechanism into an existing application. The only configuration that has to be done is providing the name(s) of the application's password parameter(s). Thus, outfitting an existing J2EE application with our solution is easily and quickly done.

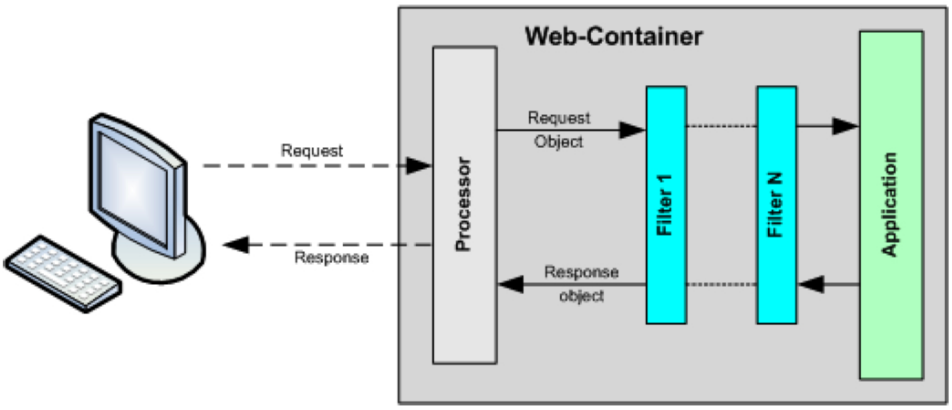

Figure 4.4: The model of J2EE filters to handle request and response objects.

We test our implementation manually using a vulnerable J2EE application. For this purpose, we choose the open-source J2EE-based Wiki JamWiki, Version 0.8.0 [80] which is susceptible to session fixation[7]. After installing the software on our test system, we verify that the installed version is in fact vulnerable, using the testing method outlined in Section 4.2.3.

In the next step, we add our J2EE filter to the installation and enter the name of JAMWiki's password parameter (`j_password`) to the filter's configuration file. Finally, after restarting the application server, we verify that after every login attempt, the JSESSIONID value indeed changes and, thus, the vulnerability is properly mitigated.

4.3.3 Protection via a Reverse Proxy

In certain situations, it is neither feasible to fix a vulnerable application's source code nor to apply a framework-level countermeasure, as described in Section 4.3.2. Such scenario include, for instance, the hosting of closed-source applications, mission-critical applications which cannot be patched timely because of otherwise expected downtime, or legacy applications that require frameworks which do not support session re-generation, such as PHP prior to version 4.3.2 [140]. Furthermore, sometimes a short-term solution is needed even if an application-inherent fix can be applied later, e.g., when an identified vulnerability is under active attack and the fix is still under development.

In all these scenarios, an application-external solution is required – a generic self-standing protection mechanism that does not necessitate alteration of the actual application or its application server. In this section, we propose a method for transparent, proxy-based protection against session fixation attacks for such scenarios.

[7]We discovered the software's vulnerability during our first experiments with session fixation. We informed the JAMWiki authors and the vulnerability is fixed.

Challenges

Several hurdles have to be overcome to implement such a solution. In this section, we list the identified problems and briefly outline our corresponding solutions.

- **Application-external solution:** The protection mechanism necessarily has to take its measures outside of the actual application as in the given situation (see above) resolving the situation directly at the application is not possible. Consequently, we designed our solution in the form of a server-side reverse proxy.
- **Complementing the application's session management:** Our proxy has no direct control over the application's internal session management, unlike our solution that we presented in Section 4.3.2. For this reason, our solution has to be able to invalidate fixed sessions while maintaining legitimate application usage. We solve this problem through the introduction of a secondary session identifier that is issued by the proxy (PSID). The proxy's identifier management component is tightly secured against session fixation and only requests which carry a valid PSID are forwarded by the proxy to the actual web application.
- **Login detection:** Similar to the framework-level solution described in Section 4.3.2, detecting that a login process has happened is crucial for the solution to function properly. We tackle this problem analogously.

In the following section we give details on how we solved the above mentioned problems.

Protection Methodology

As outlined above, we introduce a proxy which monitors the communication between the user and the vulnerable application. The proxy implements a second-level session identifier management. In addition to the SIDs that are set by the application, the proxy issues a second identifier (the *proxy SID* – PSID).

Whenever an HTTP request without a PSID value is received by the proxy, this request is regarded to be the user's very first request to the application. If the request carries any stale SID values, such data is discarded. For the corresponding HTTP response a fresh PSID value is generated and attached to the response via `set-cookie` (see Figure 4.5). In the course of the following HTTP communication, the application's responses are monitored for outgoing SID values that are to be assigned from the application to the user. If such a value is detected, the combination of the PSID and SID value is stored by the proxy. From now on, only requests that contain a valid combination of these two values are forwarded to the application (see Figure 4.6). Requests that are received with an invalid combination of SID/PSID are treated as if they would carry no session information. Consequently, they are stripped off all `Cookie` headers before sending them to the application and are outfitted with a fresh PSID value upon response.

The proxy monitors the HTTP requests' data for incoming password parameters to provide protection against session fixation. If a request contains such a parameter,

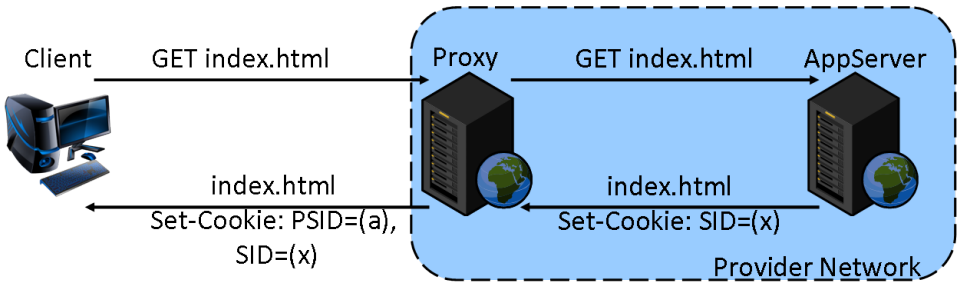

Figure 4.5: Issuing the proxy SID against session fixation attacks.

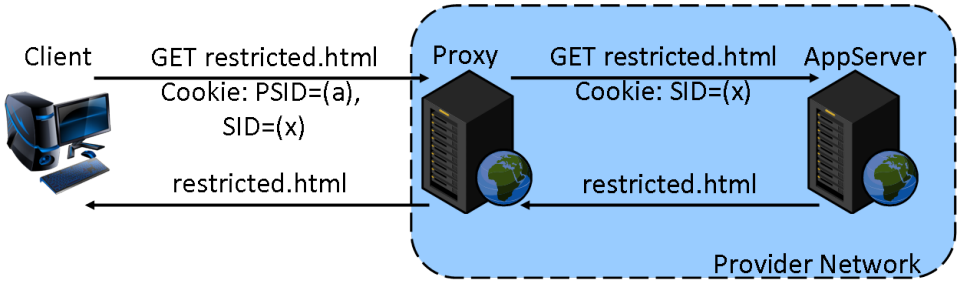

Figure 4.6: Verifying the proxy SID to overcome session fixation attacks.

the proxy assumes that an authentication process has happened and renews the PSID value, adds an according `Set-Cookie` header to the corresponding HTTP response, and invalidates the former PSID/SID combination. This way, only the session identifier is renewed whereas the session data remains unchanged. The PSID is even renewed if the authentication attempt fails. This, however, is no threat as the new PSID does not have any security requirements.

Implementation and Evaluation

We implement a prototype in Python to test our approach and utilize *CherryPy* [171] as the basis for the proxy server. CherryPy is a lightweight web framework which offers a smart interface for developers of web applications. CherryPy only provides the framework for handling incoming requests and rendering responses to clients, hence, providing the proxy's front-end. The Python module *Urllib2* [61] implements the back-end communication with the vulnerable application. It provides methods for retrieving data from a URL using either HTTP GET or POST requests and access to appended cookies.

The proxy implements the issuing and verification of PSIDs as described above. The PSID value is stripped off incoming requests before they are forwarded to the application to avoid potential problems that some applications might expose when they receive unexpected parameters or headers.

We again test our implementation against JAMWiki and verify the provided protection by observing the proxy's behavior in respect to ongoing login processes.

4.3.4 Discussion

We presented three different protective measures that can be utilized by a web application's operator to avoid session fixation problems. Each of the measures is targeted at a distinct scenario in respect to the level of control that the operator has when it comes to altering the web application's internals.

Fixing the problem within the application logic through reworking the authentication handling code, as shown in Section 4.3.1, should always be the first choice as long as no restrictions exist when it comes to altering the source code and timely applying the resulting security patch. The main problem that can be encountered in this scenario is that the authentication and session handling code in a given application might turn out to be non-trivial and spread. For the fix to function properly, it is essential that *all* code segments, in which a user's authorization level changes because of an authentication process, are addressed correctly. If one of such processes is missed in the creation of the security patch, the protection is incomplete. For this reason, the handling of session fixation should be an integral part of the software development process and addressed from the begin on.

The second proposed measure (see Section 4.3.2) is applicable if a direct alteration of the application's source code is not feasible but the utilized application server and framework are under full control. Such situations mainly arise, if the operator of the application is a different entity than the application's developer. This applies, for instance, for third-party components, closed-source applications, or legacy applications for which the original author has left the company long time ago. The described approach provides reliable protection, as every authentication process causes the framework to renew the SID value. Furthermore, the approach is very light-weight. This stems from two characteristics: For one, the mechanism is completely stateless. It does not require temporary storage of any data as it only reacts based on incoming password parameters. Furthermore, it is closely integrated to the existing framework infrastructure. Consequently, there is no need to execute any complex operations on its own – all the hard work, such as parsing the HTTP headers and parameters, is done by the application framework. These characteristics result in a runtime behavior that, at least if implemented in the form of a J2EE filter, does not cause noticeable performance overhead. Finally, as the mechanism operates completely transparent to the application, the patch of an existing application is easy and straightforward. The main drawback is that the implementation of the countermeasure is specific for an application framework. The protection might be lost and has to be reintroduced if changes in the runtime infrastructure are taken, such as exchanging the underlying application server – a characteristic that does not apply to handling session fixation directly on the source code level.

The third discussed countermeasure (see Section 4.3.3) is to be used whenever no changes at all to the vulnerable system are possible (see above for a list of reasons). It is designed to be completely self-standing and can be set up to protect arbitrary web applications simply by positioning it between the application and the user. It is reasonable to expect that the proposed mechanism can be easily integrated to an existing web application firewall (WAF) [137]. WAFs are in essence web proxies which

were introduced for the exact same scenario as the discussed countermeasure – to protect against web application attacks without altering the application itself. Consequently, WAFs already handle all operations, such as parsing incoming requests, that are required by our countermeasure. In turn, our countermeasure itself is comparatively light-weight and does not add significant complexity to a WAF's functionality.

It depends on the given situation which of the three described measures is to be taken. In general, if possible, the solution that is most closely integrated to the application's core functionality should be chosen to reduce the setup complexity and avoid potential security regression due to future changes in the application's infrastructure.

4.4 Related Work

Session fixation has received little attention in the past, mostly due to the vulnerability's obscurity and the fact that more severe XSS vulnerabilities are still very common in current web applications.

The first public paper on session fixation appeared in 2002 [99]. It describes the basic fundamentals of the attack and the most obvious attack vectors. However, it provides no information about the spreading of the vulnerability or more advanced attack schemes. Furthermore, OWASP and WASC added articles about session fixation to their security knowledge bases. The OWASP article [136] briefly names common session fixation issues and attack vectors. In contrast, the WASC article [191] also provides small code examples of different attacks. Both sources name the HTTP response splitting attack as an attack vector, but they do not discuss its impact in today's world of web applications. Furthermore, they lack a general rating of the attack.

Web application firewalls are server-side proxies that aim to mitigate security problems. However, as the OWASP best practices guide on web Application Firewalls (WAF) [137] states, current WAFs can only prevent session fixation "if the WAF manages the sessions itself." In comparison, our approach does not touch the application-level session management and only introduces additional security-related information for each request.

Furthermore, several protection techniques have been proposed that utilize proxies to mitigate related web application vulnerabilities, either to counter XSS attacks [95, 85], for CSRF protection [88, 90], or for client-side detection of SSL-stripping attacks [128].

4.5 Summary

In this section, we thoroughly examined session fixation: We described various session fixation attack vectors in detail and conducted two sets of tests (Section 4.2.3). First, we examined open-source applications for evidence that their developers were aware of the vulnerability class, i.e., we tested if session identifiers were changed after an authentication process. Only 4 out of 12 applications take this measure. Consequently, the remaining 8 applications would be vulnerable to session fixation if a supporting problem,

such as a header injection flaw, exists (one application was vulnerable out of the box due to URL support; the issue has been fixed in the meantime). Secondly, motivated by the outcome of the first set of tests, we examined various web application programming frameworks with respect to protection against header injection flaws. These tests resulted in the observation that the majority of the regarded frameworks indeed take measures to protect against header injection vulnerabilities, thus, indirectly protecting otherwise vulnerable applications against session fixation attacks.

Based on our observation that most web applications are vulnerable if an attacker is able to fixate the session ID with the victim, we proposed three distinct server-side measures against session fixation: For one, we showed how to avoid the problem in the applications' development phase (Section 4.3.1). Furthermore, we presented two approaches to fix running web applications with reasonable interference (Sections 4.3.2 and 4.3.3). These countermeasures require minimal configuration effort, which solely consists in providing the parameter names of the session identifier and password fields, thus, allowing fast and easy mitigating freshly detected session fixation issues. Our countermeasures are robust in respect to failed login attempts as the actual link between the server-side session storage and the application's user is preserved in all cases.

In sum, we provided defensive solutions for all potential scenarios in respect to control over an application's source code which can be encountered when operating a web application and, thus, achieve complete protection coverage against session fixation attacks.

4.6 Conclusion

Section 3 explained the conceptual differences between connection-oriented and connectionless protocols. We explained in Section 2.3 that web applications use HTTP cookies to implement sessions as a connection replacement. Session fixation attacks occur if the attacker can determine the victim's session ID and thus access the session using his knowledge (see Section 2.5.3). Transferred to connection-oriented protocols, the attacker would have to initiate an anonymous connection, then make his victim use this connection and authenticate, and finally take over the authenticated connection to act in the victim's name.

We showed that session fixation attacks can be thwarted if the web application issues a fresh session ID when the authentication status of the user changes. Our approach implements this kind of protection automatically on the server side. This allows an attacker to access the victim's session only as long as it is not authenticated yet. Similarly to establishing a connection, the web application assigns a new session ID upon login that is only known to the user's browser and the web application and thus excludes third parties that know the ID of the unauthenticated session.

In the next section, we will consider the problem that, by default, all authentication credentials are sent over the wire and not bound to the user or her browser. This fact makes stolen passwords and cookies reusable for attackers and thus facilitates session fixation and session hijacking attacks.

5 Augmenting Authentication Credentials Against Account Hijacking

We showed in the last section that web applications must issue a fresh session ID when the authentication status raises in order to protect the access to the established channel. However, even a new session ID is only a piece of information exchanged between the communication peers and – when stolen – sufficient to access a user's account. The same is true for the user's password for the initial authentication.

5.1 Introduction

Due to the missing connection establishment, there is a favor for one-step authentication processes in web applications: Browsers send all shared secrets in order to authenticate, first the password and then the session cookie. There is no binding of secrets to another authentication factor nor to the user's browser. Consequently, an attacker who gains access to any message carrying these secrets can fully impersonate the victim because he owns all secrets.

For the user login, the browser only sends information entered by the user. This step can be replicated by any user from any machine only by knowing the entered information. Then, the browser maintains an assigned session cookie to authenticate all subsequent steps. In this phase, the single account access token is sent back and forth plus stored by two communication parties. Again, an attacker gaining knowledge can impersonate the user.

In this section, we will show our approaches to augment the existing authentication credentials. We bind the user's knowledge – the password – to her browser to overcome phishing attacks. Then, we add user knowledge to the browser-based session tracking to authorize security-critical actions and mitigate CSRF, session hijacking, session fixation, and clickjacking attacks. Hence, user knowledge and browser-stored credentials complement each other, and an attacker needs to steal both in order to hijack a user's account.

5.2 Augmenting the Password with Transparent Browser Authentication

The term "phishing" describes a class of social engineering attacks on authentication systems, that aim to steal the victim's authentication credential, e.g., the username and password. The severity of phishing is recognized since the mid-1990's and a considerable amount of attention has been devoted to the topic. However, currently deployed or proposed countermeasures are either incomplete, cumbersome for the user, or incompatible with standard browser technology. In this section, we show how modern JavaScript API's can be utilized to build PhishSafe, a robust authentication scheme, that is immune against phishing attacks, easily deployable using the current browser generation, and requires little change in the end-user's interaction with the application. We evaluate the implementation and find that it is applicable to web applications with low efforts and causes no tangible overhead.

5.2.1 Motivation

From a security point of view, passwords are a terrible choice for authentication. They are easily stolen. Often, they are easy to guess, due to the fact that they were chosen in a fashion that allows the user to remember them (e.g., names of pets, children, or cars). And they are frequently reused, causing the compromise of one server to probably affect several independent applications as well.

However, it is an unrealistic assumption, that we will reach a situation, in which password-based authentication loses its significance, even in the presence of well designed password-less techniques, such as client-side SSL authentication, and promising new developments, such as Mozilla Persona [120].

Unlike all alternatives, the user's requirements to utilize password authentication are extremely light-weight: All she needs to logon, is to remember her username and password. Password-less authentication systems either require preconfigured state on the device, such as installed client-side certificates, the presence of specific hardware, such as smart card readers, or the possession of additional items, e.g., a cell phone to obtain out-of-band credentials [65].

This characteristic of password authentication is even amplified in the presence of web applications: The only remaining software requirement is, that on the utilized computer a web browser is installed, something that can be taken for granted since several years. Hence, no matter in which situation a user is, as long as she remembers her password and has a networked device with a web browser at her disposal, she is able to access her applications. No other system for networked applications offers similar properties. It can even be argued that the ease of password authentication was one of the success factors of the web.

However, the passwords' strength – their ease of use – is also their biggest weakness: As easily they are entered, as easily they are stolen, in case that a used password field is actually under the control of the attacker.

In variants, this class of attack, known under the term *phishing*, is probably as old as the discipline of password authentication itself, having its roots in social engineering attacks [119]. The severity of phishing is recognized since the mid-1990's and a considerable amount of attention has been devoted to the topic. However, as we will show in Section 5.2.2, currently deployed or proposed countermeasures are either incomplete, cumbersome for the user, or incompatible with standard browser technology.

In this section, we present *PhishSafe*, a light-weight approach that provides robust security guarantees, even in case that the user's password was successfully stolen. The core of our approach is a transparent browser-personalization process, that is invisible to the user. This way, unlike the majority of existing anti-phishing approaches, *PhishSafe* does not burden the user with altered authentication interaction or additional burdens, such as recognizing security indicators or visual authenticity clues. On the contrary: As long as a user predominately uses only a single browser, she won't notice a difference to the currently established, insecure scheme.

5.2.2 The Context of Phishing Attacks

While phishing attacks have a long history, phishing activity has not decreased over time (see Figure 5.2). The attackers' strategy, however, has changed to counter the anti-phishing means in use, for instance, phishing sites move faster to prevent blacklisting (see Figure 5.1). In this section, we model the attackers' capabilities, evaluate proposed anti-phishing solutions, and analyze why those solutions have not significantly reduced phishing activities.

Attacker Models

In order to estimate a phishing attacker's capabilities, we define two attackers. These attackers define the scope of our work, i.e., we present existing approaches against these kinds of attackers in Section 5.2.2 and propose PhishSafe, our countermeasure, in Section 5.2.3.

We consider a *phishing attacker* as a remote web participant. He is able to set up websites and email accounts, can send emails and messages via instant messengers (IMs). He can obtain valid SSL certificates for his domains. We do not assume timing constraints, i.e., he can react immediately on any input at all time.

Moreover, we consider an *XSS attacker*. He has all capabilities of the phishing attacker but can also inject JavaScript code into vulnerable web pages.

Neither of both has control over the user's platform nor over the network. We neglect browser vulnerabilities and respective exploits. Also, they can not break cryptography.

Current Solutions

Several approaches have been applied so far to mitigate phishing attacks. In this section, we name them and explain their strengths and weaknesses. We find that they are either incomplete, cumbersome for the user, or incompatible with standard browser technology.

Incomplete Countermeasures One class of countermeasures suffers from incompleteness in terms of false positives and false negatives, i.e., they do not protect against phishing on some sites and prevent access to genuine sites suspected to phishing.

Browser vendors, e.g. Microsoft[8] and Google[9], as well as third parties, e.g. PhishTank[10], provide lists of malicious and genuine websites. Browsers query their list upon accessing a website and check whether this site is known for phishing. The blacklists suffer from a window of vulnerability between the setup of a phishing site and its listing [160]. This window can be decreased by real-time queries towards the list providers for each unknown domain. The additional online query slows down page loading and reveals almost the complete browsing history to the list providers. List providers went over to classify websites automatically to capture phishing sites earlier [193], however, at the expense of accuracy, i.e., more false positives and false negatives [107]. The extraction of features from phishing sites provoked an arms race between phishers, who have a financial interest in passing those filters, and the blacklist providers. Among other features, phishers reduce the uptimes of their sites (see Figure 5.1) to make the blacklists come to nothing. The trend lasts and led to an average uptime of one day in 2012 leaving only very short reaction time to blacklist providers [172].

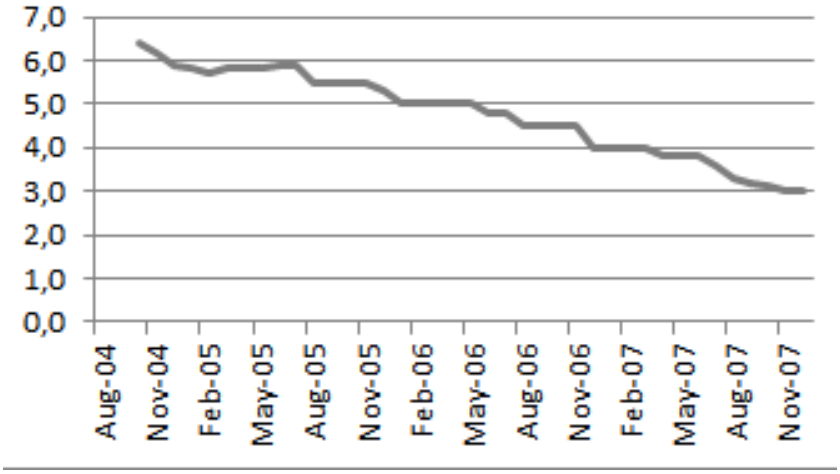

Figure 5.1: The average online time of phishing sites in days between Oct '04 and Dec '07, the time of acquisition by the APWG, *src: Regular APWG Phishing Attack Trends Reports [174].*

Countermeasures Cumbersome for the User Another class of approaches makes use of the increasing propagation of mobile devices. Users need to enter a second credential that is either received or generated by their mobile device in order to login or perform

[8] `http://windows.microsoft.com/en-US/windows-vista/Phishing-Filter-frequently-asked-questions`
[9] `https://support.google.com/chrome/answer/99020?hl=en`
[10] `http://www.phishtank.com/`

critical actions. This breaks their ongoing workflow as they need to switch to a different device. Example implementations include Google Authenticator [65] and one-time passwords sent to cell phones. Beside the fact that malware now also targets mobile devices to intercept received tokens [48], both approaches can not help against our attacker models (see above) because the attacker only needs to wait for the victim to enter her credentials and relay all gathered user data to the actual web application in real time. This way, the user serves as an oracle that provides the needed information. In this scenario, the attacker plays the role of a man in the middle without manipulation on the network layer.

Client-side SSL aims at replacing username/password-based logins. Though SSL could overcome most of currently known weaknesses in knowledge-based authentication, it has not become popular probably due to its setup complexity for non-expert users. Finally, SSL certificates are hardly portable. A user can login to web accounts from every device using an off-the-shelf browser and her password. It is rather difficult to store, carry, and use an SSL certificate securely on an untrusted computer.

Countermeasures Incompatible with Standard Browser Technology A family of approaches extends the user's browser [94, 146, 199, 45, 198, 195, 142, 76, 180, 33, 145, 112, 31, 200, 157] (see Section 5.4 for details). Browser extensions and toolbars share a number of drawbacks:

- They provide no protection by default but only protect risk-aware users after installation.
- They are inherently incompatible with standard browser technology and can only protect users of supported browsers while porting them to other browsers is hard. [141]
- The majority of browser-based solutions aims at detecting phishing websites while accessed. However, most users ignore issued warnings and more rely on the web content to estimate a website's authenticity. [194]
- Browser toolbars, that classify websites into phishing and harmless, are susceptible to false positives and false negatives, i.e. letting phishing sites pass while warning of genuine sites. Case studies showed that a high detection rate often comes with a high false positive rate. [202]
- Phishing is a particular problem on mobile devices while existing approaches are hard to port because of the limited screen size. [58]
- Phishers can evade most browser-based protection approaches by asking victims to reply by email to their inquiry.

Summary We can conclude that the existing approaches still leave room for phishing attacks. None of the current solutions offers thorough protection for all users. The volatile number of active phishing sites reflect the ongoing arms race between phishers and anti-phishing blacklist providers (see Figure 5.2). The more stable number of

phishing campaigns shows the unabated activity of phishers over a long period. Matters are complicated by more targeted spear phishing attacks which are harder to detect by generic features than common large-scale attacks.

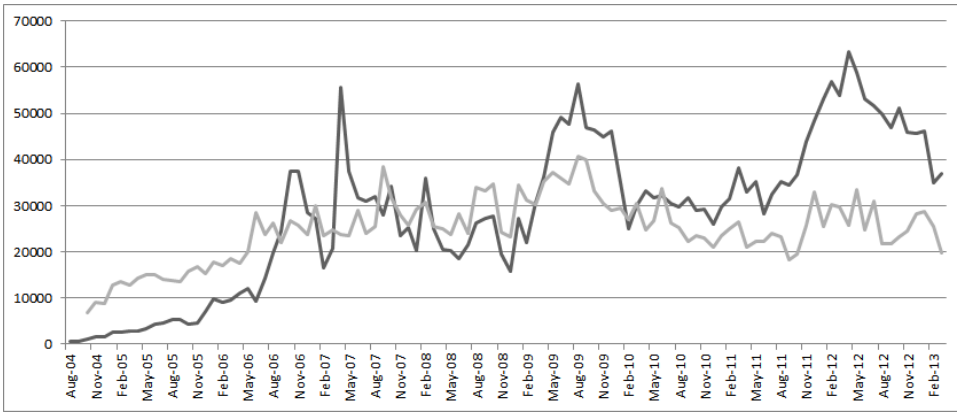

Figure 5.2: Phishing statistics since Aug. 2004 in terms of active phishing sites (dark grey) and email phishing campaigns (pale grey), *src: Anti-Phishing Working Group (APWG) Reports [174].*

Emerging consumer-oriented SSO protocols like Mozilla Persona [120], OpenID [132], and OAuth [73] decrease the user's attack surface. Nevertheless, they still require user logins with the identity provider and, thus, cannot remedy phishing attacks. SAML [106] and Shibboleth [83] target business environments and require a higher level of coordination between participants, thus, are more suitable for closed application scenarios. We provide more details in Section 5.4.1.

The Weakest Link: The User

After analyzing existing countermeasures and modern attack vectors (see Section 2.5.1), we identify the user as the weakest link. We find that phishing attacks abuse the user's misconception concerning her communication partner in the world wide web. Transferred to the physical world, a phishing attacker would set up a storefront that looks familiar to many people. In the virtual world of the world wide web, the attacker can succeed much easier for several reasons.

First of all, the user has no personal reference point in terms of location. Informally speaking, she does not know where she actually is. Most users are not familiar with domains and URLs, and even if they were, they could still be misled by exploiting weaknesses in the Domain Name System (DNS spoofing, pharming). The international domain name (IDN) homograph attack [64] even deceived skilled security experts.

Second, users learned to assess a person's trustworthiness. While this assessment can be manipulated, there is hardly any natural feeling of trustworthiness with respect to programs and machines nor do reliable indicators help. Existing approaches focus on proving an email's (e.g. DKIM [37], SenderID [117]) or a website's (e.g. https) trustworthiness but not the opposite, i.e. in an attack scenario, they do not provide

any helpful hint. Teaching users to check SSL indicators inspired phishing attackers to spoof those indicators or obtain valid certificates for similar domains, e.g. `gooogle.com`. Such indicators are missing on most mobile devices due to the limited screen size [130]. The opposite approach – warning users instead of indicating trustworthiness – made users being annoyed and ignore such warnings [51], because users want to make things happen and not think about security, so they do whatever is asked for in even unusual emails [47]. Attackers increase their chances by threatening their victims, for example, announcing bad consequences like blocking an email account or disabling the credit card. This strategy prevents that users contemplate on the message's reliability.

Third, automation allows large-scale attacks making the efforts worthwhile. The intention to classify phishing attempts led to an arms race meaning that attacks evolve and require new features to detect phishing [59].

To sum up, we conclude that the user must not play a decisive role in phishing protection nor can the user behavior be supposed to change. An algorithmic approach is needed to rule out phishing attacks.

5.2.3 PhishSafe

In this section, we describe the idea of our authentication scheme, named PhishSafe, that avoids the drawbacks identified in Section 5.2.2. Section 5.2.4 gives details of the implementation.

Design Goals

Following the lessons learned from previous approaches and current phishing techniques (see Section 5.2.2), we phrase the following design goals for PhishSafe: It

- sidesteps the arms race between phishers and the anti-phishing community,
- reduces reliance on the user,
- avoids dependence on the browser's interface,
- waives the need for additional devices and the installation of protective tools, and
- withstands the attackers defined in Section 5.2.2.

Our design goals are in parts inspired by Parno et al. [139] (see Section 5.4). In the remainder of this section, we motivate our design goals in more detail.

Sidestep the Arms Race Between Phishers and the Anti-Phishing Community It is important to quit the arms race with financially motivated phishers that are always one step ahead. The anti-phishing community can only react on new phishing techniques while phishers update their features again.

Reduce Reliance on the User We showed in Section 5.2.2 that the user is the weakest link in phishing scenarios. Hence, a reliable countermeasure must not rely on the user. Instead, it must tolerate that the user can be tricked and gives away all credentials she knows.

Avoid Dependence on the Browser's Interface Approaches relying on the browser's interface either require the installation of additional software (e.g., toolbars or extensions, thus, excluding users of not supported browsers or platforms) or can be spoofed using JavaScript or a favicon (e.g., simulating an SSL lock symbol). The interface is even hidden on mobile devices due to the limited screen size.

Waive the Need for Additional Devices and the Installation of Protective Tools The need for second devices makes processes more complex and requires considerable changes of the used logon procedure. Those devices must be always at hand, secure, and have a direct connection to the browser to transfer control. Obtaining passcodes from a second device is not an option because these can be phished and exploited.

The usage of protective software is always limited to risk-aware users utilizing a supported platform.

Withstand the Attackers Defined in Section 5.2.2 We modeled the attackers according to realistic assumptions. So, a reliable approach must provide protection against their attacks.

High-level Overview

The main idea of PhishSafe is to release the user from responsibility: she neither needs to perform special actions nor check security indicators nor keep a secret other than her password. Instead, she will use a second factor she does not know and, thus, can not disclose to a phisher. This factor is stored in her browser and attached to logins towards the genuine web application. The web application prohibits logins without proper second factor authentication. An attacker luring his victim on a phishing site can obtain her password but not the second factor credential. However, the password alone is not enough to login. The second factor is established during account setup and, if necessary, restored after visiting a URL sent by email.

Detailed Authentication Process

As emphasized above, the authentication scheme implements two-factor authentication without the user knowing about it. In order to apply our authentication scheme, the website stores a secret token in the persistent web storage of the user's browser (see below for details). The user does not have to be aware of this token nor does she have to care. The important point is that this token is subject to the same-origin policy (SOP) [150] and not accessible to web applications on foreign domains.

When the user accesses the login page, a challenge string is invisibly included in the HTML form beside the username and password input fields. The page also embeds JavaScript code that computes the second factor credential from the challenge and the secret browser token using an HMAC function [101]. The second factor is then appended to the HTML form and transmitted to the web application together with the username and password. The web application verifies the second factor by performing the same computation that happened in the browser. It denies access to the user account if the verification fails.

A phisher could lure the user into visiting his prepared page. Given that the user does not detect the attack, she enters her username and password and sends them to the attacker's site. Then, the attacker tries to log into the user's account exploiting the phished credentials. The web application, however, denies access because the necessary browser token is not available to compute the valid second factor.

There are scenarios where a browser is not only used by one user but at least two where both have an account on the same web application respectively, e.g., a family sharing one laptop (and OS account) or tablet PC. In this case, they would share the same browser token. This is also true for guests accessing the web application via this browser just once. We prevent such unintended sharing of the browser token by assigning it to the respective user account in the browser's storage, i.e., the second factor can only be computed if a browser token associated with the given username is found.

Browser Enrollment

The idea how PhishSafe proceeds has been described above. What remains is PhishSafe's bootstrapping, i.e., the process that establishes the token in the user's browser. There are two options when the token is stored: during account setup or, afterwards, whenever the user logs in from a previously unknown browser.

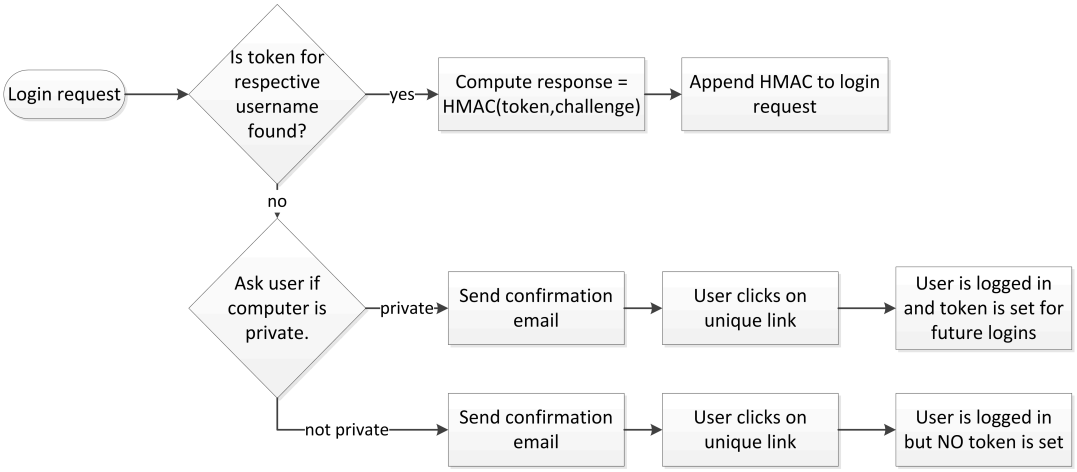

Figure 5.3: Setting and using the browser-stored authentication token to overcome phishing attacks.

The web application can set a token during the registration process unless the user opts out, e.g., because she uses a friend's device. After the user chooses username and password, the token is stored in the browser's web storage.

Restoring the Browser Token We assume that account information includes the user's email address and leverage this as a second channel for token installation. Given that the user uses more than one device to access the web application, changes her browser, reinstalls her operating system or firmware, or just deletes the browser's web storage for privacy reasons, she needs an opportunity to restore her browser token. The password alone is insufficient because the attacker can learn it and so use it to equip his browser with a valid token.

Usual second authentication tokens are not sufficient, either. Examples for this class are apps or devices that issue two-step verification numbers, e.g. Google Authenticator [65] and RSA SecurID [53], as well as one-time passcodes sent to the cell phone or by email. A phishing attacker could lure the victim on his page and at the same time request the original login page. When the victim provides her password, he forwards it and is prompted with an input field for the second authentication step. Then, he leads his victim to believe that her browser token needs to be reset and requests the same authentication credential that he is supposed to enter. Finally, he only needs to forward the user's second factor credential to finally own the password and the browser token. Note that this attack even works with passcodes sent to the user's cell phone because the application indeed sends such a code to the user (upon the attacker's request). We believe that receiving the code makes the actual phishing attack even more credible. The attacker acts as a man in the middle.

Our authentication scheme uses complete URLs that must be clicked (or copied and pasted to the browser) by the user. When the user enters her username and password on the login page but no respective browser token is found, the web application sends a confirmation email to her account. The email contains a unique URL that must be accessed within the same session context as the login request. In the attack scenario described above, the attacker's login attempt triggers the email confirmation. However, if the user clicks on the provided link, she accesses the real web application but not the phishing page. At that point, the attack becomes detectable for the web application because the session context does not match. The attacker never obtains the necessary input to obtain a valid browser token.

One-time Account Access Finally, the user might use a public computer to access the web application. So, a persistent credential in the browser's web storage is not appropriate. For this reason, PhishSafe also provides one-time access. The only difference to the above described browser token reset is that no token is stored. After the user enters her username and password and no browser token for this username is found, she is asked if she is using a public computer or if she trusts all users of this computer and uses it regularly. In both cases, the web application sends an email with a unique URL. However, if the user requests the URL from a public computer, no browser token is set

and access to the account is granted only once. The token handling logic is given in Figure 5.3.

Protection Against the XSS Attacker

The authentication process described above perfectly protects against the phishing attacker (see Section 5.2.2). The XSS attacker, however, could inject JavaScript code that is executed within the same domain context as the web application. This allows him to read the browser's web storage and obtain the secret browser token. For this reason, we move the token and all related computations to a secure subdomain. Given that the actual web application runs on `www.example.com`, a subdomain, e.g., `auth.example.com`, is responsible to handle and store the secure browser token. This subdomain only contains static JavaScript dedicated to this task and nothing else. Based on this, we consider well audited and XSS-free code to be feasible. An HTML document served from the subdomain and embedded into the main web application as an invisible iframe contains the JavaScript code. This way, we leverage the guarantees provided by the same-origin policy [150] and the postMessage API [161] to prohibit access by the XSS attacker to the browser token while enabling controlled interaction between the web application and the secure subdomain.

Hence, the challenge appended to the login form is submitted to the secure subdomain via JavaScript and the postMessage API. The code of the subdomain computes the HMAC of the challenge using the browser token. The HMAC is then sent back to the original document and attached to the subsequent login request (see Figure 5.4). Please note that all this communication happens within the browser.

5.2.4 Implementation

The implementation of our proposed authentication scheme comprises three interacting components: the *TokenManager* handles the browser token in the secure subdomain, the *Authenticator* assembles necessary input for the login, and the server-side *AccountManager* fits into legacy or new web applications in order to handle browser challenges and responses.

Client-side Components

We first describe the client-side components, the TokenManager that is loaded in the iframe from the secure subdomain and the Authenticator that delivers the web application's challenge to the TokenManager and appends the retrieved response to the HTML login form.

The TokenManager The TokenManager implements the necessary functions to fetch the browser token, compute the HMAC of the token and the web application's challenge, and return an error message if no token is found. It runs in the domain context of a secure

subdomain. We use the CryptoJS[11] library as an implementation of the cryptographic functions.

The TokenManager uses the browser's `localStorage` part of the web storage [77]. Web storage is supported by all major browsers on mobile and desktop platforms which makes our authentication scheme platform and browser independent. The localStorage is persistent, i.e., it is not cleared on a regular basis as the `sessionStorage` is. The storage is limited in size per origin between 5 and 25 Mbytes depending on the browser which, however, is far more than necessary for our purposes. Web storage is meant for pairs of identifiers and values where both must be strings. More complex data structures can be stored as JSON objects [38] that are easily converted to string and back. The TokenManager uses JSON to store the username and the associated browser token.

The communication interface of the TokenManager is restricted to a function that expects a challenge and a username as input and provides an HMAC as the output (see Figure 5.4). The Authenticator's direct access to the subdomain's localStorage is prohibited by the same-origin policy [150]. So, the Authenticator needs to use the JavaScript `postMessage` API [161] that enables two web documents in a browser to communicate across origin boundaries in a secure manner. A `postMessage(msg, target)` call expects a message string and the target origin as parameters. The receiving document needs to register an event handler to receive a message. The triggered event comes with additional metadata provided by the browser, e.g., the origin of the sender. This allows the receiver, i.e., the TokenManager, to carefully check the sender's authenticity.

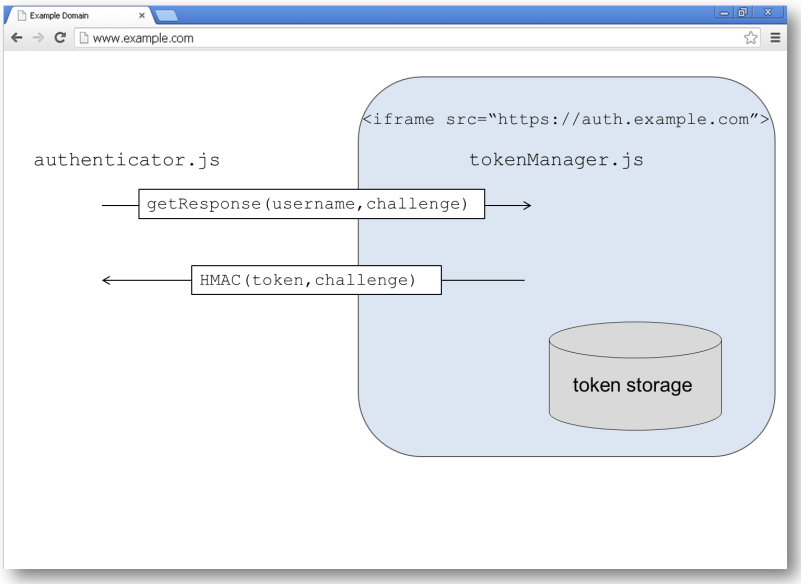

Figure 5.4: Leveraging the domain-isolated token storage to protect the authentication token against XSS attackers.

[11] `http://code.google.com/p/crypto-js/`

The Authenticator The HTML login form contains two additional hidden fields for PhishSafe: `AuthChallenge` and `AuthResponse`. The first contains the web application's challenge, the second is initially blank. The Authenticator reads the challenge and the user's username from the input field. It passes both arguments to the TokenManager and reads back the answer. The answer either contains the computed HMAC or an error. In the success case, the Authenticator rewrites the login form's `AuthResponse` field to append the response. It prompts the user if the TokenManager reported that no browser token was found (see Figure 5.5).

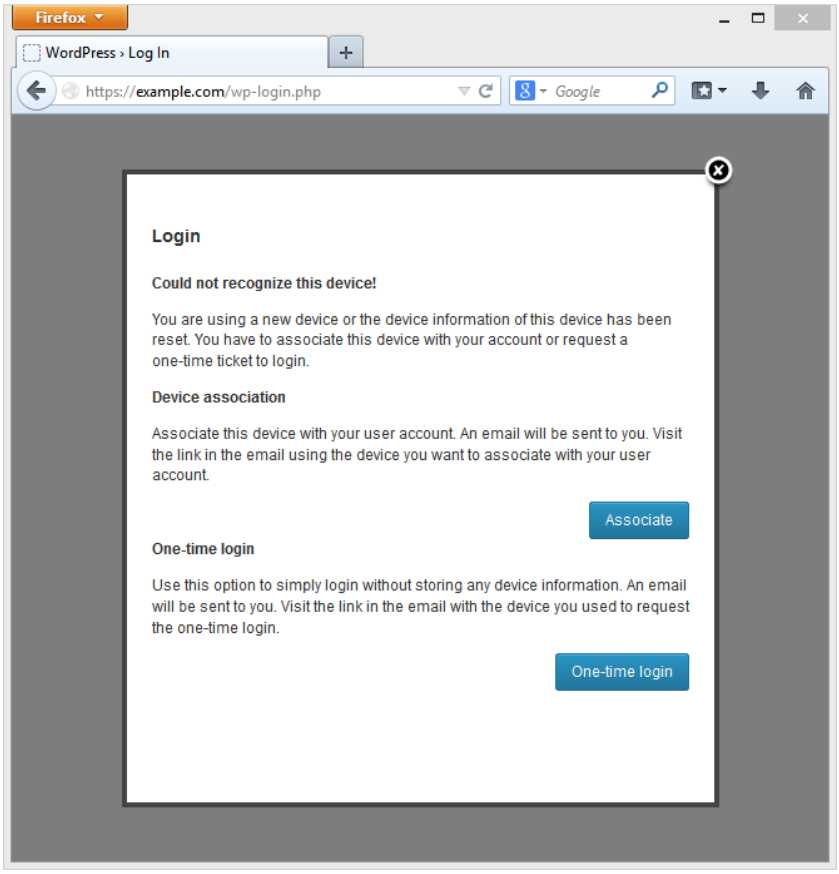

Figure 5.5: Prompting the user if no anti-phishing token is found.

Server-side Component

The server-side part of PhishSafe consists of a single component, the AccountManager.

The AccountManager The AccountManager implements the server-side part of our authentication scheme. We equipped WordPress with the AccountManager as a plugin. The integration required only reasonable efforts and no changes of the application code due to the modular architecture of WordPress together with the hooking feature. We

consider the integration into modular or new web applications as an easy task while necessary efforts might be bigger for non-modular legacy applications.

The AccountManager issues the user's browser token and adds the invisible iframe and the Authenticator to the web application's login page. It generates a new challenge for every user login, adds the `AuthChallenge` and `AuthResponse` fields to the HTML login form, and checks incoming login requests for valid responses.

The XSS attacker could inject a payload that reads the user's username, password, and the returned HMAC for authentication. Having all this information, he can log into the web application without a valid browser token. For this reason, the AccountManager sets a cookie in the user's browser. This cookie has a random value that is saved by the AccountManager together with the user's current challenge. Only login requests that carry this cookie and the valid response are processed. The cookie has the `HttpOnly` and the `Secure` flags set to prevent it from being read by the attacker's payload or during plain `http` transport.

Security Configuration Though it is not part of our attacker model, we leverage two more modern security features to shelter from SSL stripping [110] and pharming [184] attacks. A man-in-the-middle attacker performing an SSL stripping attack could prevent the user's browser from using `https` and then read transmitted information or even inject code into the document loaded in the invisible iframe. A pharming attacker could serve own code on behalf of the abused web application and so bypass the same-origin policy. Both attackers could hijack the secure browser token.

To overcome these attacks, the AccountManager adds HTTP Strict Transport Security (HSTS) [79] and Public Key Pinning (PKP) [55] policy headers to the web application's HTTP responses. HSTS makes sure that the browser only contacts the website using `https`. There is an inherent bootstrapping problem before the first request, i.e., the website must ensure that the browser eventually receives the HSTS header. Google Chromium and Mozilla Firefox overcome this problem using a static list of pre-defined domains[12]. The PKP policy prevents that a pharming attacker presents his own certificate. After receiving the policy header, the browser only accepts SSL certificates with the pinned public key.

5.2.5 Evaluation

We evaluate the security properties of our authentication scheme and validate our design goals from Section 5.2.3.

Security Evaluation

We first explain how far PhishSafe protects users against the attackers defined in Section 5.2.2. Then, we give details of further security properties, and finally, we identify open issues of the proposed scheme.

[12] `http://www.chromium.org/sts`

The Phishing Attacker The phishing attacker counterfeits the design of a web application in scope and lures victims. The latter can be done either by email or by links and ads on other websites. In any case, our proposed authentication scheme does not prevent a victim from accessing the phishing page. However, the information the phisher can obtain is not sufficient to abuse the user's account because the web application denies access if no valid response is appended to the login request.

The XSS Attacker The XSS attacker differs from the phishing attacker in his ability to execute JavaScript code on vulnerable domains. For instance, the XSS attacker could perform a reflected XSS attack by sending a specially crafted link via email or IM. The injected payload can read the username, password, the current challenge for login and the respective response if a browser token is stored in the browser. However, we assumed that the attacker can not break cryptography, thus, he can not compute the browser token from the challenge and the response. The captured data is still insufficient because it lacks the related cookie which is inaccessible to JavaScript. Finally, we consider the task to develop invulnerable static code for the secure subdomain feasible such that there is no attack vector for the XSS attacker.

Further Security Advantages The proposed authentication approach comes with additional security features.

First of all, though running purely in the browser, the scheme thwarts email-based phishing attempts. Phishers try to evade browser-based protection by asking the victims to reply to their phishing emails and give credentials in the email. The obtained username and password, however, do not give an attacker access to the user's web account. We consider phishing for the browser token to be infeasible as it requires major efforts and advanced knowledge of a user to read the token from the browser's web storage.

Phishing attacks rely on unprepared users that disclose credentials. This general observation holds true for any kind of credentials a user may know. So, second authentication factors that must be entered by the user in a web form are inherently susceptible to phishing attacks, too. Examples include one-time passcodes sent to the user's cellphone or generated by apps. Our approach utilizes complete URLs to overcome second factor phishing. A user clicking on a link not only proves access and knowledge but is also directed to the right web application.

Open Issues Next, we emphasize on potential attacks on our authentication scheme.

Our approach relies on the security of the user's email account. The attacker can request and read the confirmation URL sent to the user if he has access to the email account and the user's credentials to the target web application. We are here in line with today's best practices for password reset as virtually all web applications offer email-based processes at least if no cell phone number is given or the attacker pretends the cell phone is stolen.

An email account can not be protected if the confirmation URL is sent to the same address. There are two options to make sure that the user can always access the confir-

mation URL: First, the user can provide an alternative email account where the confirmation URL is sent to. Second, the email provider can offer application-specific passwords[13]. These passwords are chosen by the provider and not remembered by the user. Instead, they are stored by client applications to obtain access to the user's account. Given an application-specific password and an email client on a PC or mobile device, access to the confirmation URL is assured.

An attacker can try to acquire the confirmation URL sent to the user by making the user enter it into a prepared input field on the phishing site. The easiest way to avoid this is to make the user click on the link. Moreover, a highlighted warning in the email reduces the attacker's chances.

A window of vulnerability towards a pharming attacker remains before the first PKP header is received by the user's browser (see Section 5.2.4). As long as the browser did not pin the server's public key, a pharming attacker can present a spoofed certificate and submit malicious content. In the future, a similar pre-defined list of certificates might be implemented as it happened for HSTS.

Finally, though completely out of scope of our authentication scheme, fully fledged spyware can read and transmit the browser token. Less elaborate keyloggers, however, are ineffective because the browser token is never entered via the keyboard.

Validation of Design Goals

In this section, we evaluate the compliance of PhishSafe with the design goals given in Section 5.2.3.

Sidestep the Arms Race Between Phishers and the Anti-Phishing Blacklist Community Our approach does not exploit features of phishing sites or emails for classification. So, there is no motivation for actions and reactions. In fact, we do not consider phishing activities at all but only hide some piece of information from phishers. We argued in the section above that phishers can hardly learn the browser token.

Reduce Reliance on the User The security of the approach barely relies on the user. She neither needs to enter her credentials only when a dedicated indicator is shown, nor does she need to remember additional credentials. In fact, the only change compared to her used workflow is the decision if a computer is trusted or not and the click on the confirmation link. The actual login procedure does not change at all on regularly used browsers.

Avoid Dependence on the Browser's Interface PhishSafe does not depend on the browser's interface, nor does it change the browser's appearance. This feature not only avoids confusing the user but is one important aspect of cross-platform applicability (see below).

[13] `https://support.google.com/accounts/answer/185833`

Waive the Need for Additional Devices and the Installation of Protective Tools PhishSafe only needs an off-the-shelf browser and neither relies on extensions, nor toolbars, nor third-party plug-ins, like Flash or Silverlight. A second device is also not necessary.

Withstand the Attackers Defined in Section 5.2.2 We showed above that PhishSafe resists attacks by the phishing attacker and the XSS attacker.

Further Points PhishSafe runs on mobile as well as desktop browsers because all modern browsers support the WebStorage and postMessage APIs. It does not rely on visual indicators which makes it applicable on mobile devices with limited screen size.

PhishSafe can easily complement other approaches. If the browser maintains a blacklist of phishing sites, it can prevent that the user reveals her credentials. Approaches leveraging a secure password entry field to some degree also work together with PhishSafe even though details need to be sorted out.

5.2.6 Summary

In the course of this section, we analyzed the root causes for the continuing prevalence of phishing attempts and classified existing solutions into three main categories: incomplete countermeasures prone to false positives and false negatives, countermeasures cumbersome for the user compared to common logon processes, and countermeasures relying on browser extensions or toolbars, thus, expecting risk-awareness by the user and excluding users of not supported browsers and platforms.

Then, we identified the user as the weakest link when it comes to phishing protection. She can neither be expected to apply cumbersome countermeasures, nor install protective tools, nor take care of security indicators. We presented PhishSafe, a reliable approach to overcome phishing attacks, that runs in browsers out of the box and barely changes known logon processes. The user must only decide if she uses her private browser or not. This way, PhishSafe implements a two-factor authentication scheme where the second factor is only accessible to the genuine web application but not to the phisher nor to the user. Without knowing the second factor, the user cannot disclose the necessary information for account access to an attacker.

PhishSafe can be easily deployed by web application providers and is not susceptible to the chicken-and-egg problem. Moreover, it complements SSO protocols and anti-phishing blacklists.

5.3 Augmenting the Session Cookie with User Knowledge to Mitigate Web Session-based Vulnerabilities

After the initial login, web browsers authenticate to web applications by sending the session credentials with every request. Several attacks exist which exploit conceptual deficiencies of this scheme, e.g. CSRF, session hijacking, session fixation, and clickjacking. We analyze these attacks and identify their common root causes in the browser authentication scheme and the missing user context. These root causes allow the attacker to mislead the browser and misuse the user's session context. Based on this result, we present a user authentication scheme that prohibits the exploitation of the analyzed vulnerabilities. Our mechanism works by binding image data to individual sessions and requiring the submission of this data along with security-critical HTTP requests. This way, an attacker's exploitation chances are limited to a theoretically arbitrary low probability to guess the correct session image.

5.3.1 Motivation

Web applications must identify and authenticate their users in order to provide personalized services in the world wide web. Upon signing up, users generally choose a username and a password that can be used as a shared secret to establish future sessions. After the authentication of the user, the web application assigns a unique temporary token to the user. This token is stored in the browser and subsequently used by the browser and the application to tell this user and others apart. Several attacks target the browser or the token to hijack established sessions. Clickjacking and CSRF mislead the victim's browser to send requests that are determined by the attacker. Session hijacking and session fixation aim at sharing the token with the attacker.

In this section, we introduce a method to authenticate security-sensitive operations. Our approach, named *Session Imagination*, can be applied to existing web applications and mitigates the above mentioned attacks. Specifically, we apply the two steps of identification and authentication to established sessions. After login, the user is equipped with a shared secret that is not stored in her browser. The former universal token then serves as the identification that is complemented by the shared secret as the authentication for security critical operations. The shared secret can not be stolen by an attacker, and the browser can not be lured into misusing the secret.

Contribution

Our contribution is twofold: We identify the above mentioned attacks' common root causes and provide an applicable solution that implements the well-known and approved concept of identification and authentication to web sessions. This solution remedies basic deficiencies of current web session implementations. We give details about the

authentication scheme and its implementation, evaluate the approach, and show that the protection goals are achieved.

5.3.2 The Root Causes of Web Session-based Attacks

The attacks on web sessions described in Section 2.5 share common root causes.

First, session authentication means authentication performed by the browser. For the user's perception, only one authentication step happens, namely the login where she provides her username and password. The rest of the session handling is transparent to the user. As explained in Section 2.3, HTTP does not have a session feature and, thus, session handling has to be implemented using session identifiers on the application layer. This fallback solution provides authentication of the browser with every request instead of authentication of the user as it would be required. The following example illustrates this fact: One person logs into her account on a web page, then leaves her computer to have a coffee. Every other person could now interact with the web application on behalf of the user logged in because the browser will do transparent authentication. So, as long as the browser maintains the session ID, all requests are authenticated. The same person accesses a terminal next to the coffee maker. She visits the same web application but she will not be able to access her account without another login though she already authenticated towards this web application.

Second, on the opposite side, the server can not distinguish different contexts of a request. On the server side, incoming requests generated by a JavaScript command, an image tag, or the click of a user respectively are all alike. The requests do not contain evidence that they are intended by the user. The server can not decide whether the user is aware of the action that is caused by a request.

To sum up, the common root causes of session hijacking, CSRF, clickjacking, and session fixation are in fact *browser authentication* instead of *user authentication* along with the server's unability to determine a request's initiation context.

Browser-level and User-level Authentication

The authentication of HTTP requests can be divided into two classes: browser-level and user-level authentication.

Browser-level authentication is the current practice in web applications, meaning that after the user provided her credentials for login, the authentication token is cached and subsequent requests are implicitly applied by automatically sending the authentication token. In this case, the browser performs authentication on behalf of the user because the user logged in to the personalized service. Examples of implicit, e.g. browser-level, authentication are the above mentioned cookies, client-side SSL authentication, HTTP authentication (basic and digest), and authentication based on the client's IP address.

The other principle is user-level authentication. In this case, another authentication step for a user's requests is added. We require the user's explicit consent to a user-level authentication step such that this step can not be taken by the browser only but

additional action by and knowledge of the user is required. Examples of explicit, user-level authentication are re-entering the username and password and passcodes received as text messages.

We identified two attack vectors emerging from browser-level authentication. First, CSRF and clickjacking attacks make the browser send a request and authenticate on behalf of the user even though the authenticated user does not acknowledge. This problem is known as the 'confused deputy problem' [74]. The browser stores all secret information that is needed to authenticate the requests. The underlying assumption becomes evident in the attack scenarios: All requests are supposed to be only initiated by deliberate user clicks or by the browser that fetches regular content. This assumption stems from the early days of the world wide web when web applications were not personalized. The addition of web sessions and cookies turned this established assumption to a security risk. The web application can not decide whether the user deliberately initiated the requests. Uncommon request sequences may indicate CSRF attacks, clickjacking attacks simulate regular user sessions and are harder to detect.

Second, while CSRF and clickjacking are based on requests initiated by the victim's browser and without her consent, there is another attack vector that exploits the fact that browser-stored information can be easily transferred. Session hijacking and session fixation attacks strive to impersonate the user from different machines towards the web application. Both attacks share the same goal, namely the attacker and the victim share the same SID and are thus indistinguishable from the web application's point of view.

Both attack vectors are based on the same conceptual deficiency: Due to browser-level authentication, no user input is needed to supply evidence that the authenticated user intends the requested action. On the opposite, request authentication including user interaction prevents the attack vectors and remedies the conceptual deficiency.

5.3.3 Session Imagination

We implemented a new approach for user-level authentication, named *Session Imagination*, to address the root causes described in Section 5.3.2. Thereby, we focused on overcoming the vulnerabilities' root causes (see Section 5.3.2). In this section, we will describe our solution that aims at mitigating CSRF, clickjacking, session hijacking, and session fixation attacks. Session Imagination separates identification and authentication in web sessions and relies on visual authentication tokens which can be easily remembered and recognized by the user while the authentication token is not stored in the browser.

Attacker Model and Protection Goals

We model the attacker to be a regular web participant. He can send messages, access web applications and set up his own websites. However, he does neither control the other user's machine or platform nor those of the web application nor the communication infrastructure between them.

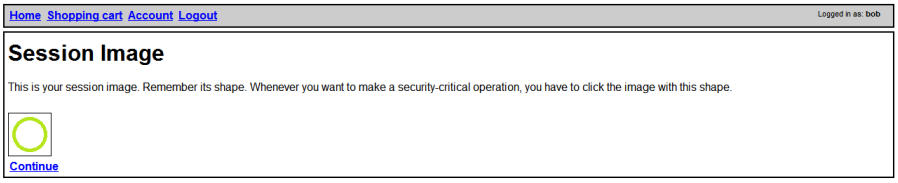

Figure 5.6: A fresh session image is given immediately after login. This image has to be remembered throughout the session and identified among a set of images to legitimate security-critical requests.

Our goal is to protect a web application and its users against CSRF (see Section 2.5.5), session hijacking (see Section 2.5.2), session fixation (see Section 2.5.3), and clickjacking (see Section 2.5.4). Protection means that an attacker's chances to reach his goals are limited to an upper bound of probability. The actual upper bound may be configurable. The attacker must not be able to increase this probability. For the sake of completeness, we must say that we aim at securing authentication tracking and do not consider an attacker who owns the login credentials. For example, a phishing attacker gaining knowledge of username and password can still use a protected web application in the victim's name. We presented an effective approach to defend against phishing attacks in Section 5.2.

The User-level Authentication Scheme

Session Imagination uses images as per-session user-level authentication tokens. That means that every user is assigned an image upon login. This image is displayed once immediately after login (see Figure 5.6).

It is then used together with a conventional session ID in a cookie to authenticate security-critical requests. For example, in an online shop, a set of critical actions is defined, e.g. sending an order or changing the shipping address. Upon requesting such an action, the user has to choose the right image among a given set before the action is executed (see Figure 5.7). In our example implementation, we used circles, triangles, hexagons, arrows, squares, and ellipses as images. One could also use more usual images like animals, shoes, or hats. We call this intermediate step the 'challenge'. A brief overview of the Session Imagination steps is given in Figure 5.8.

For every new challenge, the images' shape is slightly varied. That does not affect the user's ability to distinguish the right image from the others but makes simple image recognition, e.g. by automatic hashing, harder. As an example, consider the images in Figure 5.9 which represent the same six "classes" as those given in Figure 5.7. Differences between two images of the same class can occur in terms of orientation (where appropriate), line color, and fill color. If pictures of animals or items serve as session secret, similar classes can be used. Users are expected to be able to distinguish cats from dogs etc.

The next point of image recognizability is the *Uniform Resource Identifier (URI)*. The

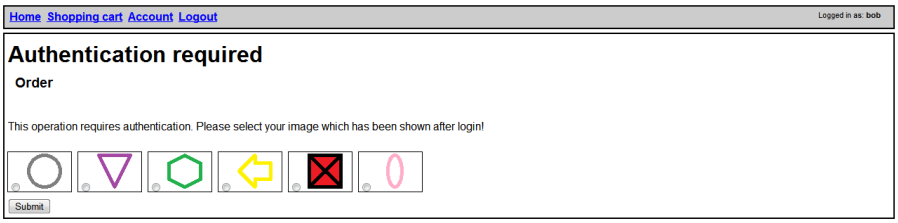

Figure 5.7: Before a critical operation is executed, the respective request has to be authenticated. Therefore, the user has to identify the correct session image.

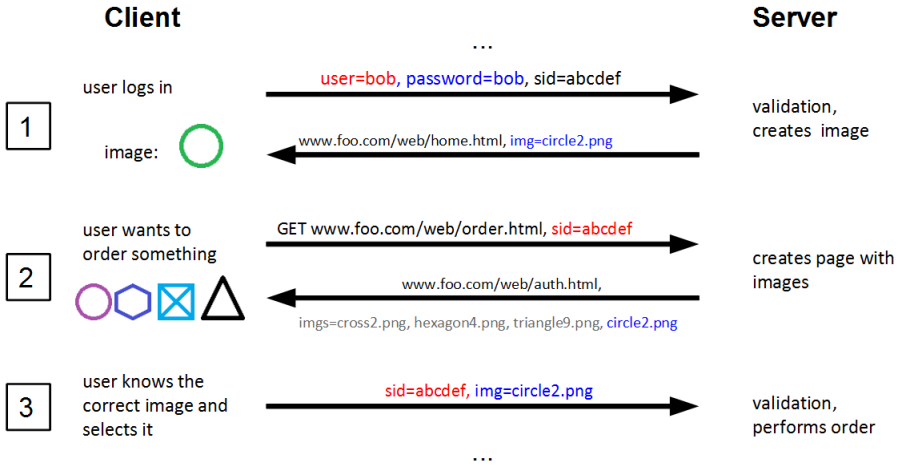

Figure 5.8: An overview of authentication steps related to Session Imagination. We used descriptive file names for the pictures for the sake of clarity.

provided images could be identified by their file names, e.g. `circle1.png`. Given that, an attacker can conclude the image shape from the name which can be stolen by an XSS attack in conjunction with the session cookie. So, we implemented random names for all provided images. The names are regenerated with every response. They serve as one-time passwords that the user does not have to remember because she can identify the correct password by the corresponding image which is valid for the whole session.

In the run of an XSS attack, the attacker could record the user's click and use a canvas element [185] to prepare an exact copy of the session image. The attacker can choose size 0 x 0 to avoid that the attack is detected by the victim. Next, the canvas is serialized and transmitted to the attacker's domain, e.g. by a hidden form or as a `GET` parameter. As a countermeasure, the images are integrated as iframes [186] from a different subdomain than the actual web page. The same-origin policy (SOP) [150] prevents that the attacker's payload injected in the web page can read the image data.

Finally, the order of images must change with every challenge to avoid recognisability by position, e.g. "always the left most image". In particular, clickjacking attacks are much easier if the sequence of images is predictable. The examples given in Figure 5.7 and Figure 5.9 illustrate how the images are re-arranged. The classes that are used in

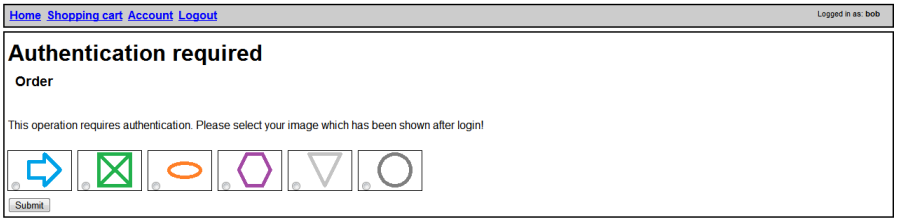

Figure 5.9: The actual shapes of the session images vary. This does not lower identifiability by users but prohibits image recognition by hashing.

both examples are the same which is necessary to prevent intersection attacks. Otherwise, the attacker could prompt several challenges and compute the intersection. The remaining set must contain the correct class because the user must always be able to choose the correct image. This way, the attacker could reduce the number of candidates with every new challenge.

Session Imagination implements a user-level authentication scheme where the browser is not able to authenticate high-security requests transparently. The conventional separation of identification and authentication is restored. The SID in the cookie serves as a temporal identification while the correct image is the authentication. As we pointed out in Section 5.3.2, a user-level authentication scheme prevents all attacks under consideration.

5.3.4 Evaluation

We evaluate Session Imagination in terms of performance, its protection properties with respect to the attacks in scope, and its usability. Also, we describe the conceptual differences to related approaches and options to increase the security gain.

Performance

The performance evaluation of Session Imagination can be restricted to the measurement of the additional steps required for the authentication of security-critical actions. The restriction to security-critical actions limits the overhead. In our prototype implementation, we considered an online shop as a use case. Putting items to the cart was possible without additional efforts while checking out and changing account information was classified as security-critical. So, for an average shopping trip, only one additional step is necessary.

Protection Goals

Next, we come back to the protection goals named in Section 5.3.3. We will show that Session Imagination is able to overcome all of the respective vulnerabilities and, thus, meet the goals. This is achieved by the introduction of identification and authentication for requests to overcome the conceptual deficiency of SID-based authentication.

Session Hijacking and Session Fixation Session hijacking and session fixation attacks both aim to steal the established session context. Session Imagination does not prevent stealing or setting the session ID. So, we consider the case that the attacker already owns the correct SID. Then, he can act on behalf of his victim unless he faces a challenge where his only chance is guessing the right image. A session hijacking attack that makes use of XSS does not increase the attacker's probability. The payload can not access the images because they are served as iframes from a different domain. The right image is not stored on the victim's machine such that the attacker can not steal or set the right image in the same way as the respective cookie.

CSRF A CSRF attacker can make the victim send a request for a security-critical operation. Though the attacker can generally not read back the application's response, he might know the application and can thus predict the form of the next request. This would be the answer to the challenge. At this point, the attacker not only has to guess the right image among the given ones but he has to guess the right image name which is a dynamic and random string of variable length. This is due to the fact that this string is used as a response parameter to decide whether the user clicked the right image and the attacker can not read the web page to learn the provided names. In this scenario, the attacker's chances are lower than guessing the right image among the provided ones.

Clickjacking A clickjacking attack prohibits the victim's context awareness which is crucial for passing the challenges.

If the attack starts before user authentication, the attacker would have to include the target web application's user login while pretending to log in on the attacker's site. Moreover, the attacker would have to make the victim provide her credentials of the target web application. We consider this to be infeasible.

If the victim is already logged in at the target web application, the attack must fail because the attacker would have to make the victim deliberately click on the session image of another web application. This task can be rendered impossible if the session images contain their web application's context, like the company's logo. If the attacker overlays the images with his own images to hide the context, the attack fails because the attacker can not link the user's session image with the respective attacker image. So, the user ends up clicking an arbitrary image which is equal to guessing the image.

To sum up, in all scenarios, the attacker can not increase his chance higher than the probability to guess the correct image.

Relation to Other Approaches

Relation to Picture-based Authentication Approaches based on password pictures differ in major aspects from our approach. First, we implement a secret on a session basis. The user thus does not have to remember another persistent password in the form of picture categories. Case studies on the long-term memorability of graphical passwords do not apply to our approach. Moreover, an attacker gaining knowledge of the user's session image can not use this after the user logs out and in again in our approach.

Second, the user is not free to choose the picture. This fact avoids that the attacker can take advantage of familiarity with the victim to guess the correct image (e.g. the respective user loves cats).

Relation to CAPTCHAs A CAPTCHA [2] denotes a "Completely Automated Public Turing test to tell Computers and Humans Apart". It is meant to provide a task that can be easily solved by a human but is hard to solve for a computer. However, a CAPTCHA contains all information that is needed to solve the task while the task consists in extracting this information. This would allow an attacker to hijack a session after stealing the session ID.

Relation to Other User-level Authentication Schemes The most wide-spread approach to make sure that the user is willing to perform the particular action is to require username and password entry again. This, however, is less secure compared to Session Imagination. First, the credentials entry form can be easily spoofed by an attacker (by XSS or phishing) which makes the victim provide her confidential login data to the attacker. Second, the username and the password can be easily stored in the browser. This makes the browser again the storage point of all information needed to hijack sessions.

The other common approach is to enter passcodes that have been received via text message. This approach has similar security properties as Session Imagination, e.g. guessing is still theoretically possible and an attacker owning the victim's platform will still succeed. However, this procedure induces additional cost and requires an additional device with GSM connectivity. Mobile and smartphones are hence excluded from accessing the respective web application because they are required as an end point of the independent second channel. Session Imagination does not require GSM availability and can be used with a single device.

Usability

We conducted a survey in order to assess the usability of Session Imagination. Therefore, we set up an online shop equipped with Session Imagination. 40 users had to provide the correct session image to check out and enter or change the shipping address. We found that 95% of them have never forgotten the correct session image. Next, we asked the test people whether they prefer another password entry ($17,5\%$), passcodes via SMS ($22,5\%$), or Session Imagination ($47,5\%$). The remaining $12,5\%$ do not like any of these. Nevertheless, $92,5\%$ would accept additional effort if this protects them from fraud. $47,5\%$ consider 2-5 challenges acceptable in the course of an online shopping trip where 45% tolerate only 1 challenge. Overall, we can say that a vast majority of all testers accept Session Imagination challenges and prefer this procedure to the alternative approaches.

Improvements to Decrease the Attacker's Chances

In our prototype implementation, we presented six images to the user, i.e. an attacker has a chance of 16.67% to guess the right image. More images can reduce the attacker's chances and increase security. As an alternative, a big picture could be presented where the user has to click a certain area to authenticate. The security level then depends on the number of areas. Further, aligned style sheets allow the provider to include many pictures while only some of them are visible to the user. This allows to increase security without lowering usability.

5.3.5 Summary

In this section, we thoroughly examined fundamental deficiencies in today's web session management and identified the common root causes of four widespread vulnerabilities, namely session hijacking with XSS, session fixation, CSRF, and clickjacking. The root causes lie in the use of browser-level authentication schemes and the missing user context on the server side.

Based on these insights, we proposed a user-level authentication scheme, named Session Imagination. It makes use of images as session-based secrets that are shared between the user and the web application. We showed its effectiveness in the sense that it mitigates the above mentioned vulnerabilities. The attacker's chances can be expressed as the probability to guess the correct session image. At the same time, this probability can be set by design to an arbitrary low value by providing a considerable number of images. The limit depends on the actual design of the user interface. We showed its usability in a survey which confirms advantages in terms of user friendliness, universal applicability, cost, and security over the two state-of-the-art approaches. Session Imagination is applicable with reasonable efforts to new and existing web applications. It is technology independent and does not create new requirements on the client side.

In sum, we provide a solution that does not tamper with the symptoms of some vulnerability but resolves the underlying problem of web session-based deficiencies. In the course of this, we achieved the mitigation of at least four vulnerabilites that are exploited in practice.

5.4 Related Work

We distinguish the related work between anti-phishing approaches that prevent the leakage of the user's password and approaches to facilitate secure sessions that prevent session hijacking and cross-site attacks.

5.4.1 Secure Login

There is a long history of approaches to overcome phishing attacks. We classify the existing body of work into three categories: approaches that augment the visible user

interface with trust indicators, approaches leveraging sophisticated authentication protocols to prevent that the real password is sent to the attacker, single sign-on protocols, and approaches aiming to distinguish between reliable and phishing sites.

Augmenting the User Interface

A number of approaches tries to protect the user from phishing attempts using individual authenticity features. The overall goal is to ensure that the user enters her password only if a pre-shared symbol indicates trustworthiness. Basic approaches just embed personalized images in the login page [149, 182].

Other approaches require the installation of client-side extensions to tune the browser's user interface. They display custom names, logos, and the certification authority (CA) of the visited website [76], open personalized windows including user-defined pictures [45], combine images with custom names of websites [199], or use colored frames to indicate the website's trust level [195, 198].

This class of approaches burdens the user with challenging tasks, including

- remembering a visual authenticity feature [149, 182, 76, 199],
- tolerating adverse impacts on usability and browsing experience [195],
- passing complex setup processes, for instance, choosing site labels, master passwords, appropriate protection service providers, and finally start the protection feature by hand [199],
- manually maintaining a list of supporting sites and compare two displayed pictures to authenticate the server before login [45], and
- install a dedicated browser [198].

Finally, these approaches are not portable and require support by the server and the client, thus being subject to a chicken-and-egg problem.

Sophisticated Authentication Protocols

The second class of phishing mitigation approaches applies changes to the common username and password based authentication. The main goal is to not submit the password in plaintext to an unauthenticated remote server but mutually authenticate client and server [68, 159, 180], utilize a zero-knowledge protocol to avoid transmitting confidential information [97], check for user-specific knowledge that changes over time [129], use trusted second devices to establish an authenticated session [139], generate site-specific passwords from a seed [146], or use bookmarks as a secure entry point [1].

The implementation of non-standard authentication protocols by design requires effort on both communication parties for support. The user either needs to store and maintain a particular bookmark for every protected website [1], remember to activate protection before entering her credentials [146], install a plugin [180] or a browser toolbar and

regularly verify that it is not spoofed [159], use a dedicated browser [97] or a second device that must be trustworthy but also able to establish a direct connection with the browser [139], or remember every past action with respect to this account [129], while some approaches are not implemented or practically evaluated [68].

Single Sign-on

A set of so-called single sign-on protocols aims at releasing the user from maintaining one unique password for each web account respectively, among them OpenID [132], Mozilla Persona [120] (aka BrowserID), SAML [106], and Shibboleth [83]. They allow a user to login once with a single authority in order to access several accounts at different providers.

The distributed authorization protocol OAuth [73] is used in some cases to log into third party web applications, too. A previous login with the provider, usually a social network, is required.

These protocols decrease the credential management overhead caused by the trend of an increasing number of web accounts. Nevertheless, the user must log in once in order to apply such a protocol. In this respect, PhishSafe complements those approaches to secure the one remaining login.

Detecting Phishing Sites

This class of approaches tries to identify phishing sites in order to warn the user and prevent information leakage. There are three main vectors for site classification: First, approaches use web crawlers to check websites for phishing features [193, 197, 59]. These approaches utilize machine learning algorithms to update their classification criteria. They generate blacklists of suspicious domains. Browsers can download those blacklists and warn the user whenever she accesses a listed site. The delay between the setup of a phishing site and the time it is listed in the browsers grants phishers a temporal advance.

Approaches of the second vector attempt to classify visited websites in real time [94, 142, 33, 145, 112]. These approaches do not suffer the time delay the blacklisting approaches have. However, they create an overhead for examination for every page access.

Finally, some approaches feed suspicious websites with bogus credentials and observe the reaction on those spoofed login requests [31, 200, 157]. The point is that phishing sites supposedly accept all combinations of username and password or always answer with an error message.

All described vectors for the classification of phishing sites are part of an arms race with phishers. The approaches rely on features that can be easily changed by phishers to circumvent classification. In a next step, the classification criteria can be adjusted and so on. Moreover, classification is always prone to mistakes, i.e., genuine websites may be classified as phishing attempts while phishing sites are treated as genuine. Nevertheless, phishing detection approaches can serve as a first line of defense and complement PhishSafe to prevent the leakage of username and password.

5.4.2 Secure Sessions

Session hijacking prevention either prohibits script access to session cookies [124, 85] or the execution of unauthorized script code [122].

Session fixation protection strives to renew the SID after authentication [99, 154, 86].

Server-side CSRF protection validates the transmitted referer [10] or request-specific nonces [90]. Client-side approaches strip off authentication information from suspicious requests [88, 151].

Clickjacking [143, 144] attacks can be partially thwarted by HTTP headers [118, 121].

However, all these approaches target only one of the attacks respectively, e.g. they protect against CSRF attacks but can not thwart clickjacking or combinations of CSRF and XSS [90]. Moreover, some of the standard defenses turned out to not provide the aimed protection level [203, 152].

There is no web-based approach with the same scope of protection as Session Imagination.

5.5 Conclusion

While the connection establishment phase of connection-oriented protocols offers to negotiate shared session secrets using well understood authentication protocols, today's web applications still send out shared secrets. In this section, we presented a combination of user knowledge and browser authentication to overcome two inherent issues of the web:

- First, users can be tricked to enter confidential information on a forged web page because they can hardly distinguish two websites with the same user interface and similar domain names. Browsers, on the other side, can easily enforce the same-origin policy and hide secrets stored in the context of one domain from other domains.

- Second, while browsers can distinguish domains, they can not know the user's intent. This makes them susceptible to confused deputy attacks [74] meaning that an attacker can make them perform actions on behalf of the user but contrary to her interest.

Using our approaches, valid authentication is only possible with the user's and the browser's approval making user impersonation attacks harder.

We will show in the next section how a shared secret negotiation phase can happen using connectionless HTTP. After setting a password, no shared secret must be transmitted for mutual authentication.

6 A Trusted Path for End-to-End Authentication

The last section pointed out that no single credentials – neither passwords nor session tokens – must be transmitted as long as they are reusable by an adversary. However, an attacker may still learn the user's password and try reusing it on web applications that do not support our approach. Also, an attacker learning the session ID can try to guess the correct image and has at least a tiny chance to succeed.

The connection-establishment phase of connection-oriented protocols allows to run sophisticated authentication protocols before proceeding to the second phase. These protocols support the mutual authentication of both communication participants and the agreement on temporary shared secrets without sending out credentials.

In this section, we will present our approach to implement a respective authentication protocol and thus simulate the connection-establishment phase of connection-oriented protocols. After the account setup, no confidential data is exchanged neither for the user login nor for session tracking.

Modern mobile devices come with first class web browsers that rival their desktop counterparts in power and popularity. However, recent publications point out that mobile browsers are particularly susceptible to attacks on web authentication, such as phishing or clickjacking. We analyze those attacks and find that existing countermeasures from desktop computers can not be easily transferred to the mobile world. The attacks' root cause is a missing trusted UI for security critical requests. Based on this result, we provide our approach, the MobileAuthenticator, that establishes a trusted path to the web application and reliably prohibits the described attacks. With this approach, the user only needs one tool to protect any number of mobile web application accounts. Based on the implementation as an app for iOS and Android respectively, we evaluate the approach and show that the underlying interaction scheme easily integrates into legacy web applications.

6.1 Motivation

Since the introduction of the original iPhone in 2008, mobile devices are first class citizens in the world of computing. Due to the impressive advances in energy consumption, mobile processor power, and display quality, the majority of the common computing tasks can nowadays be done as easily on a mobile device as on a "real" computer on the desktop.

However, while the computational power of the mobile devices is almost comparable

to their desktop counterparts, other key differences, in areas such as screen estate, UI paradigms, or operating system induced limitations, remain for the foreseeable future. These differences have a significant impact on the device's security characteristics: Reduced screen estate results in significant less space for visual security indicators that could help combating phishing attacks [4, 58]. Changed user interaction paradigms allow for different clickjacking variants [108]. Virtual keyboards on mobile devices lead to choosing insecure passwords, due to necessary, uncomfortable context switches between letters, numbers, and special characters [58]. And finally, the current restrictions in mobile operating systems and the lack of an extension model for iOS' mobile browser render most of the currently proposed attack mitigation tools impossible on mobile devices.

As we will explore in Section 6.2, these limitations especially amplify security threats against mobile web authentication. For this reason, we propose a novel authorization delegation scheme using a native application, the MobileAuthenticator, that functions as a companion application to the mobile web browser. In this section, we make the following contributions:

- We analyze how common web authentication attacks, such as phishing or clickjacking, manifest themselves in mobile scenarios and identify a common root cause – the lack of a trusted UI of the browser.
- We propose a novel authorization delegation scheme for mobile web applications that leverages a native companion application. It serves as a trust anchor for the mobile web application's client side through providing the missing trusted UI capabilities.
- We report on a practical implementation of our system as an app for the two currently dominating mobile operating systems, iOS and Android. In this context, we show how the concept can be realized through leveraging the platform-specific facilities for inter-app cooperation.

6.2 Security Threats to Mobile Web Applications

In this section, we discuss phishing, XSS, CSRF, session fixation, and clickjacking attacks in respect to how they apply to mobile web applications. Furthermore, we explore if previously proposed solutions can be adopted in a mobile environment.

6.2.1 Threat Classes

In general, mobile web applications are susceptible to the same class of threats as their desktop counterparts. However, it has been shown that several attack types, such as phishing or clickjacking, are harder to solve in the mobile scenario, due to their direct interplay with the available screen estate and web browser chrome [58, 153, 4]. In this section, we list applicable security issues and briefly discuss special aspects of the mobile case.

Phishing

It has been shown [130, 58, 4] that mobile web applications expose a higher level of susceptibility to such attacks, mainly due to the significantly reduced availability of optical indicators, such as browser chrome or SSL indicators.

Clickjacking

For the mobile case, Rydstedt et al. [153] coin the term *tapjacking* for this attack vector as users do not click but tap on their mobile devices. One of their techniques is zooming elements of the target web page. They found that the hosting (i.e., attacking) page can set a zoom factor overriding the iframe's own scaling. This way, an attacker can include a transparent "Like" or "Tweet" button fitting the entire width of the screen.

CSRF

The mobile case is similar to the desktop scenario with a slight exception: Client-side protection approaches like CsFire [151] do not work because mobile browsers have no or not sufficient extension support.

XSS

There is actually no difference between XSS attacks on mobile browsers and desktop browsers.

Session Fixation

The mobile case is very similar to the desktop case. However, sessions in some mobile web applications expire later [153], or do not expire at all but only delete the client-side session cookie upon logout [28]. This extends the attacker's control over the user's account.

6.2.2 On the Infeasibility of Existing Mitigation Approaches in Mobile Web Scenarios

In this section, we discuss several potential solutions to the outlined security problems and show their insufficiency in the realm of mobile web applications.

Client-side SSL Authentication

The current generation of – at least Android – smart phones is missing proper tools support for certificate management. Furthermore, the usage of this authentication method only solves a subset of the identified security implications, i.e., all issues that exist in connection with the potential stealing of passwords (i.e., mainly phishing). However, security problems that concern attacker-initiated state changes (e.g. caused by XSS, CSRF, or clickjacking) remain unprotected.

Browser Extensions or Plug-ins

A potential approach to overcome shortcomings of web browser-based applications is to include the security mechanism directly into the browser using a browser extension or plugins, such as Silverlight or Flash. However, the web browsers in current smart phones do not support plugins[14], and the only browser offering support for extensions is Firefox Mobile for Android with only a limited number of APIs[15].

Dedicated Modified Browsers

It is possible to deploy dedicated web browsers to mobile devices, which incorporate enhanced security mechanisms. However, they can not be used within applications that offer an integrated web-view, nor can they be set to serve as the default browser on iOS platforms, thus, excluding roughly half of all users. Finally, developing and maintaining a special browser variant is of high effort and cost, which is also a major roadblock for this potential approach.

Local Network-layer Helpers

Finally, there are several approaches that rely on local network-layer utilities, such as HTTP proxies. Such tools cannot be deployed to the current generation of mobile devices.

6.2.3 Root Cause Analysis

Generally speaking, a web application is a reactive system. The web server receives incoming HTTP requests and reacts according to the implemented business logic of the application. A subset of the incoming requests lead to changes in the server-side state while others only retrieve data stored on the server. The first case may represent security sensitive actions on the application data if received as part of an authenticated session. The handling of such requests requires special attention. Within this section, we will repeatedly utilize the term *authorized action*.

Definition 3. ***Authorized Action*** *An authorized action is a security sensitive event on the server that is triggered by an incoming authenticated request, meaning that the user authorized the web application to perform the requested action on her behalf.*

Which events have to be considered security sensitive highly depends on the internal logic of the application. Hence, the applicable set of authorized actions has to be determined on a per-application basis. Frequently encountered examples include the login to the application, changing the user's data record, and ordering and purchasing of services or goods. For all such actions, the underlying assumption is that the owner of the credential (password or authenticated SID) is the originator of the triggering event and that

[14] The Android platform offered limited support for Flash on a subset of existing devices. Adobe discontinued support by Aug 15, 2012. See `http://adobe.ly/1a1EpPH`.

[15] See `http://mzl.la/1fwQNoX` and `http://bit.ly/1k7NQOE` for details.

the details of the action have not been tampered with by unauthorized third parties. All discussed security issues have in common, that the application's back-end component (i.e., the web server) cannot distinguish authorized actions, which have been conducted intentionally by the user, from authorized actions, that have either been conducted directly by the attacker (e.g., through credentials that have been stolen via phishing or XSS) or have been initiated by the attacker via tricking the user (through clickjacking or CSRF). What web applications are missing is a *trusted path* between the user and the back-end system. The back-end system needs reliable evidence, that the initiated security sensitive actions have indeed been deliberately conducted by the user:

Definition 4. ***Trusted Path*** *An application provides a trusted path, if it can be verified on the server side that all incoming authorized actions are caused with the user's explicit consent and that their integrity is ensured.*

6.3 Mobile Authenticator

The general idea of our approach is to establish a trusted path between the user and the web application in order to protect the user against the attacks given in Section 6.2.1. We implement the approach as an app but we envisage it as an integral feature of mobile operating systems. The mobile application enables the user to communicate securely with the web application's server side using authorized actions that (1) have been explicitly initiated by the user, (2) thus are fully intended by the user, instead of being created without her consent (i.e., through clickjacking or XSS), and (3) have not been tampered with. This way, the security functionality is strongly separated from the web application's browser-based front-end, and hence, the web-specific weaknesses and limitations do not apply anymore. The actual application logic can still be implemented as a cross-platform web application which can be accessed on any web-enabled mobile device. The only part that needs to be implemented as a native application for each mobile platform is the MobileAuthenticator. The MobileAuthenticator itself provides generic security functionality. As a consequence, it can serve as a trusted interface for more than one mobile web application.

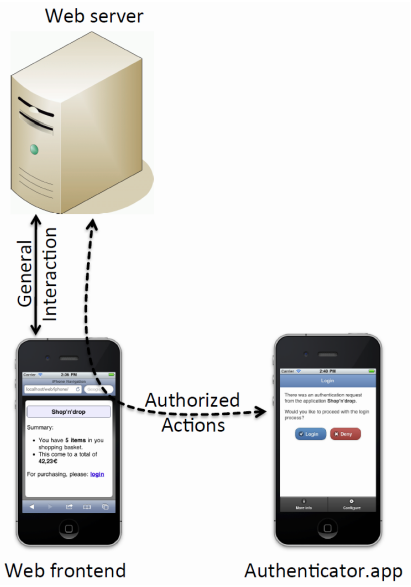

Figure 6.1: Overview of our MobileAuthenticator approach for a trusted UI and signed requests.

6.3.1 Providing a Trusted Path Through an App

We propose to introduce the MobileAuthenticator as a dedicated system app that serves as a trust anchor for the user in the communication with the web application. It establishes a trusted path between the UI and the application's back-end. However, as extending modern mobile operating systems is out of our scope, we describe the approach as an app that can be installed by the user.

Concept

The MobileAuthenticator is a dedicated application that encapsulates the user's credentials and authorization state and that maintains a trust relationship with the web server. Authorized actions are routed through the MobileAuthenticator on behalf of the web application. The mobile web browser never receives, processes, or sends credentials that can be utilized for conducting authorized actions. This way, the MobileAuthenticator serves both as a trusted UI for the mobile web interface as well as a second authentication factor, effectively elevating all supporting web applications to using an implicit two-factor authentication scheme.

Interaction Pattern

For most purposes, the interaction between the web browser and the mobile web application remains unchanged. Only in cases, when the user initiates an authorized action, the control flow is routed via the MobileAuthenticator, implementing a challenge/response scheme to capture the user's intend.

1. Using a dedicated interaction bridge between the web browser and the MobileAuthenticator, the authorized action, which is supposed to be triggered, as well as all needed parameters including the server's challenge are passed over to the app.

2. The user explicitly acknowledges the authorized action in the trusted UI of the MobileAuthenticator. This causes the MobileAuthenticator to compute the response to the server's challenge.

3. The MobileAuthenticator passes the control back to the browser including a dedicated credential which allows the triggered authorized action to be conducted.

4. This credential is passed from the web front-end to the server.

Please note: This process is only executed when authorized actions are conducted. For the vast majority of a user's web interaction, the web application remains unchanged (see Section 6.5.3). This also entails, that general authentication tracking is done the regular way, i.e., using HTTP cookies, and that application handling does not change significantly from a user's perspective.

6.3.2 Components

The overall architecture consists of three main components: The actual MobileAuthenticator that runs on the mobile device and provides the trusted UI, a server-side module that evaluates incoming requests and checks the integrity of the authentication token, and a JavaScript library that is delivered to the browser and takes care of delegation between all participants.

MobileAuthenticator

The client-side component, the MobileAuthenticator, maintains a repository of preconfigured authorized actions including a human understandable description of each action's impact. Upon receiving a security critical request from the browser, it looks up the respective action's details in its repository, displays the description to the user, and asks for consent. The MobileAuthenticator signs the request using a shared secret with the web application, and passes it back to the browser, if the user agreed.

Server-side Module

On the web application's server side a counterpart is needed that maintains a trust relationship with the user's MobileAuthenticator instance and implements the challenge/response process to accept incoming authorized actions.

AuthenticationBroker

The AuthenticationBroker is a small JavaScript library that provides the necessary interface to the application's web front-end to delegate authorized actions to the MobileAuthenticator for obtaining user consent. Upon receiving the MobileAuthenticator's response, the acknowledged request is routed to the web application for processing. It is evident that the AuthenticationBroker itself is not security critical. This is an important fact because otherwise malicious injected script code might be able to manipulate or disable the AuthenticationBroker and, thus, run an attack. The worst impact of an attack against the AuthenticationBroker, however, is a denial-of-service that prevents authenticated requests from being routed towards the MobileAuthenticator.

6.3.3 Initial Enrollment on the Mobile Device

Each instance of the MobileAuthenticator that the user wants to use has to be enrolled individually. In this process, the web application's server-side and the application instance initiate a device specific trust context, represented through a shared secret. This enrollment process works as follows.

After the account setup, the web application provides the user with a unique URL pointing back to the application, which carries parameters that identify the enrollment process. The user copies this URL to the MobileAuthenticator. The MobileAuthenticator displays the application's domain to ask the user for confirmation. The user confirms

by entering her password which is then used by the MobileAuthenticator for authentication. After the initial authentication step terminated successfully, the MobileAuthenticator and the web application compute a shared secret using the Diffie-Hellman key exchange. This secret is not only specific for the user but also for this particular MobileAuthenticator instance. The MobileAuthenticator then discards the user password as it is no longer needed. All further app-to-server interaction uses the shared secret for authentication. As long as this secret is valid, the user will not be required to enter her password again. Finally, the web application supplies a repository of configured authorized actions, including parameters and actionID, and a human understandable description of each request's impact. The MobileAuthenticator is able to maintain several of such (shared key, repository) records and can thus protect all user accounts for compatible web applications on the device.

6.3.4 User Login

After the MobileAuthenticator and the web application are synchronized, the overall login procedure adheres to the following protocol: The user first accesses the web application's login page in her mobile browser. The server can not utilize user-specific credentials at this step as the user is not authenticated yet. Instead, it issues a challenge consisting of its `AppID`, the login's `ActionID` and a timestamp. The challenge is signed using the web application's private key. The respective public key is stored in the MobileAuthenticator during enrollment.

When tapping the login button, the control is delegated by the AuthenticationBroker to the MobileAuthenticator. In this step, the server challenge is pushed to the MobileAuthenticator that takes over and asks the user whether she wants to login to this web application. A phishing attack would fail at this point, as the password is never entered to the mobile device for login.

If the signature is valid, the MobileAuthenticator compiles the response from the server's challenge, the username, and the device ID and signs it using HMAC with the shared secret, and control is transferred back to the browser. Finally, the AuthenticationBroker sends the signed login request to the web application.

Upon receiving this request, the web server extracts the username and device ID and verifies that the request was indeed signed using the shared secret and, thus, finishes the user's login process. Username and device ID are required to pick the correct shared secret for signing and verification.

6.3.5 Conducting Authorized Actions

The process for conducting further authorized actions is similar to the login process. For the login, the MobileAuthenticator witnesses the user's consent and proves the request's integrity and its own authentication by signing the request using the shared secret. The same features are necessary for authorized actions: First, in the browser, the user taps a link or a button requesting an authorized action. The respective request is then relayed to the MobileAuthenticator that obtains the user's consent, signs, and returns

the request to the browser. The AuthenticationBroker forwards the request to the web application that checks the signature and performs the requested authorized action.

6.3.6 Unknown Authorized Actions

During enrollment, the server pushes a list of allowed authorized actions to the MobileAuthenticator. If the web application has been updated since the enrollment of the MobileAuthenticator instance, it can happen that the MobileAuthenticator receives a request for an unknown authorized action. In this case, the MobileAuthenticator updates its local repository by a new list from the web application. This update process can also be triggered in a regular manner or based on push messages. After receiving the updated list from the web server, the MobileAuthenticator verifies that the requested authorized action is indeed listed. If this is not the case, the app rejects the action request.

6.3.7 Challenge and Response Formats

In this section, we briefly specify the challenge/response formats.

Server Challenge

For a given authorized action challenge, the server compiles a tuple consisting of:

$$CTuple = \{AppID, UserID, ActionID, timestamp\}.$$

The server HMAC-signs this tuple with the user-specific shared secret to allow the MobileAuthenticator to verify the challenge's authenticity. The values in this tuple have the following meanings:

- *AppID & UserID*: Identifiers of the web application and the user account, to allow the MobileAuthenticator to choose the correct authentication context.
- *ActionID*: Unambiguous identifier of the requested authorized action.
- *timestamp*: Each challenge can be assigned a dedicated lifespan to mitigate potential replay attacks.

The resulting challenge consists of the tuple and the corresponding HMAC signature:

$$SChallenge = HMAC(CTuple, shared\ secret).$$

On the server-side, the challenge is bound to the user's session and, thus, to her session identifier.

Client Response

After interacting with the user to capture her explicit consent, the MobileAuthenticator creates the response by assembling the response tuple:

$$RTuple = \{SChallenge, (Parameter_1), ..., (Parameter_i)\}.$$

Again, this tuple is HMAC-signed using the shared secret:

$$CResponse = HMAC(RTuple, shared\ secret).$$

The existence and number of the *parameters* depends on the authorized action. For instance, the login procedure requires the username and device ID, while the transfer of money in a banking application will most likely include the amount and the receiving account number in the signed response value.

6.4 Implementation

To practically evaluate the feasibility of our concept, we implemented the solution for the two leading mobile operating systems, iOS and Android. Furthermore, we outfitted the popular CMS Wordpress with server-side support for our system.

6.4.1 Client-side Implementation

In this section, we point out the platform-dependent differences between the implementations for iOS and Android respectively. Our implementation shows that the approach can be put into practice without support by platform providers though we favor an integration into the mobile platforms.

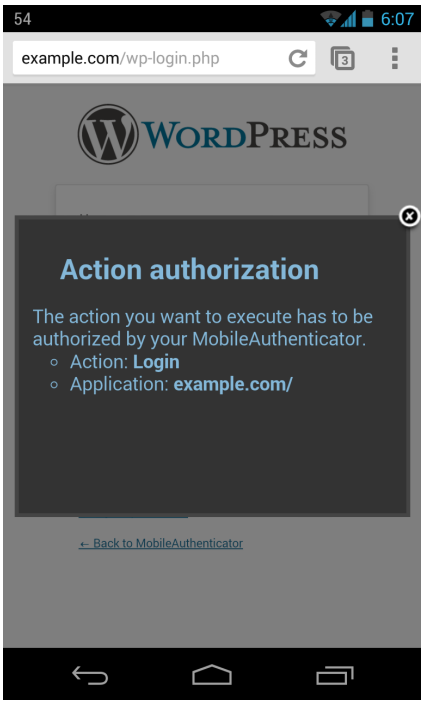

Figure 6.2: Triggering an authorized action using the MobileAuthenticator app.

Implementation for iOS

On iOS, communication between apps, such as the web browser and the MobileAuthenticator is severely limited. The only – for our purpose – usable channel is leveraging custom URL schemes: An iOS app can register a URL scheme, such as `mobileauth:`, which is registered with the operating system on app installation. When a different app accesses a URL that starts with this custom URL scheme, iOS conducts a context switch and activates the application that has registered the scheme while pushing the calling app into

background. The activated app receives the full URL in form of a string for further processing.

We use this mechanism to delegate the authorized action from the web browser to the MobileAuthenticator: The AuthenticationBroker (see Section 6.3.2) compiles a `mobileauth`-URL which carries the server's challenge and the required parameters. Furthermore, the location of the active web document is attached to the URL as the callback URL. Then, the script makes the browser request the compiled `mobileauth` URL via assigning it to `document.location`. This, in turn, causes the operating system to activate the MobileAuthenticator. After user acknowledgment, the MobileAuthenticator calls the callback (`http`-)URL and appends the `CResponse` as a hash identifier. This prevents a page reload in the browser and submits the response to the AuthenticationBroker.

Implementation for Android

The MobileAuthenticator provides a background service that is started right after the boot process completed. This service hosts a WebSocket server and is therefore accessible from the device's browser using the AuthenticationBroker and the HTML5 WebSocket API. The AuthenticationBroker establishes a WebSocket connection to the MobileAuthenticator's background service when it hooks an attempt for an authorized action. It obtains the challenge from the action's HTML meta data and pushes the request together with the challenge to the background service. The background service then launches an activity bringing the MobileAuthenticator to foreground (see Figure 6.2). After the user took a decision (either consent or denial, see Figure 6.3), the app computes the HMAC on the entire request, including the challenge, appends it and sends the whole string back to the AuthenticationBroker using the established WebSocket connection.

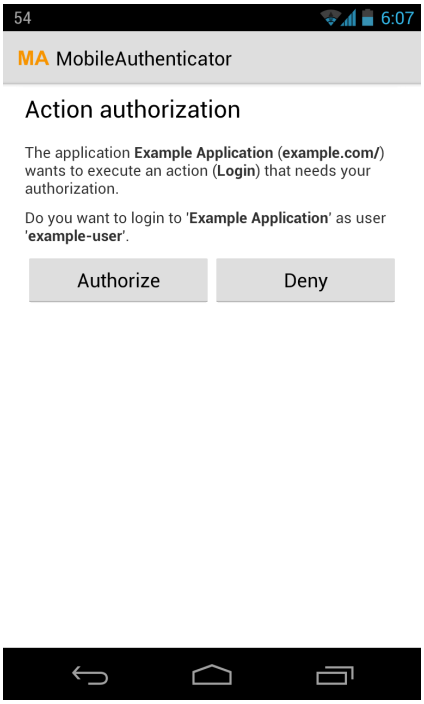

Figure 6.3: Obtaining user consent to perform an authorized action with the MobileAuthenticator.

6.4.2 Server-side Implementation

We implemented the server-side components to support the MobileAuthenticator and integrated them into the popular PHP weblog Wordpress as a plug-in. This allows to support legacy web applications without changing the existing codebase. There are three logical components of the plug-in: First, a client administration component manages the enrollment process for new devices, including a device confirmation in the user account, and the revocation of authorized device connections, e.g. because the device was lost or

stolen (see Section 6.5). Second, an action verification component issues new challenges and checks incoming requests for valid response tokens. These two components are generic and need no adaption to the particular web application. The last component, however, is application specific. It glues the above components into the legacy code, incorporates the client administration function into the user profile pages, and activates a central request filter that checks if an incoming request targets an authorized action. If so, it forwards the request to the action verification component. The Authentication-Broker is a JavaScript file that is included with every web page. It is roughly 10kb, is stored in the browser's LocalStorage together with a list of authorized actions, and hooks requests for those actions.

6.5 Evaluation

We evaluate the MobileAuthenticator with respect to its security and protection properties as well as to its usability.

6.5.1 Security Evaluation

Phishing

An attack can only succeed if the user enters credentials on a phishing site ignoring the fact that this is not necessary on her device. Expecting a redirect to the MobileAuthenticator, users become suspicious if their used comfort is missing.

Clickjacking

An attacker can still lure his victim into clicking on links but the target web application then redirects the victim to the MobileAuthenticator where the attack becomes obvious and the victim does not acknowledge the targeted authorized action.

CSRF

An attack is only detectable for a potential victim if the attacker can forward his payload to the MobileAuthenticator (see Section 6.4). Even if the attacker manages to do so, the victim suddenly faces the MobileAuthenticator asking for permission to perform an authorized action on a different website.

XSS

Injected JavaScript code can perform all actions on the user's behalf. It can raise new authorized actions and redirect the respective requests to the MobileAuthenticator. However, due to the missing shared secret, it can not sign the requests. So, as long as users do not acknowledge unintended actions, no authorized action can be triggered. The only damage an XSS attacker can cause is a denial-of-service by discarding all signed requests and, thus, preventing intended authorized actions.

Session Fixation

The attacker can still get access to the user's account. The login step elevates the session cookie to an authorized state granting access to the owner. However, the attacker can not perform authorized actions because he has no access to the shared secret.

No More Password Entry

Felt and Wagner discussed the fact that mobile keyboards actively discourage the usage of complicated, and thus secure, passwords, as the entry of numbers or special characters require cumbersome context switches [58]. Our scheme obliterates the necessity of entering passwords completely. Hence, the password cannot be stolen, as it is neither stored nor entered again. Moreover, this process allows the usage of arbitrarily complicated application (master) passwords, as the usability drawbacks upon password entry do not apply for our system.

Device-specific Credentials

As a matter of fact, mobile devices get lost or stolen from time to time. A thief or finder can use the MobileAuthenticator to log into accounts and conduct authorized actions, once he vanquished the display lock. However, there is built-in protection against this threat: During enrollment (see Section 6.3.3), the MobileAuthenticator and the web application compute a shared secret. The MobileAuthenticator does not store the user password. So, the user only has to revoke the shared secret in her account to prevent any access using the lost device. A thief, in contrast, can not exclude the user as changing the password is not possible without knowing the old password.

6.5.2 Attacking the MobileAuthenticator

We briefly discuss attacks that might apply directly to our implemented mechanism. The proposed solution as a system app is not susceptible to these attacks.

App Spoofing

An attacker may offer a malicious app via the respective platform's market, i.e., Apple's App Store or Google Play. When installed on a user's device, it could try to obtain user credentials pretending to be the legitimate MobileAuthenticator. The only occasion is the registration of new accounts in the MobileAuthenticator. This, however, is initialized by the user, usually by shortcuts on her home screen. So, as long as this malicious app is not able to replace the legitimate app shortcut with its own, the attacker can not gain confidential knowledge. We want to emphasize that spoofing the legitimate app when the AuthenticationBroker forwards the server's challenge for signing does not reveal any credentials to the malicious app, because the user only confirms or denies but does not enter anything.

The implementation as a system app can register an exclusive protocol scheme such that the registration URL is instantly forwarded to the MobileAuthenticator.

Task Interception

There is a task interception attack on Android devices. A malicious app having the necessary permissions (given by the user at installation time) can poll running tasks and display a phishing screen as soon as the target app is started. The user, expecting this screen, would probably enter the credentials. Finally, the malicious app can exit and call the genuine app. This kind of attack is not promising when run on the MobileAuthenticator because the background service is permanently running, thus, revealing no indication for the moment to spoof the MobileAuthenticator screen.

6.5.3 Usability

Felt et al. phrase crucial criteria for user-friendly interaction with respect to questions and user-based decisions [57]. We generalize and apply their criteria though they study mobile apps and their questions for permissions. In fact, the MobileAuthenticator is similar because it needs a user's decision on the permission to perform an authorized action. We show that the MobileAuthenticator complies with their criteria.

Their first point is to conserve user attention and only ask if the respective question has severe consequences. The MobileAuthenticator only comes into play when such confirmation is necessary. This way, we limit user interaction to the absolute minimum while, in the end, the web application determines the actual authorized actions (see Section 6.2.3).

Second, a usable security mechanism avoids interrupting the user's primary task with explicit security decisions. We achieve this by integrating the user question into the usual workflow. For instance, the MobileAuthenticator can ask the user for consent while presenting an overview of the purchase, including payment information, goods, shipping, etc. The user expects such a final inquiry. So, the integration of the MobileAuthenticator does not interrupt the user's primary task.

Finally, Felt et al. recommend using a trusted UI for *non-revertible*, *severe*, and *user initiated* actions. The authorized actions are generally *not revertible*, which means that the MobileAuthenticator can not let them happen and revert if needed. They are *severe*, meaning that carelessness is not an option and drawing the user's attention is justified. Finally, authorized actions are generally *user initiated*. This is an important point why one can expect the user to confirm her intent. Other, i.e., implicit, actions can not be confirmed that easily because the user does not know what to decide and why that dialogue popped up.

For instance, a usual shopping workflow and an online banking transaction only require one acknowledgment using the MobileAuthenticator respectively. This acknowledgment can be smoothly embedded in the workflow as a last step being expected by the user anyway. Social networks need to assess their users' risk: publicly posted messages on the one hand are deletable (i.e., revertible), so there is no need for a trusted UI.

On the other hand, however, annoying or insulting posts might damage the victim's reputation which is non-revertible and severe. This decision could also be left to each customer weighing her personal or business interests respectively. As a rule of thumb, an acknowledgment step using the MobileAuthenticator is at least necessary when a re-authentication (providing the password again) or second-factor authentication (e.g., via Google Authenticator, one-time passcodes, flicker codes) has been in place.

6.6 Related Work

There is no other approach covering the whole range of authentication-based attacks. Existing approaches either protect the login process against phishing [33, 45, 7, 76, 125, 147] or target session-based attacks [151, 81, 90, 165, 86, 148, 124]. Finally, the related body of work includes authentication and authorization protocols in the web [120, 106, 83, 73, 132].

GuarDroid [178] aims at establishing a trusted path between the user and the web application using a modified execution platform (firmware). It protects against malicious apps installed on the mobile device and prevents the leakage of the user's password. GuarDroid does not require changes of the installed apps nor of the remote web application, however, it can not protect against session-based attacks which still allow a malicious app to impersonate the user towards the web application. GuarDroid causes considerable network latency and requires the user to set, remember, and check a secure passphrase that authenticates the secure login form and delays the system boot process. Finally, the user is responsible to verify the target URL for login requests to prevent phishing attacks, thus, demanding a high level of awareness and increasing the risk that users just click through the dialogue.

Other existing approaches for trusted paths concerning user login [28] and user actions in authenticated sessions [22] focus on surfing web applications using desktop browsers.

6.7 Summary

In this section, we presented a web authorization delegation scheme for mobile devices that utilizes a native companion app, the MobileAuthenticator, to realize a trusted UI. For a set of predefined *authorized actions*, our system reliably mitigates state changing effects of currently known user impersonation attacks, such as phishing, CSRF, or clickjacking.

Furthermore, the MobileAuthenticator effectively becomes the user's authentication credential, obliterating the necessity to frequently enter passwords on the mobile device, thus, correcting the usability drawbacks that are observed when entering secure passwords on mobile keyboards.

The MobileAuthenticator itself is independent from specific characteristics of the protected web application and, thus, can serve as the central trust anchor for many different, independent applications. In consequence, a future integration of such a service on a

platform level into the mobile operating system is a compelling option.

6.8 Conclusion

Our approach shows that more sophisticated authentication protocols are possible over the connectionless HTTP without changing the user experience significantly. We showed in another approach that a challenge-response-based authentication protocol and signed requests for secure session tracking are possible using an off-the-shelf desktop browser [87]. Concerning our definition of connection-oriented protocols in Section 3.1, our approaches make one step towards running a connection-oriented protocol on top of HTTP. However, we left out the definition of valid messages to protect the web application's control-flow integrity. This field was out of our scope and is covered in Section 8.

The next section will shed light on the security policies applied to web sessions with differing security requirements. Not all web communication needs the highest possible protection. Instead, there are scenarios that require a higher level of security and scenarios which benefit from unrestricted communication between unauthenticated participants. We will explain how web communication covering uncritical and security-critical web sessions at the same time should be.

7 Adaptive Security Policies for Web Sessions

The last section showed how the authentication phase of connection-oriented protocols can be transferred to the connectionless HTTP. A connection-oriented version of HTTP could leverage sophisticated security features (see Section 3.3). However, there is still a part of the web that benefits from the original idea of a connectionless and lightweight communication protocol. Browsers as the general access clients must support both parts of the web: the security-critical web applications as well as the easily accessible web pages providing non-critical content.

Today, a web browser is a user's gateway to a multitude of web applications, each with its own balance between confidentiality and integrity versus cross-application content sharing. Modern web browsers apply the same permissive security policy to all content regardless of its demand for security – a behavior that enables attacks such as CSRF or sidejacking. Existing countermeasures to defend against such attacks enforce overly strict policies which expose incompatibilities with real-world web applications. As a consequence, users get annoyed by malfunctions. In this section, we show how the browser can enforce enhanced security policies, if necessary, and permit modern communication features, if possible, based on the user's authentication status and a web application's public interfaces. Our approach reliably protects the user against CSRF, session hijacking, sidejacking, and session fixation attacks while it does not decrease functionality. We present the implementation as a browser extension, named LogSec, that does not rely on server-side support and is transparent for the user.

7.1 Motivation

The past twenty-five years let the world wide web grow from a functionally limited set of isolated web pages on loosely coupled servers to a fully fledged interconnected application platform. The increasing functionality enabled new applications that are in no way inferior to their desktop counterparts – be it games, office applications, web mail, or multimedia applications. However, the upcoming new applications and their respective business models have been accompanied by an increasing number of attacks on web applications, web users, and their connection – web sessions. Looking back, the new web features are boon and bane of the world wide web.

Security researchers began to understand vulnerabilities as a misuse of functionality. Usual protection approaches include more or less targeted restrictions of implemented features to mitigate exploitation schemes. However, business models for the web differ

in their need for functionality and security. For instance, online banking websites aim to be isolated with no third-party communication and high requirements for confidentiality and integrity. On the opposite side, multimedia websites are usually mashups combining content from different origins like social network and bookmarking plugins, ads, and message boards.

In this section, we present *LogSec*, an approach that selectively applies security protection approaches depending on the actual need. We aim to permit functionality where appropriate and limit functionality where needed to serve the needs of all kinds of web applications.

A Tale of Two Webs

In the general perception, everything accessed through a web browser constitutes *"the web"*. The Google search page? Is in "the web". The new music video? In "the web". Most people's email? The access is through "the web". And online banking? As well on "the web".

However, when approaching the matter from a more abstract point of view, one can deduce two highly disparate usage scenarios for web technologies: For one, there is the traditional *web of information*, as it was envisioned in 1989 by Tim Berners-Lee, including decentralized servers, coupled by hyperlinks and iframes, providing mostly public information. On the other hand, service providers take advantage of the web browser to implement sophisticated front-ends for fully fledged *web applications* that rival the traditional host-based application architecture.

While both use cases share the exact same set of underlying technologies, namely HTTP, HTML, CSS, and JavaScript, their security and functionality requirements are highly different: The mostly stateless, information-centric web profits from the web's original focus on blurring the boundaries between physical servers. It leverages the easy, HTML-driven combination of multi-origin resources in a single web document. However, for the application-focused scenarios, robust security would demand much stricter isolation between mutual distrusting web origins than it is currently the case in unmodified browsers.

The result of this mismatch is an ever growing set of attacks on authenticated sessions of *web applications* (see Section 7.2.2). To mitigate the attacks, a set of client- and server-side countermeasures has been proposed and deployed, touching nearly all dimensions of web session handling (see Section 7.2.3). However, each of these protective measures comes with its own set of restrictions, which in turn potentially interfere with the functionality requirement of the *informational web*.

Contributions and Organization

We make the following contributions:

- We analyze existing approaches for secure session tracking and identify five central measures in respect to handling of the session credential.

- Based on this observation, we phrase a permissive policy for the *web of information* and a restrictive policy for security sensitive *web applications*.
- We describe a model for secure web session tracking, that applies both policies as required and overcomes CSRF, session hijacking, sidejacking, and session fixation attacks.
- We report on an implementation of the approach's client-side part as a Firefox extension, which applies a heuristic to waive the need for server-side support.

In the next section, we describe attacks on web session tracking and respective countermeasures. Section 7.3 describes our approach that satisfies the security and functionality requirements of the two webs. In Section 7.4, we describe our browser extension and evaluate it in Section 7.5. Section 7.6 presents related approaches before Section 7.7 summarizes and Section 7.8 concludes.

7.2 The Current State of (Secure) Web Session Tracking

We describe the current state of countermeasures against CSRF, session hijacking via XSS, sidejacking, and session fixation. Then, we condense security attributes from the countermeasures and compile two kinds of security policies: a permissive and a restrictive policy. In the next sections, we will apply those policies based on the user's authentication status.

7.2.1 Applicable Attacker Models

There are two main attacker models concerning attacks on web session tracking: The *web attacker* exploits application-level vulnerabilities, runs his own websites and injects JavaScript code into foreign web pages to steal session cookies. He embeds cross-domain resources in his applications to run CSRF attacks.

The *network attacker* targets the communication link between the browser and the web application. He extracts information from unencrypted messages and impersonates the user if possible. We assume that the network attacker is not able to break cryptography or forge SSL certificates.

7.2.2 Web Session Tracking: Attacks & Countermeasures

In this section, we briefly describe the current countermeasures to the attacks of interest. For a description of the actual attacks and their root causes, we refer to Section 2.5.

CSRF

Web applications append random tokens to URLs requiring protection. The request is only processed if the correct token is present [90]. Client-side protection strips authentication information, i.e., cookies and the HTTP authorization header, from suspicious cross-domain requests [151, 42, 104]. A cross-domain request is suspicious if it is not initiated by the user [151], the involved domains have not revealed cooperation previously [42], or the target domain accepts authenticated requests from all other domains [104]. Other approaches completely isolate different applications in the browser thus allow no cross-domain communication at all [32].

All the client-side approaches suffer from false positives: Cookies and HTTP headers are needed in the information-centric web to foster cross-application communication. Stripping them prevents the functionality of social networks' recommendation features. Existing sessions are reset meaning that entered data is lost. The latter happens when a foreign website issues a cross-domain request to the site maintaining the session. The target site receives a request without cookies and issues a new set of cookies, thus overwrites withheld cookies in the browser. For instance, the user filled her shopping cart and clicks a link on a price comparison website that leads her back to the online shop with her cart. The shop overwrites the user's session cookies aiming to initialize a new session because the last request had no cookies appended.

Session Hijacking (via XSS)

The common protection prevents JavaScript access to session cookies. This can be done using the `HttpOnly` flag [124] that is set by the server (or by a client-side heuristic [169]) and enforced by the browser. A client-side proxy can be used to maintain session cookies beyond the reach of injected JavaScript code [127].

The impact of falsely protected cookies is a lack of application functionality that relies on JavaScript access to cookies. Also, the flag is rarely used: A case study showed that only 16 % of Alexa Top 100,000 websites utilize `HttpOnly` cookies [162].

Sidejacking & SSL Stripping

The common countermeasure is a policy that makes the browser only access a website via HTTPS. Such a policy can be either pushed by the server [79] or defined on the client side using a customized set of URL rewriting rules [176].

Beside the tremendous efforts necessary to manually maintain lists of URL rewriting rules, there is a risk that outdated rules prohibit user access to websites.

Session Fixation

The most reliable protection against session fixation attacks is to issue a new session identifier during login (see Section 4). Client-side approaches can only provide partial protection, leaving cookies fixed by HTTP headers out of scope and stripping legitimate cookies set by JavaScript [43].

Overprotective approaches prevent user logins despite the absence of attacks.

7.2.3 Permissive and Restrictive Session Tracking Policies

As it can be seen in Section 7.2.2, all currently deployed and proposed countermeasures against web session attacks rely on posing restrictions in respect to the handling of either the session credential or the authenticated web data. In the remainder of this section, we systematically deduce the various individual restrictions taken by the countermeasures and derive a resulting "secure" session tracking policy.

Dimensions of Protective Session Policies

The primary focus of the regarded countermeasures is the handling of the session credential. More precisely, we can isolate five separate measures in respect to the session credential:

1. *Read access:* Read access via JavaScript to security sensitive cookies is prevented to hinder XSS-based session hijacking [127, 32, 124].

2. *Write access:* Write access to session cookies is either prevented or monitored to prevent session fixation attacks [32, 43]

3. *Cookie scope:* By default, a cookie's scope includes all valid superdomains, effectively giving a subdomain full control over the cookie's value. Attacks such as session hijacking and session fixation can be mitigated through tightening the cookie's scope to only the precise issuing domain [17].

4. *Cookie transport (HTTP):* Anti-CSRF [151, 42, 32, 88] measures prevent the browser from blindly attaching session cookies to outgoing cross-domain HTTP requests and requests to targets with overly permissive cross-domain policies [104].

5. *Cookie transport (Network):* Finally, measures have been introduced that prevent the insecure transport over unencrypted channels to avoid network-level leakage of the session credential [79, 176].

The exact specifics of how and where (browser/server) these protective measures are applied to the session credential depend on the individual techniques. For details, please refer to Section 7.2.2 and Section 7.6 as well as the corresponding publications.

Deriving Session Security Policies

Using the analysis from above, we can deduce two separate security policies. These policies are based on the discussed countermeasures and the standard native treatment of session identifiers by unmodified web browsers:

Permissive Policy The permissive policy represents the standard browser behavior: Session cookies are attached to an outgoing HTTP request only based on the request's target domain, regardless of the origin or security context of the request. The session cookie can be read and written by JavaScript if the same-origin policy is satisfied. Furthermore, all standard rules for cookies apply: The cookie's value can be read and written by subdomains, and the utilized protocol (i.e., `http` or `https`) is ignored when communicating the value over the network.

Restrictive Policy The restrictive policy is the superset of all observed protective measures: The session cookie is only attached to same-origin HTTP requests or cross-origin requests that satisfy security conditions, i.e., a non-wildcard whitelisting of the cross-domain target. The cookie's values can neither be written nor read by JavaScript. Finally, the cookie only applies to the exact domain that it has been set for and can only be communicated via the `https` protocol.

While the *permissive policy* is good for flexibility and interoperability, especially in cross-domain scenarios, it enables the various, previously discussed security problems. The *restrictive policy* on the other hand cannot be applied blindly to all HTTP cookies, as this would render various legitimate use cases impossible. Hence, a proper balance between these two extremes is needed.

7.3 Secure Web Session Tracking: How It Should Be

Section 7.2 described the tradeoff of secure and permissive web session tracking and introduced the notion of a comprehensive security policy. In this section, we go into the details of our approach that switches secure and permissive policies based on the current attack surface.

7.3.1 Goal: State-dependent Session Tracking Behavior

The overall goal of our approach is to make the browser apply the most appropriate web session tracking security policy with respect to the current threat potential. The threat potential is best estimated based on the user's authentication relation with the web application. More precisely, as long as no personal information has been exchanged between the user and the web application, a permissive policy can be applied. For instance, reading news on a website does not require elaborate protection. The potential damage that can be caused by an attacker is negligible in such a situation. Existing protection approaches are not able to allow insecure, i.e., plain HTTP, communication or extensive cross-domain communication for a better user experience in these situations.

However, as soon as a personalized context is established between the user and the web application, the communication with this application must be protected. In this phase, there is a session record on the server side that links all requests carrying the session token to the user's identity. Attackers must not be able to perform actions on

behalf of the user. The communication channel, the session credential, and cross-domain requests must undergo sanity checks to prevent user impersonation.

The distinction between harmless and security critical contexts allows the browser to enforce restrictions – if necessary – and enable rich communication features – if possible. This way, unwanted and annoying side effects of security implementations are mitigated.

7.3.2 Approach: Server-side Push of the Authentication Status

After explaining the overall goal of our approach, we now give details how we achieve this goal. While powerful security means have been proposed in the past (see Section 7.2.2 for pointers), the main issue is the determination of the authentication status. Taking into account that session records and user profiles are assigned on the server side, it must be the web application that decides on the actual authentication context. So, the web application pushes the current status to the browser. The browser can then apply the appropriate security policy. This approach has a couple of advantages.

First, the implementation can happen asynchronously. A web server can start pushing authentication information at any time in a fashion that is ignored by legacy browsers, e.g. using HTTP headers or cookies. Vice versa, a supporting browser can still access legacy web applications that do not push the user's authentication context. In both cases, the browser enforces the permissive default policy without a negative impact on the web application's functionality. As soon as the browser and the web application support authentication context switches, users benefit from the increased protection level. In this sense, the introduction of the authentication context is similar to that of the content security policy (CSP) [165].

Second, the approach does not rely on the installation of third party features like browser extensions or toolbars. This provides protection "out of the box" without awareness and actions by the user.

Finally, the standardized browser support makes the impact on web communication predictable for the application provider. The provider can start pushing the user's authentication context when the application's compatibility is proved. This prevents the frequent negative side effects of security means annoying users and reducing acceptance.

The `auth` Cookie Attribute

The client-side state is usually determined by HTTP cookies. We use a special cookie to set the session status on the client side. For this, we introduce a new cookie flag, named `auth`, indicating an authenticated session. The server issues a cookie having the `auth` flag set as soon as the session enters an authenticated or personalized state. This happens after the login process has terminated successfully or when personal data is entered without a login, for instance to purchase a flight ticket. The cookie must not be readable nor writable by JavaScript to prevent user impersonation by session hijacking or session fixation attacks. Also, the cookie is only sent back to the server over a secure channel. In this way, the `auth` flag implies the `HttpOnly` and `Secure` flags. As cookies are sent with every request, the web application can check the client-side authentication

status and correct it if necessary. The cookie-based approach comes along with benefits compared to HTTP header-based implementations:

- A cookie directly affects the browser's state instead of instructing the browser to change its state. Also, this state persists by default after the document is unloaded.
- Just like a session on the server side, a cookie has a lifetime that can be set according to the session lifetime on the server side. This way, the browser switches the authentication status when the server-side session expires without further communication.
- The authentication status depends on the domain, subdomain, and/or the path of the web application. So, the scope of a cookie fits the needs of the authentication scope definition.

Please note that we do not advocate to use cookies as the only authentication credential after login but to use one particular cookie in addition to existing authentication.

7.3.3 Restriction of Authenticated Cross-domain Communication to Public Interfaces

Given the authentication status of a session, the browser can determine which requests might have an impact on the user's account. However, modern web applications implement intended cross-domain interaction patterns, e.g. social media and bookmarking plugins. These require user authentication for cross-domain requests to assign actions to a user account. Unfortunately, such benign requests are methodically indistinguishable from malicious requests from the browser's perspective. This is the main reason why client-side protection approaches against CSRF attacks intercept too many requests leading to an unintended loss of functionality (see Section 7.2.2). Again, only web applications can reliably declare their intended interfaces for authenticated cross-domain requests. A list of intended interfaces must be pushed to the browser which then only permits intended cross-domain communication.

An authenticated request to a web application's public interface should not have an immediate impact on the user record. Instead, public interfaces allow read-only access, e.g. a list of friends who like the current web page. Such a request does not change the user's account data and thus can not cause harm if it is issued by a third-party domain. Public interfaces may also serve as landing pages to start a workflow, for instance the opportunity to like the current page in the next step. This next step is then a same-domain request.

The declaration of public interfaces is by its nature equivalent to the definition of a policy. Usually, servers deliver policies using HTTP headers. Recent examples are the cross-origin resource sharing (CORS) [181], content security policy (CSP) [165], public key pinning (PKP) [55], and HTTP strict transport security (HSTS) [79] policies. These approaches define document-wise (CORS, CSP) or site-wise policies (PKP, HSTS) that are eventually enforced by the browser. Consequently, the server pushes the list of public interfaces as an HTTP header.

7.3.4 Security Benefits

After explaining how our approach works, we show that it protects against the common attacks on web session tracking.

Cross-site Request Forgery

The core of CSRF attacks is the execution of authenticated actions on third-party web applications on behalf of the victim. Given the authentication status with the respective third-party application, the browser can ensure that authenticated cross-domain requests only target public interfaces.

Session Hijacking via XSS

For session hijacking, the attacker needs to share session credentials with his victim. Since the web application delivers the special `auth` cookie when the user is authenticated, it can require this cookie as one authentication element. This cookie is by design not accessible to injected JavaScript code and thus not susceptible to session hijacking attacks.

Sidejacking & SSL Stripping

Sidejacking attacks base on the insecure submission of session credentials. The `auth` cookie, however, as one part of the authentication is only transmitted over secure channels. Also, a man-in-the-middle attacker that redirects the user to insecure channels can not succeed because the browser detects the plain HTTP communication and avoids sending the `auth` cookie.

Session Fixation

In our approach, the attacker needs to inject a `Set-Cookie` header into an HTTPS response from the target application's domain in order to set an `auth` cookie in the victim's browser. He needs to be able to run code on the server side or control the victim's browser, e.g. via a malicious extension, to achieve this. In both cases, the attacker has more powerful options than running a session fixation attack.

7.4 Implementation of Client-side Protection

As discussed in Section 7.3, in an idealized world, the web browser would apply different policies for session handling depending on the user's authentication status. However, up to this point, browsers do not implement our adaptive session tracking policy. For this reason, we conducted a practical implementation in the form of a Firefox extension named LogSec. Implementing the measure in this fashion has two advantages:

category name	keywords
login	login, log in, signin, sign in, sign up, anmelden, nicht angemeldet, abgemeldet
logout	logout, sign out, log out, log_out, lgout, abmelden, sitzung beenden, eingeloggt
pwd	password, pwd, passwd, pass, passwort
keeploggedin	keep me signed in, keep me logged in, stay logged in, automatisch anmelden
accsettings	account settings, account einstellungen, kontoeinstellungen
...	

Table 7.1: An excerpt of LogSec's keyword categories and assigned expressions.

- For one, through using our extension, end users can benefit from the security advantages of our approach today, without having to wait for the uncertain adoption of the technique by the browser vendors.
- Furthermore, this practical implementation gives us the opportunity to gain experience using it under realistic circumstances. Consequently, this implementation was used as the basis of our practical evaluation in Section 7.5.

The main drawback of our implementation is the missing support from the server side. Currently, web servers do not provide browsers with dedicated indicators that the authentication status of the user has changed. Nonetheless, we have to make up for this absent component to provide the projected protection characteristics for end users today. Therefore, we designed a heuristic which automatically deduces authentication status changes through passively monitoring the browser's web traffic.

7.4.1 Detecting Session Status

LogSec implements a binary session status detection: Either the user is authenticated on a domain or not. This section outlines the details of the implemented heuristic.

Keywords & Categories

The basic indicators of our session status detection heuristic are used keywords. We found that not only visible page content but also embedded links reveal information regarding the session status. Strong indicators confirm the last user action in plain text (e.g. "You successfully logged in") or offer the respective status changing action as a link (e.g. "Logout"). When the user clicks a respective link, the HTTP request contains keywords, thus indicates an ongoing status change. We subsume synonymous words, respective abbreviations, and short phrases in categories (see Table 7.1).

Scoring

In order to evaluate the keywords found in requests and page content, we assign numeric values to each category of keywords. Keywords that suggest an authenticated session have a positive value, keywords that feature an unauthenticated session have a negative value (see Table 7.2). LogSec maintains a score value for each domain the user visits. Initially, the score is 0 for every domain, and the status is *not authenticated*. For each request and each page load, the extension adds up the values of all finds. Different weights are assigned to keywords found in requests and page content (see Table 7.2). We name the result the request offset and content offset respectively. Each offset can be positive or negative depending on the occurrence of login and logout indicators. There are two cases:

- Either the user is not authenticated, yet. Then, a positive offset is added to the domain score. The status switches to *authenticated* if the score exceeds a given threshold. A negative offset, however, is ignored as it confirms the current status and contains no new information.
- Or the user is already authenticated. Then, accounting happens the other way around. Only a negative offset has an impact while positive ones provide no new information. The status switches to *not authenticated* if the score falls below a given threshold.

After each status switch, the score is reset to 0. Finally, there are evident indicators for a change of the status. An HTTP POST request having a `username` and a `password` parameter makes LogSec switch to *authenticated* immediately. The recognized names of the parameters are again sets of keywords, e.g. `user`, `userid`, and `passwd`, `pwd`, `pass` respectively. If the login fails, there is a short phase where the user is protected though not authenticated. After the next page load, the status switches back to *not authenticated* if the page contains a login form.

Another evident indicator is the `Authorization` header in an HTTP request [63]. LogSec sets the status to *authenticated* for the target domain of the request. In case the provided credentials are not valid (or no credentials are provided at all), the web application answers with HTTP status code `401 Unauthorized`, the last unambiguous indicator. The status then switches to *not authenticated* for the requested domain.

7.4.2 Protection Features

The achievable security level of the LogSec extension is lower than the security level of the approach described in Section 7.3. The extension can not rely on server-side support and thus assumptions concerning the authentication cookie and a web application's public interfaces. Nevertheless, it provides reliable protection against CSRF and sidejacking, and at least partial protection against session hijacking via XSS and session fixation.

In order to identify session cookies among all cookies set by the web application, we leverage a heuristic [127] that relies on the cookie's name, the length and the entropy of the value. The underlying assumption is that session cookies must not be guessable.

category	found in ...	influences	impact
logout	hyperlink	content offset	+5
logout	source code	content offset	+4
login and *pwd*	source code	content offset	–4
accsettings	source code	content offset	+1
pwd, *keeploggedin*, *forgot*	source code	content offset	–1
login	form action	content offset	–2
login	request URL	request offset	+2
logout	request URL	status	false
credentials	POST request	status	true
Auth. header	request	status	true
HTTP 401	response	status	false

Table 7.2: LogSec's keyword categories and their related weighting.

CSRF

Basically, only the same-origin policy is enforced and cookies are appended, if the session status with the target domain is *not authenticated*. If, however, the user is authenticated on the target domain, the request passes through a series of checks (see Figure 7.1). The basic rationale behind these checks is to allow authenticated cross-domain requests if the domains provide an indication for collaboration.

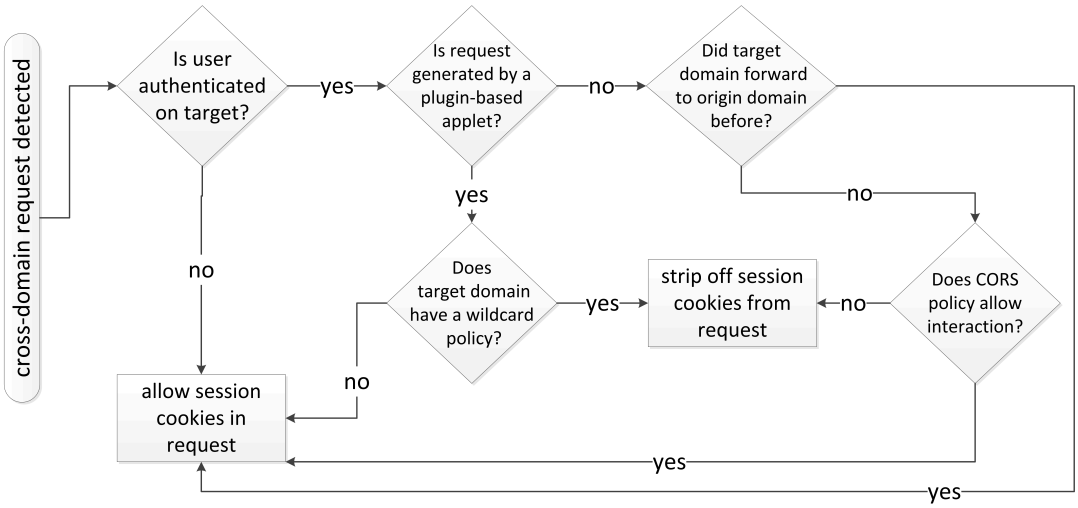

Figure 7.1: The CSRF protection policy of the LogSec browser extension: combining observable cross-domain interaction with authentication status information.

Requests issued by Flash and Silverlight are not subject to the same-origin policy nor to the CORS policy headers. Instead, each target domain defines its own cross-domain policy, i.e., a set of domains that is allowed to perform authenticated actions on the target domain. A domain providing a wildcard policy allows every other domain to

perform actions on behalf of the user. So, cross-domain requests that are generated by those plugins must either target a domain that does not have a wildcard cross-domain policy or must not be authenticated [104]. LogSec strips session cookies from cross-domain requests to wildcard domains. If, however, the target domain only allows a limited set of other domains to send authenticated cross-domain requests, we assume a trust relationship and let the request pass unmodified.

Cross-domain requests that are generated by HTML content need to pass the following check. An authenticated cross-domain request from B to A is allowed if:

- Either A previously forwarded the user to B, i.e. B returns control now [42]. A forwards to B if it redirects the user to a URL on B or if the user sends an HTTP POST request from A to B, for instance to process the payment after a purchase.
- or A authorizes cross-domain requests by explicitly listing B in an `Access- Control-Allow-Origin` HTTP response header. We assume that the explicit authorization using CORS is sufficient to allow authenticated cross-domain requests. A wildcard ($*$) CORS policy is not sufficient.

In general, LogSec allows authenticated cross-domain communication if it finds hints for collaboration between both domains. This approach copes without a list of public interfaces (see Section 7.3.3).

Sidejacking

The aim of sidejacking protection is to prevent that session cookies are sent over an insecure channel if the user is authenticated on the target domain. LogSec allows unrestricted communication before the login and prevents the leakage of session cookies after the login via unencrypted channels.

Session Hijacking via XSS

It is not an option to blindly mark all cookies as `HttpOnly` because the web application might rely on JavaScript access to some of them. Instead, LogSec flags identified session cookies after login if no other cookie is marked `HttpOnly`. The rationale is that either developers are not aware of this attribute then no cookie is flagged, or developers are aware and flag at least one cookie then the rest must be accessible by JavaScript to prevent a loss of functionality [169].

Session Fixation

A web application is vulnerable if it does not issue fresh session identifiers during user login. Most web applications, however, deliver more than one cookie that matches our session cookie criteria. Since LogSec does not know the meaning of all these cookies, there is a tradeoff between security and usability: On the one hand, LogSec could issue a session fixation warning if at least one session cookie remains unchanged after login. In this case, it covers all possible session fixation vulnerabilities but also suffers false

positives, i.e., cases where the actual session cookie is changed but another cookie is not. On the other hand, LogSec could only issue a warning if all session cookies remain unchanged after login. In this case, every warning is a true session fixation vulnerability but some vulnerable sites may be missed. We implement a conservative approach and only trigger the session fixation warning if no session cookie is renewed or freshly issued upon login. In this case, the vulnerability is proved. We implemented an optional session fixation protection feature that finds the logout link, performs the logout to invalidate the fixed session, requests the login page without cookies appended to obtain a new set of session cookies, and sends the login credentials again.

7.5 Evaluation

We evaluate our implemented approach, LogSec, in terms of its login detection accuracy, functionality and provided security.

7.5.1 Login Detection Quality

The login detection accuracy is important for the protective impact of LogSec and the improved functionality of websites where the user is not authenticated.

Evaluated Websites

We used the Alexa Top 300 worldwide list (effective 31-08-2013) and removed apparently illegal and suspect sites, for instance file sharing and streaming of material protected by copyright, to determine the overall login status detection rate. Afterwards, we registered an account for every single site where possible. We failed to resolve Russian captchas and to provide a Chinese cell phone number that was required to receive an activation token. Finally, a number of sites do not offer the possibility to create a user account. We used the remaining set of 100 different domains as a basis to evaluate the login detection rate. We performed a manual login on every website and checked whether LogSec determined the status correctly.

Detection Accuracy

Our evaluation shows that LogSec is able to correctly recognize authenticated sessions on 96 % of all sites. The detection of authenticated sessions is crucial for security because web sessions falsely identified as *not authenticated* are not protected by LogSec.

The session is properly identified as *not authenticated* on 82 % of the websites. These sites benefit from the relaxed security protection and thus from better usability.

The used domain set includes many websites with different languages for which we do not have any keywords. Currently, the list contains English and German keywords. Nevertheless, LogSec's login detection rate on these sites is similar to the rate on English and German sites. We identified the reason in the fact that action parameters in links or credentials in POST requests are still English expressions. As a matter of fact,

adding support for more languages is straightforward: All words that are added to a category are recognized as an indicator. So, one only needs to add respective keywords and abbreviations in any language for "login", "logout", "password", etc. to the list of keywords (see Table 7.1).

We found that the detection of the login status works correctly on many websites implementing a login via single sign-on (SSO). We tested *Yahoo*, *Facebook*, *Google*, and *Twitter* as identity providers and *flickr*, *4shared*, and *Pinterest* as relying parties to log in. Also, Google implements a similar protocol to log in to *Youtube*. LogSec successfully detected the login and logout on all these sites. Finally, LogSec successfully detected SSO logins for Microsoft's *live* apps.

Lessons Learned from Status Detection

In order to improve the authentication detection rate, we analyzed the reasons for false positives, that are identified as *authenticated* but the session is actually not, and false negatives, that are identified *not authenticated* though the session actually is.

- Initially, LogSec counted all request and content offsets independently from the current status. So, indicators supporting the current status were counted again and again with every user request. As a result, a high number of request-response-pairs without a status change added to extremely high or low scores that did not switch when a status change finally happened. We fixed this issue by only counting indicators for the opposite status, as described in Section 7.4.1.

- We found that some images, for instance representing the logout action, have characteristic names, e.g. `logout.jpg`. Those images are only loaded when the user is logged in to visualize the logout action. However, LogSec identified the request for a resource with a characteristic name as a strong indicator for a logout step. We went over to ignoring requests for images and gained a significantly better detection rate.

- Another issue occurred when the browser terminated. By default, Firefox keeps cookies across browsing sessions as long as they do not expire. LogSec, however, reset the session status for all domains when Firefox was closed. As a consequence, the session was maintained in the browser but the extension lost track. We coupled the lifetime of the session status on the cookie management to overcome this problem.

- Finally, LogSec is not able to determine the context of keywords. Every find contributes to the offset. In one example, the article of an online newspaper contained a huge number of keywords that alter the score and thus the session status. This is the only issue we could not fix yet.

7.5.2 Security

We consider the security features of LogSec (see Section 7.4.2) and analyze how an attacker may bypass protection. We want to stress that the approach presented in Section 7.3 provides reliable protection against all of the attacks.

CSRF

A CSRF attacker attacking web application A must find another web application B to plant his payload. B must be either whitelisted in one of A's cross-domain policies (CORS, Flash, or Silverlight) or the target of a redirection from A. In the latter case, the payload must be injected into a web page that is accessed by the user as part of the respective cross-domain workflow. The overall attack surface is substantially restricted if A does not have an open redirect function.

Sidejacking

A sidejacking attacker can only succeed if the target web application does not change the session ID after login – which results in a session fixation vulnerability, see below. Then, the attacker can reuse the session cookie that is transmitted insecurely before the login. A sidejacking attack on the authenticated session cookie must fail because it is only sent using a secure connection.

Session Hijacking via XSS

A session hijacking attack is only possible if a web application marks at least one cookie as `HttpOnly` but not the necessary session cookie. LogSec sets the `HttpOnly` flag for all session cookies if no cookie has the flag set.

Session Fixation

As mentioned in Section 7.4.2, we had to trade off security against user annoyance. LogSec is able to provide complete protection if false alarms are tolerable, or provide partial protection with no false alarms. During our evaluation, LogSec correctly identified a session fixation vulnerability in the Stud.IP campus management system of the University of Passau. We reported our finding and the vulnerability is fixed by now.

Manipulation of LogSec's Status Tracking

An attacker may embed cross-domain resources to make the browser send respective requests and switch the status as detected by LogSec. For instance, an (invisible) image from `example.com/logout.jsp` could trigger LogSec's logout detection on `example.com` even if this resource does not even exist. If successful, the user's account on `example.com` would no longer be protected. For this reason, LogSec's logout detection ignores cross-domain requests. A web application provider may still confuse our status tracking on

his own website. In his domain, however, a provider has more powerful attack options anyway.

7.5.3 Functionality of Websites

In order to evaluate a possible negative impact on functionality, we performed a manual analysis of those websites where LogSec determined the login status correctly (see Section 7.5.1). We say that a website *behaves as expected* if typical actions can be conducted without any noticeable interference. Typical actions for a web mail service for instance are signing in, checking the inbox, and sending an email. We found that 94 % of all sites behave as expected with LogSec enabled.

We checked social network buttons as a special case for cross-domain communication: The user is authenticated on the target web application but there is no hint (in terms of cross-domain policy whitelisting or redirects) for cooperation between the currently visited website and the social network. So, the first request targeting the social network's domain contains no session cookies. Usually, this request loads an iframe, so that the user's like action happens within this iframe as a same-domain request. The social buttons of Twitter and Google work flawlessly. However, utilizing Facebook's *Like* button is not possible. The button is visible as are all social plugins but clicking the button has no effect. During our first checks with LogSec, we did not have issues with the *Like* button. At some time, Facebook switched to only accept Likes after the iframe is loaded in the respective user context. The introduction of public interfaces (see Section 7.3.3) would preserve the whole functionality.

7.6 Related Work

There is a number of approaches to overcome insecure cross-domain interaction in the browser. None of them considers the session status as a factor for security relaxation. Instead, they isolate content from different origins more strictly then today's common browsers.

FF^+ [26] is a formal approach to model client-side session integrity. While the protection goals overlap ours, the focus is more on security than on preserving usability with existing security approaches. The implementation SESSINT heavily interferes in web communication: It strictly separates all domains in the browser unless there is explicit user action that serves as an indicator for intended cross-domain interaction. It enforces HTTPS for all connections after submission of a login form and marks received cookies as `HttpOnly` and `Secure`. Users must explicitly justify the initial page load to avoid its classification as a cross-domain request.

Gazelle [189] and its predecessor OP Browser [70] are functionally complete web browsers that apply OS design principles. Different origins are separated like processes, and communication between them must use explicit communication channels. Tahoma [35] goes in a similar direction but also emphasizes an additional layer for separation of the browser and the underlying OS. Also, it strengthens the user's knowledge

about and the control over a web application's code and content. In OMash [36], web pages are treated like objects in object-oriented programming languages, i.e., each page can have private and public interfaces to communicate with other pages. MashupOS [188] introduces four types of content isolation, the respective application is supposed to depend on the trust level between the content provider and the content integrator.

While these approaches describe a thorough and secure restart of browser technology and cross-domain interaction, they are not compatible with most of modern web applications. Other approaches modify existing browsers to mitigate selective issues: SOMA [131] extends the same-origin policy for inclusions of third-party content by a mutual confirmation step of the content provider as well as the integrator. This way, SOMA wants to prevent malicious cross-domain requests and mitigate CSRF and XSS attacks at the cost of one additional HTTP request per third-party domain. Doppelganger [158] aims at deriving the privacy-optimized cookie policy for any web application. It detects differences in the application's response depending on whether the respective request has cookies or not. The rationale is that no cookies are needed if the response is equal. Security considerations are not part of this approach.

Calzavara et al. [30] did a thorough analysis of real-world cookies to classify them as either authentication cookies or not. They compare the results of several heuristics, among those our session cookie classification heuristic, with their verified results. Our approach turns out to be over-approximating, meaning that it identifies the vast majority of authentication cookies correctly but misclassifies some other cookies as authentication cookies. As a result, our approach provides thorough protection but still has an impact on usability.

7.7 Summary

In this section, we revisited the fundamentals of secure web session tracking. After reviewing the existing body of work on this subject, we condensed two applicable security policies: the *permissive policy*, representing the default behavior of the web browser, and the *restrictive policy*, the superset of all measures taken by existing attack mitigation techniques. Furthermore, we observed a mismatch in how the web is used: For one, there is the *information-centric* use case, which profits from the web's original focus on cross-server navigation and communication flows. In the *application-centric* use case, however, web technologies are utilized to create sophisticated applications. While the former case's functionality requirements are hurt by the restrictive policy, the latter case's security potentially suffers under the permissive policy. We propose an adaptive switching of the applicable session tracking policy, depending on the authentication status of the user, in order to create a reasonable balance between these extremes.

We presented our technique in two fashions. For one, we showed how browsers could implement the policies natively using server-supplied meta information whenever the user's authentication status changes. Our approach is easy to integrate. Legacy web applications can upgrade using a server-side module or a reverse proxy if code changes are not an option. It allows a stepwise integration without breaking existing functionality.

Supporting browsers and web applications benefit from a higher security level while communication is left unchanged as long as one communication party does not support our approach. The browser's behavior is predictable for the web application provider at any time. This is important to prevent annoying side effects and malfunction as it happens with purely client-side approaches.

Furthermore, we presented LogSec, an extension for the Firefox browser, that makes the approach's benefits available for end users immediately. We substituted the server's precise authentication status indication with a heuristic method, which deduces authentication status changes automatically through passively monitoring the user's web browsing.

Regardless of the chosen implementation approach, we are able to provide authenticated users with all essential protective measures through applying adaptive session tracking policies. These measures robustly mitigate widespread session attacks without affecting the general functionality requirements of the public web.

7.8 Conclusion

Our approach unifies the best of two worlds: the lightweight flexibility of the connectionless HTTP and the enforcement of security features to protect communication channels as an approximation of established connections. This way, it has regard to the dual use of today's web and not only to the maximum security level.

With this section, we finalize our steps towards secure authentication and session tracking in the web. The next section will address the missing alignment of valid messages by the connectionless HTTP. We will show how web applications can protect against state manipulations.

8 Request Filtering to Preserve Control-flow Integrity

The previous sections contributed to a better user authentication and session tracking. In this section, we explain an inherent problem of connectionless protocols in combination with stateful applications: The set of valid messages is not adapted by the protocol because connectionless protocols have a static set of valid messages (see Section 2.4). Stateful applications, however, have dynamic interfaces meaning that for each step in the execution of a workflow, only a subset of all valid messages must be processed because the processing of other messages would lead to an unforeseen state. In sum, an attacker can exploit the connectionless nature to manipulate an application's state in his interest if the application does not prevent respective attempts. We will show how web applications can filter unforeseen requests and preserve control-flow integrity.

8.1 Introduction

Modern web applications frequently implement complex control flows, which require the users to perform actions in a given order. Users interact with a web application by sending HTTP requests with parameters and in response receive web pages with hyperlinks that indicate the expected next actions. If a web application takes for granted that the user sends only those expected requests and parameters, malicious users can exploit this assumption by crafting harming requests. We analyze recent attacks on web applications with respect to user-defined requests and identify their root cause in the missing explicit control-flow definition and enforcement. Then, we evaluate the most prevalent web application frameworks in order to assess how far real-world web applications can use existing means to explicitly define and enforce intended control flows. While we find that all tested frameworks allow individual retrofit solutions, only one out of ten provides a dedicated control-flow integrity protection feature.

Based on this result, we provide our first approach, a control-flow monitor that is applicable to legacy as well as newly developed web applications. It expects a control-flow definition as input and provides guarantees to the web application concerning the sequence of incoming requests and carried parameters. It protects the web application against race condition exploits, a special case of control-flow integrity violation. Moreover, the control-flow monitor supports modern browser features like multi-tabbing and back-button usage. We evaluate our approach and show that it induces a negligible overhead.

Our second approach, named *Ghostrail*, is also a control-flow monitor. It works as a

proxy and is meant to be applied when access to and knowledge of the web application business logic is unavailable. It observes incoming requests and lets only those pass that were provided as next steps in the last web page. Ghostrail protects the web application against race condition exploits, the manipulation of HTTP parameters, unsolicited request sequences, and forceful browsing. We evaluate the approach and show that it neither needs a training phase nor a manual policy definition while it is suitable for a broad range of web technologies.

8.2 Root Causes for Attacks on Control-flow Integrity

Several kinds of attacks exploit the fact that attackers can craft arbitrary requests instead of clicking on provided hyperlinks. Real-world examples of control-flow integrity violations are race conditions (see Section 2.5.6), manipulated HTTP parameters (see Section 2.5.7), unsolicited request sequences (see Section 2.5.8), and the compromising use of the browser's "Back" button (see Section 2.5.10).

Web application developers assume that users first request one of possibly several application entry points, e.g. the base directory at `http://www.example.com`. Upon the first request, the web application sends a given response containing a set of hyperlinks or a redirect instruction to the browser. As users tend to click on hyperlinks in order to navigate through the application, developers might assume that only the given requests will be accessed next. However, the user is technically not bound to click on one of the provided hyperlinks but she can still send requests that are not provided within this response. Sent requests can differ from provided hyperlinks in terms of addressed methods and HTTP parameters. Vulnerable web applications fail to handle unintended user behavior in terms of sequences of requests.

More formally, web application developers implement implicit control-flow graphs. In each state, sending a request leads to a subsequent state in the graph. Executing a step corresponds to changing the server-side state. Control-flow weaknesses occur if an attacker is able to address at least one method, i.e., cause a state-changing action, that is not meant to be addressed in the respective session state. This transition does not exist in the respective control-flow graph due to the developer's assumption that the request does not happen at that time. Vice versa, a web application implementing a control-flow graph with transitions for all requests in every state is not susceptible to control-flow weaknesses.

Forceful browsing attacks and some cases of HTTP parameter manipulation can be overcome with access control. The other attack vectors, however, include only requests that are in the scope of the user's rights. Access control mechanisms prevent users from accessing sensitive API methods at all times. Control-flow integrity protection, however, prohibits access to regular API methods in an unsolicited order or context.

The measure to achieve this can partially overlap with CSRF protection: Web applications can issue tickets in the form of nonces that must be appended to requests [90]. A request without a ticket is not processed. This prevents that CSRF attackers can craft requests that are finally executed on behalf of the victim. In some cases, this can also

prevent attacks on control-flow integrity: First, nonces must be unique for every request. Some web applications use only one ticket for a user session to save server-side resources. While a session-wide ticket reliably prevents CSRF attacks, it can not prohibit attacks on control-flow integrity. Second, a ticket must be bound to the whole request including all parameters. Otherwise, an attacker could tamper with unprotected parameters and change a request's context. The first example concerning HTTP parameter manipulation given in Section 2.5.7 describes such an attack. Third, the ticket must be invalidated immediately after use to prevent race condition exploits and faults due to "Back" button usage. Both of these scenarios use correct request-ticket combinations but more often than expected. Finally, even if all these measures are taken properly, there is still an open attack vector: The user can start the same workflow in different sessions up to the point where a race condition exploit should be run. Then, he can perform the next step in all sessions in parallel with all requests equipped with correct tickets.

Existing web applications enforce the intended control flow based on session-contained parameters. This allows only the implicit definition of workflows. The previous actions are assumed to set the parameters and, thus, allow the execution of next actions. The actual workflows are not explicitly determined preventing the proper assessment of enabled workflows. The central and explicit definition of facilitated workflows provides guarantees of request sequences to the relying web application. One crucial aspect of reliable request sequences are controlled HTTP parameters as we show by the attacks in Section 2.5.7.

To sum up, we can say that uncontrolled sequences of user requests might cause confusions on the web application's state if it does not take care of handling even unprovided requests. In the next section, we dive deeper into precautions provided by web application frameworks.

8.3 Survey: Control-flow Integrity Means in Web Application Frameworks

Modern web applications are usually developed with the help of web application frameworks. Such frameworks encapsulate basic functionality that can be reused for application development at a large granularity level. Typical features include session initialization and cookie delivery as well as HTTP communication and HTML content generation support. The application code then implements the actual business logic and uses high-level functions provided by the framework.

Almost every web application that implements a business logic spanning several request-response round trips has a need for control-flow integrity. So, a control-flow integrity module should be reusable. Web application frameworks provide sets of reusable features to facilitate web application development. In this section, we examine the ten most prevalent web application frameworks on their support for control-flow integrity. This gives us an insight how far the majority of web applications can use and add control-flow integrity protection without changing the application or the underlying framework.

Looking at it the other way round, missing support requires developers to manually implement protection means, which, as history shows, leads to more weaknesses because the implementation is often either omitted or flawed. We also check two crucial aspects of control-flow integrity: parameter integrity, which means that malicious users can not tamper with the HTTP parameters' data type, and race condition protection, which mitigates attack vectors based on the same request sent multiple times in parallel.

8.3.1 Probed Web Application Frameworks

In this section, we describe our survey on control-flow integrity protection means of the most prevalent web application frameworks. We tested the top 10 web application frameworks according to the BuiltWith index [27] on 12 Jan 2013. The list contains the most common server technologies among the 10,000 most popular websites. However, it also includes technologies that are out-of-scope for our survey because they only denote the platform, e.g. PHP. We are aware that PHP itself does not provide any control-flow integrity means, thus, we omitted all technologies that do not fall within the following definition:

> "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes." [89]

The in that way derived frameworks are Apache Tapestry [175], Google Web Toolkit [67], Spring [164], CodeIgniter [52], CakePHP [29], Kohana [98], ASP.NET [113] (Web Forms [115], MVC [114], and Web Pages [116]), and Ruby on Rails [41]. At the time of publication, Django [46] reached considerable popularity such that we quickly go into Django as well.

The testing procedure included first a check of the manuals on hints concerning control-flow integrity means. More precisely, we looked for existing functionality that can be configured, e.g. by providing a policy, and then enforces control-flow integrity features. The customer should not be required to implement but only configure enforcement. We compiled a chain of basic web pages that are connected via links and buttons and supplied a control-flow integrity policy whenever an enforcement feature is mentioned. Next, we tried to overcome the intended control flow by crafting requests.

Then, we tested each framework for race condition protection means which are a crucial part of control-flow integrity (see Section 2.5.6). We crafted a web page that accepts user requests and expects a textual parameter. The content of this parameter is posted to a message board, and a message counter keeps track on the number of posts. We allowed a maximum of five messages. A small script quickly sent message requests to that page trying to post more messages than actually allowed.

Finally, we wanted to learn how the request parser behaves. Therefore, we changed the given HTTP GET and POST parameters to see whether there is any enforcement based on the data type or a constant value.

Enforcing Sequences of Actions

In this section, we describe our findings on control-flow integrity means in the top 10 web application frameworks (see above). Our first reference point is each framework's manual. In case of promising hints, we conducted our practical test run, a simple flow definition and violating requests.

An incoming request can cause a sequence of server-side operations in **Apache Tapestry** [**175**]. Every request is first handled by a master dispatcher which forwards the request to the respective processing and page rendering routines. These routines can trigger new events (*event bubbling*). The web application reaches a stable state when all events finished processing. However, there is no enforcement mechanism to control the sequence of user actions.

Google Web Toolkit [**67**] allows the developer to write Java code which is then translated to server-side Java classes and client-side JavaScript code by the GWT SDK (Software Development Kit). Most operations and all user interaction happen on the client side. The client-side code communicates with the web server using AJAX requests (*Asynchronous JavaScript and XML*) [123]. These requests are called *remote procedure calls* because they call procedures on the server side. There is no enforcement mechanism concerning the sequence of processed requests.

Spring [**164**] is actually a modular Java framework. It becomes a web application framework by including the web module. In that combination, Spring implements a model-view-controller (MVC) architecture without any control-flow integrity protection. However, Spring is extensible by so-called projects[16] among which Spring Web Flow [163] is meant to provide flow control for web applications. It inserts a special web flow controller into the MVC-based application in order to ensure that every incoming request can be checked for policy compliance. Developers can define intended control flows as XML or as Java code. A control-flow definition contains a number of states and for each state its outgoing transitions. Processed requests trigger a state transition if they contain the respective *flowExecutionKey* and *eventID*. The *flowExecutionKey* denotes the access key to the control flow while the *eventID* is the transition's identifier. Both are transmitted as HTTP parameters. This allows Spring Web Flow to distinguish between tabs and, thus, allow multiple control flows in separate browser tabs without interference. It can also control side effects caused by the usage of the browser's "Back" button in such a way that it prevents accidental re-execution of the last action (see Section 2.5.10). In our practical test runs, we made sure that the flow definition was properly enforced. We crafted requests to all existing actions but no spoofed request was processed.

CodeIgniter [**52**] is a PHP-based web application framework implementing an MVC architecture. A dispatcher receives all incoming requests and forwards them to their respective controller. A file named `routes.php` does the assignment of requests to controllers. The included *security library*[17] processes all incoming requests and outgoing responses after the dispatcher and before the controller. However, it only sanitizes user

[16] See `http://www.springsource.org/projects` for a complete list.
[17] See `http://ellislab.com/codeigniter/user-guide/libraries/security.html` for details.

input to prevent XSS and equips links in outgoing responses with nonces to prevent CSRF. A control-flow integrity enforcement mechanism is not part of the framework.

CakePHP [29] like CodeIgniter is a PHP-based web application framework implementing an MVC architecture. The basic request processing is also similar: A dispatcher forwards all incoming requests to controllers according to the configuration file `routes.php`. CakePHP comes with a *security component*[18] that can be used by controllers to prevent CSRF and form tampering, require given HTTP methods (i.e. GET, POST, PUT, and DELETE) or SSL, or restrict the communication between controllers. None of these features, however, allows the enforcement of control-flow integrity properties.

Kohana [98] also falls into the category of PHP-based frameworks that implement an MVC architecture. The central configuration file is named `Bootstrap.php`. It gathers the basic configuration, lists included modules which provide additional functionality, and defines responsible controllers based on the requested URL. The supplied *security class*[19] offers protection routines against XSS, SQL injection, and to check input conformity. Control-flow integrity protection is not offered.

ASP.NET [113] is a web application framework built on the .NET framework for Windows operating systems. It allows to implement web applications in the programming languages C# and VB.NET. ASP.NET comprises three distinct application paradigms:

- ASP.NET Web Forms [115] generates web applications that consist of objects called *pages*. Pages contain HTML code and server-side controls. Those controls are triggered on incoming requests and perform data processing before a response is rendered and sent back to the client. The provided *state management*[20] offers data storage options across request-response round trips, similar to cookies and session records. There is no control concerning state transitions.

- ASP.NET MVC [114] again follows the model-view-controller architecture. The central dispatcher is named `Global.asax`. It assigns incoming requests to their respective controllers. An *authorization filter*[21] can be executed before the request is processed by the assigned controller. This filter checks a user's access rights to the requested action but does not control the sequence of actions.

- ASP.NET Web Pages [116] is the most lightweight web application framework of the ASP.NET family. Its application model is similar to Web Forms. Web Pages contain more HTML code enriched by dynamic server-side features while Web Forms generate most HTML elements dynamically. From a control-flow integrity point of view, there is no big difference between both.

[18] See `http://book.cakephp.org/2.0/en/core-libraries/components/security-component.html` for details.
[19] See `http://kohanaframework.org/3.3/guide/kohana/security` for details.
[20] See `http://msdn.microsoft.com/en-us/library/75x4ha6s.aspx` for details.
[21] See `http://msdn.microsoft.com/en-us/library/dd505057(v=vs.98).aspx` for details.

With **Ruby on Rails** [41], a developer implements model-view-controller-based web applications in Ruby. The underlying principle is equivalent to the above described MVC-based web application frameworks: The *action dispatch* component forwards requests to controllers based on a given configuration file, named `routes.rb`. Filters can be applied before and after the execution of the controller. However, there is no given control-flow integrity protection mechanism.

Django [46] is also MVC-based and uses regular expressions to assign requests to views. The request can be checked by *middleware* components before and after being processed by the view.

In summary, it can be stated that Spring with Web Flow offers the only control-flow integrity protection feature in the field of common web application frameworks. Common security features are anti-CSRF tokens, authorization management, and input validation against XSS and SQL injection. It seems to us that control-flow integrity has not yet received much attention and is overlooked in web application development.

Race Condition Protection

Section 2.5.6 shows that race conditions can be a severe problem in web applications. Roughly speaking, they occur whenever some action can be executed next but only a limited number of times. This is usually the case for repetition-bounded state changing actions. It depends on the application's business logic which actions are concerned. So, a web application framework should offer means to define such actions and respective requests in order to make the framework process them sequentially instead of parallel. We could endorse the results given above that none of the frameworks offers such protection with the exception of Spring Web Flow which we will take a deeper look at in this section.

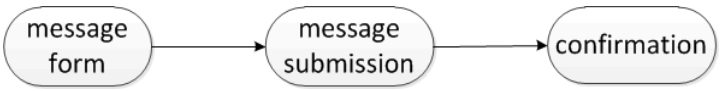

Figure 8.1: The intended flow for sending a message. First, the message text is entered. Next, the message is transmitted, and finally, a confirmation is given.

We implemented a number of web pages that allow the user to first enter a message text. Then, the message is sent via HTTP POST to the message board and a confirmation is given in the last step. The intended flow is given in Figure 8.1. We crafted the respective Spring Web Flow policy. Listing 8.1 shows the pseudo code of the method that receives the request.

```
if db.sentMessages < 5 {
  board.includeMessage(m);
  db.update(sentMessages++);
}
```

Listing 8.1: Pseudo code of the message processing method

The goal was to send a high number of messages and make more than five accepted for the message board. In a first attempt, we requested the message form, learned the request target and parameters for the message submission and sent 20 messages almost in parallel. The result shows that only one of the messages was accepted. It seemed that the *flowExecutionKey* and *eventID* were checked before the actual application code handled the request.

In a next attempt, we started the same flow in ten distinct browser tabs, thus obtaining ten different *flowExecutionKey*s, as Web Flow is able to handle multi-tabbed browsing. We were able to sent eight messages upon virtually clicking "Send" simultaneously in all ten tabs. Just for the record, we repeated the last experiment using ten different browsers instead of browser tabs and succeeded again. The difference between the last two configurations from the server's point of view is that all ten requests belong to the same user session in the first case and to ten different user sessions in the second case. In both scenarios, the actual flow definition was not violated because all steps were performed in the right order and there were no interfering requests within each single flow. The actual exploit happened on a logical level. The number of parallel executions of the same control flow within the same session or the same user account was not limited. There is no policy statement to define such restrictions. So, developers need to take care and implement customized solutions.

Parameter Enforcement

Next, we checked whether changes of the expected data type in request parameters lead to faults in web applications. For instance, we sent a request `http://www.example.com/controller/action/foo` while the application expected a numerical parameter, e.g. `http://www.example.com/controller/action/13`.

Our observation shows that the underlying programming language plays a decisive role: The Java-based frameworks fail while casting the unexpected string type to the integer variable. Apache Tapestry can not find an appropriate handler for our request and responded with a default page. Google Web Toolkit and Spring (incl. Web Flow) raise exceptions, `undeclared` and `NoMatchingTransitionException` respectively. The type-safe nature of Java in this case prohibits unintended user input, albeit the request is processed in the opposite case: A method expecting a string also accepted a number which is then, however, interpreted as a string.

The situation is different for PHP-based frameworks, because PHP does not have inherent type safety. The web application frameworks, however, all offer type matching expressions. CodeIgniter knows types `:num` and `:any` which include numerical values and all values respectively. CakePHP and Kohana suggest to enforce data types by means of regular expressions. The expression `'param' => '[0-9]+'` makes sure that only integers are accepted for parameter `param`.

There is another problem for ASP.NET web applications because they can be implemented in C# or VB.NET, thus not benefit from underlying data types. The attempt to maintain type safety is similar to the PHP world. So-called *constraints* can define regular expressions. The integer definition looks like the following: `param = @ "\d +"`

where `d` is the symbol for a digit.

Ruby on Rails also accepts *constraints*, i.e. regular expressions defining the range of accepted values for parameters. The integer definition is `:param => /[0-9]+/`

Finally, Django assigns requests to views based on regular expressions, i.e. requests with forged parameters can be sorted out before they are processed.

We can conclude that web application frameworks contribute to type safety in web applications. This makes those attacks harder which rely on request processing weaknesses based on parameter type manipulation.

Summary

Our tests show that the support for control-flow integrity in web application frameworks is insufficient. Existing approaches relying on implicit control-flow enforcement are dangerous for several reasons: First, modules are per se not reusable because their control-flow settings do not apply in a new context. Second, setting values to indicate that some action has been performed can have side effects allowing also subsequent actions of the same workflow or the repeated execution of the next action. Finally, authorization must always be distributed because the permission is given in one method while the check is performed in a different method. The need for framework-inherent control-flow integrity can only be fulfilled by Spring Web Flow (see Table 8.1).

Framework	Version	CFI	RC	Param.	Language
Apache Tapestry	5	–	–	+	Java
Google Web Toolkit	2.5	–	–	+	Java
Spring/Web Flow	3.2.2/2.3.0	–/+	–/≈	+	Java
CodeIgniter	2.1.3	–	–	+	PHP
CakePHP	2.3.0	–	–	+	PHP
Kohana	3.3.0	–	–	+	PHP
ASP.NET Web Forms	4.5	–	–	+	C#, VB.NET
ASP.NET MVC	4	–	–	+	C#, VB.NET
ASP.NET Web Pages	2	–	–	+	C#, VB.NET
Ruby on Rails	1.9.3	–	–	+	Ruby
Django	1.5.1	–	–	+	Python

Table 8.1: The results of our survey on control-flow integrity enforcement in modern web application frameworks. A plus (+) denotes that the protection feature is provided in the framework. A minus (–) means that there is no regular support for such protection. CFI is the property to enforce the right order in request processing. RC stands for race condition protection. Param. is the ability to ensure type safety of received request parameters. The Spring Web Flow race condition protection is a special case because it can only protect against single-flow race conditions.

Nevertheless, almost all frameworks in scope provide suitable execution points to hook

into. The central dispatchers of the MVC-based frameworks can observe every request passing by. Equipping those dispatchers with a control-flow integrity feature seems natural. Moreover, most of the frameworks have filters, that are executed before and after the controller processes the request. Table 8.2 gives a list of dispatchers and filters.

Framework	Dispatcher	Filters
Apache Tapestry	Master Dipatcher	–
Google Web Toolkit	Web.xml	–
CodeIgniter	routes.php	pre_controller, post_controller
CakePHP	routes.php	beforeFilter, afterFilter
Kohana	Bootstrap.php	before, after
ASP.NET Web Forms	Global.asax	–
ASP.NET MVC	Global.asax	OnActionExecuting, OnActionExecuted
ASP.NET Web Pages	Global.asax	–
Ruby on Rails	ActionDispatch	beforeFilter, afterFilter
Django	URLconf	Middleware

Table 8.2: The frameworks have single points of processing determined by their design, so-called dispatchers. Some even provide filter routines that are executed before and after request processing.

8.3.2 Summary

Our findings on the current support for control-flow integrity in the most prevalent web application frameworks show that this problem does not yet receive the attention it deserves. All frameworks but Spring with the Web Flow project lack related properties. No framework provides race condition protection features beyond single-flow request sequences. Only the type safety of received HTTP parameters is commonly supported.

8.4 Enforcing Control-flow Integrity in Web Application Frameworks

In the last section, we explained that the integration of a control-flow integrity enforcement module in modern web application frameworks is reasonable and necessary to prevent all related attacks. However, we found no sufficient support in the most prevalent frameworks. For this reason, this section presents a novel approach to enforce control-flow integrity on the framework level.

The specific contributions are as follows:

- First, we define a formal language for specifying explicitly the control flow of a web application.

- Second, we define a control mechanism that makes sure that only client requests that comply with the control-flow specification are executed.
- Third, we integrate the control mechanism in a framework based on the model-view-controller (MVC) model, making our approach both easy to use for newly developed applications and easy to integrate in already existing applications.
- Finally, we show that our approach is effective and practical by demonstrating that it enables the removal of several kinds of real-world security problems, while having a low run-time overhead.

8.4.1 Preserving Control-flow Integrity

Race condition exploits (see Section 2.5.6), HTTP parameter manipulation attacks (see Section 2.5.7), unsolicited request sequences (see Section 2.5.8), and compromising back button usage (see Section 2.5.10) are accomplished by user actions that violate given control flows. This section provides detailed information of how we prevent an unintended action from getting executed and, thus, from violating the integrity of a control flow. Roughly speaking, our approach checks every received request against a control-flow policy in order to determine whether it is part of a legitimate workflow.

Technical Background

For every web application, the application developer knows the intended control flow. This control flow can be denoted as a sequence of actions. Considering each action as a transition in a graph, we finally obtain the control-flow graph of the web application. So, the application developer deploys the control-flow graph of the web application.

The enforcement of the intended control flow requires a central entity that takes care of each incoming request. The popular MVC architecture provides such an entity by design (see Figure 8.2). Every request has to pass the application's controller, which encapsulates the business logic. The controller consists of several classes, each containing various methods. Therefore, one action of a control flow in our definition language is defined as `<class name>.<method name>`. From a granularity view, this is appropriate because a request addresses one method. In sum, a control-flow graph is given as a sequence of methods of controller classes.

Protection Goals

Our approach protects web applications from malicious users that perform attacks using arbitrary request sequences. As a side effect, the approach protects honest users against CSRF attacks to some extent because attackers have to follow the intended control flow to finally commit their abusive request. In more detail, our approach has the following goals:

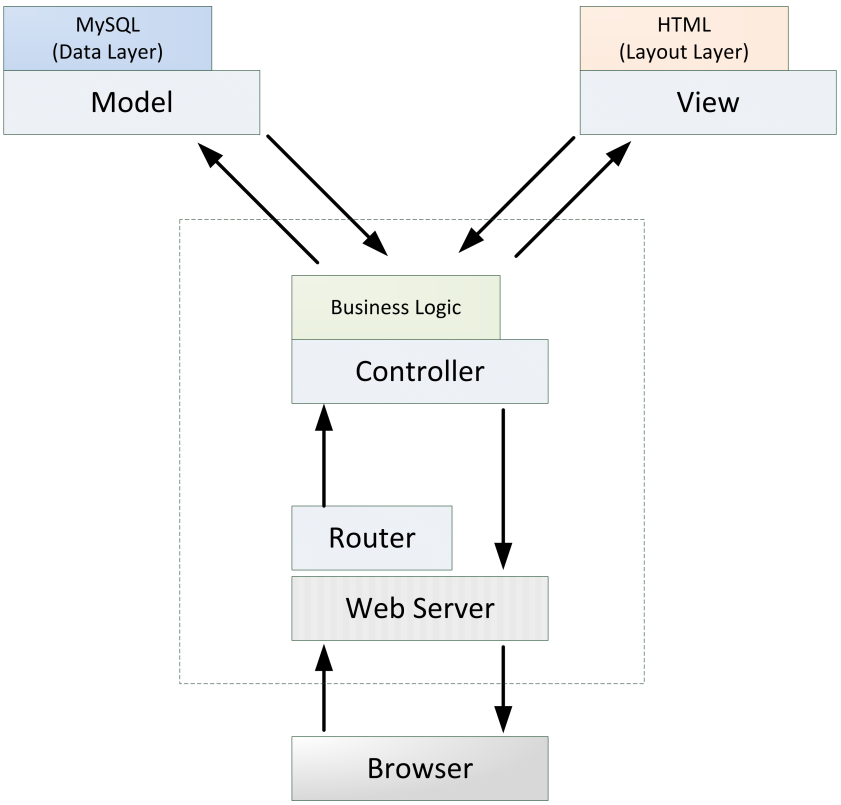

Figure 8.2: The design pattern of MVC-based web applications.

- *Request coverage:* Upon each incoming request, the approach shall determine the control-flow context of this request and take a decision whether the request is permitted. It updates the context accordingly if the request is allowed to pass.
- *Feature support:* The approach must be usable with state-of-the-art browser features, including the use of a back button as well as multiple browser tabs (multi-tabbing) for the same session. Each tab shall be permitted to use a different control flow.
- *Race condition protection:* The approach must prevent race conditions for actions that might serve as a target for an attacker. These actions can be specified in the control-flow graph.
- *HTTP parameter check:* The approach must be able to control HTTP parameters and their values.
- *Uncritical resources:* All web applications have unclassified resources such as the "About us" information. These resources shall be accessible without restrictions, independently of ongoing workflows.

Enforcing Control-flow Integrity

In this section, we provide details how we achieve the above mentioned goals. We propose an architecture based on an explicit control-flow specification and server-side enforcement. Our approach, a control-flow monitor, combines several mechanisms to enforce control-flow integrity. We show that all user interactions are intercepted and checked. Besides simple checks that sequences of requests are compatible with sequences in the control-flow graph, several situations require dedicated treatment, as explained in the following.

Back-button Support A widespread feature of modern browsers is the back button that allows the user to view the last web page again. As users are used to click that button whenever they feel like revisiting the last page, we implemented support for this step in our monitor. Therefore, the control-flow monitor records the trace of steps of the user. A request is considered a step backwards if it addresses the last method and this method is not meant as a next step in the control-flow graph. However, the control-flow monitor by default prohibits the backwards traversal due to the issues described in Section 2.5.10. Instead, the usage of the back button has to be allowed in the control-flow graph for each step.

Multi-tabbing Support Modern web browsers usually allow several tabs in the same window. As these tabs share the client-side data, e.g. cookies [102], across all instances, they are hardly distinguishable from the server side. Hence, without multi-tabbing support, actions in one tab would violate the control flow in another. In order to overcome this drawback, the control-flow monitor inserts client-side identifiers for different tabs to tell them apart. This way, each tab can be treated individually though logged in at the same web application.

Race Condition Prevention The monitor prevents the exploitation of race condition vulnerabilities (see Section 2.5.6) by disabling the parallel execution of susceptible actions. In general, these are actions that add, update, or delete data after reading. We achieve this goal with a locking mechanism. The control-flow monitor creates a temporary lock named by the session ID of the user. This means race condition protection on session level. Moreover, protection on control-flow and user level is possible by using a control-flow ID and the user ID respectively. Even a system-wide protection can be implemented using one unique ID file for all users.

Parameter Validation The client-side manipulation of HTTP parameters can lead to unintended application states (see Section 2.5.7). Thus, request parameters have to be checked for validity on the server side before they are processed. Instead of leaving this task to each method, the control-flow monitor provides means to centrally enforce given parameter properties. First, the data type of each parameter can be defined. As a side effect, this feature also mitigates *injection attacks (XSS, SQL injection)* that

need to transmit control characters. Second, parameters can be marked as "write once read many" (WORM). This allows to set the parameter's value once but not change it afterwards, meaning that this value is immutable for the rest of the session. This provides an invariant guarantee to the web application. One use case is the user ID that is supposed to not change during a session. Third, parameter names can be excluded for given workflows. This feature can protect web applications from unintended data manipulation. For instance, it prevents the setting of control flow-invariant parameters.

Definition of Uncritical Methods All web applications contain uncritical methods. Accessing these methods does not harm the application's control-flow integrity. For instance, a chat function can be allowed beside the enforced workflow. Similarly, AJAX calls that update the user's view but do not change the application's state can also be allowed.

Control-flow Definition In this section, we provide details on the syntax of the control-flow graph definition language. The following clauses and operators can be combined recursively.

- `Method1` → `Method2` — After accessing `Method1`, the user is allowed to access `Method2`.
- `(Method1|Method2)` — The user is allowed to access `Method1` or `Method2` in the first place, but she is not allowed to change her decision after clicking the back button.
- `(&Method1|&Method2)` — Like above but the user is allowed to change her decision after clicking the back button – denoted by the `&` symbol.
- `@Method{x}` — The user is allowed to access `Method` repeatedly. It is possible to define a maximum number `x` of allowed executions.
- `?Method` — The back button support for `Method` is enabled, i.e. the user can navigate one step backwards after having called this method.
- `!Method` — The race condition protection is active for this method. As long as this method is executed, no other protected method is executed in the context of the same session, user account, or system-wide (see above).
- `Method[+par1=type1,*par2=type2]` — Only parameters `par1` and `par2` are allowed for `Method` where they can be sent via POST (`+`) or GET (`*`) and have data types `type1` and `type2` respectively. Predefined data types include `bool`, `numeric`, and `string`. A policy for the whole control flow can be set by `addParameterTypeGlobal("*par=type")`.
- `addForbiddenParameters("par")` — Parameter `par` must not occur in the whole control flow.

- `addParametersGlobal("par")` — Parameter `par` can be set once but is immutable afterwards.

The nesting of clauses allows for defining complex control-flow policies. We provide simple examples below and a more sophisticated case in Section 8.4.2.

The Implementation

Several challenges need to be addressed in order to implement our control-flow monitor. Most importantly, the monitor has to be integrated into an application framework, which can be a complex task especially for existing applications. In addition, handling race conditions and multi-tabbing also deserve more detailed attention.

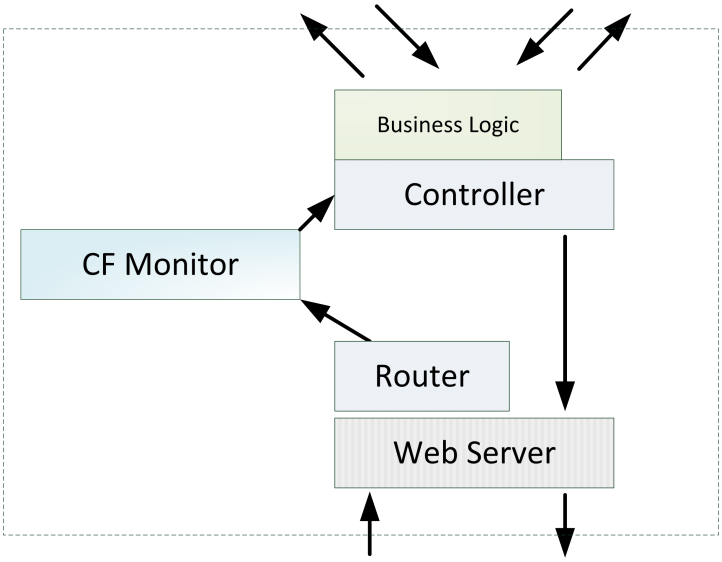

Figure 8.3: Our implemented modification of the design pattern of MVC-based web applications w.r.t. Figure 8.2.

Integration into Web Applications We implemented our control-flow monitor as a PHP module. It is run by the router (see Figure 8.3) before the controller class is called. This strategic position makes sure that, first, all requests have to pass our control-flow monitor before being processed by the web application and, second, the monitor is easy to integrate into existing web applications.

As a proof of concept, we integrated the monitor into a web application that is based on the CodeIgniter framework [52]. In fact, the only change on an existing web application affects the one line of code that calls the responsible controller. This line has to be slightly modified to include our monitor (see Listing 8.2).

We use *Aspect Oriented Programming (AOP)* to inject the control-flow monitor as a processing step into the call sequence of all controllers. This allows the developer to apply changes on the application the same way as if there were no control-flow monitor.

```
include(APPPATH.'controllers/'.$RTR->fetch_directory().$RTR->fetch_class()
    .'.php');
//must be changed to
AOP::process(APPPATH.'controllers/'.$RTR->fetch_directory().$RTR->
    fetch_class().'.php',
  $_SESSION['''atom_parentFramework''']->getCacheFolderName());
```

Listing 8.2: Dynamic Inclusion of Controller Classes in the CodeIgniter Framework [52]

Multi-tabbing Support As explained before, multi-tabbing requires the unique identification of tabs. This identification is implemented in JavaScript. Moreover, a tab handler is implemented on the server side as part of the control-flow monitor. The JavaScript code triggers an AJAX message whenever a tab is opened, closed, or a tab switch is performed by the user. A tab switch message by the client makes the tab handler change the tab context on the server side. When the user opens a new tab by clicking on the "open link in new tab" option in the browser, this tab is assigned a session-unique identifier. We use the `window.name` property of the `window` DOM object to store the identifier. An AJAX request transmits the new identifier to the tab handler. The new tab is assigned the advanced position in the control-flow graph while the first tab holds the former position. Both tabs then run the same control flow, however, it is enforced individually, i.e. a control-flow violation in one tab has no effect on the other tab as the respective tab record is duplicated when the new tab is opened.

The control-flow monitor stores flow-related information per tab, i.e. the active control-flow graph that is currently enforced in this tab and the respective position in the graph. The user's session ID and other high-level information is still stored in the session record. This allows, for instance, to consider several products in different tabs, then add some of them in the same shopping cart and finally check out in one tab that starts the checkout control flow.

An attacker stripping or manipulating the embedded tracking code can not trick the system to gain advantages. The code only signals the current tab to the web application. A manipulation would cause the web application to assign the next request to a different tab. This, however, is equivalent to perform the request in the respective tab. The intended action is only executed if the request is allowed there. Then, however, the attacker has not increased his scope of action. In all other cases, the manipulation leads to voiding the current control flow.

Race Condition Prevention Whenever a protected method is executed, the control-flow monitor tries to create a file with the current session ID. If this creation fails due to an existing file with the same name, the request is not processed and an error page is displayed. After processing the protected method, the file lock is released again. This allows the next protected method to be executed.

The race condition protection mechanism does not prevent the processing of unprotected methods, e.g. in a different tab. The fine granularity of the locking makes sure that a single locked method has no impact on the usability of other sessions of the user

```
SMS.showForm -> !SMS.validateAndSendForm
```

Listing 8.3: Control-flow Definition to Protect SMS Delivery from Race Conditions

```
Checkout.logIn
-> Payment.chooseMethod
-> Payment.validateStatus
-> Checkout.completeOrder
```

Listing 8.4: Control-flow Definition to Prevent Adding Items after Checkout

or the interactions of other users.

Examples

In this section, we show the usage of our control-flow definition language. We give examples with respect to the real-world scenarios in Sections 2.5.6 and 2.5.8 but assume a simplified technical implementation to keep the control flows simple and clear. We give details on the application of our control-flow monitor in the context of the Amazon checkout process in Section 8.4.2.

Preventing Race Conditions in SMS Delivery In the first example [138], attackers managed to bypass the delivery limit of an SMS portal by exploiting a race condition vulnerability. We assume the following control flow to send an SMS: First, the user requests the SMS input form. Then, after entering all necessary information, the user submits the form. The related control-flow definition ensures that, first, the input form has to be accessed before the submission, and, second, the submission must be protected against race condition attacks, see Listing 8.3.

The control-flow definition allows access to the method `validateAndSendForm` only after requesting `showForm`. This prevents the attacker from sending the message information directly to the delivery gateway. Of course, a capable attacker might send the requests to the `showForm` method in an automated fashion. However, as the `validateAndSendForm` method is protected against race condition attempts, e.g. on the user level, the attacker's requests will only be processed sequentially. This avoids sending more messages than actually allowed.

Prevent Adding Items to the Shopping Cart Between Checkout and Payment A more complex example is given by Wang et al. [190]. After requesting the checkout, the user was able to add more items to her shopping cart. These items were not charged. In order to prevent this sequence of requests, the checkout workflow has to be properly defined. The method that adds goods to the cart must not be accessible during this workflow. The respective control-flow definition is given in Listing 8.4.

After the authentication, i.e. login, the user chooses her favorite payment method and is redirected to a payment service provider. The actions on the payment service

```
1 Login.index
2 -> (Address.chooseExisting|Address.addNew)
3 -> Shipping.preferences
4 -> ((Payment.chooseExisting
  |Payment.addNewCreditCard)
  |Payment.addNewDebitCard)
5 -> (Billing.chooseExisting|Billing.addNew)
6 -> Order.placeOrder
```

Listing 8.5: Definition of Amazon's Checkout Control Flow

provider's site are not part of the definition because they happen on a different domain that is not controlled by the same control-flow monitor. The next request within the scope of the definition is the payment status validation after the user's return. Finally, the order is completed, the goods are shipped to the user, and the cart is reset. During the whole process, no addition of items to the cart is granted.

8.4.2 Discussion & Evaluation

In this section, we discuss the properties of our control-flow monitor. We show that it produces a negligible overhead and evaluate the protection goals defined in Section 8.4.1. Finally, we explain its possible application scenarios and limits.

Performance Evaluation

As described in Section 8.4.1, the control-flow monitor is applied between the router and the controller of the web application. It examines the received HTTP request with respect to the requested method and the parameters and checks these against a given policy. This application overhead is independent of the web application's execution time. The delay relates to the complexity of the given policy, though.

We used Xdebug (version 2.1.2) [196] to determine the control-flow monitor's overhead in a virtual machine with Debian 6 as the operating system and Apache2 as the web server with PHP 5.5.3.3-7 on an Intel Core-i7-2600 (Intel-VT activated) with 3.4 GHz and 2 GB RAM. For evaluation purposes, we implemented the checkout process of Amazon. Therefore, we analyzed the control flow on `amazon.com` by hand and derived the control-flow definition given in Listing 8.5. Note that the controller and method names are simplified for readability reasons. The control-flow definition does not allow usage of the back button because Amazon prohibits it, too.

We measured the runtime overhead ten times and computed the average for each step in the control flow (see Table 8.3). The respective graph shows a peak in the fourth state due to the triple branching (see Figure 8.4). Branches, namely alternative paths through the control flow, cause most of the overhead, the earlier a branch occurs the bigger its overhead. This is the reason why step 2 causes more overhead than step 5. We assume that some overhead can be saved by a more efficient policy parsing algorithm. Overall, the induced delay ranges between 8.9 and 9.6 milliseconds per request. We consider

	Runs										
Step	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	avg
1	8.9	8.2	8.4	10.2	11.0	8.7	8.2	9.4	8.3	7.7	8.9
2	10.2	9.9	9.3	9.8	10.1	9.8	9.0	8.2	9.5	9.1	9.5
3	10.1	9.2	10.9	8.6	9.6	8.2	9.5	9.0	9.0	8.4	9.2
4	8.3	10.1	10.0	10.2	9.4	9.8	10.3	7.8	10.6	9.0	9.6
5	8.8	11.0	10.1	8.3	8.4	10.0	8.6	7.9	7.7	9.8	9.1
6	10.0	8.5	8.1	8.5	8.4	8.4	9.6	9.7	8.0	10.4	9.0

Table 8.3: The overhead caused by the control flow monitor in $[ms]$

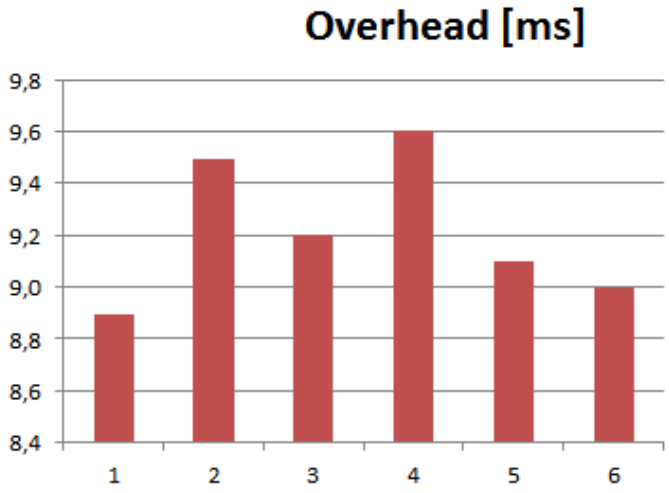

Figure 8.4: Induced overhead by the control-flow monitor in order to protect the Amazon checkout process.

this an acceptable effort with respect to the security gain. In order to determine the monitor's scalability to several user sessions, we set up 100 parallel user sessions and repeated the measurement. While the overall response time increased, we found out that there is no measurable difference to the scenario with only one user in terms of the monitor's overhead.

There is a one-time overhead for the generation of the temporary controller class file (see Section 8.4.1). This overhead occurs once whenever a new controller class is added or an existing class is modified. The first call on this class takes 60% to 90% more time than the subsequent calls. For usability concerns, this overhead can be neglected because the web application provider could easily initiate an appropriate request, thus, preventing all users from facing the mentioned delay.

```
Mage_Checkout_CartController.indexAction ->
Mage_Checkout_OnepageController.indexAction ->
Mage_Checkout_OnepageController.saveBillingAction ->
Mage_Checkout_OnepageController.saveShippingAction ->
Mage_Checkout_OnepageController.saveShippingMethodAction ->
Mage_Checkout_OnepageController.savePaymentAction ->
Mage_Checkout_OnepageController.reviewAction ->
Mage_Checkout_OnepageController.successAction
```

Listing 8.6: Control-Flow Definition of the Magento [109] Online Shop

Discussion

In this section, we evaluate our findings with respect to the protective goals defined in Section 8.4.1. We have to note that the monitor is responsible for control-flow integrity while other tasks like session management and user authentication are handled by the framework in place.

Every incoming HTTP request has to pass the router in the assumed MVC architecture. So, all requests are finally processed by our control-flow monitor. Our security evaluation showed that in fact all requests are treated by the monitor and accepted or rejected appropriately. The control-flow monitor achieves complete protection against maliciously crafted requests as well as erroneous navigation attempts.

However, the protection level depends on the sound definition of control flows. The definition has to be provided by the application developer. The implications from this fact are twofold. First, the definition requires a deep knowledge of the web application and its methods. The knowledge and understanding of the web application must already exist to implement and maintain the web application. This allows developers to provide accurate control-flow policies. So, we consider this a feasible task for an expert. Second, the necessary policy-definition efforts stay within reasonable bounds. The `class.method`-based policy language abstracts from the implementation of functional modules but is still close to the web application's architecture. We crafted the control-flow policy for the checkout of the open source shop Magento [109] (see Listing 8.6) in order to estimate the complexity in real-world use cases, and find that the policy definition is even feasible for us who are not the developers of Magento.

Our control-flow monitor provides multi-tabbing and back-button support, thus proves usable with modern browser features. This increases the usability and ensures acceptance by the end users. This way, security is not achieved at the expense of a limited user experience.

To the best of our knowledge, our approach is the first to effectively protect against race condition exploits. The control-flow monitor allows the flexible definition of the protection level, ranging from control flow-based over user-level up to system-wide protection.

Policies on HTTP parameters can be defined including both GET and POST parameters. Policy rules can apply in terms of the data type, the limitation on a single value assignment, and the exclusion of parameters for given workflows. Our parameter control

means are suitable to prevent the attacks described in Section 2.5.7.

The definition of uncritical methods allows the monitor to focus on a comprehensible set of relevant method calls. For instance, there can be unhindered access to pictures because they are not part of the business logic. Confidential data can be protected by access control means. AJAX requests can be divided into state-changing and other requests. The state-changing requests can be covered by the control-flow definition, the others are excluded and pass the control-flow monitor. As AJAX requests also call server-side methods, their control-flow definition is straightforward with respect to the web application's control flow.

Our approach is easily applicable at the development phase though one of its most advantageous features is its usability with legacy web applications. We implemented a PHP-based proof of concept. Nevertheless, a Java-based implementation can be achieved with acceptable effort, e.g. by a J2EE filter. Even non-MVC-based web applications can be equipped with the monitor. However, the integration causes more overhead if a central request handler is missing. Then, all calls on server-side actions have to be intercepted separately.

The control-flow monitor does not aim at replacing the web application's business logic. As a matter of fact, it provides reasonable and reliable guarantees concerning the sequence of requests and properties of provided parameters. The web application still has to make sure that user-generated content fits the expected information. For instance, a sequence of requests containing semantic garbage but matching the defined control flow will still succeed to finally request the intended method.

8.4.3 Summary

We explained the complex problem of control-flow vulnerabilities and showed its high practical relevance by real-world examples, i.e. existing vulnerabilities and attacks. We identified the root causes in the modular addressability of web applications together with the implicit and scattered definition of workflows. Our solution overcomes this problem by the explicit definition and enforcement of intended workflows. To the best of our knowledge, it is the first approach that covers the whole bandwidth of related vulnerabilities, including race conditions, HTTP parameter manipulation, unsolicited request sequences, and the compromising use of the back button. Moreover, it is the first approach that properly handles client-side features like back-button usage and multi-tabbing. We showed that this approach can prevent all described attacks and causes negligible overhead.

In sum, we provided a thorough approach that is applicable to existing and newly developed web applications and provides guarantees to the developer concerning the sequences of incoming requests as well as the format and values of parameters. This allows to separate web application semantics from control-flow integrity. As a side effect, the presented approach mitigates CSRF and injection (XSS, SQL injection) attacks.

We suppose that write access to the web application framework is granted. In multi-tenant environments, however, the underlying framework may be used but not adapted. In this case, an external approach is required which we will present in the next section.

8.5 Providing Ad-hoc Control-flow Integrity for Web Applications

This section presents an alternative approach for avoiding problems related to control-flow integrity in web applications. Based on the assumption that all client-side requests are potentially compromised and benign usage only includes mouse clicks and form input after visiting the site's entry page, the approach replicates a user's mouse clicks and form input in a server-side sandbox. Requests triggered by the sandbox are trustworthy because they adhere to the assumed user interaction with the web application. We show that this approach provides ad-hoc protection against attacks on the web application's control flow without the need for a learning phase or a manual policy definition. It functions as a reverse proxy and is thus independent of the server-side technology. No adaptations are required on the web application making the approach applicable to new and legacy applications. The induced load can be outsourced to scalable, e.g. cloud-based, platforms if necessary.

8.5.1 Preserving Control-flow Integrity Ad Hoc

In this section, we present *Ghostrail*, our ad-hoc approach to overcome race condition exploits (see Section 2.5.6), HTTP parameter manipulation including file inclusion attacks (see Section 2.5.7), unsolicited request sequences (see Section 2.5.8), and forceful browsing (see Section 2.5.9). We give details on the implementation in Section 8.5.2.

High-level Overview

The idea behind Ghostrail is the ad-hoc enforcement of control-flow integrity based on the developer's assumptions phrased in Section 8.2: Users first request one entry point of the web application, e.g. `www.example.com`, and then click on links and buttons or fill in forms. We assume that the attacker is a web user that controls all client-side data and applications within his domain. However, he can not bypass reverse proxies. He can send messages to the server but does not control the server-side platform.

Ghostrail operates as a traffic monitor on the server side. It protects a web application by filtering out incoming requests that are not generated by user clicks and form entries. In order to determine whether a request arises from regular interaction between the user and the current web page, Ghostrail analyzes the last web page delivered by the web application. A request is accepted if the web page contained the respective link. Otherwise – the page did not contain the requested URL – the request is considered crafted and, thus, possibly malicious because it violates the assumption that the user only interacts with the web application using clicks and filling in forms. Ghostrail has a three-tier whitelisting approach to derive regular requests:

- First, Ghostrail queries an application-wide whitelist of always allowed requests. The list contains the application's entry points, e.g. the start page, and possibly all requests to public resources that do not change the application's state.

- Second, Ghostrail parses the last web page delivered by the web application. It compiles a list of static references found in HTML and CSS documents. Those references denote hyperlinks, i.e., possible next user clicks, or embedded resources that are needed by the browser to render the web page, e.g. images. We give more details on static reference extraction below.

- Third, Ghostrail renders web pages in server-side sandboxes to determine dynamically generated requests, e.g. using AJAX and JavaScript. Those requests can not be determined by the static parser in step 2. Ghostrail accepts requests from the client side if the sandbox triggered the same request. We describe the sandbox-based request detection below.

Ghostrail lets only requests found in any of these three lists pass. This way, it enforces the assumption that users only interact with the web application using mouse clicks and form input. Due to the fact that the whitelists in step 2 and step 3 are compiled ad hoc, Ghostrail neither needs a pre-release learning phase to generate its control-flow policy nor a hand-crafted control-flow definition. However, as Ghostrail operates on automatically generated whitelists of references, every reference it fails to extract may degrade the usability of the web application. It is therefore crucial to extract as many references as possible. By extraction we mean the analysis, classification and storage of reference information that is embedded in content delivered by the web application. In the remainder of this section, we provide details on how Ghostrail extracts references statically and dynamically.

Extraction of Static References

In this section, we give details on how Ghostrail extracts static URLs from delivered web pages. Static references usually occur in HTML and CSS files. Other web resources like JavaScript and Flash, i.e., ActionScript, mainly utilize dynamic URL generation. They assemble the requests based on user input or the client-side state. We explain dynamic URL tracking in the next section. Finally, media files like images are ignored because they do not contain links for subsequent requests.

HTML is a tag-based language, i.e., the elements of a web page are described as tags. There is a limited number of HTML tags that may contain URLs: `<a>`, `<link>`, `<iframe>`, `<script>`, `<img>`, `<area>`, `<embed>`, `<form>`, `<base>`, and `<meta>`. Ghostrail parses these tags and extracts URLs found. However, not all HTML content is trustworthy. Web applications often allow users to provide own content, e.g. in the form of comments that are embedded in the HTML output. An attacker could easily abuse such a feature by posting the URLs he wants to request next. It is therefore possible and necessary to configure Ghostrail so that it excludes user-provided content from reference extraction within HTML documents. The second type of static content that may contain references is CSS data. As this is limited to only one syntax element (`url()`), an adequate regular expression performs the reference extraction.

There can be different URLs that are semantically identical, e.g. `http://example.org/?par1=foo&par2=bar` and `http://example.org/?par2=bar&par1=foo`. Ghostrail

normalizes URLs in order to prevent misclassification.

During reference extraction, Ghostrail tags whitelisted URLs either as a *transition* or as an *extension*. Transitions make the browser replace the current web page while extensions only update a part of the page, e.g. in an iframe or by an AJAX request, or load an additional page in a new browser window (or tab) without modifying the parent window. A transition invalidates all previous whitelisted URLs whereas an extension adds new URLs to the whitelist.

Replication of Client-side Execution

Beside the static references, dynamically generated requests play an important role within modern web applications. Applications that used to be installed and executed on a local machine, for instance office apps, move to the web and become accessible via a web browser. The synchronization of the client and server state as well as seamless interface updates require dynamic request generation and response processing, known as AJAX. The same is true for the search-as-you-type feature during user input. Such dynamically generated requests can not be determined using the static reference extraction described above because the static analysis of JavaScript code is fault-prone and requires manual code annotations [71]. Ghostrail, however, aims to protect web applications without the need to change the application code. Instead, we equipped Ghostrail with a server-side replica of the user's browser to track the execution of JavaScript and derive the respective requests. Ghostrail maintains one replica for each user session. Each replica runs in a sandbox and virtually performs the same actions that happen in the user's browser.

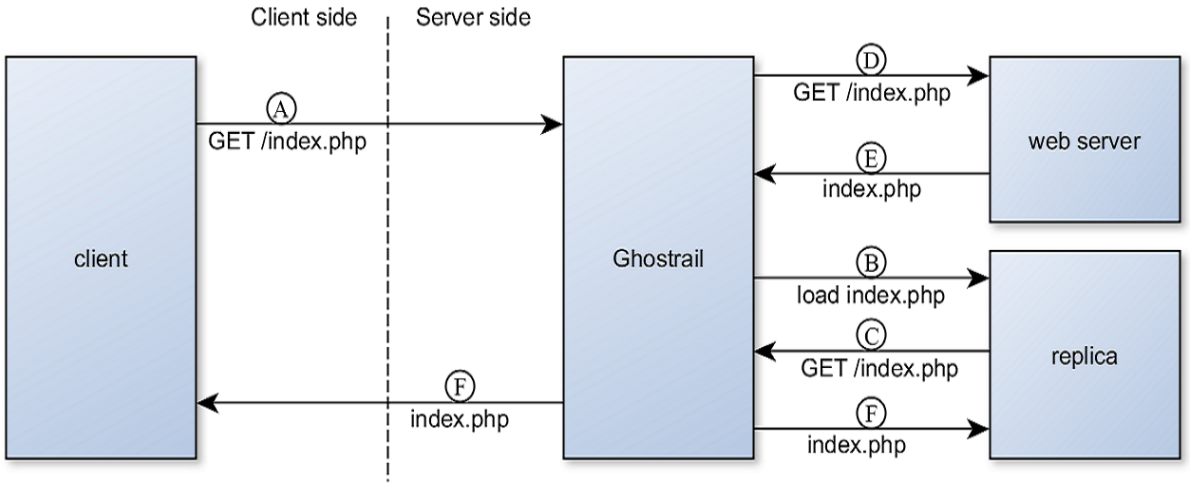

Figure 8.5: Initial loading of a web page in the sandbox.

In order to monitor a user's actions, Ghostrail injects a few lines of JavaScript code into every delivered web page. This code monitors all user actions that can trigger JavaScript events. This is necessary because JavaScript has an event-driven execution paradigm: Code is not executed linearly but triggered by user actions, timing, or state

changes of the web page. While timing and state changes also happen in the server-side replica without further ado, user actions must be transmitted to keep track. Interesting user actions include mouse movements, mouse clicks, and keystrokes.

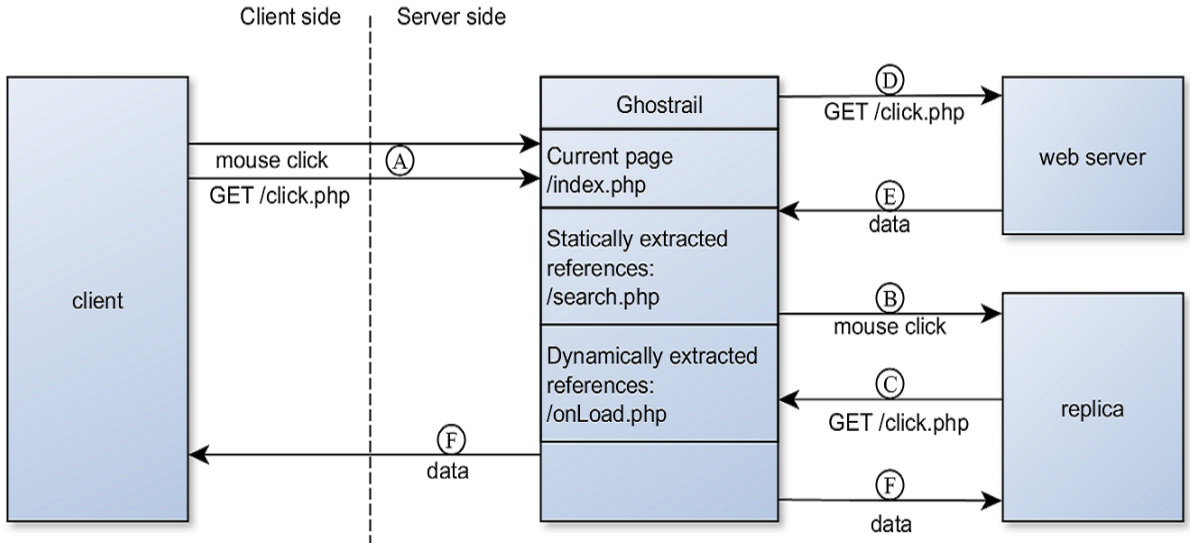

Figure 8.6: Dynamic reference extraction by replicating a user's action.

The server-side replica virtually renders the same web page as the user, it executes the same JavaScript code, and it simulates the user's mouse movements, clicks, and form input, i.e., only expected – thus benign – user actions. The requests from the replica are the condensed set of expected user requests. So, Ghostrail adds them to the user's whitelist. It is important to stress that Ghostrail only receives user actions from the client side but not the respective state change in the user's browser. This is a crucial point because it limits a malicious user's scope: Transmitting state changes allows an attacker to modify his browser such that it finally generates an attack request. For instance, a hash function may compute a URL parameter. The modified browser would always output `/etc/passwd`, independent of the input. Transmitting the output would allow the attacker to inject crafted requests into Ghostrail's whitelist. Limited to user actions, he can only spoof mouse clicks, movements and keystrokes on the web page. However, all these actions are within the scope of expected user interaction so there is no attack even if these actions are spoofed.

Figure 8.5 shows the initial loading of a web page in the sandbox. The replica is initiated with the start of the user session (steps A/B). After the page has been fetched from the web application (steps D/E), it is send as a response to the client and the replica (step F). The replica does not interact directly with the web application to prevent a double impact on the application state. Also, the duplication of the application's response (step F) ensures that both the client and the replica share the same content. After the page load, the user can interact with the web page. Figure 8.6 shows an example how Ghostrail classifies dynamic references. The user clicks on an element (step A). The details are transmitted to Ghostrail and forwarded to the replica (step B)

which simulates the same mouse click. The subsequent request to `click.php` (step C) is recorded by Ghostrail as legitimate because it is the result of intended user interaction. So, Ghostrail accepts and forwards the user request (step D).

8.5.2 Implementation

In this section, we describe the implementation of Ghostrail and the sandboxed replica. We implemented Ghostrail as a reverse proxy using Node.js (version 0.10.0) [91]. This reverse proxy manages all requests and responses and directs the replicas as well as the static reference parser. This design allows to outsource CPU- or memory-intensive processes to other machines. The reverse proxy buffers incoming requests, queries the three-tiered list of regular requests (see Section 8.5.1) and finally accepts or rejects the request. It forwards the web application's responses to the static reference parser and the respective replica for further analysis. In the remainder of this section, we focus on the implementation of the replicas and the handling of client-side data.

The Sandboxed Replica

Ghostrail initiates a fresh replica for each user session – and destroys replicas when the session ends. We implemented the replicas using PhantomJS [78], a fully-fledged, GUI-less, and WebKit-based browser. Due to the WebKit basis, the replicas support all major web technologies like JavaScript, AJAX, CSS, JSON and SVG. Replicas do not need to run on the same machine as Ghostrail. They can be distributed to cloud-based computing platforms to scale with varying load. The communication between the replicas and Ghostrail is based on HTTP and WebSockets.

Ghostrail injects a small piece of JavaScript code into every web page that is delivered to the user. This code establishes a WebSocket connection with Ghostrail to transmit user actions. Ghostrail records three kinds of user actions:

- **Mouse clicks:** Ghostrail records mouse clicks by injecting the `onclick` event handler for the whole web page. In order to simulate the click in the right page area, the (x, y) coordinates relative to the browser window are appended. Also, the dimensions of the browser window must be transmitted to configure the replica with the same size.

- **Mouse movements:** Mouse movements can trigger `onmouseover` events on a web page. Hence, Ghostrail records mouse movements above a configurable threshold using the `onmousemove` event handler. The threshold is necessary to avoid an overload of Ghostrail.

- **Key strokes:** Requests can contain user-defined data from an HTML form. Ghostrail must record every key stroke (using `onkeypress` and `onkeyup` event handlers) to simulate the user input. This is the only option to relate the resulting request to regular user behavior, i.e., entering data into form fields.

While Ghostrail injects new event handlers into the web page, there could be other event handlers that fire first and bypass Ghostrail's events. For instance, an event handler that redirects the browser to another page is executed first such that Ghostrail loses track and forbids future regular requests. We overcome this issue the following way: There are two options for the order of cascading event handlers, namely *event bubbling* and *event capturing* [82]. Event bubbling triggers the innermost event of the DOM tree first, i.e., for nested elements where each has an `onclick` event handler, the event of the inner element fires first when the user clicks. Event capturing has the reverse execution order. Ghostrail enforces event capturing and assigns its event handlers to the outermost element of the DOM tree, i.e., `document`.

Handling Browser Cache and History

Browsers cache web content locally in order to improve performance. Upon the next page access, they first query their local cache and restore the page without the need to request it again from the website. This, however, poses a problem to Ghostrail if it can not observe the local page load and extract the references from the cached page. We implemented a twofold cache management in Ghostrail to overcome this issue: First, Ghostrail adds the `Cache-Control: no-cache` HTTP header [60] to each response to prevent caching on the client side. More precisely, the browser may cache the respective content but must revalidate every usage with the server. However, we found that the Chrome and the Firefox browser still cache at least the last visited page and reuse it without revalidation. Second, the client-side code detects the click that loads the cached resource. Then, the replica performs the same click and loads the same content from the local cache provided by PhantomJS.

Beside the local cache, browsers also maintain a local browsing history. This history allows users to navigate back and forth. PhantomJS supports the browsing history since version 1.8 such that the replica can emulate the navigation through the browsing history.

8.5.3 Evaluation

We evaluate the security gain by Ghostrail, investigate a possible impact of Ghostrail on the protected web application's availability, and give results of our performance measurements.

The Security Gain by Ghostrail

In order to evaluate the security gain by Ghostrail, we first describe how Ghostrail protects web applications against race condition exploits, HTTP parameter manipulation including file inclusion attacks, unsolicited request sequences, and forceful browsing. Then, we explain our practical evaluation using the intentionally vulnerable web application Google Gruyere [66].

Race Conditions An attacker who exploits race conditions in web applications must send the same request many times in parallel. If the request is compiled dynamically, he must prepare the respective input in his browser and send the form using a mouse click to make the request be also sent by the replica and thus be whitelisted. In any case, as soon as Ghostrail accepts the attacker's first request, it immediately discards the whitelist of requests and waits for the application response to refill the list of expected requests. So, Ghostrail rejects the second request unless it is extracted from the subsequent page.

HTTP Parameter Manipulation HTTP parameter manipulation attacks rely on the user's ability to freely change the parameters of a given request, e.g. to change the given account ID or message ID. While an attacker can still craft arbitrary requests in his browser's address line, Ghostrail rejects all requests that do not match an extracted request from the current page.

Unsolicited Request Sequences Unsolicited request sequences occur if an attacker can assemble a request to call application functions when they are not supposed to be called. In the example given in Section 2.5.8, the attacker knows the request that adds items to his shopping cart. As Ghostrail discards previously allowed requests, the crafted request is not whitelisted after checkout and thus rejected.

Forceful Browsing A forceful browsing attacker also needs to craft a targeted request that is not part of the current web page. So, Ghostrail rejects forceful browsing attempts by design.

Case Study: Protection of Google Gruyere In order to evaluate Ghostrail's protection in practice, we set up Google Gruyere that is vulnerable to forceful browsing, file inclusion, and reflected cross-site scripting (XSS) attacks. With Ghostrail in place, none of the attacks on Gruyere worked. However, we want to emphasize that injection attacks like XSS and SQL injection are out of scope for Ghostrail. If the attacker enters the payload into a form field, Ghostrail regards the resulting request as benign. So, protected applications still need to sanitize user input from free text form fields. Nevertheless, Ghostrail limits possible user input via drop-down menus or radio buttons to the given options.

Ghostrail's Compatibility with SSL/TLS As Ghostrail plays the role of a traffic monitor, it must be the server-side endpoint of SSL connections with the client side. Given that it runs in the same domain as the protected web application, we do not consider this point a serious issue. If needed, the communication between Ghostrail and the web application, as well as between Ghostrail and the replicas, can be encrypted again.

The Impact of Ghostrail on the Availability of the Protected Web Application

We evaluated whether Ghostrail has a negative impact on the web application's availability. A negative side effect can occur if Ghostrail fails to extract a regular reference that is accessed by the user in the next step. In that case, Ghostrail mistakenly classifies the user request as unexpected (false negative). The evaluation is threefold: First, we set up a demo web application that implements a broad range of modern web technologies, i.e., redirects, CSS, jQuery as a representative JavaScript library, dynamic page updates via AJAX, and the navigation to dynamically generated URLs. This approach is meant to find out whether there are general compatibility issues of Ghostrail with any web technology. We used Selenium [156] to direct an instance of Firefox and Chromium respectively. Each browser performed virtually 1,000 user actions, resulting in 20,648 requests overall (each user action can trigger several requests). Afterwards, we analyzed Ghostrail's log files and did not find any blocked request. Please note that every blocked request would be a false negative because the virtual users only clicked on links or filled forms – what we defined as compliant behavior.

Second, we set up Ghostrail as a reverse proxy for the Alexa Top 20 websites. We used Selenium again to make Firefox perform 200 user actions on each website. Overall, we recorded 18,319 requests with a false rejection rate of 17.57%. Our analysis showed that blocked requests contained customized elements. Some websites perform a kind of client fingerprinting, i.e., they read browser and system features that differ for Firefox and the replica and add such information to the requests. The replica proved able to simulate the more common `User-Agent:` string for client classification. Another source for blocked requests are client-side timestamps and random numbers generated by JavaScript and appended to requests. In these cases, the outcome of the code execution differs for the browser and the replica. While we had to consider each web application as a black box, the application provider can configure Ghostrail more appropriately to avoid most of the false rejections, e.g. by adding a rule that ignores differing parameters if they match the expected pattern and if their processing may not cause harm. We avoid transmitting the random numbers and timestamps from the client side to the replica for synchronization because this would allow a malicious user to inject arbitrary HTTP parameters.

Third, we accessed three websites manually to learn the perceivable impact of Ghostrail. This is important because the raw number of blocked requests does not make a point concerning the impact on the web application's availability.

- *Google search:* The search function was usable without interference. Only the auto completion did not work due to differing request parameters.
- *Amazon:* We were able to search items, add them to our cart and checkout. However, we did not see product recommendations. The almost complete functionality of Amazon is particularly interesting because we experienced the highest number of falsely rejected requests ($\approx 60\%$). This result calls the significance of the raw number of false rejections into question.
- *Wikipedia:* We did not experience any issues on the availability.

We found that Ghostrail is able to allow workflows which span several domains. For instance, Ghostrail may protect an online shopping web application. When the user is redirected to a third-party cashier like PayPal, Ghostrail can not track the payment process (however, the cashier may run another instance of Ghostrail). Hence, the first request that leads the user back to the shopping application must be whitelisted as an application entry point.

Instead of redirecting, a web application may include third-party content in its own pages. Then, the replica fetches the same content but does not provide cookies nor authenticating HTTP parameters to avoid requests on behalf of the user.

Performance

Finally, we evaluated the performance impact of Ghostrail. We measured the HTTP round trip time between sending a request and receiving the response. We used the testbed with our demo application described above in order to avoid independent factors on the performance. The size of the served web pages ranges from 225KB to 450KB, and the round-trip overhead was between 250ms and 360ms. For applications with real-time requirements, e.g. online games, Ghostrail may only be an intermediate solution. For other applications, it is possible to scale the number of Ghostrail and replica instances with the load.

8.5.4 Summary

We identified the root causes of control-flow integrity compromises in the attacker's possibility to craft arbitrary requests at any time together with the developer's assumption that users only follow provided links. Ghostrail overcomes this problem by the ad-hoc generation of next-step policies. It is the first approach that neither needs a repeated training phase nor a manual policy definition and covers the whole bandwidth of related vulnerabilities, including race conditions, HTTP parameter manipulation, unsolicited request sequences, and forceful browsing. Ghostrail is compatible with all modern web technologies including mash-up's and JavaScript libraries while it is applicable to all new and legacy web applications without any changes on the application code. For high-traffic applications, the induced load can be moved to any appropriate platform. In sum, we provided a thorough approach that provides guarantees to the developer concerning the sequences of incoming requests including the values of parameters. As a side effect, Ghostrail mitigates CSRF and injection (XSS, SQL injection) attacks in most cases.

8.6 Related Work

The *Open Web Application Security Project (OWASP)* coined the term *failure to restrict URL access* [133] to describe a similar vulnerability as our control-flow weakness.

However, it is more focused on access-control flaws that can be exploited by *forced-browsing attacks* [134] to find a *deep link* [25] to a high-privilege web page. Workflows and control-flow integrity play a tangential role in the description.

We divide other related work in navigation-restriction means, detection of server-side state violation, protection against and detection of client-side manipulation, race condition detection, and access-control mechanisms that mitigate direct URL access attempts.

There are different names for the respective attacks and vulnerabilities though not big differences in their technical details. In some cases, the attack allowing a malicious user to compose his own sequence of actions is called *workflow violation attack* [34], *state violation attack* [105], *workflow attack* [9, 84] or the attack exploiting *web application logic vulnerabilities* [56]. Partial overlap exists with *HTTP parameter pollution attacks* [5] and *parameter tampering attacks* [16].

8.6.1 Navigation-restriction Means

These approaches restrict the web application's request surface towards the user. They limit the accepted requests to a predefined set and prevent arbitrary navigation by users.

BAYAWAK [84] is a powerful tool to enforce request integrity. The basic idea is to prevent access to all server-side resources by giving them unique temporary interface identifiers (IID). The IIDs are changed with every request. In each response, the hyperlinks carry the necessary IID to address the intended next resources. Requests to arbitrary resources are prevented due to missing identifiers. BAYAWAK appends the IID as an HTTP parameter, e.g. `?IID=x`. All necessary attributes in all web pages have to be modified to include the IID. It remains open how dynamically generated requests are equipped with the IID. By design, multi-tabbing and back-button support as well as page reloads can not be granted as the session-bound IID must be outdated. Race condition protection depends on the actual implementation of this concept, namely whether the parallel execution of requests with the same IID is possible or excluded.

Hallé et al. propose a model checking-based approach to prevent navigation errors [72]. They explain their navigation state machines that allow the execution of given actions only immediately after a preceding action. For example, the modification of user accounts is only admitted if requested right after listing all user accounts. Moreover, parameter values can be defined as a prerequisite for actions. The approach focuses more on unintentionally caused errors than on security issues based on malicious user behavior. Complete workflows can not be defined explicitly. Instead, only ordered pairs of actions can be set. Multi-tabbing and race conditions are not handled.

8.6.2 State Violation Detection

The approaches that we describe in this section aim at detecting unintended or unusual server states. They infer the intended application states during a training phase or by static code analysis and raise an alarm as soon as the detected state deviates from

the known states, but they do not intend to make workflows explicit and control the interactions with users.

MiMoSA [9] detects violations of workflow integrity if intended workflows are enforced based on PHP session variables, request parameters, and database tables. It uses a cascade of dynamic and static analysis of PHP code together with model checking techniques to identify program paths that finally lead to an insecure state – either due to workflow attacks or due to injection attacks, like XSS and SQL injection.

Swaddler [34] detects anomalous combinations of session states and code execution points in PHP-based web applications after a learning phase. It assumes that attacks lead to observable differences in the application's state with respect to a threshold. In that sense, it is comparable to the functioning of an intrusion detection system (IDS).

BLOCK [105] follows a black-box approach to detect state violation attacks based on input/output invariants. In this case, input means the requested action, input parameters, and the session state while the output is the new session state and the HTTP response. The invariants are derived during an attack-free training phase. Discrepancies between the observed input/output and known invariants cause an alarm.

Waler [56] follows a similar but white-box approach. It attempts to infer invariants by running dynamic analysis. Invariants are determined by `if` statements and equality relations between session variables and database entries. Finally, Waler uses model checking to find invariants-violating program paths.

With existing approaches, every change on the web application needs a new pre-release learning phase to derive the policy automatically [9, 34, 56, 105]. Their policies can never be sound because training phases always miss unusual scenarios. Also, all such approaches must be fuzzy by design because they neglect the actual request context, e.g. HTTP parameters like an ID that change case-by-case but must not be changed by the user. Ghostrail and our control-flow monitor are able to enforce exact parameter matching without a need for policy updates.

8.6.3 Client-side Manipulation Detection

Malicious users not only craft individual HTTP requests or manipulate request headers to achieve their goals. Depending on the business logic of the web application, changes on the client-side JavaScript code can cause damage to the application provider. Existing approaches statically analyze JavaScript to determine the expected sequence of requests [71] or check the web application for exploitable HTTP parameter pollution vulnerabilities [5]. Two approaches replicate the client-side computation on the server side to detect deviations: *NoTamper* [16] focuses on input validation of HTML forms, while *Ripley* [183] follows a similar approach to Ghostrail. It replicates client-side JavaScript events in a server-side replica. However, Ripley is only applicable during development but not for legacy applications. Also, it relies on a distributing compiler thus excludes non-fitting technologies. Technically, Ripley ignores mouse movements on the client side and can not track respective events. As it uses event bubbling, it misses client-side events that redirect the browser. Ripley can not handle JavaScript code from different domains like common JavaScript libraries, mash-ups, and the `postMessage` API for communica-

tion between iframes. In that sense, Ghostrail is the consequent next step after Ripley because it covers modern application scenarios and all relevant user actions, thus makes less assumptions. Ripley and Ghostrail still share the same issues with randomness and timestamps.

8.6.4 Race Conditions

Race conditions [135] are explained in detail in Section 2.5.6. An attacker exploiting this vulnerability can execute one function more often than intended by the application developer.

Paleari et al. [138] describe an approach to detect race condition vulnerabilities in LAMP[22]-based web applications. They dynamically log SQL queries at runtime and analyze the log file to find possible race conditions based on the series of SQL clauses.

8.6.5 Access Control Mechanisms

As we pointed out in the beginning of this section, attackers might run forced-browsing attacks to gain access to restricted resources. They exploit weaknesses in static access limitations. Ghostrail has a similar goal, however, our access rules are dynamic and change with respect to the past actions.

The approach of Sun et al. [166] finds access control vulnerabilities in web applications. Therefore, they compute the difference of sitemaps for privileged and unprivileged users and try to directly address pages of this set.

Nemesis [40] implements an information flow-based approach to prevent access control and authentication bypass attacks. The respective server-side language interpreter (e.g. PHP) has to be modified to track the information flow through the web application. This way, the users' rights are mapped to the file system and database.

Desmet et al. [44] verify the protection provided by a WAF against forced-browsing attacks. They first model the read and write operations of web application components to the shared session record and statically verify this model. Next, they apply static verification to show that all web traffic passing the WAF, i.e. adhering to the WAF's enforcement policy, is compliant with the components' access to the shared session record, meaning that there is no unintended sequence of access attempts. Finally, they dynamically enforce the WAF's policy at runtime.

8.7 Conclusion

In this section, we first showed that current web application frameworks do not provide sufficient means to protect a web application's control-flow integrity. Based on this observation, we developed two options for protection: First, a control-flow monitor can be plugged in frameworks adhering to the MVC architecture. It expects a policy defining request sequences, and filters the requests not matching the definition. Second,

[22]LAMP stands for *Linux, Apache, MySQL, PHP*, the classical web server architecture.

our control-flow proxy is applicable without providing a policy. It learns the expected next requests by examining the last visited web page and thus enforces the common assumption that users only access a web application on dedicated entry pages and then click on links and buttons.

The combination of our control-flow monitor and the advanced authentication protocol described in Section 6 implements the desired features of connection-oriented protocols over the connectionless HTTP: The authentication protocol provides mutual authentication, preserves message integrity and confidentiality, and obtains the user's consent to critical cross-application requests, the control-flow monitor provides control-flow integrity and prevents state manipulation by unauthorized request sequences. So, we can conclude that we achieved the intended security features given the real-world protocols. More research is necessary concerning the consequences of pushing control-flow integrity policies to browsers. On the one hand, the client-side enforcement of such policies can save server load and improve the reaction time, on the other hand, received requests must still be validated to detect manipulations bypassing the enforcement on the client side.

9 Conclusion

We conclude the thesis with a summary of the past sections, name future work and problems that remain open after this thesis, and finally give an outlook on current developments in the field of web communication security.

9.1 Summary

In this thesis, we identified crucial security requirements of modern web applications that are not covered by HTTP on the application layer: message integrity and confidentiality, user and application authentication, control-flow integrity, and application-to-application authorization. We showed that those requirements are best met by a connection-oriented protocol. However, the connection-oriented protocols of the web stack can only cover message integrity and confidentiality, and the exchange of the web's dominating protocol is out of scope to achieve the other requirements. Taking the connectionless HTTP protocol as given, we came up with practical approaches to improve the security of modern web communication.

We showed in Section 3 that SSL/TLS can preserve message integrity and confidentiality. However, the ultimate reliance on a vast number of certificate authorities and the tremendous implementation issues (see Poodle, Heartbleed, a.o.) are severe problems that create the need for more sustainable solutions. Our approach implementing secure user and application authentication (see Section 6) provides all necessary elements to sign and encrypt messages before sending. It does not establish a secure connection but works on a message level. There is no need for a public key infrastructure because the respective keys are exchanged on a peer-to-peer basis. However, a trust anchor is required for the initial account setup.

In order to implement secure user and application authentication in the short run, web applications need to issue fresh session identifiers upon user login (see Section 4) and bind user credentials entered into web forms and browser-stored credentials together to avoid that an attacker can easily reuse stolen passwords and session cookies (see Section 5). This mitigates the conceptual problem that secrets are currently entered and sent to non-authenticated communication partners – and thus hardly remain secret. Also, a user's acknowledgment to security-critical actions delivers the browser from its role as a confused deputy. Our more sophisticated approach overcomes the fundamental issue concerning the transmission of confidential data (see Section 6). It supplies the needs of modern web applications in the long run because it requires adaptions on the client side and on the server side – the well-known chicken-and-egg problem. Our implementations prove the applicability on mobile devices. A desktop edition can be implemented as a

browser extension, natively in the browser, or as a standalone tool.

While the established standard for message integrity and confidentiality suffers considerable weaknesses, the official standard for secure application-to-application authorization CORS [181] still lacks support. For the time being, our approach presented in Section 7 implements a heuristic to estimate the risk of e.g. cross-application requests. It lets uncritical requests pass while authenticated, i.e. critical, requests undergo a series of sanity checks one of them the check for an allowing CORS policy. This way, the approach helps immediately and bridges the time until CORS is introduced widely. The more CORS is introduced the more accurate our approach can prevent cross-application attacks and let intended communication pass.

Finally, control-flow integrity is an inherent requirement of web applications that implement workflows. We showed that attacks on an application's control-flow integrity can have severe consequences. Unprotected applications can apply ad-hoc protection using our self-learning proxy (see Section 8.5). It neither requires access to the code nor a policy as input and thus perfectly fits as a quick fix. The runtime overhead, however, will make it a temporary solution in most cases. In the long run, control-flow integrity must be taken into account at development time. As one part of the web application development process, a control-flow integrity policy must be phrased and shipped together with the application. This policy is enforced by a respective module of the underlying web application framework as are all kinds of regular tasks demanded by the majority of web applications. We presented an appropriate module in Section 8.4.

9.2 Future Work and Open Problems

There is an ongoing trend to use web technologies for home automation, so-called *smart homes*. For instance, vendors of heaters and washing machines install web servers on their devices which provide a user interface to control and 'program' the devices. This use case can be considered the next business model driving the development of the web, and again, the simple nature of HTTP together with the universal availability of client devices, i.e. all networked devices with a browser, paves the way for this step. Looking ahead, the smart home scenario will bring a new set of challenges concerning secure communication:

Secure Machine-to-Machine Authentication We explained the pitfalls of current user authentication towards web applications and the other direction: the authentication of a web application towards a user. In smart home scenarios, however, a number of devices need to communicate with each other in order to provide holistic services. During this communication, authentication is crucial to prevent spoofing attacks leading to harmful results for the user. The example of Vaillant recently showed the impact of a security breach. In this case, the heater control was world readable and writable[23]. In machine-to-machine authentication scenar-

[23] See `http://www.hotforsecurity.com/blog/vulnerability-in-vaillant-heating-systems-allows-unauthorized-access-5926.html`

ios, there is no secret credential that is only known by the user. Considering the basic user-centric authentication means, i.e. knowledge, biometry, and ownership, there must be a secret piece of information stored on each device serving as the authentication factor. The challenges include the definition of an authentication protocol as well as the protection against physical attacks.

Secure Session Migration There is an increasing number of mobile devices in most households, each with its own pros and cons. Cell phones, for instance, are usually taken along while tablet PCs are sometimes too bulky. At home, however, the bigger screen of a tablet PC is favorable. So, a user might start the heater before coming home and check the content of the fridge using her cell phone while she switches to her tablet PC after arrival to tune the home entertainment system. Current web technologies require a new user authentication when the user exchanges the device. The scenario described above, however, demands usable, i.e. seamless, and secure session migration.

The Domain Paradigm Web technologies have always relied on the well-known domain principle: The same-origin policy separates content from different domains and SSL/TLS assigns a public key to a domain. In a smart home scenario, however, the separation of devices and services into domains is not straightforward:

- Assigning a unique domain to each device means applying the strongest separation. Hence, cross-device communication in the user's browser is severely limited. Such communication is however wanted because a user's action might have an impact on more than one device which means that the information about this action must spread.
- Unifying all devices of the same household in one domain, on the other hand, means almost unrestricted cross-device communication – for the prize of unrestricted access of each device to personal information provided by other devices. Also, a security breach of one device immediately threatens the security of all smart devices in the house.

History shows that technical decisions like the separation of devices into domains must not be left to the user because users tend to prefer functionality over security and can hardly estimate the impact of their decision.

Attacker Model It is necessary to understand how an appropriate attacker model is for the smart home scenario, i.e. how far are conventional attacker models applicable to smart homes. For instance, at a first glance, the local nature of a smart home could serve as a starting point for trust. Then, each person coming close to the devices gains access including visitors, intruders, and sometimes neighbors. On the contrary, attackers might regularly take the position of a man in the middle or take control of a previously trusted device by stealing it.

Off-topic Challenges Related challenges concerning the security of smart homes include preserving the privacy of users and patching vulnerable services. We omit a further

discussion of these topics because they are out of our scope.

Finally, developing a secure input form for user credentials is still an open problem. Such a form must not be forgeable by an attacker – which usually means that its design must not be predictable – but easily verifiable by each user – meaning that it must have a constant look – without burdening the user with the verification of secret authentication properties. The login forms used by browsers for HTTP authentication as well as today's HTML form-based login forms are easy to spoof and thus not suitable. Without a secure input form, however, phishing attacks will not end unless no user credentials must be entered at all for authentication. We want to emphasize that even protocols implementing sophisticated authentication protocols without transmitting the password (e.g. BetterAuth [87]) are susceptible to phishing attacks without a secure input form.

9.3 Outlook

In order to render a modern web page, the browser must issue an average of 100 HTTP requests[24]. For this reason, the focus of the new HTTP version 2.0 [11] lies on performance, a more effective usage of network resources, and the reduction of latency on the client side[25]. Nevertheless, it continues the tradition of backwards compatibility. The goals are supposed to be achieved by HTTP header compression, connection multiplexing using parallel streams, i.e. issuing several requests right after the other via the same TCP connection, and a new feature that allows web servers to push content that has not been requested yet. Servers can push elements of a requested web page like JavaScript and CSS files together with the response anticipating the client's request for these elements next.

From a security point of view, the new version will not contribute to the requirements of modern web applications we discussed in this thesis. While Google's draft SPDY [12], which is the blueprint of HTTP 2.0, provided at least the mandatory use of SSL/TLS, this feature was removed in the draft of the HTTP 2.0 specification for compatibility reasons concerning intermediaries, e.g. proxies, that provide authentication, caching, security scanning, and the like. Finally, there is no user or application authentication, message integrity and confidentiality depends on the optional usage of SSL/TLS, the application-to-application authorization can eventually be implemented using CORS – which at least runs faster thanks to HTTP 2.0 – and preserving control-flow integrity remains each application's own business.

Summing up, the contributions of this thesis are still valid after the introduction of HTTP 2.0. All approaches are applicable with HTTP 2.0 though the clear structure of request-response round trips will be watered down. Nevertheless, web applications will react on requests and perform respective actions. In this context, request authentication and control-flow integrity remain crucial elements of web communication security. We believe that the message-based security provided by our authentication protocol in

[24] see `http://httparchive.org/trends.php`, Nov 2014
[25] `https://http2.github.io/`

Section 6 will also have performance benefits compared to the conventional SSL/TLS channel due to the new connection multiplexing with parallel streams. However, the example of SPDY shows that only a big player in the web market can establish changes on the client side and on the server side, for instance, to introduce a sophisticated authentication protocol.

Bibliography

[1] Ben Adida. BeamAuth: Two-Factor Web Authentication with a Bookmark. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[2] Luis Von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: using hard AI problems for security. In *Eurocrypt'03*, pages 294–311, 2003.

[3] Wade Alcorn. Inter-Protocol Exploitation. Whitepaper, NGSSoftware Insight Security Research (NISR), `http://www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf`, March 2007.

[4] Chaitrali Amrutkar, Patrick Traynor, and Paul C. van Oorschot. Measuring SSL Indicators on Mobile Browsers: Extended Life, or End of the Road? In *Proceedings of the Information Security Conference 2012 (ISC'12)*, 2012.

[5] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, 2011.

[6] D. Balfanz and R. Hamilton. Transport Layer Security (TLS) Channel IDs. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-channelid-01`, Version 01, June 2013.

[7] D. Balfanz, D. Smetters, M. Upadhyay, and A. Barth. TLS Origin-Bound Certificates. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-obc-01`.

[8] D. Balfanz, D. Smetters, M. Upadhyay, and A. Barth. TLS Origin-Bound Certificates. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-obc-01`, Version 01, November 2011.

[9] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-Module Vulnerability Analysis of Web-based Applications. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[10] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[11] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol version 2. Internet Draft, `https://tools.ietf.org/html/draft-ietf-httpbis-http2-16`.

[12] Mike Belshe and Roberto Peon. SPDY. The Chromium Projects, `http://www.chromium.org/spdy`, accessed 15-01-2015.

[13] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, `http://tools.ietf.org/html/rfc1945`.

[14] Tim Berners-Lee. HTTP 0.9. [online], `http://www.w3.org/Protocols/HTTP/AsImplemented.html`, accessed 24-11-2014, 1991.

[15] Tim Berners-Lee and Robert Cailliau. WorldWideWeb: Proposal for a HyperText Project. [online], `http://www.w3.org/Proposal`, accessed 24-11-2014, Nov 1990.

[16] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[17] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin Cookies: Session Integrity for Web Applications. In *Proceedings of the WEB 2.0 SECURITY AND PRIVACY 2011 Workshop (W2SP 2011)*, 2011.

[18] Bastian Braun, Patrick Gemein, Hans P. Reiser, and Joachim Posegga. Control-Flow Integrity in Web Applications. In *Proceedings of the 2013 International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*, pages 1–16. Lecture Notes in Computer Science (LNCS), Springer, February 2013.

[19] Bastian Braun, Caspar Gries, Benedikt Petschkuhn, and Joachim Posegga. Ghostrail: Ad Hoc Control-Flow Integrity for Web Applications. In *Proceedings of the 29th IFIP International Information Security and Privacy Conference (IFIP SEC 2014)*, pages 264–277. IFIP Advances in Information and Communication Technology, Springer, June 2014.

[20] Bastian Braun, Martin Johns, Johannes Köstler, and Joachim Posegga. A Trusted UI for the Mobile Web. In *Proceedings of the 29th IFIP International Information Security and Privacy Conference (IFIP SEC 2014)*, pages 127–141. IFIP Advances in Information and Communication Technology, Springer, June 2014.

[21] Bastian Braun, Martin Johns, Johannes Köstler, and Joachim Posegga. PhishSafe: Leveraging Modern JavaScript API's for Transparent and Robust Protection. In *Proceedings of the Fourth ACM Conference on Data and Application Security and Privacy (ACM CODASPY 2014)*, pages 61–72, March 2014.

[22] Bastian Braun, Stefan Kucher, Martin Johns, and Joachim Posegga. A User-Level Authentication Scheme to Mitigate Web Session-Based Vulnerabilities. In

Proceedings of Trust, Privacy and Security in Digital Business (TrustBus '12), pages 17–29. Lecture Notes in Computer Science (LNCS), Springer, September 2012.

[23] Bastian Braun, Korbinian Pauli, Joachim Posegga, and Martin Johns. LogSec: Adaptive Protection for the Wild Wild Web. In *Proceedings of the 2015 ACM Symposium on Applied Computing (SAC 2015)*. ACM, April 2015.

[24] Bastian Braun, Christian v. Pollak, and Joachim Posegga. A Survey on Control-Flow Integrity Means in Web Application Frameworks. In *Proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec 2013)*, pages 231–246. Lecture Notes in Computer Science (LNCS), Springer, October 2013.

[25] Tim Bray. Deep Linking in the World Wide Web. [online], `http://www.w3.org/2001/tag/doc/deeplinking.html`, accessed 29-05-2012.

[26] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably Sound Browser-Based Enforcement of Web Session Integrity. In *Proceedings of the Computer Security Foundations Symposium 2014 (CSF'14)*, 2014.

[27] builtWith. Framework Usage Statistics – Overview of Statistics for Framework Technologies. [online], `http://trends.builtwith.com/framework`, accessed 12-01-2013.

[28] Elie Bursztein, Chinmay Soman, Dan Boneh, and John C. Mitchell. SessionJuggler: Secure Web Login from an Untrusted Terminal Using Session Hijacking. In *Proceedings of the 24th International World Wide Web Conference (WWW)*, 2012.

[29] Cake Software Foundation, Inc. CakePHP. [online], `http://cakephp.org/`, accessed 15-01-2015.

[30] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. Quite a Mess in My Cookie Jar! Leveraging Machine Learning to Protect Web Authentication. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*, 2014.

[31] Madhusudhanan Chandrasekaran and Ramkumar Chinchani. PHONEY: Mimicking User Response to Detect Phishing Attacks. In *International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM 2006)*, 2006.

[32] Eric Y. Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. 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.

[33] Neil Chou, Robert Ledesma, Yuka Teraguchi, Dan Boneh, and John C. Mitchell. Client-Side Defense against Web-based Identity Theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, 2004.

[34] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2007.

[35] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A Safety-oriented Platform for Web Applications. In *2006 IEEE Symposium on Security and Privacy*, 2006.

[36] Steven Crites, Francis Hsu, and Hao Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[37] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, `http://tools.ietf.org/html/rfc6376`.

[38] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, `http://tools.ietf.org/html/rfc4627`.

[39] Arshan Dabirsiaghi. Cross-Protocol XSS with Non-Standard Service Ports. Tech-Note, `http://i8jesus.com/?p=75`, accessed 05-02-2010, August 2009.

[40] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *USENIX Security*, 2009.

[41] David Heinemeier Hansson. Ruby on Rails. [online], `http://rubyonrails.org/`, accessed 15-01-2015.

[42] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against csrf attacks. In *European Symposium on Research in Computer Security (ESORICS 2011)*, 2011.

[43] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: Self-reliant Client-side Protection against Session Fixation. In *12th IFIP International Conference on Distributed Applications and Interoperable Systems*, 2012.

[44] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Bridging the Gap Between Web Application Firewalls and Web Applications. In *Proceedings of the fourth ACM Workshop on Formal Methods in Security*, November 2006.

[45] Rachna Dhamija and J. D. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Proceedings of the 2005 Symposium on Usable Privacy and Security (SOUPS '05)*, 2005.

[46] Django Software Foundation. Django. [online], `https://www.djangoproject.com/`, accessed 15-01-2015.

[47] Julie S. Downs, Mandy B. Holbrook, and Lorrie Faith Cranor. Decision Strategies and Susceptibility to Phishing. In *Symposium On Usable Privacy and Security (SOUPS)*, 2006.

[48] Dr. Web. New Trojan steals short messages. [online], `http://news.drweb.com/show/?i=3549`, accessed 12-09-2013.

[49] Peter Eckersley. How secure is HTTPS today? How often is it attacked? [online], `https://www.eff.org/deeplinks/2011/10/how-secure-https-today`, accessed 16-02-2012, October 2011.

[50] Peter Eckersley and Jesse Burns. The (Decentralized) SSL Observatory. Invited Talk, Usenix Security 2011, `http://static.usenix.org/events/sec11/tech/slides/eckersley.pdf`, accessed 16-02-2012, August 2011.

[51] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, 2008.

[52] EllisLab, Inc. CodeIgniter. [online], `http://www.codeigniter.com/`, accessed 15-01-2015.

[53] EMC Corporation. RSASecurID. [online], `http://www.emc.com/security/rsa-securid.htm`, accessed 12-09-2013.

[54] David Endler. The Evolution of Cross-Site Scripting Attacks. Whitepaper, iDefense Inc., `http://www.cgisecurity.com/lib/XSS.pdf`, accessed 19-01-2015, May 2002.

[55] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. Internet-Draft, `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-08`, accessed 10-09-2013.

[56] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX Security*, 2010.

[57] Adrienne Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to Ask for Permission. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security (HotSec 2012)*, 2012.

[58] Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices. In *Web 2.0 Security and Privacy (W2SP)*, 2011.

[59] Ian Fette, Norman Sadeh, and Anthony Tomasic. Learning to Detect Phishing Emails. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*, 2007.

[60] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, June 1999.

[61] Python Software Foundation. URLlib2 - Python HTTP URL opener library. [software], `http://docs.python.org/library/urllib2.html`, accessed 23-06-2010, April 2010.

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

[63] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, `http://www.ietf.org/rfc/rfc2617.txt`, 1999.

[64] Evgeniy Gabrilovich and Alex Gontmakher. The Homograph Attack. *Communications of the ACM*, 45, 2002.

[65] Google. Authenticator. [online], `http://code.google.com/p/google-authenticator/`, accessed 12-09-2013.

[66] Google. Gruyere. [online], `http://google-gruyere.appspot.com`, accessed 23-01-2014.

[67] Google, Inc. Google Web Toolkit. [online], `http://www.gwtproject.org/`, accessed 15-01-2015.

[68] Mohamed G. Gouda, Alex X. Liu, Lok M. Leung, and Mohamed A. Alam. SPP: An anti-phishing single password protocol. *Computer Networks*, 51(13):3715 – 3726, 2007.

[69] Robert Graham. SideJacking with Hamster. [online], `http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html`, accessed 08-02-2010, August 2007.

[70] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *In Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[71] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*, 2009.

[72] Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, 2010.

[73] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849, `http://tools.ietf.org/html/rfc5849`, April 2010.

[74] Norm Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22:36–38, October 1988.

[75] Edward Henning. Trustwave issued a man-in-the-middle certificate. [online], `http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate-1429982.html`, accessed 16-02-2012, February 2012.

[76] Amir Herzberg and Ahmad Jbara. Security and identification indicators for browsers against spoofing and phishing attacks. *ACM Transactions on Internet Technology (TOIT)*, 2008.

[77] Ian Hickson. Web Storage. [online], `http://www.w3.org/TR/webstorage/`, accessed 10-09-2013.

[78] Ariya Hidayat. PhantomJS. [online], `http://phantomjs.org`, accessed 22-01-2014.

[79] Jeff Hodges, Colin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS). [IETF draft], `http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-04`, Version 04, January 2012.

[80] Ryan Holliday. JAMWiki. [software], `http://jamwiki.org/`, Version 0.8.0, accessed 22-06-2010, December 2009.

[81] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *21st USENIX Security Symposium*, August 2012.

[82] Ilya Kantor. JavaScript Tutorial - Bubbling and capturing. [online], `http://javascript.info/tutorial/bubbling-and-capturing`, accessed 22-01-2014.

[83] Internet2. Shibboleth. [online], `http://shibboleth.net/`, accessed 29-05-2013.

[84] Karthick Jayaraman, Grzegorz Lewandowski, Paul G. Talaga, and Steve J. Chapin. Enforcing Request Integrity in Web Applications. In *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2010)*, 2010.

[85] Martin Johns. SessionSafe: Implementing XSS Immune Session Handling. In *European Symposium on Research in Computer Security (ESORICS 2006)*, September 2006.

[86] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable Protection Against Session Fixation Attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011)*, pages 1531–1537. ACM, March 2011.

[87] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. BetterAuth: Web Authentication Revisited. In *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC 2012)*, December 2012.

[88] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In *OWASP Europe 2006*, 2006.

[89] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, volume 1, 1988.

[90] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks (SecureComm 2006)*, 2006.

[91] Joyent, Inc. Node.js. [online], `http://nodejs.org`, accessed 22-01-2014.

[92] Samy Kamkar. phpwn: Attack on PHP Sessions and Random Numbers. Security Advisory, `http://samy.pl/phpwn/`, accessed 09-02-2010, August 2009.

[93] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, `http://tools.ietf.org/html/rfc5996`.

[94] Engin Kirda and Christopher Kruegel. Protecting Users agains Phishing Attacks with AntiPhish. In *29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, 2005.

[95] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.

[96] Amid Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Whitepaper, Sanctum Inc., `http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`, accessed 04-09-2008, March 2004.

[97] Paul Knickerbocker. Combating Phishing through Zero-Knowledge Authentication. Master's thesis, Graduate School of the University of Oregon, 2008.

[98] Kohana Team. Kohana. [online], `http://kohanaframework.org/`, accessed 15-01-2015.

[99] Mitja Kolsek. Session Fixation Vulnerability in Web-based Applications. Whitepaper, Acros Security, `http://www.acrossecurity.com/papers/session_fixation.pdf`, accessed 19-01-2015, December 2002.

[100] Korbinian Pauli. All About Clickjacking. [Seminar Paper], Seminar on Secure Cloud Computing, University of Passau, Germany, March 2014.

[101] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, `http://tools.ietf.org/html/rfc2104`, February 1997.

[102] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109, `http://www.ietf.org/rfc/rfc2109.txt`, February 1997.

[103] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965, `http://www.ietf.org/rfc/rfc2965.txt`, October 2000.

[104] Sebastian Lekies, Nick Nikiforakis, Walter Tighzert, Frank Piessens, and Martin Johns. DEMACRO: Defense against Malicious Cross-Domain Requests. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2012)*, 2012.

[105] Xiaowei Li and Yuan Xue. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011)*, 2011.

[106] Hal Lockhart and Brian Campbell. SAML V2.0. [online], `https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf`, accessed 29-05-2013, March 2008.

[107] Christian Ludl, Sean McAllister, Engin Kirda, and Christopher Kruegel. On the Effectiveness of Techniques to Detect Phishing Sites. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '07)*, 2007.

[108] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking attacks on web in android, ios, and windows phone. In *Foundations and Practice of Security*, 2012.

[109] Magento Commerce. [online], `http://demo.magentocommerce.com/`, accessed 24-09-2012.

[110] Moxie Marlinspike. New Tricks For Defeating SSL In Practice. Talk at BlackHat '09, `http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf`, accessed 10-09-2013.

[111] Rob McCool. Server Scripts. [www-talk mailing list], `http://1997.webhistory.org/www.lists/www-talk.1993q4/0485.html`, accessed 19-01-2015, Nov 1993.

[112] Eric Medvet, Engin Kirda, and Christopher Kruegel. Visual-Similarity-Based Phishing Detection. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm '08)*, 2008.

[113] Microsoft. ASP.NET. [online], `http://www.asp.net/`, accessed 15-01-2015.

[114] Microsoft. ASP.NET MVC. [online], `http://www.asp.net/mvc`, accessed 15-01-2015.

[115] Microsoft. ASP.NET Web Forms. [online], `http://www.asp.net/web-forms`, accessed 15-01-2015.

[116] Microsoft. ASP.NET Web Pages. [online], `http://www.asp.net/web-pages`, accessed 15-01-2015.

[117] Microsoft. SenderID. [online], `http://www.microsoft.com/senderid`, accessed 03-09-2013.

[118] Microsoft. X-Frame-Options. [online], `http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx`, accessed 20-05-2011.

[119] Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security.* John Wiley & Sons, 2002.

[120] Mozilla. Persona. [online], `https://developer.mozilla.org/en-US/docs/Mozilla/Persona`, accessed 03-09-2013.

[121] Mozilla. X-Frame-Options response header. [online], `https://developer.mozilla.org/en/the_x-frame-options_response_header`, accessed 20-05-2011.

[122] Mozilla. CSP (Content Security Policy). Mozilla Developer Network, `https://developer.mozilla.org/en-US/docs/Web/Security/CSP`, accessed 19-01-2015, March 2009.

[123] Mozilla Developer Network. AJAX. [online], `https://developer.mozilla.org/en-US/docs/AJAX`, accessed 15-01-2015.

[124] MSDN. Mitigating Cross-site Scripting With HTTP-only Cookies. [online], `http://msdn.microsoft.com/en-us/library/ms533046(VS.85).aspx`, accessed 08-06-2012.

[125] Netcraft. Anti-Phishing Extension. [online], `http://toolbar.netcraft.com/`, accessed 30-05-2013.

[126] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 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'12)*, 2012.

[127] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the Third international conference on Engineering Secure Software and Systems (ESSoS'11)*, 2011.

[128] Nick Nikiforakis, Yves Younan, and Wouter Joosen. HProxy: Client-side detection of SSL stripping attacks. In *Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, 2010.

[129] Nikos Nikiforakis, Andreas Makridakis, Elias Athanasopoulos, and Evangelos P. Markatos. Alice, What Did You Do Last Time? Fighting Phishing Using Past Activity Tests. In *Proceedings of the 3rd European Conference on Computer Network Defense*, 2009.

[130] Yuan Niu, Francis Hsu, and Hao Chen. iPhish: Phishing Vulnerabilities on Consumer Electronics. In *Proceedings of the 1st Conference on Usability, Psychology, and Security (UPSEC '08)*, 2008.

[131] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. SOMA: Mutual Approval for Included Content in Web Pages. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, 2008.

[132] OpenID Foundation. OpenID. [online], `http://openid.net/`, accessed 14-02-2012, February 2012.

[133] OWASP. Failure to Restrict URL Access. [online], `https://www.owasp.org/index.php/Top_10_2010-A8-Failure_to_Restrict_URL_Access`, accessed 11-05-2012.

[134] OWASP. Forced Browsing. [online], `https://www.owasp.org/index.php/Forced_browsing`, accessed 04-05-2012.

[135] OWASP. Race Conditions. [online], `https://www.owasp.org/index.php/Race_Conditions`, accessed 23-05-2012.

[136] The Open Web Application Security Project (OWASP). Session Fixation. TechNote, `http://www.owasp.org/index.php/Session_Fixation`, accessed 05-03-2010, February 2009.

[137] OWASP German Chapter. OWASP Best Practices: Use of Web Application Firewalls. [whitepaper], `http://www.owasp.org/index.php/Category:OWASP_Best_Practices:_Use_of_Web_Application_Firewalls`, accessed 07-03-2010, July 2008.

[138] Roberto Paleari, Davide Marrone, Danilo Bruschi, and Mattia Monga. On Race Vulnerabilities in Web Applications. In *Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2008)*, 2008.

[139] Bryan Parno, Cynthia Kuo, and Adrian Perrig. Phoolproof Phishing Prevention. In *Proceedings of the 10th International Conference on Financial Cryptography and Data Security (FC'06)*, 2006.

[140] PHP Group. session_regenerate_id(). PHP documentation, [online], `http://www.php.net/manual/de/function.session-regenerate-id.php`, accessed 04-04-2010, June 2010.

[141] Thomas Raffetseder, Engin Kirda, and Christopher Kruegel. Building Anti-Phishing Browser Plug-Ins: An Experience Report. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems (SESS '07)*, 2007.

[142] Venkata Prasad Reddy, V Radha, and Manik Jindal. Client Side Protection from Phishing Attack. *International Journal of Advanced Engineering Sciences and Technologies (IJAEST)*, pages 39–45, 2011.

[143] Robert Hansen. Clickjacking. [online], `http://ha.ckers.org/blog/20080915/clickjacking/`, accessed 20-05-2011.

[144] Robert Hansen and Jeremiah Grossman. Clickjacking. [online], `http://www.sectheory.com/clickjacking.htm`, accessed 20-05-2011.

[145] Angelo P. E. Rosiello, Engin Kirda, Christopher Kruegel, and Fabrizio Ferrandi. A Layout-Similarity-Based Approach for Detecting Phishing Pages. In *Proceedings of the third International Conference on Security and Privacy in Communication Networks (SecureComm 2007)*, 2007.

[146] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger Password Authentication Using Browser Extensions. In *Proceedings of the 14th Usenix Security Symposium (USENIX 2005)*, 2005.

[147] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger Password Authentication Using Browser Extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[148] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. IETF Internet-Draft, `http://tools.ietf.org/html/draft-ietf-websec-x-frame-options`, accessed 19-01-2015, February 2013.

[149] RSA Data Security. SiteKey. [Hosted at Bank of America], `https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go`, accessed 01-08-2013.

[150] Jesse Ruderman. The Same Origin Policy. [online], `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`, accessed 07-05-2014.

[151] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Proceedings of the 2010 International Symposium on Engineering Secure Software and Systems (ESSoS '10)*, 2010.

[152] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Proceedings of W2SP 2010*, 2010.

[153] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks. In *Proceedings of the 4th Workshop on Offensive Technologies (wOOt 2010)*, 2010.

[154] Michael Schrank, Bastian Braun, Martin Johns, and Joachim Posegga. Session Fixation – the Forgotten Vulnerability? In *Proceedings of Sicherheit 2010: Sicherheit, Schutz und Zuverlässigkeit*, pages 341–352. Lecture Notes in Informatics (LNI), Springer, 2010.

[155] SecurityFocus. Ruby on Rails 'redirect_to' HTTP Header Injection Vulnerability. TechNote, `http://www.securityfocus.com/bid/32359`, accessed 01-03-2010, December 2009.

[156] SeleniumHQ. Browser Automation. [online], `http://docs.seleniumhq.org`, accessed 23-01-2014.

[157] Hossain Shahriar and Mohammad Zulkernine. PhishTester: Automatic Testing of Phishing Attacks. In *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2010.

[158] Umesh Shankar and Chris Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, 2006.

[159] Mohsen Sharifi, Alireza Saberi, Mojtaba Vahidi, and Mohammad Zorufi. A Zero Knowledge Password Proof Mutual Authentication Technique Against Real-Time

Phishing Attacks. In *Third International Conference on Information Systems Security (ICISS 2007)*, 2007.

[160] Steve Sheng, Brad Wardman, Gary Warner, Lorrie Faith Cranor, Jason Hong, and Chengshan Zhang. An Empirical Analysis of Phishing Blacklists. In *Sixth Conference on Email and AntiSpam (CEAS 2009)*, 2009.

[161] Eric Shepherd. window.postmessage. [online], `https://developer.mozilla.org/en/DOM/window.postMessage`, accessed 12-02-2012, October 2011.

[162] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[163] Spring Projects. Spring Web Flow. [online], `http://www.springsource.org/spring-web-flow`, accessed 15-01-2015.

[164] SpringSource. The Spring Framework. [online], `http://www.springsource.org/`, accessed 15-01-2015.

[165] Brandon Sterne and Adam Barth. Content Security Policy. [W3C Working Draft], `http://www.w3.org/TR/CSP/`, November 2011.

[166] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *USENIX Security*, 2011.

[167] Sun Microsystems Inc. J2EE - Java Platform Enterprise Edition 5. [online], `http://java.sun.com/javaee/technologies/javaee5.jsp`, accessed 04-06-2007, 2007.

[168] Andrew S. Tanenbaum. *Computer Networks*. Pearson, 2003.

[169] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[170] Ryan Tate. Apple's Worst Security Breach: 114,000 iPad Owners Exposed. [online], `http://gawker.com/5559346/`, accessed 19-01-2014.

[171] CherryPy Team. CherryPy - Lightweight, pythonic web framework. [software], `http://www.cherrypy.org/`, accessed 23-06-2010, April 2010.

[172] The Anti-Phishing Working Group (APWG). Global Phishing Survey: Domain Name Use and Trends in 2H2012. [online], `http://docs.apwg.org/reports/APWG_GlobalPhishingSurvey_2H2012.pdf`, accessed 03-09-2013.

[173] The Anti-Phishing Working Group (APWG). Phishing Activity Trends Report, 1st Quarter 2013. [online], `http://docs.apwg.org/reports/apwg_trends_report_q1_2013.pdf`, accessed 03-09-2013.

[174] The Anti-Phishing Working Group (APWG). Phishing Attack Trends Reports. [online], `http://www.apwg.org/resources/apwg-reports/`, accessed 03-09-2013.

[175] The Apache Software Foundation. Tapestry. [online], `http://tapestry.apache.org/`, accessed 15-01-2015.

[176] The Electronic Frontier Foundation. HTTPS Everywhere. [Browser Extension], `https://www.eff.org/https-everywhere`, accessed 05-10-2013.

[177] The New York Times. Thieves Found Citigroup Site an Easy Entry. [online], `http://www.nytimes.com/2011/06/14/technology/14security.html`, accessed 24-05-2012.

[178] Tianhao Tong and David Evans. GuarDroid: A Trusted Path for Password Entry. In *Mobile Security Technologies (MoST) 2013*, 2013.

[179] Jochen Topf. The HTML Form Protocol Attack. TechNote, `http://www.remote.org/jochen/sec/hfpa/hfpa.pdf`, accessed 05-02-2010, August 2001.

[180] Hicham Tout and William Hafner. Phishpin: An Identity-Based Anti-Phishing Approach. In *International Conference on Computational Science and Engineering (CSE '09)*, 2009.

[181] Anne van Kesteren. Cross-Origin Resource Sharing. W3C Working Draft, `http://www.w3.org/TR/cors/`, April 2012.

[182] Laura Varteressian. Yahoo! Sign-In Seal. [online], `http://security.yahoo.com/sign-seal-000000996.html`, accessed 01-08-2013.

[183] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, 2009.

[184] Bob Violino. After Phishing? Pharming! [online], `http://www.csoonline.com/article/220629/after-phishing-pharming-`, accessed 10-09-2013.

[185] W3C. HTML5 - The canvas element. [online], `http://www.w3.org/TR/html5/the-canvas-element.html`, accessed 29-09-2011.

[186] W3C. HTML5 - The iframe element. [online], `http://www.w3.org/TR/html5/the-iframe-element.html#the-iframe-element`, accessed 29-08-2011.

[187] Robert Wagner and Jeff Bryner. Address Resolution Protocol Spoofing and Man-in-the-Middle Attacks. [online], `http://www.sans.org/reading-room/whitepapers/threats/address-resolution-protocol-spoofing-man-in-the-middle-attacks-474`, accessed 17-11-2014.

[188] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, 2007.

[189] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, 2009.

[190] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *IEEE Symposium on Security and Privacy*, 2011.

[191] The Web Application Security Consortium (WASC). Session Fixation. TechNote, `http://projects.webappsec.org/Session-Fixation`, accessed 05-03-2010, January 2010.

[192] Heiko Webers. Header Injection And Response Splitting. TechNote, `http://www.rorsecurity.info/journal/2008/10/20/header-injection-and-response-splitting.html`, accessed 01-03-2010, October 2008.

[193] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-Scale Automatic Classification of Phishing Pages. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS '10)*, 2010.

[194] Min Wu, Robert C. Miller, and Simson L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*, 2006.

[195] Min Wu, Robert C. Miller, and Greg Little. Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In *Proceedings of the Second Symposium on Usable Privacy and Security (SOUPS '06)*, 2006.

[196] Xdebug. [online], `http://xdebug.org/`, accessed 05-06-2012.

[197] Guang Xiang, Jason Hong, Carolyn P. Rose, and Lorrie Cranor. CANTINA+: A Feature-rich Machine Learning Framework for Detecting Phishing Web Sites. *ACM Transactions on Information and System Security (TISSEC)*, 2011.

[198] Zishuang (Eileen) Ye and Sean Smith. Trusted Paths for Browsers. In *Proceedings of the 11th USENIX Security Symposium (USENIX 2002)*, 2002.

[199] Ka-Ping Yee and Kragen Sitaker. Passpet: Convenient Password Management and Phishing Protection. In *Proceedings of the Second Symposium on Usable Privacy and Security (SOUPS '06)*, 2006.

[200] Chuan Yue and Haining Wang. Anti-Phishing in Offense and Defense. In *Annual Computer Security Applications Conference (ACSAC 2008)*, 2008.

[201] Michal Zalewski. Cross Site Cooking. Whitepaper, `http://www.securiteam.com/securityreviews/5EP0L2KHFG.html`, accessed 03-02-2010, January 2006.

[202] Yue Zhang, Serge Egelman, Lorrie Cranor, and Jason Hong. Phinding Phish: Evaluating Anti-Phishing Tools. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[203] Yuchen Zhou and David Evans. Why Aren't HTTP-only Cookies More Widely Deployed? In *Proceedings of W2SP '10*, 2010.