PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Black-Box Security Testing of Browser-Based Security Protocols

Avinash Sudhodanan

Advisor:

Alessandro Armando, Full Professor, Università degli Studi di Genova

Co-Advisors:

Roberto Carbone, Researcher, Fondazione Bruno Kessler, Italy

Luca Compagna, Product Security Researcher, SAP Labs, France

April 2017

Abstract

Millions of computer users worldwide use the Internet every day for consuming web-based services (e.g., for purchasing products from online stores, for storing sensitive files in cloud-based file storage web sites, etc.). *Browser-based security protocols* (i.e. security protocols that run over the Hypertext Transfer Protocol and are executable by commercial web-browsers) are used to ensure the security of these services. Multiple parties are often involved in these protocols. For instance, a browser-based security protocol for Single Sign-On (SSO in short) typically consists of a user (controlling a web browser), a Service Provider web site and an Identity Provider (who authenticates the user). Similarly, a browser-based security protocol for Cashier-as-a-Service (CaaS) scenario consists of a user, a Service Provider web site (e.g., an online store) and a Payment Service Provider (who authorizes payments).

The design and implementation of browser-based security protocols are usually so complex that several vulnerabilities are often present even after intensive inspection. This is witnessed, for example, by vulnerabilities found in various browser-based security protocols such as SAML SSO v2.0, OAuth Core 1.0, etc. even years after their publication, implementation, and deployment. Although techniques such as formal verification and white-box testing can be used to perform security analysis of browser-based security protocols, currently they have limitations: the necessity of having formal models that can cope with the complexity of web browsers

(e.g., cookies, client-side scripting, etc.), the poor support offered for certain programming languages by white-box testing tools, to name a few. What remains is black-box security testing. However, currently available black-box security testing techniques for browser-based security protocols are either scenario-specific (i.e. they are specific for SSO or CaaS, not both) or do not support very well the detection of vulnerabilities enabling replay attacks (commonly referred to as logical vulnerabilities) and Cross-Site Request Forgery (CSRF in short).

The goal of this thesis is to overcome the drawbacks of the black-box security testing techniques mentioned above. At first this thesis presents an attack pattern-based black-box testing technique for detecting vulnerabilities enabling replay attacks and social login CSRF in multi-party web applications (i.e. web applications utilizing browser-based security protocols involving multiple parties). These attack patterns are inspired by the similarities in the attack strategies of previously-discovered attacks against browser-based security protocols. Second, we present manual and semi-automatic black-box security testing strategies for detecting 7 different types of CSRF attacks, targeting the authentication and identity management functionalities of web sites. We also provide proof-of-concept implementations of our ideas. These implementations are based on OWASP ZAP (a prominent, free and open-source penetration testing tool).

This thesis being in the context of an industrial doctorate, we had the opportunity to analyse the use-cases provided by our industrial partner, SAP, to further improve our approach. In addition, to access the effectiveness of the techniques we propose, we applied them against the browser-based security protocols of many prominent web sites and discovered nearly 340 serious security vulnerabilities affecting more than 200 web sites, including the web sites of prominent vendors such as Microsoft, eBay, etc.

Acknowledgements

The research presented in this thesis is a joint effort. First, I would like to express my sincere gratitude to my three supervisors: Prof. Alessandro Armando, Dr. Roberto Carbone and Dr. Luca Compagna (it has been a pleasure working with all of you). Next I would like to thank *(i)* Nicolas Dolgin for his contributions towards Blast and CSRF experiments, *(ii)* Adrien Hubner for transforming my Python code for Blast to a proper object-oriented form and *(iii)* Umberto Morelli for his assistance in testing Alexa top web sites (and also for being a great friend).

Special thanks to Laura Segalla, Sylvine Eusebi, Manuela Bacca, Silvia Tomassi, Adriana Betti, Dr. Francesca Belton, Dr. Andrea Stenico and other administrative staff members for helping me overcome the bureaucratic nightmares of visa, *nulla osta*, *permesso di soggiorno*, lodging, travel, health insurance and medical appointments.

The generous funding from Marie Sklodowska-Curie Actions program, Poste Italiane and University of Trento was indeed helpful in covering the expenses associated to my PhD.

The high-quality facilities offered by Fondazione Bruno Kessler and SAP Labs France deserves special mention (thanks to all those comfortable office spaces, high-end PCs, nutritious food, technical support, etc.).

Finally, I would like to thank my family for loving me unconditionally, my friends for making my life more joyful and my yoga teachers for ensuring my physical and mental well being.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The research presented in this thesis has been carried out in the context of the SECENTIS Project [26], under the topic *"Automatic Analysis of Browser-Based Security Protocols"*. The SECENTIS Project is a European Industrial Doctorate Program on Security and Trust of Next Generation Enterprise Information Systems. The industrial partner of the SECENTIS Project is SAP Labs France. The SECENTIS Project is financed by the European Union grant 317387, under FP7-PEOPLE-2012-ITN. In this thesis, we propose new approaches for *black-box security testing* of *browser-based security protocols.* In particular we focus on black-box security testing techniques for detecting logical vulnerabilities and vulnerability enabling Cross-Site Request Forgery attacks. We focus not only on the theoretical aspects of these techniques but also evaluate their practical applicability on industrial use-cases such as the ones provided by SAP.

1.1 Context

Browser-Based Security Protocols: Security protocols are communication protocols that aim to achieve some security goals through the use of cryptographic primitives. Browser-based security protocols are protocols that run over the Hypertext Transfer Protocol [53] (HTTP in short) and are executable by commercial web-browsers. By supporting the development of zero-footprinting applications (i.e. applications that do not require special-purpose clients), browser-based protocols greatly simplify the deployment and usage of applications based on the Software-as-a-Service (SaaS in short) paradigm. For this reason they are subject to a growing attention by industry. Browser-based security protocols ensure that the right people (or applications) access the right resources, thereby greatly simplifying the design and implementation of complex, on-line applications. A number of browser-based security protocols have been developed and standardized. For instance, the SAML SSO v2.0 Web-Browser Profile [61] is a standard, inter-operable, protocol for Single Sign-On (SSO) which is now supported by all major software vendors (e.g., Google, SAP, etc.). Similarly, PayPal Express Checkout is a proprietary protocol developed by PayPal for Cashier-as-a-Service (CaaS) scenario (i.e. payment via a trusted third-party).

1.2 The Problem

The design and analysis of browser-based security protocols is usually so complex that severe vulnerabilities are often present even after intensive inspection. This is witnessed, for example, by vulnerabilities found in various SSO protocols, such as SAML SSO v2.0 [45], OAuth Core 1.0 [67], even years after their publication, implementation, and deployment. While some model checking techniques exist for the automatic security analysis of security protocols (see, e.g., [47, 46]) and can be used to analyse browser-based security protocols, a significant amount of manual intervention is necessary to cope with the complexity of browsers (cookies, reliance on SSL/TLS channels, client-side scripting, to name a few). This hinders the adoption of these techniques in the software development life-cycle.

Another available option is that of white-box security testing techniques that can detect security vulnerabilities by analysing the source code of web applications (e.g., [91, 71]). However, there are several challenges in applying these techniques to test browser-based security protocols. For instance, the currently-available white-box security testing techniques for browser-based security protocols are only applicable to the CaaS scenario (e.g., the approach mentioned in [91]). Additionally, the unavailability of source code and the presence of multiple parties supporting different programming languages in browser-based security protocols makes it difficult to apply white-box security testing techniques.

The remaining option is represented by black-box security testing techniques. They analyse the HTTP traffic of web applications for detecting security vulnerabilities (e.g., [65, 82, 94]). Due to their application-agnostic nature, they are widely-preferred at the software development industry. However, quite similar to white-box techniques, the currently available black-box techniques for security testing browser-based security protocols are either scenario-specific (e.g., the approach mentioned in [94] is applicable only to SSO scenario and the approach proposed in [82] has been applied only to CaaS scenario) or it has been shown (e.g., in [81, 52]) that they do not support very well the detection of vulnerabilities causing replay attacks (also known as logical vulnerabilities [81]) and Cross-Site Request Forgery [98] (CSRF in short).

The above-mentioned limitations of the available security testing techniques clearly highlights the need for a general-purpose, application-agnostic, security testing technique for browser-based security protocols that can detect vulnerabilities that are not very well supported by currently available security testing tools (e.g., logical vulnerabilities, CSRF, etc.).

1.3 Proposed Solutions

The application-agnostic nature of black-box security testing techniques when combined with the fact that they are widely-preferred at the software industry, makes them an ideal candidate for conducting research on extending them. In fact this is what we did. We mainly focussed on proposing black-box security testing techniques for browser-based security protocols. In particular, first we propose a black-box security testing technique based on attack patterns for generating attack test cases against browser-based security protocol implementations. A summary of this approach is provided in Section 1.3.1. Additionally, we propose manual and semi-automatic black-box security testing techniques for detecting vulnerabilities enabling CSRF in the authentication and identity management functionalities of web sites. Section 1.3.2 provides a summary of these techniques.

1.3.1 Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications

We present an automatic technique based on attack patterns for black-box security testing of Multi-Party Web Applications (MPWAs in short). MPWAs are web applications implementing browser-based security protocols involving multiple parties. Examples of MPWA scenarios include web applications implementing SSO, CaaS, etc. Our approach stems from the observation that attacks against popular MPWAs share a number of similarities, even if the underlying protocols and services are different. We primarily target six different replay attacks and a social login Cross-Site Request Forgery (CSRF) attack. Firstly, we propose a methodology in which security experts can create attack patterns from known attacks. Secondly, we present a security testing framework that leverages attack patterns to automatically generate test cases that can be used to automate the security testing of MPWAs. We implemented our ideas (as a proof-of-concept tool named Blast) on top of OWASP ZAP (a popular, open-source penetration testing tool), created 7 attack patterns that correspond to 13 prominent attacks from the literature and discovered 21 previously unknown vulnerabilities in prominent MPWAs (e.g., `twitter.com`, `developer.linkedin.com`, `pinterest.com`), including MPWAs that do not belong to SSO and CaaS families. Finally, we extended Blast to meet the security testing needs at SAP.

1.3.2 Large-scale Analysis & Detection of Authentication Cross-Site Request Forgeries

During our experiments on MPWAs (see Section 1.3.1 above), our social login CSRF attack pattern discovered CSRF vulnerabilities in the login processes of top web sites. This inspired us to conduct further research on CSRF attacks affecting the authentication and identity management functionalities of web sites (what we refer to as *Authentication CSRF* attacks). We started by collecting several Auth-CSRF attacks reported in the literature, then analyzed their underlying strategies and identified 7 security testing strategies that can help a manual tester uncover vulnerabilities enabling Auth-CSRF. In order to check the effectiveness of our testing strategies and to estimate the incidence of Auth-CSRF, we conducted an experimental analysis considering 300 web sites belonging to 3 different rank ranges of the Alexa global top 1500. The results of our experiments are alarming: out of the 300 web sites we considered, 133 qualified for conducting our experiments and 90 of these suffered from at least one vulnerability enabling Auth-CSRF (i.e. 68%). We further generalized our testing strategies, enhanced them with the knowledge we acquired during our experiments and implemented them as an extension (namely CSRF-checker) to the open-source penetration testing tool OWASP ZAP. With the help of CSRF-checker, we tested 132 additional web sites (again from the Alexa global top 1500) and discovered that 95 of them were vulnerable to Auth-CSRF (i.e. 72%). Our findings include serious vulnerabilities among the web sites of Microsoft, Google, eBay, etc. Finally, we responsibly disclosed our findings to the affected vendors and helped them fix the issues we discovered. We even received monetary rewards for our findings (e.g., we received a $1500 cash award from Microsoft).

1.4 Overview of our contributions

The important contributions of this thesis are listed below:

- We propose a novel approach for black-box security testing multi-party web applications that implements browser-based security protocols.
- We propose manual and semi-automatic black-box security testing techniques for detecting vulnerabilities causing CSRF attacks in the authentication and identity management functionalities of web sites. These techniques are a good addition to the widely-used web application security testing guides (e.g., the one provided by OWASP [33]).
- We provide proof-of-concept implementations of the automatic and semi-automatic back-box security testing techniques we propose. Our implementations are based on OWASP ZAP (a popular, free and open-source penetration testing tool). This helps in the swift adoption of the techniques we propose by the security testing community.
- To measure the effectiveness of our approaches, we applied them to test the security of prominent web sites of the Internet and discovered nearly 340 serious security vulnerabilities affecting more than 200 web

sites, including the web sites of prominent vendors such as Microsoft, eBay, Linkedin, Stripe, Pinterest, etc.

- We responsibly disclosed our findings to the affected vendors and helped them fix the issues. For instance, we received a $1500 monetary award from Microsoft for discovering a CSRF attack in their browser-based security protocol underlying prominent online services such as `skype.com`, `outlook.com`, etc.
- We extend our proof-of-concept implementations based on the needs of our industrial partner, SAP and currently SAP is continuing to invest resources for further developing our prototypes.

1.5 Structure of the thesis

The remainder of this manuscript is organized as follows. In Chapter 2, we present our approach for black-box security testing MPWAs. In Chapter 3, we present our study on large-scale analysis and black-box detecting of vulnerabilities causing Authentication CSRF. In Chapter 4 we present the impacts of our research at the software industry, mainly focusing on SAP (the industrial partner of the research associated to this thesis). Finally, in Chapter 5 we conclude with some final remarks.

Chapter 2

Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications

This chapter presents an approach we developed (and presented at [89, 90]) for black-box security testing browser-based security protocol implementations. In particular, we focus on browser-based security protocols involving multiple parties.

2.1 Introduction

An increasing number of business critical, online web applications leverage trusted third parties in conjunction with web-based security protocols to meet their security needs. For instance, many online applications rely on authentication assertions issued by identity providers to authenticate users using a variety of web-based single sign-on (SSO) protocols (e.g., SAML SSO v2.0, OpenID Connect). Similarly, online shopping applications use online payment services and Cashier-as-a-Service (CaaS) protocols (e.g., Express Checkout [21] and PayPal Payment Standard [22]) to obtain proof-of-payment before delivering the purchased items. We refer to this broad class of protocols as security-critical *Multi-Party Web Applications* (MP-WAs). Three entities take part in the protocols: the *User* (through a web browser B), the web application (playing the role of Service Provider, SP), and a trusted third party (TTP).

The design and implementation of the protocols used by security-critical MPWAs are notoriously error-prone. Several vulnerabilities have been reported in the last few years. For instance, the incorrect handling of the OAuth 2.0 access token by a vulnerable SP can be exploited by an attacker hosting another SP [96]. If the victim *User* logs into the attacker's SP, the attacker obtains an access token (issued by TTP) from the victim and can replay it in the vulnerable SP to login as the victim. A similar attack was previously discovered in the SAML-based implementation deployed by Google [45] (here the SAML authentication assertion is replayed instead of the OAuth 2.0 access token). Similar attacks have also been detected in CaaS-enabled scenarios [91, 82]. For instance, a vulnerability in osCommerce v2.3.1 that allowed an attacker to shop for free has been reported in [82]: the attacker controls a SP and obtains an account identifier from PayPal for paying herself; later on, she replays this value in a subsequent session with a vulnerable SP where she purchases a product by paying herself. Recently, a token fixation attack in PayPal Express Checkout flow was discovered [36] which is very similar to the session fixation attack in OAuth 1.0 [16]. The problem is exacerbated by the large number of deployments. As a matter of fact, over 20% of the top twenty-thousand Alexa top US websites have a vulnerable implementation of the Facebook SSO [101].

The aforementioned attacks have been discovered through a variety of domain-specific techniques with different levels of complexity, ranging from formal verification [45], white-box analysis [91], black-box testing [82], to manual testing [36]. In this chapter, we explain how we pursue a different approach and propose an automatic black-box testing technique for security-critical MPWAs. Our approach is based on an observation and a conjecture. The observation is that, regardless of their purpose, the security protocols at the core of MPWAs share a number of features:

1. By interacting with *SP* (and/or *TTP*), *User* authenticates and/or authorizes some actions,
2. *TTP* (*SP*, resp.) generates a security token,
3. the security token is dispatched to *SP* (*TTP*, resp.) through the web browser, and
4. *SP* (*TTP*, resp.) checks the security token and completes the protocol by taking some security-critical decisions.

The conjecture is that the attacks found in the literature (and possibly many more still to be discovered) are instances of a limited number attack patterns. We conducted a detailed study of attacks discovered in MPWAs of real-world complexity and analyzed their similarities. This led us to identify a small number of application-independent attack patterns that concisely describe the actions performed by attackers while performing these attacks.

To assess the generality and the effectiveness of our approach, we have developed a security testing framework based on OWASP ZAP[1], a popular open-source penetration testing tool, and run it against a number of prominent MPWA implementations. Our tool has been able to identify:

- two previously unknown attacks against websites integrating Linkedin's Javascript API-based SSO that causes an access token replay attack and a persistent XSS attack;
- a previously unknown redirection URI fixation attack against the implementation of the OAuth 2.0 protocol in PayPal's "Log in with

[1]`www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`

PayPal" SSO solution which allows a network attacker to steal the authorization code of the victim and replay it to login as the victim in any *SP* website using the same SSO solution;

- a previously unknown attack in the payment checkout solution offered by Stripe (integrated in over 17,000 websites [31]); the attack allows an attacker to impersonate a *SP* to obtain a token from the victim User which is subsequently used to shop at the impersonated *SP*'s online shop using the victim's credit card; and
- seven previously unknown vulnerabilities in a number of websites (e.g., `developer.linkedin.com`, `pinterest.com`, `websta.me`) leveraging the SSO solutions offered by Linkedin, Facebook, and Instagram.

Besides the SSO and the CaaS scenarios, we investigated a popular MPWA scenario, namely the Verificaton via Email (VvE), which is often used by websites to send security-sensitive information to users via email. By testing the security of Alexa global top 500 websites[2] we found that a number of prominent websites such as `twitter.com`, `open.sap.com` are vulnerable to login CSRF attacks. The following are the main contributions of this chapter:

1. We show that the attack strategies behind thirteen prominent MPWA attacks can be represented using seven attack patterns, and these attack patterns are general enough to discover similar attacks in MPWAs implementing different protocols and in different MPWA scenarios. For instance, an attack pattern inspired by various SSO attacks from the literature was able to automatically discover a new attack in the CaaS scenario.
2. The idea that prior attacks proposed on SSO and CaaS share commonalities is not new [97, 60]. However, ours is the first *black-box* security

[2]http://www.alexa.com/topsites

testing approach that has experimental evidence of applicability in both SSO and CaaS domains.

3. Prior work on security analysis of MPWAs is focused only on SSO and CaaS scenarios. We evaluate the MPWA scenario in which websites send security-sensitive information to users via email and show that eight out of the top Alexa global 500 websites are vulnerable to login CSRF attacks.

4. We have developed a fully functional prototype of our approach on OWASP ZAP, a widely-used open-source penetration testing tool. The tool is available online (upon request) at the companion website.[3]

5. We have been able to identify 11 previously unknown vulnerabilities in security-critical MPWAs leveraging the SSO and CaaS protocols of Linkedin, Facebook, Instagram, PayPal, and Stripe.

Structure of the chapter. In Section 2.2, we introduce some background information about MPWAs and details about various attacks from the literature. The idea of creating attack patterns from concrete attacks is explained in Section 2.3. In Section 2.4 we show how the attack patterns we defined can be used to carry out black-box testing of MPWAs. In Section 2.5, we provide some details about our prototype implementation. An illustrative example demonstrating the usage of our security testing approach is shown in Figure 2.6. We discuss the experimental evaluation in Section 2.7. In Section 2.8 we discuss the related work and in Section 2.9 we discuss the limitations and future work of the research presented in this chapter.

[3]`https://sites.google.com/site/mpwaprobe/`

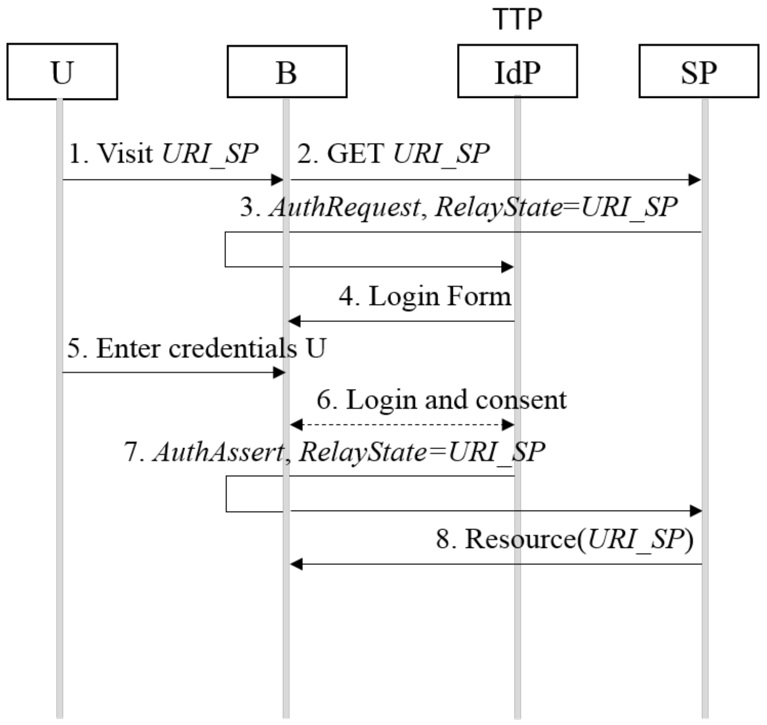


Figure 2.1: SAML-based SSO

2.2 Background

Figures 2.1, 2.2 and 2.3 provides pictorial representations of example MP-WAs leveraging SSO, CaaS, and Verification via Email (VvE) protocols. They all feature *(i)* a user U, operating a browser B, who wants to consume a service from a service provider SP and *(ii)* a service provider SP that relies on a trusted-third-party TTP to deliver its services. TLS (and valid certificates at TTP and SP) are used to securely exchange messages.

Figure 2.1 shows the SAML 2.0 SSO protocol [61], where SP relies on TTP (the Identity Provider, IdP for short) to authenticate a user U before granting the user access to one of its resources. The protocol starts (steps 1-2) with U asking SP for a resource located at URI_SP. SP in turn redirects B to IdP with the authentication request $AuthRequest$ (step 3). The $RelayState$ field carries URI_SP. IdP then challenges B to provide valid

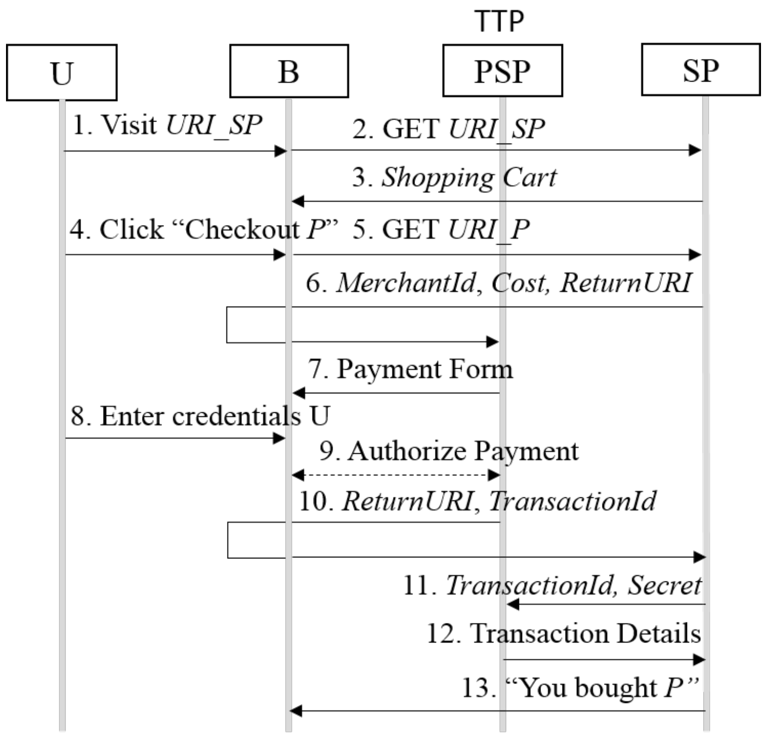


Figure 2.2: PayPal Payments Standard CaaS

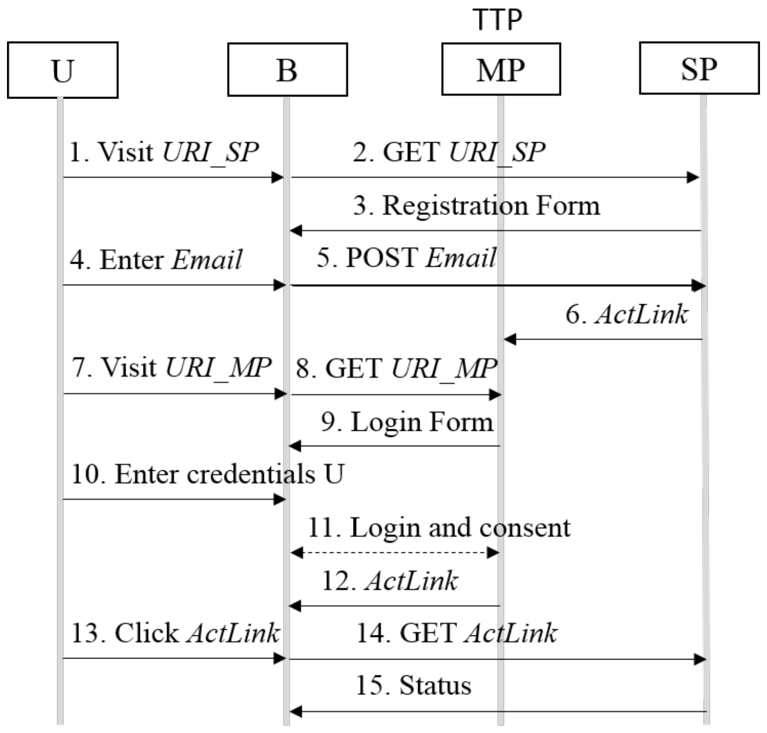


Figure 2.3: Email notification and acknowledgment

credentials that are entered by U (steps 4-6). If the authentication succeeds, *IdP* issues a digitally signed authentication assertion (*AuthAssert*) and instructs the user to sent it (along with the *RelayState*) to the *SP* (step 7). *SP* checks the assertion and delivers the requested resource (step 8).

A severe man-in-the-middle attack against the SAML-based SSO for Google Apps was reported [45]. The attack, due to a deviation from the standard whereby *AuthAssert* did not include the identity of *SP* (for which the assertion was created), allowed a malicious agent hosting a *SP* (say SP_{M}) to reuse *AuthAssert* to access the resource of the victim U (say U_{V}) stored at Google, the target *SP* (say SP_{T}). More in detail, after a session S_1 of the protocol involving U_{V} and SP_{M}, in which SP_{M} receives the *AuthAssert* from U_{V}, the malicious agent starts another session S_2 playing the role U_{M} and mischievously reuses the assertion obtained in S_1 in S_2 to trick Google (SP_{T}) into believing he is U_{V}.

Figure 2.2 illustrates a typical MPWA running the PayPal Payments Standard CaaS protocol [22] where *TTP* authorizes U to purchase a product P at *SP*. Here, *TTP* is a Payment Service Provider (*PSP*) played by PayPal. *SP* is identified by PayPal through a merchant account identifier (*PayeeId*). U places an order for purchasing P (steps 1-5). *SP* sends the *PayeeId*, the cost of the product (*Cost*) and a return URI (*ReturnURI*) to *TTP* by redirecting B (step 6). By interacting with *PSP*, U authorizes the payment of the amount to *SP* (steps 7-9). The transaction identifier (*TransactionId*) is generated by *PSP* and passed to *SP* by redirecting B to *ReturnURI* (step 10). The *TransactionId* is then submitted by *SP* to *TTP* to get the details of the transaction (steps 11-12). Upon successful verification of the transaction details, *SP* sends U the status of the purchase order (step 13).

A serious vulnerability in the integration of the PayPal Payments Standard protocol in osCommerce v2.3.1 and AbanteCart v1.0.4 that allowed a malicious party to shop for free was discovered in [82]. The attack is as follows: from a session S_1 of the protocol involving the *PSP* and the malicious party controlling both a user ($\mathrm{U_M}$) and a *SP* ($\mathrm{SP_M}$), the malicious party obtains a payee (merchant) identifier. Later, in the checkout protocol session S_2 between $\mathrm{U_M}$ and the target SP ($\mathrm{SP_T}$), the malicious agent replays the value of *PayeeId* obtained in the other session and manages to place an order for a product in $\mathrm{SP_T}$ by paying herself (instead of $\mathrm{SP_T}$).

While MPWAs for SSO and CaaS scenarios received a considerable attention (see, e.g., [60, 87, 91, 95, 94, 97, 82]), there are several other security critical MPWAs that are in need of close scrutiny. For instance, websites often send security-sensitive URIs to their users via email for verification purposes. This scenario occurs very frequently for account registration: an account activation link is sent via email to the user who is asked to access her email and click on the link contained in the email message. An illustration of this scenario is provided in Figure 2.3. Here, *TTP* is a mailbox provider *MP* that guarantees *SP* that a user *U* is in control of a given email address (*Email*). During registration, *U* provides *Email* to *SP* (steps 1-5). *SP* sends the account activation URI (*ActLink*) via email to *U*, when *U* visits her inbox at *MP* he gets access to *ActLink* (steps 6-12) and by clicking it, the status of the account activation is loaded in *U*'s browser (steps 13-15). This scenario is not just limited to account activation as the same process is followed by many *SP*s to verify the authenticity of security-critical operations such as password reset. For generality, we refer to this scenario as Verification via Email (in short, VvE). Persona, a SSO solution by Mozilla [14] was (currently no longer active) based on the VvE scenario.

Quite surprisingly, prominent SPs (e.g., `twitter.com`) do not properly perceive and/or manage the risk associated to the security-sensitive URIs sent via email to their users. It turns out that some of these URIs give direct access to sensitive services skipping any authentication step. For instance, when a user has not signed into twitter for more than 10 days, `twitter.com` sends emails to the user about the tweets the user missed and this email contains security-sensitive URIs that directly authenticates the user without asking for credentials. Such a URL can be used by an attacker to silently authenticate a victim to an attacker controlled twitter account. This attack is widely known as login CSRF.

2.2.1 Attacks

Table 2.1 presents ten prominent attacks that were discovered in literature on SSO- and CaaS-based MPWAs. It includes the two attacks mentioned above (excluding login CSRF in `twitter.com`), corresponding to #1 for SAML SSO, and #3 for PayPal Payments Standard. We do not consider here XSS and XML rewriting attacks (see Section 2.8 for details). Hereafter, we briefly describe the other attacks.

#2 The attacker hosts SP_{M} to obtain the *AccessToken* issued by the *TTP* Facebook for authenticating U_{V} in SP_{M}. The very same *AccessToken* is replayed against SP_{T} to authenticate as U_{V}.

#4 The attacker completes a transaction T_1 at SP_{T}, and the order id (*OrderId*), issued by the *TTP* PayPal for completing this transaction, is reused by the attacker to complete another transaction T_2 at SP_{M} without payment.

#5 The attacker completes a transaction T_1 at SP_{T} and the payment *Token* issued by the *TTP* PayPal for completing this transaction is reused by the attacker to complete another transaction T_2 at SP_{M} without payment. In [82], the interaction with PayPal was completely skipped during T_2. Here, we focus on the replay attack strategy used.

#6 The attacker spoofs the *AppId* of $\mathrm{SP_T}$ in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$ to obtain *AccessToken* of $\mathrm{U_V}$. The very same *AccessToken* is then replayed by the attacker in a session between $\mathrm{SP_T}$ and $\mathrm{U_M}$ to authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$. In [94], a logic flaw in flash was applied to capture the *AccessToken*. Here, we focus on the replay attack strategy used.

#7 Initially, the attacker obtains an authentication assertion (*AuthAssert*) from the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$. Then the attacker forces victim's browser to submit *AuthAssert* to $\mathrm{SP_T}$ to silently authenticate $\mathrm{U_V}$ as $\mathrm{U_M}$ at $\mathrm{SP_T}$.

#8 The attacker obtains the value of *AuthCode* during the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$. The attacker forces $\mathrm{U_V}$'s browser to submit this value to $\mathrm{SP_T}$ to silently authenticate $\mathrm{U_V}$ as $\mathrm{U_M}$ at $\mathrm{SP_T}$.

#9 The attacker replaces the value of *RedirectURI* to a malicious URI (MaliciousURI) in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$. *TTP* sends *AuthCode* of $\mathrm{U_V}$ to MaliciousURI and the attacker obtains it. The *AuthCode* is then replayed in the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$ to authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$.

#10 The attacker replaces the value of *RedirectURI* to a malicious URI (MaliciousURI) in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$. *TTP* sends *AccessToken* of $\mathrm{U_V}$ to MaliciousURI and the attacker obtains it. The *AccessToken* is then replayed in the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$ to authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$.

Table 2.1: Attacks against security-critical Multi Party Web Applications

#	Vulnerable MPWA	Description of the Attack	Attacker's Goal
1	SPs implementing Google's SAML SSO [45, §4]	Replay $\mathrm{U_V}$'s *AuthAssert* for $\mathrm{SP_M}$ in $\mathrm{SP_T}$	Authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$
2	SPs implementing OAuth 2.0 implicit flow-based Facebook SSO [96, §5.2.1]	Replay $\mathrm{U_V}$'s *AccessToken* for $\mathrm{SP_M}$ in $\mathrm{SP_T}$	Authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$
3	PayPal Payments Standard implementation in SPs using osCommerce v2.3.1 or AbanteCart v1.0.4 [82, §IV.A.1]	Replay *PayeeId* of $\mathrm{SP_M}$ during transaction T at $\mathrm{SP_T}$	Complete T at $\mathrm{SP_T}$
4	SPs implementing CaaS solutions of 2Checkout, Chrono-Pay, PSiGate and Luottokunta (v1.2) [91, §V.A]	Replay *OrderId* of transaction T_1 at $\mathrm{SP_T}$ during transaction T_2 at $\mathrm{SP_T}$	Complete T_2 at $\mathrm{SP_T}$
5	PayPal Express Checkout implementation in SPs using OpenCart 1.5.3.1 or TomatoCart 1.1.7 [82, §IV.A.2]	Replay *Token* of transaction T_1 at $\mathrm{SP_T}$ during transaction T_2 at $\mathrm{SP_T}$	Complete T_2 at $\mathrm{SP_T}$
6	SPs implementing OAuth 2.0 implicit flow-based Facebook SSO [94, §4.2]	Replay *AppId* of $\mathrm{SP_T}$ in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$ to obtain *AccessToken* of $\mathrm{U_V}$ which is then replayed to $\mathrm{SP_T}$.	Authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$
7	developer.mozilla.com (SP) implementing BrowserID [49, §6.2]	Make $\mathrm{U_V}$ browser send request to $\mathrm{SP_T}$ with $\mathrm{U_M}$'s *AuthAssert*	Authenticate as $\mathrm{U_M}$ at $\mathrm{SP_T}$
8	CitySearch.com (SP) using Facebook SSO (OAuth 2.0 Auth. Code Flow) [50, §V.C]	Make $\mathrm{U_V}$ browser send request to $\mathrm{SP_T}$ with $\mathrm{U_M}$'s *AuthCode*	Authenticate as $\mathrm{U_M}$ at $\mathrm{SP_T}$
9	Github (TTP) implementing OAuth 2.0 Authorization Code flow-based SSO [1, Bug 2]	Replace the value of *RedirectURI* to MaliciousURI in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$ to obtain *AuthCode* of $\mathrm{U_V}$ and replay this *AuthCode* in the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$	Authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$
10	*SP*s implementing Facebook SSO [2]	Replace the value of *RedirectURI* to MaliciousURI in the session between $\mathrm{U_V}$ and $\mathrm{SP_M}$ to obtain *AccessToken* of $\mathrm{U_V}$ and replay this *AccessToken* in the session between $\mathrm{U_M}$ and $\mathrm{SP_T}$	Authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$

2.2.2 Threat Models

The attacks shown in Table 2.1 can be discovered by considering the *Web Attacker* threat model introduced in [44] and outlined hereafter according to our context:

Web Attacker. He/She can control a *SP* (referred to as the $\mathrm{SP_M}$) that is integrated with a *TTP*. The $\mathrm{SP_M}$ can subvert the protocol flow (e.g., by changing the order and value of the HTTP requests/responses generated from her *SP*, including redirection to arbitrary domains). The *web attacker* can also operate a browser and communicate with other *SPs* and *TTPs*. Notice also that none of the attacks discussed requires the threat scenario in which the *TTP* can be played by the attacker [77]. We do not consider this threat scenario. However, we will show in Sections 2.4.1 and 2.5.2 that we also had to deal with the *network attacker* and the *browser history attacker* threat models.

2.3 From Attacks to Attack Patterns

A close inspection of the attacks in Table 2.1 reveals that:

1. they leverage a small number of nominal sessions of the MPWA under test, namely those played by $\mathrm{U_V}$, $\mathrm{U_M}$, $\mathrm{SP_T}$, and $\mathrm{SP_M}$, which we concisely represent by $(\mathrm{U_V}, \mathrm{SP_T})$, $(\mathrm{U_M}, \mathrm{SP_T})$, $(\mathrm{U_V}, \mathrm{SP_M})$, $(\mathrm{U_M}, \mathrm{SP_M})$.[4]
2. they amount to combining sessions obtained by tampering with the messages exchanged in one nominal session or by replacing some message from one nominal session into another.

By *session* we mean any sequence of HTTP requests and responses corresponding to an execution of the MPWA under test. Our goal is to identify recipes, called *attack patterns*, that specify how nominal sessions can be tampered with and combined to find attacks on MPWAs. We start by identifying and comparing attack strategies for the attacks in Table 2.1 and then we abstract them into general, i.e. application-independent, attack patterns.

Attack strategies are built on top of the following three operations:

[4]For the sake of simplicity we leave *B* and the *TTP* implicit since we identify the browser with the user. The *TTP*, according to the threat model considered, is assumed to be trustworthy.

- REPLAY x FROM S_1 IN S_2: indicating that the value of the HTTP element x extracted while executing session S_1 is replayed into session S_2;
- REPLACE x WITH v IN R: denoting that the HTTP element x (e.g., `SID`) is replaced with the value v (e.g., `abcd1234`) while executing the sequence of HTTP requests R; and
- REQUEST-OF x FROM R: indicating the extraction of the HTTP request transporting the HTTP element x while executing the sequence of HTTP requests R.

Table 2.2: Known Attacks Strategies against MPWAs

Id	Attack Strategy
1	REPLAY *AuthAssert* FROM (U_V, SP_M) IN (U_M, SP_T)
2	REPLAY *AccessToken* FROM (U_V, SP_M) IN (U_M, SP_T)
3	REPLAY *PayeeId* FROM (U_M, SP_M) IN (U_M, SP_T)
4	REPLAY *OrderId* FROM (U_M, SP_T) IN (U_M, SP_T)
5	REPLAY *Token* FROM (U_M, SP_T) IN (U_M, SP_T)
6	REPLAY *AccessToken* FROM S IN (U_M, SP_T) where S = REPLAY *AppId* FROM (U_M, SP_T) IN (U_V, SP_M)
7	REPLACE x WITH REQUEST-OF *AuthAssert* FROM (U_M, SP_T) IN $[U_V$ SEND $x]$
8	REPLACE x WITH REQUEST-OF *AuthCode* FROM (U_M, SP_T) IN $[U_V$ SEND $x]$
9	REPLAY *AuthCode* FROM S IN (U_M, SP_T) where S = REPLACE *RedirectURI* WITH MALICIOUSURI IN (U_V, SP_T)
10	REPLAY *AccessToken* FROM S IN (U_M, SP_T) where S = REPLACE *RedirectURI* WITH MALICIOUSURI IN (U_V, SP_T)

For the sake of simplicity, we present in the overall chapter the replay of a single element, but our attack patterns actually support simultaneous replay of combinations of elements. By abusing the notation, we use (U, SP)

in place of R to indicate the sequence of HTTP requests underlying the session (U, SP).

The attack strategies corresponding to the attacks described in Table 2.1 are given in Table 2.2.

In attack strategy #1 (and #2), the attacker runs a session with the victim user $\mathrm{U_V}$ playing the role of the service provider $\mathrm{SP_M}$ and replays $AuthAssert$ ($AccessToken$, resp.) into a new session with a target service provider $\mathrm{SP_T}$. The attacker tries thus to impersonate the victim ($\mathrm{U_V}$) at $\mathrm{SP_T}$.

Attack strategy #3 is analogous to the previous ones, the difference being that the user role in the first session is played by the malicious user and the replayed element is $PayeeId$. Here the goal of the attacker is to use credits generated by TTP, in the first session, for $\mathrm{SP_M}$ on $\mathrm{SP_T}$.

Attack strategy #4 (and #5) differs from the previous ones in that the $User$ and the SP roles are played by $\mathrm{U_M}$ and $\mathrm{SP_T}$ respectively in both sessions. In doing so, the attacker aims to "gain" something from $\mathrm{SP_T}$ by re-using the $Token$ ($OrderId$, resp.) obtained in a previous session with the same $\mathrm{SP_T}$.

Attack strategy #6 is the composition of two basic replay attack strategies. The element $AppId$, obtained by running a session between the victim user $\mathrm{U_V}$ and the malicious service provider $\mathrm{SP_M}$, is replayed to get the $AccessToken$ which is then in turn replayed by the attacker $\mathrm{U_M}$ to authenticate as $\mathrm{U_V}$ at $\mathrm{SP_T}$. Thus, the result should be the same obtained by completing a session between $\mathrm{U_V}$ and $\mathrm{SP_T}$.

In attack strategy #7 (and #8), the HTTP request (cf. REQUEST-OF keyword) transporting $\mathrm{U_M}$'s $AuthAssert$ ($AuthCode$, resp.) for $\mathrm{SP_T}$ is replaced on a sequence comprising a single HTTP request in which $\mathrm{U_V}$ sends a HTTP request to $\mathrm{SP_T}$ (denoted as [$\mathrm{U_V}$ SEND req]). This single HTTP request is the representation of the cross-site request sent by the attacker's web page loaded at the U_V's web browser. If the attack is successful, then the result should be the same obtained by completing a session between U_M and SP_T.

In attack strategy #9 (and #10), the attacker includes a malicious URI (MaliciousURI) in the session between U_V and SP_T. In doing so, the credential *AuthCode* (*AccessToken*, resp.) is received by the attacker. By replaying this intercepted *AuthCode* (*AccessToken*, resp.) in the session between U_M and SP_T, the attacker aims to authenticate as U_V in SP_T. Thus, the result should be the same obtained by completing a session between U_V and SP_T.

We have distilled the attack strategies in Table 2.2 into a small set of general, i.e. application-independent, attack patterns which are summarized in Table 2.3. To illustrate, consider the attack pattern RA1. This pattern has been obtained from attack strategy #1 (#2) in Table 2.2 by abstracting the element to replay, i.e. *AuthAssert* (*AccessToken*, resp.) into a parameter x.

The generation of all other attack patterns go along the same lines. For the creation of the attack pattern LCSRF we were clearly inspired by attacks #7 and #8. It turns out that this attack pattern is a bit more general than what it was created for. In fact, it can uncover general CSRF based on POST requests. An example of this will be discussed in Section 2.6 (see Table 2.9).

A key step in the execution of an attack pattern is the selection of the elements to be replaced or replayed. For instance, when executing RA1 against a given MPWA, the parameter x can be instantiated with any element occurring in the HTTP trace resulting from the execution of (U_V, SP_M). Trying them all is clearly not acceptable. To tackle the problem, we inspect the sessions and enrich the elements occurring in the HTTP trace with syntactic, semantic, location and flow labels whose mean-

Table 2.3: Attack Patterns

Name	Attack Strategy	Precondition	Postcondition
RA1	REPLAY x FROM $(\mathrm{U_V}, \mathrm{SP_M})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$	$(\mathrm{TTP\text{-}SP} \in x.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{UU}) \in x.\mathsf{labels})$	$F(\mathrm{U_V}, \mathrm{SP_T})$
RA2	REPLAY x FROM $(\mathrm{U_M}, \mathrm{SP_M})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$	$(\mathrm{SP\text{-}TTP} \in x.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{AU}) \in x.\mathsf{labels})$	$F(\mathrm{U_M}, \mathrm{SP_T})$
RA3	REPLAY x FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$	$(\mathrm{TTP\text{-}SP} \in x.\mathsf{flow}$ AND $\mathrm{SU} \in x.\mathsf{labels})$	$F(\mathrm{U_M}, \mathrm{SP_T})$
RA4	REPLAY y FROM S IN $(\mathrm{U_M}, \mathrm{SP_T})$ where	$(\mathrm{SP\text{-}TTP} \in x.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{AU}) \in x.\mathsf{labels}$ AND	$F(\mathrm{U_V}, \mathrm{SP_T})$
	$S =$ REPLAY x FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN $(\mathrm{U_V}, \mathrm{SP_M})$	$\mathrm{TTP\text{-}SP} \in y.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{UU}) \in y.\mathsf{labels})$	
RA5	REPLAY x FROM $(\mathrm{U_V}, \mathrm{SP_T})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$	$(\mathrm{TTP\text{-}SP} \in x.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{UU}) \in x.\mathsf{labels}$ AND	$F(\mathrm{U_V}, \mathrm{SP_T})$
		$x.\mathsf{location} = \textsc{request}\mathrm{URL})$	
LCSRF	REPLACE req WITH REQUEST-OF y	$(\mathrm{TTP\text{-}SP} \in y.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{UU}) \in y.\mathsf{labels})$	$F(\mathrm{U_M}, \mathrm{SP_T})$
	FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN $[\mathrm{U_M}$ SEND $req]$		
RedURI	REPLAY y FROM S IN $(\mathrm{U_M}, \mathrm{SP_T})$	$(\mathrm{SP\text{-}TTP} \in x.\mathsf{flow}$ AND $\mathrm{RURI} \in x.\mathsf{labels})$ AND	$F(\mathrm{U_M}, \mathrm{SP_T})$
	where $S =$ REPLACE x WITH x' IN $(\mathrm{U_V}, \mathrm{SP_T})$	$\mathrm{TTP\text{-}SP} \in y.\mathsf{flow}$ AND $(\mathrm{SU}\|\mathrm{UU}) \in y.\mathsf{labels})$	

Legend: The notation $(x|y) \in S$ is used to abbreviate $(x \in S$ OR $y \in S)$.

ing is summarized in Tables 2.4 and 2.5. The preconditions in Table 2.3 determine how these elements are selected for each pattern.

For instance, since RA1 is a replay attack that tries to replay an element from $(\mathrm{U_V}, \mathrm{SP_M})$ to $(\mathrm{U_M}, \mathrm{SP_T})$, it is reasonable to replay only those elements that flow from *TTP* to *SP*, i.e. data flow label TTP-SP. Indeed, these are the ones that are likely to comprise specific values that *TTP* issues for $\mathrm{U_V}$. In addition, it would make little sense to replay elements whose values do not change over different traces. This is why that pattern selects only elements in the trace that are tagged either as session unique (SU) or user unique (UU) (the users are different among the sessions where the replay takes place). The precondition of RA2 is analogous to that of RA1, but since RA2 replays an element from $(\mathrm{U_M}, \mathrm{SP_M})$ to $(\mathrm{U_M}, \mathrm{SP_T})$, then that element must flow from *SP* to *TTP*. Similar reasoning holds for other attack patterns. Notice that for RedURI pattern (which is inspired by attacks #9 and #10 of Table 2.1), we consider only the URLs that are chosen by the $\mathrm{SP_T}$, but can be changed by the users (see definition of RURI label in Table 2.4). We discuss how we implemented the core logic of this attack pattern— i.e. replacing a URL (shown as x in row 6, column 2 of Table 2.3) to its malicious form (shown as *x'* in row 6, column 2 of Table 2.3)— in Section 2.5.

The attack pattern replace these URLs (generalized as x in Table 2.3) with its malicious form (shown as *x'* in row 6, column 2 of Table 2.3). We achieve this by replacing https URLS into http so that a *web attacker* residing in the same network as the victim user and can read all the unencrypted messages (hereinafter referred to as the network attacker) can read the

In Table 2.3, we have also introduced a new attack pattern named RA5 which is inspired by the "credential leak in browser history" threat model which is mentioned in the OAuth 2.0 threat model and security considerations document [42]. According to this threat model, $\mathrm{U_M}$ and $\mathrm{U_V}$ share the same browser. In the attack strategy, $\mathrm{U_M}$ replays (to $\mathrm{SP_T}$) the HTTP elements that are issued by the *TTP* to $\mathrm{SP_T}$ for $\mathrm{U_V}$. Notice that in the preconditions it is mentioned that the security critical parameters which are used in this attack strategy must be located in the request URL. The request URLs of a browsing session are likely to be stored in the browser history. Last, but not least, attack patterns need a way to automatically determine whether the attack strategy they executed was successful to detect any attack. The postconditions included in Table 2.3 serve this purpose. The idea is that each one of the four nominal sessions is associated with a *Flag* (an oracle) that defines what determines the successful completion of it. For instance, a string "Welcome Victim" could be the *Flag* for the nominal session $(\mathrm{U_V}, \mathrm{SP_T})$ of a MPWA implementing a SSO solution (assuming that "Victim" is the name provided by $\mathrm{U_V}$ at $\mathrm{SP_T}$). The concept of *Flag* will be further clarified in the next section. The postcondition is just an oracle program that checks whether a certain *Flag* is captured or not while executing the strategy. It corresponds to checking whether the goal of performing an attack (see column 3 of Table 2.1) has been met or not. A value of the form $F(U, SP)$ in the column Postcondition (of Table 2.3) stands for this program checking for the *Flag* associated with (U, SP).

It must be noticed that the definition of postcondition depends on the specific MPWA under test.

Table 2.4: Syntactic and Semantic Labels

Type	**Label**	**Short Description**	**Example**
Syntactic[†]	URL	a Uniform Resource Locator	`redirect_uri=http://google.com`
	BLOB	an alphanumeric string with (optional) special characters	`code=vrDK7rE4`
	WORD	a string comprised only of alphabetic characters	`response_type=`token
	EMAIL	an email address	`usrname=jdoe@example.com`
	EMPTY	an empty value	`state=`
	NUMBER	a number	`id=5`
	BOOL	a boolean value	`new=true`
	UNKNOWN	none of the other syntactic labels match this string	`#target`.
Semantic[‡]	SU	SU stands for Session Unique, meaning the element is assigned different values in different sessions	*AuthAssert* of Figure 2.1
	UU	UU stands for User Unique, meaning the element is assigned the same value in the sessions of the same user	*username* and *password* of a user
	AU	AU stands for App Unique, meaning the element is assigned the same value in the sessions of a single *SP*	*MerchantId* of Figure 2.2
	MAND	MAND stands for Mandatory, meaning the element must occur in the HTTP Traffic for the protocol to complete successfully	*AuthAssert* of Figure 2.1
	RURI	RURI stands for Redirect URI, meaning the element must be MAND, it must be a URL that is passed as a parameter in a request uri and it is later found in the Location header of a redirection response	*ReturnURI* of Figure 2.2
Data flow*	SP-TTP	it means that the corresponding element has been received from *SP* and then sent to *TTP*	*MerchantId* of Figure 2.2
	TTP-SP	it means that the corresponding element has been received from *TTP* and then sent to *SP*	*AuthAssert* of Figure 2.1

[†] Syntactic labels provide type information. Most of the syntactic labels presented here are borrowed from [94, 82]

[‡] Semantic labels provide information on the role played by the element within the MPWA. While the SU and UU labels are borrowed from [94], the AU and RURI labels are new. The MAND label generalizes the SEC label introduced in [94], where it was used to indicate a secret specific to the current session and necessary for the success of the authentication, while here MAND is not just secret and SU.

Table 2.5: Data Flow and Location Labels

Type	Label	Short Description	Example
Data flow*	SP-TTP	it means that the corresponding element has been received from *SP* and then sent to *TTP*	*MerchantId* of Figure 2.2
	TTP-SP	it means that the corresponding element has been received from *TTP* and then sent to *SP*	*AuthAssert* of Figure 2.1
Location†	REQUEST.URI	this label is given for elements that are present in the URL of HTTP requests	the element `code=aXyz` of the URL `https://sp.com?code=aXyz`
	REQUEST.HEADER	it means that the corresponding element is a header in a HTTP request	`X-CSRF-Token: ejkX`
	REQUEST.BODY	meaning the element is present in the body of a HTTP request	*Email* of Step 5 of Figure 2.3
	RESPONSE.HEADER	it means that the corresponding element is present in the header of a HTTP response	`Location: www.sp.com?code=XyZi`
	RESPONSE.BODY	this label is given to elements present in the body of a HTTP response	the element `token: ‘‘XcJ"` of `{uid: ‘‘23e", token: ‘‘XcJ"}`

* Flow labels represent the data flow properties of an element in the HTTP traffic. Currently we have two flow labels: TTP-SP and SP-TTP. Label TTP-SP (SP-TTP, resp.) means that the corresponding element has been received from *TTP* (*SP*, resp.) and then sent to *SP* (*TTP*, resp.).

† Location labels denotes the location in the HTTP Message where the element has been found. The labels that we use are REQUESTURI, REQUESTHEADER, REQUESTBODY, RESPONSEHEADER and RESPONSEBODY indicating the location of the element as request URI, request header, request body, response header and response body respectively.

2.4 Approach

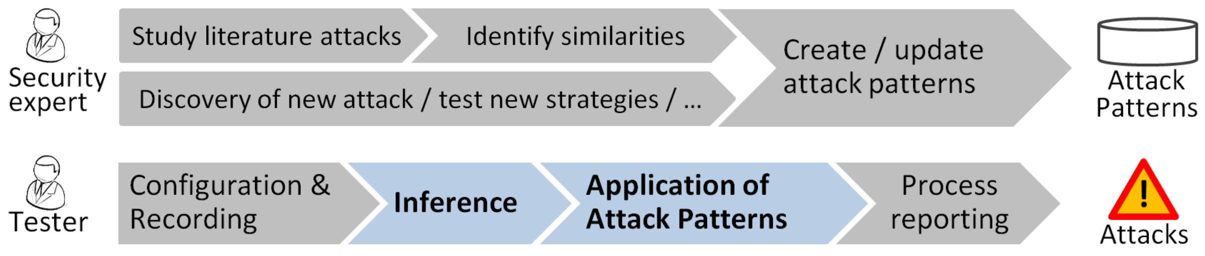

Security expert
Study literature attacks
Identify similarities
Discovery of new attack / test new strategies / ...
Create / update attack patterns
Attack Patterns
Tester
Configuration & Recording
Inference
Application of Attack Patterns
Process reporting
Attacks

Figure 2.4: Approach

Figure 2.4 outlines the two processes underlying our approach. In the first one, executable attack patterns are created, reviewed, and improved by security experts (see Section 2.4.1). The second process enables testers to identify security issues in their MPWAs. In a nutshell, the testers (e.g., developers of a MPWA) take advantage of the *security knowledge* embedded within the executable attack patterns. We will see that what is requested to testers is not much more of what they have to do anyhow in order to test the business logic of their MPWAs. See Section 2.4.2 for details.

2.4.1 Creating, reviewing, and improving Attack Patterns

Working on our attack patterns require web application security knowledge and implementation skills. Security experts, in particular those who perform penetration testing of web applications, clearly have both. Security experts can thus read and understand attack patterns like those sketched in Table 2.3. Improving an attack pattern, by changing few things here and there to e.g., make it a bit more general, is also a straightforward follow-up step. Creation of attack patterns asks for some more effort and, more importantly, for inspiration. As discussed in Section 2.3, with the only exception of RA5, all attack patterns in Table 2.3 have been inspired by attacks reported in literature. The discovery of a previously unknown attack not yet covered by our *catalog* of attack patterns is, of course, another source of inspiration. In general, security experts can craft attack patterns capturing novel attack strategies to explore new types of attacks. This is the case for attack pattern RA5, which we developed to explore the "credential leak in browser history" threat model (e.g., see [42, §4.4.2.2]). This threat model, referred to as the *browser history attacker*, is important because browsers can be shared (e.g., public libraries, internet cafes). To the best of our knowledge, we are the first to include this threat model in a black-box security testing approach.

A *browser history attacker* shares the same browser with other *Users*. It is assumed that the user does not always clear her browser history, but she properly signs out from her login sessions. The attack pattern RA5 leverages this threat model by replaying all the elements issued by the *TTP* that the attacker can collect from the browser history of the victim. As we will see in Section 2.7, by using this threat model, we have been able to detect two attacks that could not be discovered automatically using other state-of-the-art black-box security testing techniques.

2.4.2 Security Testing Framework

The different phases of our security testing framework are described below. Additionally, an illustrative example is provided in Section 2.6.

(P1) Configuration. The tester configures the testing environment so to be able to collect traces for the four nominal sessions: $S_1 = (\mathrm{U_V}, \mathrm{SP_T})$, $S_2 = (\mathrm{U_M}, \mathrm{SP_T})$, $S_3 = (\mathrm{U_V}, \mathrm{SP_M})$, and $S_4 = (\mathrm{U_M}, \mathrm{SP_M})$. To this end, the tester creates two user accounts, $\mathrm{U_V}$ and $\mathrm{U_M}$, in her service provider $\mathrm{SP_T}$ and in a reference implementation $\mathrm{SP_M}$ (the purpose of $\mathrm{SP_M}$ is to represent the *SP* controlled by the malicious party). Notice that, this step does not require a strong security background and normally does not add-up any additional cost for the tester that wants to functionally test her MPWA. All major *TTP*s provide reference implementations—e.g., [10, 9, 15, 7]—to foster adoption of their solutions. In case a working official reference implementation is not available, another *SP* (running the same protocol) can be used.

(P2) Recording. In order to enable the testing engine to automatically collect the necessary HTTP traffic, the tester records the user actions (*UAs* for short) corresponding to sessions S_1 to S_4. This amount to collecting the actions $\mathrm{U_V}$ and $\mathrm{U_M}$ perform on the browser *B* while running the protocol with $\mathrm{SP_T}$ and $\mathrm{SP_M}$. Additionally, for each sequence of *UAs*, the tester must also identify a *Flag*, i.e. a regular expression representing a pattern in the HTTP traffic which can be used to determine the successful execution of the user actions. *Flag*s must be different between each other so to be able to ensure which session was completed without any ambiguity. Standard web browser automation technologies such as Selenium WebDriver [28] and Zest [35] can be used for recording *UAs*. Such technology could be extended to allow the tester to define *Flags* by simply clicking on the web page elements (e.g., the payment confirmation form) that identify the completion of the user actions. Off-the-shelf market tools already implement this kind of feature to determine the completion of the login operation.

(P3) Inference. The inference module automatically executes the nominal sessions recorded in the previous phase and tags the elements in the resulting HTTP traffic with the labels in Tables 2.4 and 2.5. We do not exclude that in the future more information (e.g., inference of the observable workflow of the MPWA [82]) could be necessary to target more complex attacks. While we borrow the idea of inferring the syntactic and semantic properties from [94] and [82], we introduce the concept of inferring flow labels to make our approach more automatic (compared to [94]) and efficient (less no. of test cases for detecting the same attack mentioned in [82]).

The inference results of sessions S_1 to S_4 are stored in a data structure named labeled HTTP trace.

(P4) Application of Attack Patterns. Labeled HTTP traces (output of inference) are used to determine which attack patterns shall be applied and corresponding attack test cases are executed against the MPWA.

(P5) Reporting. Attacks (if any) are reported back to the tester and the tester evaluates them.

2.5 Implementation

We implemented our approach on top of OWASP ZAP (ZAP, in short). In this way, the two core phases of our testing engine (cf. **P3** and **P4** in previous section) are fully automated and take advantage of ZAP to perform common operations such as execution of *UAs*, manipulating HTTP traffic using proxy rule, regular expression matching over HTTP traffic, etc. Figure 2.5 outlines the high-level architecture of our testing engine.[5]

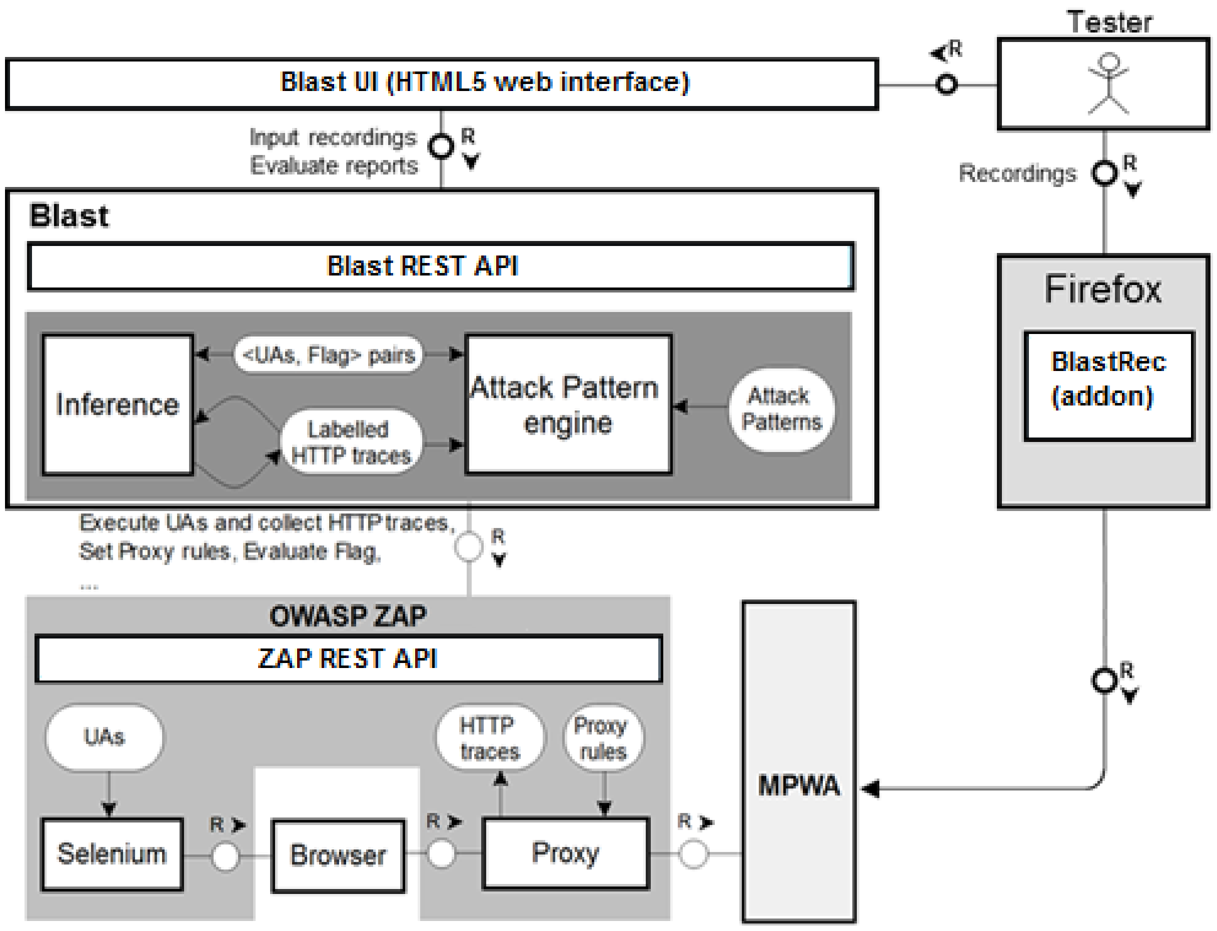


Figure 2.5: Testing Engine Architecture

[5]The “R” with the small arrow in Figure 2.5 is a short notation of the request-response channel pair that clarifies who are the requester and the responder of a generic service.

An overview of the purpose of each component (developed by us and presented in Figure 2.5), along with their implementation details are shown in Table 2.6. Hereinafter we discuss them in detail. The *Tester* records the *UAs* and *Flags* via BlastRec (our extension of the Selenium IDE firefox plugin [27]) for running a test. The recorded *UAs* must be stored as Python Web Driver scripts [29]. Our tool also support *UAs* written as Zest scripts [35]. The recorded *UAs* and *Flags* are provided to the testing engine via a HTML-5 graphical user interface (namely Blast UI). This interface is available in the `localhost` of the computer in which the testing engine is installed and the tester can visit this interface via a web browser. The *Testing Engine* employs *OWASP ZAP* to probe the *MPWA*. In particular, the *Testing Engine* invokes the API exposed by ZAP to perform the following operations:

- (**Execute user actions and collect HTTP traces.**) *UAs*, expressed as Zest script, can be executed via the Selenium WebDriver module in ZAP and the corresponding HTTP traffic can be collected from ZAP.

- (**Proxy rule setting.**) Proxy rules can be specified, as Zest scripts, to mutate HTTP requests and response passing through the built-in proxy of ZAP.

- (**Evaluate *Flag*.**) Execute regular expression-based pattern matching within the HTTP traffic so to, e.g., evaluate whether the *Flag* is present in the HTTP traffic.

Hereafter, we detail the two core phases (**P3** and **P4**) of our *Testing Engine* that follow the flow depicted in Figure 2.6. Each step is tagged by a number to simplify the presentation of the flow.

Table 2.6: Core Components of Our Testing Engine

Component	Short Description	Implementation
BlastRec	Our extension of Firefox Selenium IDE plug-in [27] that enables a tester to record the *UAs* and *Flags* necessary for running a test (see **P1** of Section 2.4.2 for details). The recorded *UAs* are stored as Python Web Driver scripts.	*Javascript, CSS, Mozilla's XML UI Language* [40]
Blast UI	HTML5-based Graphical User Interface (GUI) that simplifies the interaction between the tester and Blast. Through this GUI the tester can *(i)* launch a new test campaign by providing the necessary *UAs* and *Flags*, *(ii)* check the progress of a running test, *(iii)* have a dashboard-style overview of completed tests (for instance, view the results of inference and application of attack patterns phases), *(iv)* re-run a particular attack test case of an attack pattern (e.g., for attack verification), *(v)* re-run a particular phase (e.g., running only the application of attack patterns phase after patching the MPWA under test for an attack discovered in a previous test campaign with Blast).	*SAPUI5* [37]
Blast REST API	Mediates the communication between Blast UI and the back-end of Blast which mainly consists of the inference and attack pattern engine.	*Python 2.7*
Inference Engine	Executes *UAs* to collect HTTP traffic from OWASP ZAP for inferring the properties (defined in Tables 2.4 and 2.5) of HTTP elements	*Python 2.7*
Attack Pattern Engine	Executes the attack strategies shown in Table 2.3 by manipulating the HTTP traffic by setting proxy rules in OWASP ZAP via ZAP's REST API.	*Python 2.7*

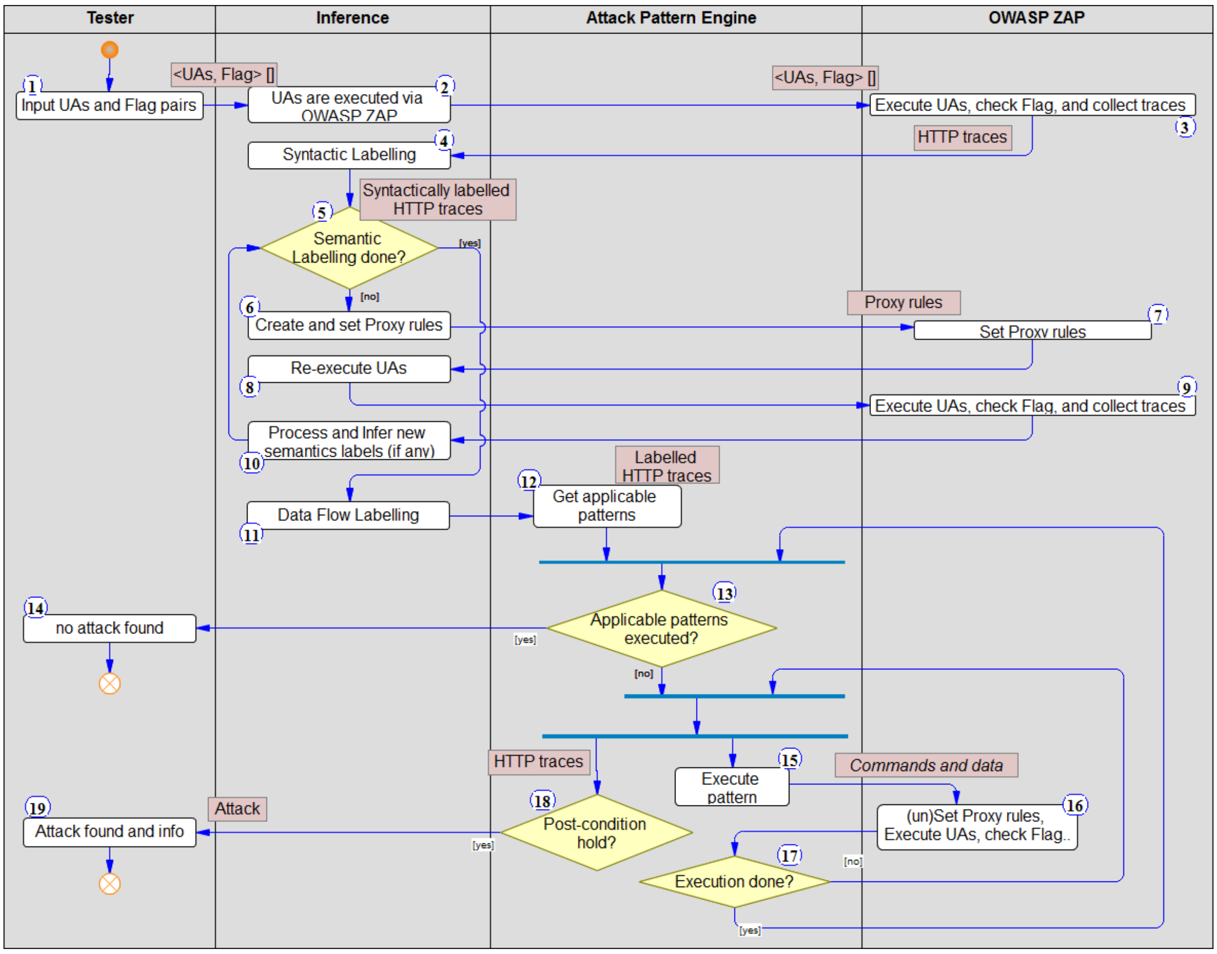

Tester
Inference
Attack Pattern Engine
OWASP ZAP
1 Input UAs and Flag pairs
<UAs, Flag> []
2 UAs are executed via OWASP ZAP
<UAs, Flag> []
3 Execute UAs, check Flag, and collect traces
HTTP traces
4 Syntactic Labelling
Syntactically labelled HTTP traces
5 Semantic Labelling done?
[yes]
[no]
6 Create and set Proxy rules
Proxy rules
7 Set Proxy rules
8 Re-execute UAs
9 Execute UAs, check Flag, and collect traces
10 Process and Infer new semantics labels (if any)
Labelled HTTP traces
12 Get applicable patterns
11 Data Flow Labelling
14 no attack found
13 Applicable patterns executed?
[yes]
[no]
HTTP traces
15 Execute pattern
Commands and data
16 (un)Set Proxy rules, Execute UAs, check Flag..
19 Attack found and info
Attack
18 Post-condition hold?
[yes]
17 Execution done?
[no]
[yes]

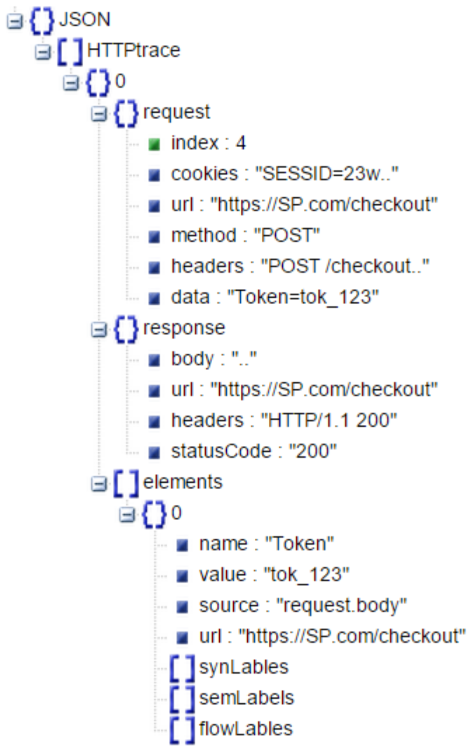
Figure 2.6: Testing Engine Flow

2.5.1 Inference

With reference to the steps of Figure 2.6, the following activities are performed by the inference module after the tester inputs (step 1) the four $\langle UAs, Flag \rangle$ corresponding to sessions S_1, S_2, S_3, and S_4 in **(P2)**.

Trace collection (steps 2-3) The input UAs are executed and corresponding HTTP traces are collected. The *Flags* are used to verify whether the collected traces are complete. We represent the collected HTTP traces as $HT(S_1)$, $HT(S_2)$, $HT(S_3)$, and $HT(S_4)$. The traces are stored as an array of $\langle request, response, elements \rangle$ triplets. Each triplet comprises the

HTTP *request* sent via ZAP to the MPWA, the corresponding HTTP *response*, and details about the HTTP *elements* exchanged. An excerpt of a trace related to our illustrative example (Figure 2.8) is depicted in Figure 2.7 in JSON format. For simplicity, we present only one entry of the trace array and only one HTTP element. We assume the reader is familiar with standard format of the HTTP protocol. Here we focus on the HTTP elements. For each of them we store the name (“`name`”), the value (“`value`”), its location in the request/response (“`source`”, e.g., `source:"request.body"` indicates that the element occurs in the request body of the HTTP request), the associated request URL (“`url`”), its data flow patterns, syntactic and semantic labels that are initially empty and will be inferred in the next activities. For instance, the element illustrated in Figure 2.7 is the *Token* shown in step 10 of Figure 2.8.



```
{} JSON
  [] HTTPtrace
    {} 0
      {} request
          index : 4
          cookies : "SESSID=23w.."
          url : "https://SP.com/checkout"
          method : "POST"
          headers : "POST /checkout.."
          data : "Token=tok_123"
      {} response
          body : ".."
          url : "https://SP.com/checkout"
          headers : "HTTP/1.1 200"
          statusCode : "200"
      [] elements
        {} 0
            name : "Token"
            value : "tok_123"
            source : "request.body"
            url : "https://SP.com/checkout"
          [] synLables
          [] semLabels
          [] flowLables
```

Figure 2.7: HTTP trace with empty labels (an excerpt)

Syntactic and Semantic Labeling (steps 4-10) The collected HTTP traces are inspected to infer the syntactic and semantic properties of each HTTP element, reported in Table 2.4. While syntactic labeling is carried out by matching the HTTP elements against simple regular expressions, semantic labeling may require (e.g., for MAND) active testing of the MPWA. For instance, to check whether an element e occurring in $HT(\mathrm{U_M}, \mathrm{SP_T})$ is to be given the label MAND, the inference module generates a proxy rule that removes e from the HTTP requests (step 6). By activating this proxy rule (step 7), the inference module re-executes the *UA* corresponding to the session $(\mathrm{U_M}, \mathrm{SP_T})$ and checks whether the corresponding *Flag* is present in the resulting trace (steps 8-9). For instance, the element $Token$ (see Figure 2.7) is assigned the syntactic labels BLOB and the semantic labels SU and MAND.

Data Flow Labeling (step 11) After syntactic and semantic labeling, the data flow properties of each MAND element in the trace are analyzed to identify the data flows (either TTP-SP or SP-TTP). In order to identify the protocol patterns, it is necessary to distinguish *TTP* and *SP* from the HTTP trace. We do this by identifying the common domains present in the HTTP trace of the two different *SPs* ($\mathrm{SP_T}$ and $\mathrm{SP_M}$) implementing the same protocol and classifying the messages from/to these domains as the messages from/to *TTP*.

The output of the inference phase is the labeled HTTP traces of sessions S_1 to S_4 (represented as $LHT(S_1)$, $LHT(S_2)$, $LHT(S_3)$, and $LHT(S_4)$).

2.5.2 Attack Pattern Engine

For the simplicity of explanation, we represent our attack patterns in the same way as the attack graph notation introduced in [83]. Each attack pattern has a `Name`, the underlying `Threat model`, `Inputs` used, the `Goal`

Listing 2.1: Attack Pattern for RA1

```
Name:  RA1
Threat Model:  Web Attacker
Inputs:    UAs($U_V$,$SP_M$), LHT($U_V$,$SP_M$),
           UAs($U_M$,$SP_T$), Flag($U_V$,$SP_T$)
Preconditions:  At least one element x in LHT($U_V$,$SP_M$)
  is such that (TTP-SP $\in$ x.flow AND (SU|UU) $\in$x.labels)
Actions:
  For each x such that preconditions hold
  e = extract(x,UAs($U_V$,$SP_M$))
  HTTP_logs = replay(x,e,UAs($U_M$,$SP_T$))
  Check Postconditions;
Postconditions:Check Flag($U_V$,$SP_T$) in HTTP_logs
               Report(e,UAs($U_M$,$SP_T$),Flag($U_V$,$SP_T$))
```

the attacker (who follows the attack strategy defined in the pattern) aims to achieve, `Preconditions`, `Actions` and `Postconditions`. The `Inputs` to the attack pattern range over the *LHTs* (labeled HTTP traces generated by the inference module), *UAs* of the nominal sessions, and the corresponding *Flags*. The `Goal`, `Preconditions`, `Actions` and `Postconditions` are built on top of the `Inputs`. The pattern is applicable if and only if its `Preconditions` hold (steps 12-14 of Figure 2.6). As soon as the pattern `Preconditions` hold, the `Actions` are executed (steps 15-17 of Figure 2.6). The `Actions` contain the logic for generating proxy rules that mimics the attack strategy. The generated proxy rules are loaded in ZAP and *UAs* are executed. The execution of *UAs* generates HTTP requests and responses. The proxy rules manipulates the matching requests and responses. As last step of the `Actions` execution, the `Postconditions` are checked. If they hold (step 18 of Figure 2.6), an attack report is generated with the configuration that caused the attack (step 19 of Figure 2.6).

Example on Attack Pattern for RA1. To illustrate, let us consider the

Listing 2.2: Extract function

```
value extract(id x, uas UAs){
  rb = generate_break_rule(x)
  load_rule_ZAP(rb)
  HTTP_logs = execute_ZAP(UAs)
  e = extract_value(x, HTTP_logs)
  clear_rules_ZAP
  return e}
```

Replay Attack pattern RA1 reported in Table 2.3. In Listing 2.1, we show the pseudo-code describing it.

The `Threat model` considered is the *web attacker*. To evaluate the applicability of the pattern, the output of the inference phase is sufficient ($LHT(\mathrm{U_V}, \mathrm{SP_M})$): the attack pattern is executed in case at least one element x has the proper data flow and semantic label (lines 6-7). For each selected element x (line 9), the function extract(x, UAs($\mathrm{U_V}$,$\mathrm{SP_M}$)) (line 10) executes UAs($\mathrm{U_V}$, $\mathrm{SP_M}$), returning the value e associated with x. This value e is then used by the function replay(x, e, UAs($\mathrm{U_M}$, $\mathrm{SP_T}$)) (line 11) to replay the value of e while executing UAs($\mathrm{U_M}$, $\mathrm{SP_T}$), and generating the corresponding HTTP traffic logs (HTTP_logs). This logs are finally used in the `Postconditions` to check whether Flag($\mathrm{U_V}$, $\mathrm{SP_T}$) occurs. To clarify how the attack pattern engine leverages the API exposed by ZAP to interact with the built-in proxy, the pseudo-codes corresponding to the extract and replay functions are reported in Listing 2.2 and Listing 2.3, respectively. In Listing 2.2, at first, the function generate_break_rule(x) is invoked. Given an element x, it returns a proxy rule rb which sets a break point to the execution of the user actions in ZAP, when an occurrence of x is detected. The proxy rule includes regular expressions for uniquely identifying an elements in the HTTP traffic. Then, the ZAP API call load_rule_ZAP(rule) loads

Listing 2.3: Replay function

```
HTTP_logs replay(id x, value e, uas UAs){
  rr = generate_replay_rule(x,e)
  load_rule_ZAP(rr)
  HTTP_logs = execute_ZAP(UAs)
  return HTTP_logs}
```

rb in ZAP. The ZAP API call execute_ZAP(UAs) executes the UAs in ZAP and returns the generated HTTP_logs. The HTTP_logs are taken as input by the function extract_value(x, HTTP_logs) extracting from them the value e, associated to x. In Listing 2.3, the function generate_replay_rule(x, e) returns the proxy rule rr used to detect and replace the value of the element x with e. Then, the ZAP API call load_rule_ZAP(rule) loads rr in ZAP. The ZAP API call execute_ZAP(UAs) executes the UAs in ZAP and returns the generated HTTP_logs.

Notice that, besides the functions mentioned above, in order to help the *security expert* in defining new attack patterns, we provide several functions.[6]

The implementation of other attacks patterns (see Table 2.3) follows similar line of reasoning as RA1. However, implementing the core idea behind the RedURI attack pattern—i.e. converting a URL to its malicious form so that an attacker can steal the sensitive information contained in the HTTP requests send to the malicious URL— is challenging. For handing this case, we considered the *network attacker* threat model presented in [44]. In this threat model, the attacker has all the capabilities of a web attacker. Additionally, the attacker can read: *(i)* the contents of all unencrypted HTTP requests originating from the victim's web browser (due

[6]The full list of functions that can be used in the definition of attack patterns is available at `https://sites.google.com/site/mpwaprobe`.

to the fact that the attacker and victim resides in the same network), *(ii)* the contents of encrypted HTTP requests that are directed to the web sites (including *SP*s) that are controlled by the attacker.

2.6 Illustrative Example

In this section we explain how the different phases of our security testing framework concretely apply on the following illustrative example: The developer Diana has implemented the Stripe checkout solution in her web application. She is required to ensure that *(r1)* the new feature works as it should and *(r2)* it does not harm the security of her web application. Diana feels confident for *(r1)* as the Stripe API is documented and there are several demo implementations available in the Internet that she can use as references. However, she does not for *(r2)* as she does not have a strong security background.

Let us see how our approach empowers people like Diana (referred to as the tester) to do a systematic usage of the body of knowledge collected by security experts.

The Stripe checkout protocol is illustrated in Figure 2.8. It is slightly different than the PayPal Payments Standard presented in Figure 2.2. Hereafter how the Stripe protocol works. In steps 1-5, the user *U* visits *SP*—an e-shopping application—at *URI_SP* and initiates the checkout of a product item *I*—the item is identified by *I_ID*. Upon receiving the checkout request, *SP* returns a payment form embedded with a unique identifier (*DataKey*) issued by Stripe to *SP* (step 6). The user provides credit card details (*Credentials*) to Stripe and *DataKey* is sent in this request (steps 7-8). After verifying the validity of *Credentials*, Stripe returns a token (*Token*) which is specific to the *SP* (steps 9-10). Upon presenting *Token* and *Secret* (a secret credential possessed by each *SP* integrating the Stripe checkout solution) and *Amt* (cost of *I*), *SP* withdraws *Amt* from the user's credit card (steps 11-12). Finally, the status of the transaction is sent to the user (step 13).

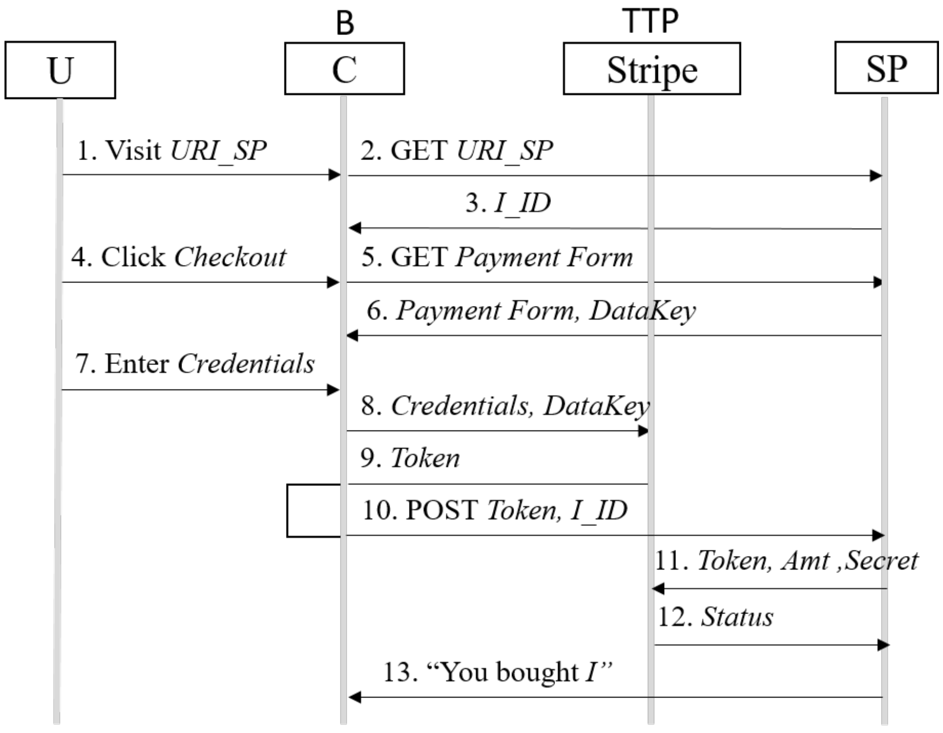


Figure 2.8: Stripe checkout protocol

(P1) Configuration. Diana uses the *SP* she implemented as $\mathrm{SP_T}$ and the official reference implementations provided by Stripe [30] as $\mathrm{SP_M}$. For each of them, she creates the two user accounts $\mathrm{U_V}$ and $\mathrm{U_M}$.

(P2) Recording. Table 2.7 summarizes the *UAs* and *Flags* collected by Diana during the recording phase. Note that the *UAs* are obtained from steps 1, 4, and 7 of Figure 2.8, while the *Flag* is derived from step 13 in Figure 2.8 (I_1-I_4 indicate four different items).

Table 2.7: User Actions and Flags of Stripe Checkout

No.	*Session*	*UAs*	*Flag*
S_1	$(\mathrm{U_V}, \mathrm{SP_T})$	1. Visit *URI*_$\mathrm{SP_T}$ 2. Click Checkout 3. Enter credentials $\mathrm{U_V}$	"`bought` I_1"
S_2	$(\mathrm{U_M}, \mathrm{SP_T})$	1. Visit *URI*_$\mathrm{SP_T}$ 2. Click Checkout 3. Enter credentials $\mathrm{U_M}$	"`bought` I_2"
S_3	$(\mathrm{U_V}, \mathrm{SP_M})$	1. Visit *URI*_$\mathrm{SP_M}$ 2. Click Checkout 3. Enter credentials $\mathrm{U_V}$	"`Enjoy` I_3"
S_4	$(\mathrm{U_M}, \mathrm{SP_M})$	1. Visit *URI*_$\mathrm{SP_M}$ 2. Click Checkout 3. Enter credentials $\mathrm{U_M}$	"`Enjoy` I_4"

Table 2.8: Excerpt of Inference on Stripe Checkout

Element	Data Flow	SynLabel	SemLabel
DataKey	SP-TTP	BLOB	MAND, AU
Token	TTP-SP	BLOB	MAND, SU

(P3) Inference. An excerpt of the inference results of the protocol underlying Diana's implementation of the Stripe checkout protocol is shown in Table 2.8.

(P4) Application of Attack Patterns. The result of applying each attack pattern of Table 2.3 on this example is reported in Table 2.9.

Table 2.9: Attack Pattern Application on Stripe Checkout

RA1	REPLAY *Token* FROM $(\mathrm{U_V}, \mathrm{SP_M})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$. This attack pattern reports no attacks. When the attack test-case reaches step 10 of Figure 2.8, $\mathrm{U_V}$'s *Token* which was actually issued for $\mathrm{SP_M}$ is replayed by $\mathrm{U_M}$ against $\mathrm{SP_T}$. The *TTP* Stripe identifies a mismatch between the owner of *Secret* and the *SP* for which *Token* was issued and returns an error status at step 12.
RA2	REPLAY *DataKey* FROM $(\mathrm{U_M}, \mathrm{SP_M})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$. No attacks reported. Similar reasons as the previous one: the attacker replays *DataKey* belonging to $\mathrm{SP_M}$ in the checkout session at $\mathrm{SP_T}$. Hence the *Token* returned by *TTP* cannot be used by $\mathrm{SP_T}$ to receive a success status at step 12.
RA3	REPLAY *Token* FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$. No attack reported. In Stripe checkout, the validity of a *Token* expires once it is used. Reuse of *Token* returns an error.
RA4	REPLAY *DataKey* FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN S where S = REPLAY *Token* FROM $(\mathrm{U_V}, \mathrm{SP_M})$ IN $(\mathrm{U_M}, \mathrm{SP_T})$. This attack pattern reports an attack as there is no protection mechanism in the Stripe checkout solution that prevents spoofing of the *DataKey* by another *SP*. Initially, the attack test case replays the *DataKey* from $(\mathrm{U_M}, \mathrm{SP_T})$ into $(\mathrm{U_V}, \mathrm{SP_M})$. When the *Token* obtained in this session by $\mathrm{SP_M}$ is replayed into session $(\mathrm{U_M}, \mathrm{SP_T})$, Stripe does not identify any mismatch and returns a success status at step 12. This allows the attacker $\mathrm{U_M}$ to impersonate $\mathrm{U_V}$ and to purchase a product at $\mathrm{SP_T}$.
RA5	This attack strategy is not applicable to Stripe as there are no elements with data flow TTP-SP that also have REQUESTURL as location (basically none of those elements would be present in the browser history).
LCSRF	REPLACE *req* WITH REQUEST-OF *Token* FROM $(\mathrm{U_M}, \mathrm{SP_T})$ IN $[\mathrm{U_V}$ SEND *req*$]$. This pattern detects an attack. The test case generated sends a HTTP POST request corresponding to step 10 with an unused *Token*. This request alone is enough to complete the protocol and to uncover a CSRF. In our experiment, this was discovered on the demo implementation of Stripe. Indeed it is not unusual that this kind of protections are missing in the demo systems. We do not know whether any productive MPWAs suffer from this. Determining this would require specific testing users on the productive system and the buying of real products.
RedURI	This pattern is not applicable as there are no URIs that have data flow TTP-SP and semantic property RURI.

(P5) Reporting. The RA4 and LCSRF attacks are reported to Diana. Execution details of attack patterns are logged and can be inspected.

2.7 Evaluation

To test the effectiveness of our approach, we ran our prototype implementation against a large number of real-world MPWAs. In Section 2.7.1, we explain the criteria based on which we selected our target MPWAs. Next, in Sections 2.7.2 and 2.7.3, we explain the attacks we discovered (both automatically and with manual support) and finally, in Section 2.7.4, we provide some information on how we (responsibly) disclosed our findings to the affected vendors.

2.7.1 Target MPWAs

We selected SSO, CaaS and VvE (see Figure 2.3) scenarios as the targets of our experiments. For the SSO scenario, we adopted the Google dork strategy mentioned in [12] to identify *SPs* integrating SSO solutions offered by Linkedin, Instagram, PayPal and Facebook. Additionally, we prioritized the Google dorks results using the Alexa rank of *SP*s. For the CaaS scenario, we targeted open-source e-commerce solutions and publicly available demo *SP*s integrating 2Checkout and Stripe checkout solutions. For the VvE scenario, we selected the websites belonging to the Alexa Global Top 500 category.[7]

2.7.2 Results

We have been able to identify several previously-unknown vulnerabilities and they are reported in Table 2.10. We have promptly notified our findings

[7]`www.alexa.com/topsites`

to the flawed *SPs* and *TTPs* and most of them acknowledged our reports and patched their solutions accordingly. Additional information regarding the disclosures is given in Section 2.7.4. Screencasts of the attacks and the details about our interactions with the vendors are available at the companion website. Some *SP*s have not patched the vulnerabilities yet, and thus in Table 2.10 we have anonymized their names.

We cluster the attacks into four classes (see last column of Table 2.10) according to their similarities with respect to known attacks. This allows us to show the capability of our approach to not only detect attacks that are already known in literature, but also to find similar attacks in MPWAs implementing different protocols and in different MPWA scenarios.

New kind of attack (N)

The RA5 pattern that leverages the *browser history attacker* threat model discovered an attack in the integration of the Linkedin JS API SSO solution at `developer.linkedin.com` (#a2). The presence of the non-expiring user id of the victim in the browser history allows an attacker to hijack the victim's account. Another *SP* website that appears in the Alexa top 10 e-commerce website category[8] is also vulnerable to the same attack (#a1).

Attacks to different scenarios (NS)

A known kind of attack has been applied to a different MPWA scenario. By applying the RA4 attack pattern, we were able to detect a previously unknown attack in the CaaS scenario (#a3 of Table 2.10). It must be noted that RA4 is inspired by an attack in SSO scenario (see #6 of Table 2.1), and our protocol-independent approach allowed us to detect it in CaaS scenario. In particular, we identified the attack in the payment checkout solution offered by Stripe: the attack allows an attacker to impersonate a

[8]www.alexa.com/topsites/category/Top/Business/E-Commerce

SP by replaying its publicly available API key (*DataKey* in Figure 2.8) to obtain a payment token (*Token* in Figure 2.8) from the victim user which is subsequently used to shop at the impersonated *SP*'s online shop using the victim's credit card. As reported in Table 2.10, this attack is applicable to all *SP*s implementing the Stripe checkout solution [30]. Similarly, using our login CSRF attack pattern (inspired by attacks in SSO), we tested the VvE scenario and discovered the following (#a4):

- login CSRF attack in the account registration process of `twitter.com`, `open.sap.com` and six other *SP*s (all having Alexa Global rank less than 500). One of the vulnerable *SP* is a popular video-sharing website. The account activation link (*ActLink* of Figure 2.3) issued by this website not only activated the account, but also authenticated the user without asking for credentials. An attacker can create a fake account that looks similar to the victim's actual account and authenticate the victim to the fake account (this can be done when victim visits attacker's website). As mentioned in [51], this enables the attacker to keep track of the videos searched by the victim and use this information to embarrass the victim.

Additionally, we found another similar attack vector in `twitter.com` for mounting the same attack. The following is the detail:

- `twitter.com` sends an email to a user if he/she has not signed into twitter for more than 10 days. The URLs included in this email directly authenticates the user without asking for credentials. This is a perfect launchpad for performing login CSRF attacks. The authors of [50] had discovered a standard form-based login CSRF attack against `twitter.com` and demonstrated how a login CSRF attack in twitter.com becomes a login CSRF vulnerability on all of its client websites.

Attacks to different protocols (NP)

A known kind of attack is applied to different protocols or implementations of the same scenario (SSO, CaaS, or VvE). Using the RA1 attack pattern which is inspired by the attacks against Google's SAML SSO (cf. #1 of Table 2.1) and Facebook's OAuth SSO (cf. #2 of Table 2.1), we discovered a similar issue in the integration of the Linkedin JS API SSO solution at INstant [10] (#a6) and another *SP* (#a5) which has an Alexa US Rank[9] less than 55,000. The vulnerable *SP*s authenticated the users based on their email address registered at Linkedin and not based on their *SP*-specific user id.

We discovered login CSRF attacks in two *SP*s (#a8, both having Alexa Global Rank less than 1000) integrating the Instagram SSO solution and another *SP* (#a9 of Table 2.10, with Alexa Australia rank[10] less than 4200) integrating the Linkedin OAuth 2.0 SSO. The attack pattern that discovered these attacks is inspired by login CSRF attacks against *SP*s integrating the Browser Id SSO and Facebook SSO solutions (see #7 and #8 of Table 2.1).

Our attack pattern that tampers the redirect URI (inspired by #9 of Table 2.1) reported that in Pinterest's implementation of the Facebook SSO, it is possible to leak the OAuth 2.0 authorization code of the victim to the network attacker by changing the protocol of the redirect URI from "https" to "http" (#a10 of Table 2.10). This attack was possible due to the presence of a Pinterest authentication server that is not SSL protected. The same vulnerability was found in all *SPs* implementing the "Login with PayPal" SSO solution [8] (#a11 of Table 2.10). However, in this case it was due to incorrect validation of the redirect URI by the *IdP* PayPal.

[9]http://www.alexa.com/topsites/countries/US
[10]http://www.alexa.com/topsites/countries/AU

Attacks to new *SPs* (NA)

A known kind of attack on a specific protocol is applied to new *SPs* (still using the same protocol offered by the same *TTP*). This shows how our technique can cover the kinds of attacks that were reported in literature. For instance, in [91], the authors mention that a logical vulnerability in the 2Checkout integration in osCommerce v2.3 enables an attacker to reuse the payment status values of the paid order to bypass payment for future orders (cf. #4 of Table 2.1). We tested the 2Checkout integration in the latest version of OpenCart (v2.1.0.1) and noticed that our RA3 attack pattern discovered a similar attack (#a12 of Table 2.10).

2.7.3 Manual Findings

In [94], the authors were able to manually discover exploit opportunities in SSO integrations by analyzing the inference results of the HTTP traffic. Since our inference module is an extension of [94], we were also able to manually identify two attacks. We created one single attack pattern that generalizes the XSS attack strategy reported in [48, §4]. While writing the preconditions and the attacker strategy was straightforward, the postcondition was more challenging. Indeed establishing whether a XSS payload is successfully executed is a well-known issue in the automatic security testing community. In our preliminary experiments, we just relied on the tester to inspect the results of the pattern and to determine whether the XSS payload was successfully executed. By doing so, we uncovered a XSS vulnerability in the INstant website [10] integrating the Linkedin JS API SSO. Additionally, we manually analyzed the data flow between *SP* and *TTP* in *SP*s integrating Linkedin REST API SSO to identify tainted data elements. We replaced the value of tainted elements with XSS payloads and identified another XSS vulnerability in a *SP* that has Alexa Global

Table 2.10: Attacks discovered

#	Attack Pattern	SP	TTP (& protocol)	Element(s)	Class
a1	RA5	AlexaEcommerce-10	Linkedin JS API SSO	UId, $Email$	N
a2	RA5	developer.linkedin.com	Linkedin JS API SSO	$MemberId$, $AToken$	
a3	RA4	All *SP*s	Stripe Checkout	$DataKey$, $Token$	
a4	LCSRF	twitter.com, open.sap.com, other 6 SPs in Alexa Global Top 500	Gmail	$ActLink$	NS
a5	RA1	AlexaUS-55000	Linkedin JS API SSO	$Email$	
a6	RA1	INstant	Linkedin JS API SSO	$AccessToken$	
a7	XSS	INstant	Linkedin JS API SSO	$Fname$, $LName$	
a8	LCSRF	AlexaGlobal-1000a, AlexaGlobal-1000b	Log In With Instagram	$Code$	NP
a9	LCSRF	AlexaAu-4200	Linkedin OAuth 2.0 SSO	$Code$	
a10	RedURI	pinterest.com	Facebook SSO Auth.Code Flow	$RedUri$	
a11	RedURI	All *SP*s	PayPal Log In	$RedUri$	
a12	RA3	OpenCart v2.1.0.1	2Checkout	$Order\_number$, Key	NA
a13	XSS	AlexaGlobal-300	Linkedin REST API SSO	$AboutMe$	

rank less than 300 (#a13).

2.7.4 Disclosures

Pinterest acknowledged our report about the redirect uri fixation attack and recently they updated their Facebook SSO implementation. The redirect uri fixation attack against all *SP*s integrating the PayPal SSO was due to the deviation from the OAuth 2.0 standard by PayPal. Even though PayPal acknowledged our report, we did not win the bug bounty as another security researcher simultaneously reported the attack. However, none of the details regarding this attack was publicly available and we have the screencast of the attack in our website to support our claim. The attack against online shopping websites integrating Stripe checkout was appreciated by Stripe and they rewarded us for our findings. Linkedin updated the Linkedin Developers website after receiving our report about the attack by the *browser history attacker*. OpenSAP acknowledged our report about the login CSRF attack in the account registration process of `open.sap.com` and fixed the issue. We reported the XSS attacks we discovered against the *SPs* integrating the Linkedin SSO to the corresponding vendors. Linkedin was partially responsible for this attack as it was possible to create a Linkedin account and provide XSS payload as the value of user information fields (e.g., first name, last name). However, it was the responsibility of *SP*s to properly filter and encode the user information received from Linkedin. After notifying Linkedin about the issue, we noticed that they enforce restrictions in the usage of HTML characters in input fields. Login CSRF is out of scope for Twitter's vulnerability rewards program [38]. Hence, we did not win a bounty for our report. However, in Section V.F of [50], it is mentioned that the authors discovered a standard form-based login CSRF in the login form of `twitter.com` (which was already known) and the authors explain how this causes a login CSRF in *SP*s integrating Twitter's

SSO solution. Further details about the disclosures are available at our website.

2.8 Related Work

2.8.1 Attack pattern-based Black-Box Techniques

One of the prominent works in this domain is by Pellegrino et al. [82] who proposed the idea of black-box detection of logical vulnerabilities in e-shopping applications. The proposed approach creates an abstract model of the web application from the HTTP traffic, identifies the applicability of predefined behavioral patterns and generate test cases misusing these patterns. The generated test cases are mainly based on the attacks against Cashier-as-a-Service based web stores discovered by Wang et al. [95]. We follow a different *complementary* approach by neglecting the application model and directly focusing on replay attacks (among others). We reckon that, in principle, there could be control-flow attacks that [82] could detect and we may not (even if there is no experimental evidence for this). However, it is interesting to note that the strategy behind all the exploitable attacks discovered by [82] falls under the category of replay attacks (precisely those covered by our RA2 and RA3 attack patterns). We experimentally verified this by running our tool against the e-commerce web applications containing exploitable vulnerabilities and reported in [82]. We also verified that our attack on Stripe checkout protocol would require not-so-obvious extensions of [82]: consider malicious *SP* as we do and generate online test-cases to deal with short-lived/one-time tokens.

Somorovsky et al. [87] conducted an in-depth analysis of 14 different SAML frameworks and developed a framework for testing the security of SAML implementations. The testing framework automatically generated various SAML attack patterns by permuting the positions of the original and malicious elements in a SAML assertion. Currently our tool does not support XML-based attacks (e.g., XML signature wrapping attack, XSW in short). Adding this support to our tool requires the following: our inference module should be able to automatically identify base64 and URL encoded XML, have attack patterns that correctly decode the XML, apply the attack strategy (e.g., XSW) and correctly encode the XML, etc. However, currently there are tools such as SAMLRaider [55] and WS-Attacker [78] that can be easily configured to perform these attacks. Hence we did not invest much in this direction.

Bozic et al. [58] proposed attack pattern-based combinatorial testing for detecting XSS vulnerabilities in web applications. In order to increase the coverage of our attack patterns, we applied the concept of combinatorial testing, as mentioned in Section 2.3.

2.8.2 Other Black-Box Techniques

Wang et al. [94] identified many vulnerabilities in the integration of web SSO systems. The proposed technique analyzes the HTTP traffic going through the browser, infers syntax and semantics of the traffic parameters, checks the applicability of three different attack strategies and provides an overview to assist a security expert in manually identifying concrete attacks. In our approach, we adopted their inference concept, further enhanced it with data flow patterns and automated the process of attack discovery.

Prithvi et al. [56] proposes a black-box technique for exposing vulnerabilities in the server-side logic of web applications by identifying various parameter tampering opportunities and by generating test cases corresponding to the identified opportunity. However, this technique required manual effort to convert these exploit opportunities to actual ones.

Zhou et al. [101] proposed SSOScan, a tool for automatically testing *SP* websites that implements Facebook SSO. SSOScan probes the *SP* website for detecting the presence of 5 vulnerabilities that are specific to Facebook SSO. SSOScan is useful in conducting large-scale security testing of *SP*s implementing the same SSO solution. Even though our input collection module requires more manual effort compared to that of SSOScan, the concept of application agnostic attack patterns extends the generality of our approach by enabling the testing framework to detect attacks in multiple scenarios (SSO, CaaS, etc.).

None of the above mentioned black-box techniques provides experimental evidence of the applicability of the approach in multiple MPWA scenarios (CaaS, SSO, etc.) as we do.

2.8.3 Other Techniques

Bai et al. proposed AUTHSCAN [49] for automatically extracting formal specifications from the implementations of authentication protocols and verify it using a model checker to identify vulnerabilities. AUTHSCAN uses sophisticated techniques such as analyzing the available client-side code in order to increase the correctness of the automatically extracted formal model. However, the authors mention that due to the issue of false positives, manual effort was required for checking inconsistencies between the actual implementation and the extracted formal model. This requires the tester to be knowledgeable on formal specification. Our approach does not have such a strong requirement and its applicability is not limited to authentication protocols.

WebSpi [50] is a library for modeling web applications using a variant of the applied pi-calculus. These formal models were verified using the ProVerif tool to discover a variety of attacks in the integration of OAuth-based Single Sign-On solutions. The authors of [50] also proposed the idea of automatically obtaining the formal specification of applications written in a subset of PHP and JavaScript. This work also emphasized the importance of considering CSRF and open redirectors while evaluating the security of web-based security protocols.

Sun et al. [60] proposed to detect logical vulnerabilities in e-commerce applications through static analysis of the available program code. Even though the level of automation in [60] is higher than our approach, we were able to detect similar attacks without requiring the source-code of the application.

Recently, there have been some efforts [97, 60] to prevent the exploitation of logical vulnerabilities in the integrations of CaaS and SSO APIs. However, these techniques requires changes to be made in the way applications are deployed. Our approach does not have this requirement as we are focusing on detecting the attacks rather than preventing them.

2.9 Limitations and future directions

Coverage is a general issue of the black-box security testing tools. Though each of our attack pattern can state precisely what it is testing, our approach is not an exception in this respect. Additionally, it can only detect known types of attacks because our attack patterns are inspired by known attacks. Creative security experts could craft attack patterns capturing novel attack strategies to explore new types of attacks. Two cases can be foreseen here. The new attack patterns (new recipes) can be built (cooked) on top of the available preconditions, actions, and postconditions (ingredients). In this case it should be pretty straightforward for security experts to cook this new recipe. If new ingredients are necessary, extensions are needed. These can range from adding a simple operation on top of OWASP

ZAP up to extending the inference module with e.g., control-flow related inferences and similar. Another research direction could focus on integrating fuzzing capabilities within some of our attack patterns. A clear drawback is that this extension will likely make the entire approach subject to false positives. A more challenging research direction could focus on automated generation of attack patterns. Though this may look as a Holy Grail quest, there may be reasonable paths to explore. For instance, when considering replay attacks and the patterns we created for them, it is clear that the attack search space we are covering is far from being complete. How many sessions and which sessions should be considered in the replay attack strategy as well as which goal that strategy should target remain open questions. However, attack patterns could be automatically generated to explore this combinatorial search space.

A few attacks reported in the MPWA literature are not covered by our attack patterns. In fact, Table 2.1 does present neither XML rewriting attacks [87] nor XSS attacks, e.g., [48, §4]. For XSS we did not invest too much in that direction as there are already specialized techniques in literature that are both protocol- and domain-agnostic. As explained in 2.8, by adding XML support, new attack patterns can be created to target also XML rewriting attacks as in [87]. This can be a straightforward extension of our approach and prototype especially considering that OWASP ZAP supports Jython [34]. Basically, all Java libraries can be run within OWASP ZAP so that Java functions performing transformations on the HTTP traffic (e.g., base64 encoding/decoding, XML parsing) can be used in the attack patterns.

Our approach can also be extended to handle postMessage [5]: frames would be considered as protocol entities and their interactions as communication events. While there are no conceptual issues to perform this extension, there is technical obstacle as, at the moment, OWASP ZAP provides only partial support to intercept postMessages.

Another interesting future direction is to extend our approach to identify attacks enabling replay attacks in mobile applications. Although mobile applications communicate with their back-end using the HTTP protocol, extending our approach to the mobile app scenario would require extensions such as the consideration of inter-app communication channels such as the Android Intent.

Additionally, the approach presented in this chapter is not fully automated because it requires the tester to provide the initial configurations. The quality of these configurations has a direct impact on the results. For instance if the *Flags* are not chosen properly, our system may report false positives.

Still, as shown, the approach is effective and we plan to further refine it to overcome these kinds of issues.

Chapter 3

Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries

In Section 2.7 of the previous chapter, we showed that out attack pattern for Login CSRF was able to detect Login CSRF attacks against many prominent web sites implementing the VvE scenario. This made us curious to conduct a large-scale analysis of the web for measuring the pervasiveness of web sites not protecting their authentication and identity management functionalities from CSRF. After all, browser-based security protocols underlies the processes associated to these functionalities. Additionally, it was shown in [52] that state-of-the-art black-box security testing tools (for web applications) have low detection rate for vulnerabilities causing CSRF. This motivated us to do further research on black-box security testing for CSRF. Hereinafter, we present this research.

3.1 Introduction

Cross-Site Request Forgery (CSRF) is one of the top threats to web applications and has been continuously ranked in the OWASP Top Ten [19].

In a CSRF attack, the attacker makes a victim's web browser silently send a forged HTTP request to a vulnerable web site and cause an undesired state-changing action. The term victim refers to an honest user of the vulnerable web site.

CSRF attacks can be classified as post- and pre-authentication CSRF attacks, depending on whether the undesired state-changing action requires the victim to have an already-established authenticated session or not, respectively. Post-authentication CSRF is known at least since 2001 [41] and has received a lot of attention from the web community. For instance, the OWASP Testing Guide [80] devotes an entire section to this vulnerability (cf. Testing for CSRF in [80]). On the contrary, pre-authentication CSRF is not mentioned in the OWASP Testing Guide although severe exploits have been reported in the literature, including:

- the execution of malicious JavaScript code at the victim's web browser [51],
- the association of the victim's financial details (or Google Search History) with the attacker's PayPal account (or Google account, respectively) [51], and
- the tracking of the videos watched by the victim by the attacker [90, 66].

In this chapter, we focus on state-changing CSRF attacks that affect web sites' authentication and identity management functionalities. If carried out successfully, these attacks can enable an attacker to *(i)* authenticate as the victim on the vulnerable web site for post-authentication actions or *(ii)* authenticate the victim into an attacker-controlled account on the vulnerable web site for pre-authentication actions. We refer to them collectively as *Authentication CSRF* (Auth-CSRF for short). We will refer to pre-authentication Auth-CSRF attacks as *preAuth-CSRF* and

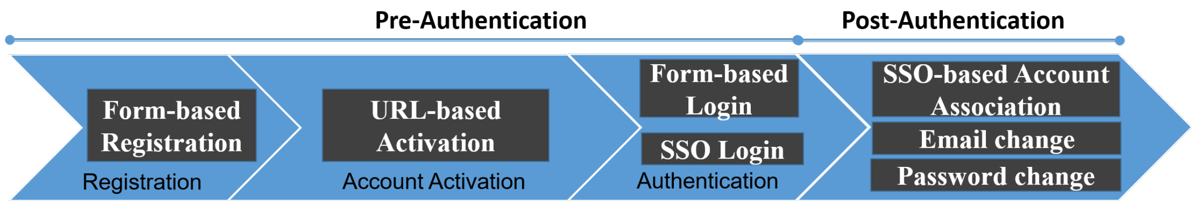


Figure 3.1: Commonly-found Auth-CSRF-vulnerable processes

post-authentication Auth-CSRF attacks as *postAuth-CSRF*.

We begin by analyzing the Auth-CSRF attacks reported in literature, we have been able to *(i)* identify 7 commonly-found processes whose vulnerable implementation causes Auth-CSRF attacks (an overview of them is shown in Figure 3.1), *(ii)* rationally reconstruct 7 process-based testing strategies that can be followed by testers to uncover Auth-CSRF vulnerabilities and also included in widely-used web application testing guides (to spread awareness of Auth-CSRF attacks), *(iii)* generalize the 7 process-based testing strategies into 2 (one pre and one postAuth-CSRF testing strategy) with the prospect of automating them and thereby reducing the manual effort required in applying them. Additionally, we review the (semi-) automatic CSRF-prevention mechanisms proposed in the literature (see, e.g., [62, 75, 64]), establishing that only a subset of the Auth-CSRF attack vectors can be blocked by them.

To estimate the incidence of Auth-CSRF in online web applications and evaluate the effectiveness of our process-based testing strategies, we run an experimental analysis considering 300 Alexa global top web sites (see `www.alexa.com/topsites`) belonging to 3 different rank ranges of the Alexa top 1500. The results of our experimental analysis are alarming: out of the 300 web sites we considered, 133 qualified for conducting our experiments (many web sites were skipped because of language barriers, duplicates, etc., see Section 3.5 for details) and 90 of these suffered from at least one vulnerability enabling Auth-CSRF (i.e. around 68% of the tested

Table 3.1: Excerpt of our Findings

Type	Vulnerable Web Sites	Impact
preAuth-CSRF	e-Government web site for tax filing	Victim can be tricked to reveal his/her private financial information
	Search engines (Google, Bing)	Web Search history of the victim can be accessed by the attacker
	Video-sharing (YouTube)	History of the videos searched & watched by the victim is accessible to the attacker
postAuth-CSRF	Online Dating (Twoo)	Attacker can compromise the victim's Twoo account and access the victim's chat history, know the victim's dating preferences, sexual orientation, etc.
	Online shopping (eBay)	Attacker can access victim's eBay account and shop using the financial information associated to that account.
	Smartphone company's web site	Attacker can compromise the victim's account at the phone company's web site and access the victim's phone data remotely, including SMS, contacts list, gallery items, location info, etc.

web sites are vulnerable). In total, we discovered 150 vulnerabilities in this phase and some of our most-severe findings are mentioned in Table 3.1.

Motivated by the challenges we encountered while conducting our experiments (e.g., identifying the relevant HTTP requests to test for Auth-CSRF, altering the requests based on stored v/s reflected CSRF criteria, etc.), we developed *CSRF-checker*, a proof-of-concept testing tool based on OWASP ZAP [20] that assists a tester to (semi-)automatically detect vulnerabilities causing Auth-CSRF attacks. Using CSRF-checker, we assessed 132 additional web sites (from the Alexa global top 1500) and identified 95 vulnerable ones that are susceptible to Auth-CSRF attacks. In these experiments, CSRF-checker helped in uncovering 168 vulnerabilities.

All the experiments reported in this paper have been conducted in a responsible and non intrusive way (explained in Section 3.8). We reported our findings to the affected vendors and some of them already acknowledged our findings (their identities are disclosed in the paper). For instance, Microsoft and Twoo (a prominent dating web site) paid us $1500 and $500 bug bounties respectively. Similarly, eBay fixed an Auth-CSRF based account hijack vulnerability (on `eBay.com`) we reported. Additionally, LiveJournal and a prominent smartphone company offered non-monetary rewards (e.g., free subscriptions, gift cards, etc.) for our findings related to Auth-CSRF in their web sites.

The contribution of the research presented in this chapter is many-fold:

- we provide a comprehensive analysis of Auth-CSRF attacks taking into account both pre-authentication and post-authentication processes (to the best of our knowledge, something like this has not been done yet);
- we provide precise security testing strategies for Auth-CSRF; introducing these strategies within, e.g., the OWASP Testing Guide may help increase awareness and ultimately improve CSRF protection in web sites;
- we present a proof-of-concept prototype that supports the (semi-) automatic discovery of Auth-CSRF;
- we report on a large-scale experimental analysis for Auth-CSRF we conducted on popular websites from the Alexa global top 1500; this provides experimental evidence to our statements;
- combining the results of all our experiments shows that there are 318 exploitable Auth-CSRF vulnerabilities affecting 185 web sites from the Alexa global top 1500 (among the 265 web sites we tested), i.e. around 70% of the web sites we tested were vulnerable to Auth-CSRF; we also report on our responsible disclosure experience.

Structure of the chapter. In Section 3.2, we introduce some background information about CSRF attacks and defenses. We continue in Section 3.3 discussing Auth-CSRF over key processes of a web application. Section 3.4 defines precise security testing strategies to detect Auth-CSRF attacks. We discuss our first experimental evaluation of Alexa top web sites in

Section 3.5, and some relevant case studies in Section 3.6. In Section 3.7 we present our prototype for assisting users toward testing for Auth-CSRF. Our ethics and responsible disclosure experience is reported in Section 3.8. In Section 3.10 and Section 3.9 we discuss some limitations of our work and a comparison with the related work.

3.2 Background

This section provides background knowledge of different types of CSRF attacks and the widely-used defenses to prevent them.

3.2.1 CSRF Attacks

In a CSRF attack, an attacker makes a victim user's web browser send a forged HTTP request to an honest web site and cause an undesired state-changing action at the web site. The seriousness of a CSRF attack depends on the consequences of the state-changing action that was initiated by the attacker. For instance, in [98], it was shown that while being logged in on prominent web sites like `nytimes.com` (an online newspaper web site), `INGdirect.com` (a famous banking web site), etc., if a victim user loads an attacker-controlled web page in his/her web browser, the attacker can send forged HTTP requests to these web sites and (1) steal the victim's personal email address stored at nytimes, (2) transfer money from the victim's ING bank account to the attacker's account, etc.

From 2001 (first reporting of CSRF [41]) to 2008, it was widely-considered that only the state-changing actions that can be caused by authenticated users need to be protected from CSRF attacks. This is mainly due to the assumption that only authenticated users can execute actions having high-impacts (e.g., transferring money from one account to another). However, in 2008, Login CSRF attack was introduced [51]. In a Login CSRF attack, an attacker makes a victim's browser send a HTTP request to the authentication end point of a web site (e.g., action URL of the login form) with the attacker's *username & password* for the web site and thereby authenticating the victim into the web site as the attacker. By doing so, the attacker can keep track of the actions performed by the victim on the vulnerable web site until the end of the session, enabling the attacker to steal sensitive information. The authors demonstrated this by showing that the Google search history (or bank account details) of another user can be stolen by mounting a Login CSRF attack on Google (or PayPal respectively). The authors even demonstrated the possibility of executing malicious JavaScript code at the victim's web browser by exploiting a Login CSRF in iGoogle. Recently, several variants of Login CSRF attack have also been reported in the Single Sign-On (SSO) domain [44, 50].

Pre- and Post-authentication CSRF Attacks: It is interesting to note that the victims of Login CSRF attacks (and its variants [44, 50]) are not authenticated users (unlike previously-reported CSRF attacks). Based on these findings, we divide CSRF attacks into two categories: CSRF attacks that do not require the victim to have an authenticated session with the vulnerable web site (hereafter what we refer to as *pre-authentication CSRF* attacks) and the type of CSRF attacks that require the victim to have an authenticated session (hereafter referred to as *post-authentication CSRF attacks*). We will show in Section 3.2.2 that this distinction is important when it comes to protecting web sites from CSRF attacks.

Reflected and Stored CSRF Attacks: Depending on the way in which the attacker makes the victim send the forged HTTP request, CSRF attacks can be classified into *reflected CSRF attacks* and *stored CSRF attacks* [59, 85]. While in reflected CSRF attacks, the attacker uses a medium other than the vulnerable web site—for instance a malicious web site controlled by the attacker—to make the victim's browser send the state-changing HTTP request, in stored CSRF attacks, the attacker can either directly use the vulnerable web site or a web site running in a related domain [57] to make the victim's browser send the state-changing HTTP request. As we will show in Section 3.2.2, this distinction is important in understanding the drawbacks of CSRF defenses.

3.2.2 Defending against CSRF Attacks

There are three main defense methods for protecting web sites from CSRF attacks, namely secret validation token, HTTP `Referer/Origin` header validation, and custom HTTP header validation. Hereafter, we briefly describe them (interested readers can refer to [51] for more details), also reporting some (semi-)automatic CSRF protection mechanisms that leverage these defenses.

Secret Validation Token: This method helps a web site to maintain session integrity by validating a secret token. Whenever a user starts a session with a web site, the web site generates (1) a unique identifier for the session (what we refer to as session identifier) and (2) a non-guessable secret token that is cryptographically bound to the session identifier. The session identifier is stored on the web browser associated to the session (e.g., using the `Set-Cookie` HTTP header) and the secret token is embedded in all HTTP responses to the web browser. Whenever the user executes an important operation that will cause a state-changing action at the web site, a HTTP request is sent to the web site containing both the session identifier and the secret token. Upon receiving the request, the web site checks whether the secret token and the session identifier maintain the expected cryptographic relationship. Only if this condition is satisfied the state-changing operation will be executed. Even though this is an effective defense, we will show in Section 3.5 that many web sites implement this defense incorrectly and thereby enable CSRF attacks.

Referer/Origin Header Validation: In this method, whenever a web site receives an HTTP request associated to a state-changing action, the web site checks whether the request originated from a trusted domain. This can be done by checking the value of either the `Referer` header or the `Origin` header present in the HTTP request. The `Referer` header carries the value of the URL of the web page that caused the request. The `Origin` header contains only the *scheme*, *host*, and *port* of the URL of the web page that caused the request. If the value in the `Referer/Origin` header does not belong to that of a trusted domain, the request is dropped. We will show in Section 3.3 that there are certain special scenarios (e.g., URL-based account activation) where the web site will have to execute state-changing actions upon receiving HTTP requests from unknown domains and erroneous logic of the implementation of these scenarios can cause CSRF attacks. Additionally, certain browser-based vulnerabilities can also enable an attacker to spoof the `Referer/Origin` headers (e.g., [68] explains a PDF reader-based vulnerability enabling CSRF). We do not consider browser-based vulnerabilities in this paper.

Custom HTTP Header Validation: A web site adopting this defense must implement all state-changing actions via *XMLHttpRequests* [39]. In this way, the web site can add custom HTTP headers to all state-changing HTTP requests and validate these headers to ensure that the request has not been forged. In [51], it is suggested to validate the presence of the `X-Requested-By` header (a header present in all XMLHttpRequests) to ensure that the request originated from a trusted domain. The logic behind this idea is that an attacker's web page loaded at a victim's web browser will not be able to send XMLHttpRequests with the `X-Requested-By` header to another web site (which is not under the control of the attacker) unless the web site suffers from Cross-Site Scripting vulnerabilities (commonly referred to as XSS), or if the web site defines erroneous cross-domain policies [73], or if the victim uses a vulnerable web browser (beyond the scope of this paper, see [68] for details). In [51] it is also suggested to drop all state-changing requests that do not contain the `X-Requested-By` header. However, we will show in Section 3.5 that many web sites implementing their login actions via XMLHttpRequest do not reject the request even if it does not contain the `X-Requested-By` header. This allows an attacker to send forged login requests.

(Semi-)automatic CSRF Protection Mechanisms: Manually implementing the above mentioned CSRF defenses is not only tedious, but also error prone. Even though there are web site development frameworks like ASP.NET that supports swift adoption of CSRF defenses, it was pointed out in [76, §2.1.2] and [62, §3.1.1] that these frameworks have many exceptions (e.g., no protection for SSO [76, §2.1.2]). Another interesting option available for web site developers and users to avoid CSRF attacks is to make use of (semi-)automatic CSRF protection mechanisms (e.g., [69, 72, 62, 79]).

These mechanisms are implemented either at the client-side (e.g., as a browser plugin) or at the server-side of a web site. They can be broadly seen as having two parts. First they use certain heuristics to identify suspicious requests (e.g., [69] considers all cross-domain requests as suspicious) and then they perform certain operations on the suspicious HTTP requests (e.g., [69] removes authentication credentials from the header of suspicious requests). We will show in Section 3.3.3 that many CSRF attacks cannot be prevented by these mechanisms.

In the following section we explain the subclass of CSRF attacks we focus in this chapter.

3.3 Authentication CSRF Attacks (Auth-CSRF)

As shown in the previous section, CSRF attacks can affect any sensitive process of a web site, and their impact can have different levels of severity depending on the process considered, and a number of defenses can be put in place to defend against CSRF. It is thus difficult to evaluate whether all the sensitive processes of a web site are protected from CSRF.

For our purposes, we identified a significant subclass of CSRF attacks that affects the authentication and identity management processes of web sites. We refer to them as Authentication CSRF attacks (or Auth-CSRF in short). In Auth-CSRF attacks, the attacker exploits a CSRF vulnerability on a web site to cause either the (1) victim to be authenticated as the attacker on the target web site or (2) attacker to be authenticated as the victim on the target web site.

The reason why we considered this subclass is manifold. Auth-CSRF attacks are pervasive (as shown by recent studies [86, 76]) and affect both pre- and post-authentication processes of web sites (cf. Figure 3.1). In addition, verifying the success of the application of an Auth-CSRF attack strategy on a web site can be done easily by checking whether *(i)* the victim has been authenticated as the attacker or *(ii)* the attacker can authenticate as the victim. Last but not least, given that Auth-CSRF attacks affects authentication, the impact of Auth-CSRF attacks can be even more serious than other kinds of CSRF attacks. In the following section we explain this in more detail.

3.3.1 Impacts of Auth-CSRF Attacks

CSRF attacks causing the victim to be authenticated as attacker are often underestimated. This is mainly due to the fact that it is not clear how an attacker can exploit this scenario. Some of the possible exploitations that are reported in the literature are as follows.

- In [51] it was shown that by logging the victim into the attacker's Google (or PayPal) account, an attacker can steal the Google search history (or bank account details respectively) of the victim, execute arbitrary JavaScript code on the victim's browser, etc. Interestingly, another researcher [66] showed that by exploiting this vulnerability an attacker can host a malicious flash file and steal the search history of the victim's actual YouTube account.

- In [90] it was shown that an attacker can authenticate the victim to the attacker's account on a famous video-sharing web site and thereby track the videos watched by the victim.

- It was shown in [76] that by logging the victim into the attacker's Facebook account, an attacker can associate his/her Facebook account to the victim's StackExchange account. This enables the attacker to sign into the victim's StackExchange account via the "log in with Facebook" option on StackExchange.

CSRF attacks causing the attacker to be authenticated as the victim are obviously serious. They allow an attacker to have complete access to the victim's account on a web site. An attacker can purchase items using the victim's credit card if the vulnerable web site is an online shop where victim has associated his/her credit card. Similarly, the attacker can access the victim's confidential files if the vulnerable web site is an online file-sharing web site.

We will present additional impacts of Auth-CSRF attacks from our experimental analysis in Section 3.6.

3.3.2 Selection of Auth-CSRF Attacks and Associated Processes

Before defining of security testing strategies for Auth-CSRF, we started with a scrutiny of several Auth-CSRF attacks reported in the literature and the selection of the processes affecting authentication. Table 3.2 shows the Auth-CSRF attacks that cause the victim to be authenticated as the attacker and Table 3.3 shows the attacks that cause the attacker to be authenticated as the victim. The process whose vulnerable implementation caused the attack is shown in the last column of Tables 3.2 and 3.3. An overview of all the processes considered in Tables 3.2 and 3.3 is illustrated in Figure 3.1. Hereafter, we describe each process (labeled as P_1 to P_7) and the associated attacks.

Form-based Registration (P_1): In many web sites, users can create accounts by providing the necessary information (such as username, desired password, etc.) in a registration form. We refer to this process as *form-based registration*.

Attack #1: The attack was discovered in `www.localize.io` (Localize). The web site's Sign Up form was not protected from CSRF attacks and the web site directly authenticates a user upon submitting the registration form with valid data. When the victim visits the attacker's web site, a forged HTTP request corresponding to the submission of the registration form is sent from the victim's web browser. Being a reflected CSRF attack (see Section 3.2 for details), the origin of the forged attack request (shown as Atk Req—meaning Attack Request—in Table 3.2) will be a URL associated to the attacker's web site (shown as AtkWS—abbreviation of Attacker's Web Site—in column 4, row #1 of Table 3.2) and the HTTP request containing the username, password and other registration information chosen by the attacker, similar to that of the victim (represented

Table 3.2: Auth-CSRF attacks causing Victim to be Authenticated as Attacker

#	Reference	Referer/Origin		Credentials in Atk Req	Vulnerable Process
		Benign Req	**Atk Req**		
1	Localize.io's Sign up form [43]	VulnWS	AtkWS	Body[$uname_A$, $pass_A$, $info_A$]	Form-based Registration
2	openSAP's account activation URL [90, §IV.B.2]	TrustWS	AtkWS	URL[act_token_A]	URL-based Account Activation
3	Twitter's [50, §IV.E] and Google's [51, §3] Login Form	VulnWS	AtkWS	Body[$email_A$, $pass_A$]	Form-based Login
4	Facebook's Login Form [51, §4.2]	VulnWS	[ ]	Body[$email_A$, $pass_A$]	
5	Facebook's Login Form [76, §2.2.1]	VulnWS	AWPVulnWS	Body[$email_A$, $pass_A$]	
6	Two web sites implementing Mozilla's BrowserID [49, §6.2]			Body[$auth\_assert_A$]	
7	Many web sites implementing Open ID [51, §6.1]	TrustWS	AtkWS	Body[$token_A$]	SSO Login
8	Stanford's WebAuth implementation [44, §IV.E]			URL[id_token_A]	
9	Many web sites implementing OAuth protocol [90, §VI.B.3], [50, §V.C], [92, §4.4], [86, §3.1]			URL[$code_A$]	

Legend: (1) VulnWS: Vulnerable Web Site, (2) AtkWS: Attacker's Web Site, (3) TrustWS: Trusted Web Site (e.g., an IdP), (4) AWPVulnWS: Attacker-configurable Web Page on the Vulnerable Web Site, (5) [ ]: empty `Referer` Header

Table 3.3: Auth-CSRF attacks causing Attacker to be Authenticated as Victim

#	Reference	Referer/Origin		Credentials in Atk Req	Vulnerable Process
		Benign Req	Atk Req		
10	Web site implementing OAuth-based account association feature [84, 13],[96, §5.2.1(A6)]	TrustWS	AtkWS	Body[$code_A$], Hdr[$cookie_V$]	SSO-based Account Association
11	Primary Email change in MetaFilter[98, §3.3]	VulnWS	AtkWS	Body[$email_A$], Hdr[$cookie_V$]	Primary Email Change
12	Web sites having Password change forms without CSRF protection [3]	VulnWS	AtkWS	Body[new_pass_A], Hdr[$cookie_V$]	Password Change

Legend: (1) TrustWS: Trusted Web Site (e.g., an IdP), (2) AtkWS: Attacker's Web Site, (3) VulnWS: Vulnerable Web Site as $uname_A$, $pass_A$ and $info_A$ in column 5, row #1 of Table 3.2). A benign version of this forged request (shown as Benign Req—meaning Benign Request—in Table 3.2) is supposed to originate from the vulnerable web site (shown as VulnWS—abbreviation of Vulnerable Web Site—in column 3, row #1 of Table 3.2) which in this case is Localize. Upon receiving the request, the victim is authenticated as attacker on Localize.

URL-based Account Activation ($\mathbf{P_2}$): On many web sites, whenever a user creates an account, the web site sends a URL containing a secret activation token to the email address provided by the user during registration. The user is then instructed to click on the link. This procedure helps the web site to verify whether the user is actually the owner of the provided email address. When the user passes this verification, the newly-created account is fully activated. We refer to this process as *URL-based account activation* and to the URL with the secret activation token as the *activation link*.

Attack #2: This attack was found in `open.sap.com` (openSAP in short). When the user clicks on the account activation link sent by openSAP, the web site not only activates the account but also authenticates the user. The attacker can create an account on openSAP that looks (visually) similar to the victim's actual openSAP account (in openSAP this can be done by keeping the firstname and lastname of the victim's openSAP account as the firstname and lastname of the spoofed account). After creating the account, the attacker receives an activation link containing the secret activation token (act_token_A). The attacker embeds this link on the attacker's web site. When the victim visits the attacker's web site, the attacker makes the victim's browser sends a forged HTTP request corresponding to clicking the activation link containing (act_token_A) and the victim is authenticated as the attacker on openSAP.

Form-based Login ($\mathbf{P_3}$): In many web sites, the user can authenticate by providing a user identifier—in most cases this is the email address—and a password on a login form provided by the web site. We refer to this process as *form-based login*.

Attack #3: This attack is known as Login CSRF attack and the attack was discovered in `twitter.com` (Twitter) and `google.com` (Google) due to the absence of CSRF protection in the login forms. The description of the attack is as follows. The attacker creates an account on Google (or Twitter) with the attacker's email address ($email_A$) and password ($pass_A$). The newly created account looks (visually) similar to the victim's actual Google (or Twitter) account (for instance, a Google or Twitter account created with the same first name and last name as that of the victim's actual account). When the victim visits the attacker's web site, the attacker makes the victim send a forged HTTP request corresponding to the submission of the login form on Google (or Twitter) with $email_A$ and $pass_A$. Upon receiving the request, the victim is authenticated as the attacker on `google.com` (or `twitter.com`).

Attack #4: This attack was found in `facebook.com` (Facebook). Facebook protects its login form from CSRF attacks by checking the `Referer` header of the login requests (to understand whether the request originated from a web page associated to Facebook). However, if the `Referer` header is missing in the request, Facebook allows the request. This allows an attacker to perform an attack similar to attack #3 of Table 3.2 but with the difference that, while sending the forged login request with the attacker's login credentials, the attacker abuses a browser trick to send the request without the `Referer` header (see [69, §3.1], [51, §4.2.1]) and thereby authenticating the victim as the attacker on Facebook.

Attack #5: This attack was also discovered in `facebook.com`. The attack is similar to attack number #4 of Table 3.2. The description is as follows. The attacker creates a Facebook canvas app running on a domain with pre-fix `apps.facebook.com`. The app is configured in such a way that when the victim visits the web page associated to the app, a POST request is send to the attacker's web site (running on say `attacker.com`). Upon receiving the POST request, `attacker.com` sends a 307 redirection response to the login end point of `facebook.com` with the attacker's Facebook credentials and thereby authenticating the victim as the attacker on Facebook. The attack succeeds because `facebook.com` accepts login requests with `Referer` header values belonging to the subdomains of `facebook.com` and the 307 redirection response maintains the `Referer` header of the source request (i.e. the POST request from `apps.facebook.com`) in the subsequent request. The web page configured by the attacker and running on `apps.facebook.com` is represented as AWPVulnWS—meaning the Attacker-configurable Web Page on the Vulnerable Web Site—in column 4, row #5 of Table 3.2.

SSO Login (P_4): Many web sites depend on trusted third-party web sites for authentication. An example is SSO where a Service Provider (SP) web site (e.g., `pinterest.com`) depends on an Identity Provider (IdP) web site (e.g., `facebook.com`) for authenticating a user. When a user initiates SSO on a SP web site, the SP redirects the user's browser to the SSO authentication end point of the IdP. At this point the user is required to provide his/her login credentials to the IdP. If the provided credentials are correct, IdP redirects the user back to the SP web site with authentication data that will help the SP to uniquely identify the user and thereby authenticate him/her. We refer to this process as *SSO login*.

Attack #6: This attack was discovered in two web sites integrating Mozilla's BrowserID SSO protocol. When the victim visits the attacker's web site, the attacker forges a HTTP request to the SSO authentication end point of the vulnerable web site (which is acting as the SP) with the attacker's authentication assertion (represented as $auth\_assert_A$ in column

5, row #6 of Table 3.2) issued by the IdP in the body. The SP validates the submitted $auth\_assert_A$ and authenticates the victim as the attacker.

Attack #7: Similar to attack #6 but with the difference that the underlying SSO protocol is OpenID and the authentication data sent by the attacker to the vulnerable SP (via the victim's browser) is the OpenID token of the attacker (represented as $token_A$ in column 5, row #7 of Table 3.2).

Attack #8: Similar to attacks #6 and #7 but with the difference that the underlying SSO protocol is WebAuth, the authentication data sent by the attacker to the vulnerable SP web site (via the victim's browser) is the WebAuth id token of the attacker (represented as id_token_A in column 5, row #8 of Table 3.2) and id_token_A is located in the URL of the forged request.

Attack #9: Similar to attack #8 but with the difference that the underlying SSO protocol is OAuth 2.0 and the authentication data sent by the attacker to the vulnerable SP web site (via the victim's browser) is the OAuth 2.0 authorization code of the attacker ($code_A$).

SSO-based Account Association (P$_5$): In many web sites, the user has the possibility to authenticate both via form-based login and SSO login. This is achieved by allowing users who do form-based login to later associate their SSO account. The association is done by executing a protocol that has a flow similar to that of the SSO Login flow. The description is as follows. The user must authenticate to the trusted web site (i.e. the IdP) and the IdP makes the user's browser send certain authentication data of the user to the SP web site at which the user wants to do account association. We refer to this process as *SSO-based Account Association.* Note that SSO-based account association can be executed only while the user is logged in on the SP web site.

Attack #10: This attack was found in web sites implementing SSO-based account association via the OAuth 2.0 protocol and lacking CSRF protection. When the victim visits the attacker's web site while being logged in on the vulnerable SP web site, the attacker forges a HTTP request to the account association end point of the vulnerable SP web site with the attacker's authorization code (issued by the IdP) for account association (represented as $code_A$). Since the victim is logged in on the vulnerable SP web site, the authenticated session identifier of the victim on the vulnerable SP web site (represented as $cookie_V$) is also present in the header of the forged request. Upon receiving the forged request, the vulnerable web site associates the attacker's IdP account with the victim's form-based login account. This enables the attacker to login (via SSO) to the victim's account (on the vulnerable web site) using the attacker's IdP credentials.

Primary Email Change & Password Change ($\mathbf{P_6}$, $\mathbf{P_7}$): In many web sites, users perform form-based login by providing the email address and password associated to the user's account. Some web sites allows the user to set new values for the email and password associated to the user's account. We call these processes as *primary email change* and *password change*. The user must be authenticated on the web site while executing these processes. Note that these processes are different from the "forgot email/password" processes in web sites that do not require the user to be logged in.

Attack #11: An attack was discovered in `metafilter.com` (MetaFilter) in which the form for primary email change was not protected from CSRF attacks. When the victim visits the attacker's web site while logged in on the vulnerable web site, the attacker forges a HTTP request to the vulnerable web site that will change the primary email address associated to the victim's account. In particular, the new value of the primary email address will be the attacker's email address (represented as $email_A$). This allows the attacker to obtain a fresh password for the victim's account (via

the “forgot password” feature) and have access to the victim’s account on MetaFilter. Note that the authenticated session identifier of the victim for the vulnerable web site (represented as $cookie_V$) is automatically sent by the victim’s browser along with the forged request.
Attack #12: Same as #11 but the forged request changes the victim’s account’s password at the vulnerable web site to a value chosen by the attacker (represented as new_pass_A). This enables the attacker to login to the victim’s account (provided that the attacker knows the username of the victim’s account).

3.3.3 Preventing Auth-CSRF: Challenges

The following is our comparison of the defenses proposed for preventing CSRF attacks (explained in Section 3.2.2) and the Auth-CSRF attacks shown in Tables 3.2 and 3.3. The secret validation token method if carefully implemented can defeat all attacks (#1 to #12). However, it was shown in [86, 92, 84] that many developers do not implement this defense to protect their SSO Login and SSO-based Account association processes and thus leaving these web sites vulnerable to attacks #9 and #10. This raises the question of whether this trend of developers not implementing CSRF defenses is also applicable to other processes.

The `Referer/Origin` header validation method is suitable for preventing standard reflected Auth-CSRF attack vectors. However, the ambiguity in handling scenarios like empty or related-domain [57] values for `Referer/Origin` leaves web sites vulnerable to attacks like #4 and #5. Additionally, the `Referer/Origin` header validation method is not suitable for protecting processes such as URL-based account activation mainly due to the unpredictable nature of the `Referer/Origin` (the value of the `Referer/Origin` is chosen by the third-party mailbox provider). Lastly, browser-based vulnerabilities enabling `Referer/Origin` header spoofing (see [68]) is also a threat to this defense.

As we explained in Section 3.2, the custom header validation approach can be considered to be an effective CSRF defense only in the absence of XSS vulnerabilities, erroneous cross-domain policies and browser-based vulnerabilities. Past studies (e.g., [19, 73]) show that at least the first two issues are hard to avoid.

In [76], it was shown that the default CSRF protection offered by web site development frameworks like ASP.NET cannot prevent attacks like #9, #10, etc.

When it comes to (semi-)automatic defenses, it was shown (e.g., in [62, 64]) that while many of the proposed techniques [63, 70, 85, 69] break normal cross-domain behavior such as SSO Login, others (e.g., [98]) suffer from drawbacks of either being too permissive or restrictive. We noticed that some of them [62, 75, 64] cannot detect attacks like #2. Similarly, stored CSRF is not supported by [72].

As shown above, existing defenses for Auth-CSRF are either insufficient or prone to implementation errors. Hence, there is a strong need for good security testing strategies that can detect vulnerabilities causing Auth-CSRF. Although there exist many web vulnerability scanners, it has been shown (e.g., [52]) that they have low detection rate for CSRF in general. It is in this context that we propose manual and (semi-)automatic testing strategies for Auth-CSRF.

3.4 Manually Testing for Auth-CSRF Attacks

By carefully analyzing the attacks we discussed in Section 3.3, we have been able to distill testing strategies for processes P_1 to P_7 explained in Section 3.3.2. A tester can manually apply these testing strategies to detect vulnerabilities causing Auth-CSRF on any Web site Under Test (WUT).

Prerequisites. We assume that the tester is in control of a web browser and, using a proxy (e.g., OWASP ZAP [20]), is capable of intercepting and modifying HTTP traffic between the browser and WUT. Moreover, the tester owns credentials associated with two separate accounts (having unique usernames and passwords) on the WUT. We will refer to these accounts as AttAcc and VictAcc as they represent the accounts of an attacker and of a victim on the WUT. The tester should also have a social account enabling SSO login to the WUT (if this option is available on the WUT). We will refer to this account as AttAccSoc (as it represents the social account of the attacker). The last step of each test strategy is a check of the success criteria. A positive answer to this check is an indication that the corresponding process on the WUT is vulnerable. Hereinafter we define each testing strategy.

The general idea is to first run the selected process as the attacker. This allows us to intercept a HTTP request, that can be used as a reference to forge the one to test for Auth-CSRF attacks. After some experiments, we noticed that the following fields of the intercepted HTTP request must be kept unchanged: HTTP method, URL, `Content-Type` and `Content-Length` headers, and the request body. It is then necessary to alter the `Referer/Origin` header according to the different scenarios (see Table 3.4).

Let us first consider the strategies for detecting preAuth-CSRF attacks:

TS$_1$: Test Strategy for Form-based Registration

(1) Visit the *registration page* of WUT

(2) Submit *registration details* (including login-credentials) for AttAcc

(3) Intercept the HTTP request containing the *registration details*

(4) Copy the HTTP method, URL, `Content-Type`, `Content-Length` and body of the intercepted request

(5) Clear browser cookies and reset the intercepting proxy

(6) Visit WUT
(7) Send a new HTTP request with a forged `Referer` (based on A_1, A_2 and A_3 of Table 3.4), the same HTTP method, URL, `Content-Type`, `Content-Length` and body as those in the intercepted request
(8) Check: Is it logged in as AttAcc?

TS$_2$: Test Strategy for URL-based Account Activation
(1) Register an account AttAcc on WUT
(2) Receive *account-activation URL* at the email-address used for registration
(3) Clear browser cookies
(4) Visit WUT
(5) Visit *account activation URL*
(6) Check: Is it logged in as AttAcc?

TS$_3$: Test Strategy for Form-based Login
(1) Visit the *login page* of WUT
(2) Submit *login-credentials* for AttAcc
(3) Intercept the HTTP request containing the *login-credentials*
(4) Copy the HTTP method, URL, `Content-Type`, `Content-Length` and body of the intercepted request
(5) Clear browser cookies and reset the intercepting proxy
(6) Visit WUT
(7) Send a new HTTP request with a forged `Referer` (based on A_1, A_2 and A_3 of Table 3.4), the same HTTP method, URL, `Content-Type`, `Content-Length` and body as that of the intercepted request
(8) Check: Is it logged in as AttAcc?

TS$_4$: Test Strategy for SSO Login

(1) *SSO login* to AttAcc account on WUT via AttAccSoc
(2) Intercept the HTTP request containing the *authentication token* of AttAccSoc
(3) Copy the HTTP method, URL, `Content-Type`, `Content-Length` and body of the intercepted request
(4) Clear browser cookies and reset the intercepting proxy
(5) Visit WUT
(6) Send a new HTTP request with a forged `Referer` (based on A_1, A_2 and A_3 of Table 3.4), the same HTTP method, URL, `Content-Type`, `Content-Length` and body as that of the intercepted request
(7) Check: Is it logged in as AttAcc?

Let us now consider the strategies for detecting postAuth-CSRF attacks:

TS$_5$: Test Strategy for SSO-based Account Association

(1) Login to AttAcc on WUT
(2) Visit *SSO-based account association page* on WUT
(3) Run *SSO account association process* using AttAccSoc
(4) Intercept the HTTP request containing the *authentication token*
(5) Copy the HTTP method, URL, `Content-Type`, `Content-Length` and body of the intercepted request
(6) Clear browser cookies and reset the intercepting proxy
(7) Login to VictAcc on WUT
(8) Send a new HTTP request with a forged `Referer` (based on A_4, A_5 and A_6 of Table 3.4), the same HTTP method, URL, `Content-Type`, `Content-Length` and body as that of the intercepted request
(9) Clear browser cookies and reset the intercepting proxy
(10) Check: Is it possible to perform a SSO Login to VictAcc with the credentials used in (3)?

TS$_6$ & TS$_7$: Test Strategy for Email/Password-change
(1) Login to AttAcc on WUT
(2) Visit the *page for Email/Password-change* of WUT
(3) Submit a *new Email/Password* as AttAcc
(4) Intercept the HTTP request containing the *new Email/Password*
(5) Copy the HTTP method, URL, `Content-Type`, `Content-Length` and body of the intercepted request
(6) Clear browser and reset the intercepting proxy
(7) Login to VictAcc on WUT
(8) Send a new HTTP request with a forged `Referer` (based on A$_4$, A$_5$ and A$_6$ of Table 3.4), the same HTTP method, URL, `Content-Type`, `Content-Length` and body as that of the intercepted request
(9) Clear browser cookies and reset the intercepting proxy
(10) Check: Is it possible to access VictAcc on WUT with new Email/-Password?

We have been able to generalize all seven testing strategies mentioned above down to two, namely preAuthTS (a common testing strategy for the pre-authentication processes P$_1$ to P$_4$) and postAuthTS (a common testing strategy for the post-authentication processes P$_5$ to P$_7$). We reported them in Figures 3.2a and 3.2b, respectively.

We call *Candidate HTTP Request (CandidateReq)* a HTTP request that is generated by the browser while executing any of the processes P$_1$ to P$_7$. A *CandidateReq* always contains a security token (or credential) either as a query parameter in the request URL or as a parameter in the request body. Hence, *CandidateReq* is an ideal candidate for mounting an Auth-CSRF attack.

Strategy preAuthTS consists in running a pre-authentication process

Listing (3.1) Attack Pattern for RA1

```
Run P as AttAcc
Intercept CandidateReq
Clear cookies
Visit WUT
Alter CandidateReq
Send CandidateReq
Check Success Criteria
```

(a) preAuthTS

Listing (3.2) Attack Pattern for RA1

```
Login to AttAcc at WUT
Run P as AttAcc
Intercept CandidateReq
Clear cookies
Login to VictAcc at WUT
Alter CandidateReq
Send CandidateReq
Check Success Criteria
```

(b) postAuthTS

Figure 3.2: Testing strategies

P as AttAcc, intercepting the *CandidateReq* issued by the browser and corrupting the CSRF prevention mechanisms occurring in the header by applying the changes given in Table 3.4. In particular, A_1 is used to perform attacks like #3 of Table 3.2 where the forged HTTP request is sent from an attacker's web site (which is simulated by changing the Referer/Origin header in the request to `attacker.com`). Similarly, A_2 is used to perform attacks like #5 of Table 3.2 where the forged request originated from a web page on the vulnerable web site. This is done by changing the `Referer`

Table 3.4: Alterations

#	Referer/Origin	CSRF Type Covered
A_1	`attacker.com`	Reflected preAuth-CSRF
A_2	WUT	Stored preAuth-CSRF
A_3	Empty	preAuth-CSRF with empty Referer
A_4	`attacker.com`	Reflected postAuth-CSRF
A_5	WUT	Stored postAuth-CSRF
A_6	Empty	postAuth-CSRF with empty Referer

Table 3.5: Testing Strategy Information

P	*CandidateReq*	Success Criteria	
P_1	Body/URL[*regpass*]	Authenticated as attacker	preAuthTS
P_2	URL[*acttoken*]		
P_3	Body/URL[*loginpass*]		
P_4	Body/URL[*ssotoken*]		
P_5	Body/URL[*ssotoken*]	Account Associated	postAuthTS
P_6	Body/URL[*newemail*]	Email Changed	
P_7	Body/URL[*newpass*]	Password Changed	

header to a non-existing URL in the domain of the WUT. This URL will represent the web page in the WUT that is configurable by the attacker (e.g., similar to the feature offered by `apps.facebook.com` explained in Section 3.3). A_3 is to consider attacks like #4 of Table 3.2 where the attacker manages to send the forged HTTP request without a `Referer` header. Once forged, the corrupted *CandidateReq* is submitted and finally the success criteria is checked. The form of *CandidateReq* and the success criteria for each process are given in Table 3.5.

Strategy postAuthTS consists in logging-in with AttAcc credentials, running a post-authentication process P and intercepting the *CandidateReq* issued by the browser, logging-in using VictAcc credentials, replaying a variant of *CandidateReq* obtained by corrupting the CSRF prevention mechanisms as in the previous case, and finally checking the success criteria.

In the following section we explain our experiments of applying the testing strategies TS_1 to TS_7 to the Alexa top web sites, focusing only on reflected Auth-CSRF attacks (as the attack surface for mounting stored CSRF attacks is relatively low), i.e. applying only A_1 and A_4 of Table 3.4.

3.5 Experiments (Manual)

Selection. For this initial experimental analysis we focused on a corpus of 300 popular web sites drawn from the following three ranges of Alexa global top 1500 ranking:

(R1) 1-100 as the top 100 in Alexa Top 500 category,

(R2) 501-600 as the top 100 in Alexa Top 501 to 1000 category, and

(R3) 1001-1100 as the top 100 in Alexa Top 1001 to 1500 category.

This selection allowed us to target the most popular web sites—cf. range (R1)—expected to have good security measures in place and to compare them with relevant set-ranges lower in the ranking—cf. ranges (R2) and (R3)—by a fixed offset (in our case, 400 web sites lower). The idea was to evaluate whether a lower Alexa rank meant a higher chance of CSRF vulnerabilities. We will show in Section 3.7.3 that we also conducted experiments on other rank ranges but with more automation.

Result Overview. Figure 3.3 shows an overview of the results. Among the 300 web sites in this corpus, we could successfully test 133 and 90 have been found vulnerable and exploitable to at least one of the testing strategies discussed in Section 3.4 and focusing only on reflected Auth-CSRF, i.e. applying only A_1 and A_4 of Table 3.4. The remaining web sites have been skipped because of language barriers (90), lack of account creation feature (17), domain duplicates such as between `google.com` and `google.co.in` (31), high requirements such as payment for account creation (17), etc. The tested web sites are well distributed over the three selected Alexa ranges: 45 web sites for (R1), 48 for (R2) and 40 for (R3). Our results indicate that overall around 68% of the tested web sites are vulnerable to the attacks similar to the ones mentioned in Tables 3.2 and 3.3 . This percentage starts at a lower 53% for range (R1) and goes up to 75% for (R2) and (R3), indicating that there is indeed some difference between the most popular web sites—i.e. web sites in (R1)—and the others. However there is no significant difference in the aggregated results between (R2) and (R3).

Though the severity of each attack strongly depends on the vulnerable web site, these numbers are in general quite alarming.

Pre- Versus Post-authentication CSRF. The figures become even more interesting when comparing the incidence of pre-authentication versus post-authentication attacks, see Figure 3.4. Overall 66% of the tested web sites are vulnerable to and exploitable through pre-authentication CSRF and only 19% to post-authentication CSRF. These percentages start slightly lower with (R1): 53% for pre-authentication and 6% for post-authentication, followed by a slight increase for (R2) and (R3): 69% for pre-authentication and around 25% for post-authentication attacks. These results indicate that there is a significant difference between pre- and post-authentication CSRF incidence and seem to confirm our hypothesis that pre-authentication CSRF has not received much attention from the web community.

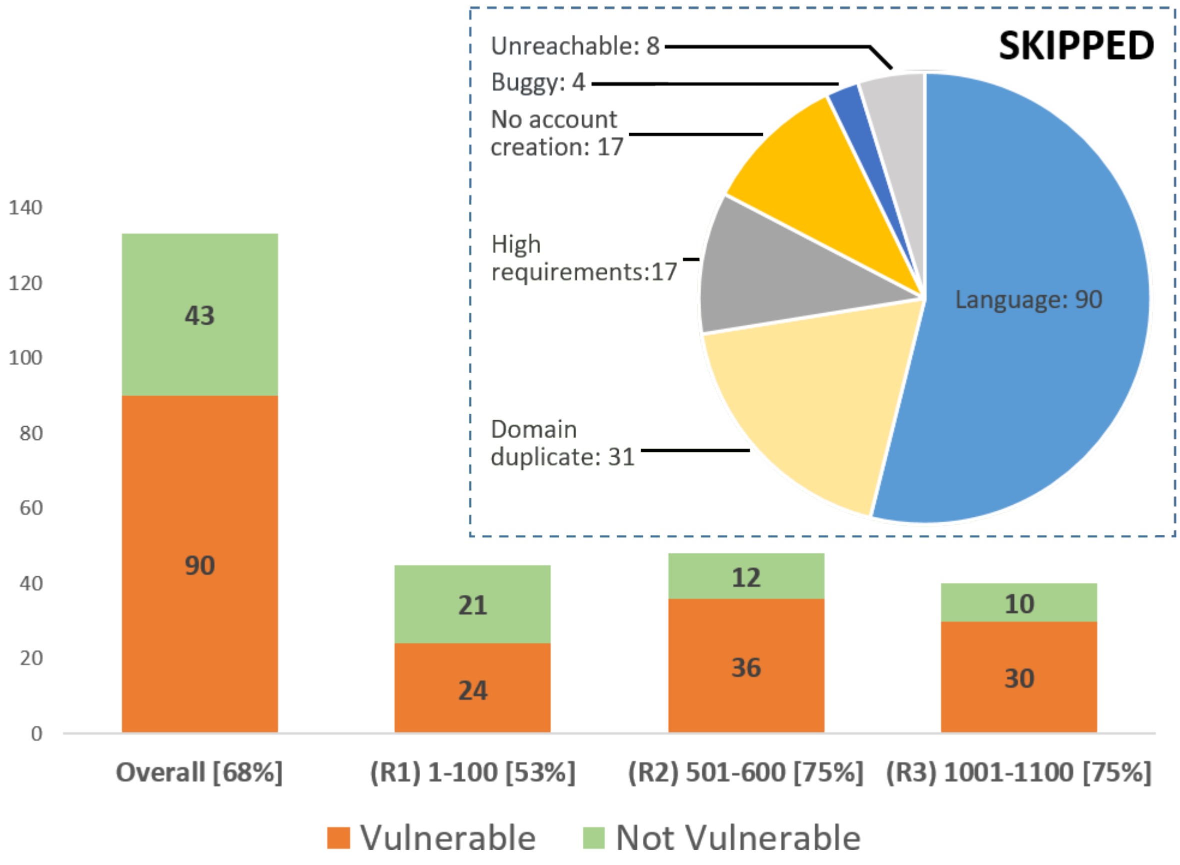


Figure 3.3: Result overview

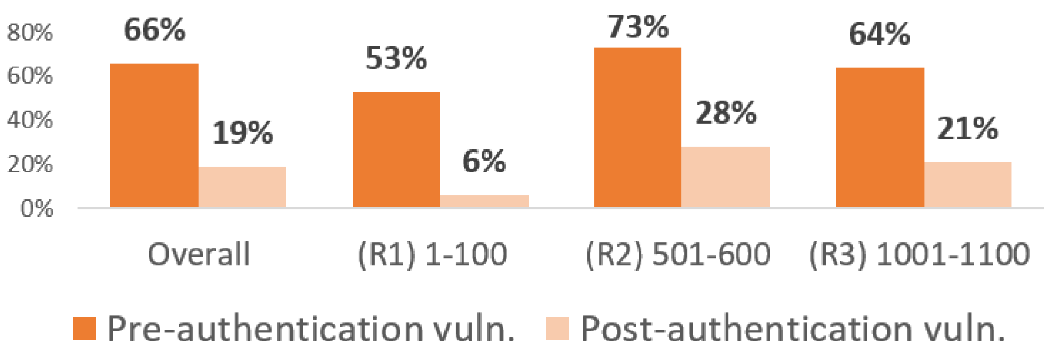


Figure 3.4: Result comparison

Results Per Testing Strategy. As already mentioned we applied all the security testing strategies TS_1 to TS_7 of Section 3.4 against our corpus of web sites (focusing mainly on reflected Auth-CSRF attacks). Each security testing strategy aims to probe whether a web site is subject to a specific Auth-CSRF attack. Figures 3.5 and 3.6 present the incidence of each one of these pre-authentication and post-authentication attacks, both in general and over the three individual Alexa ranges that we considered.
URL-based Account Activation. Over our corpus of 133 testable web sites, 71 send an email with an account activation link after registration. After applying our TS_2 testing strategy, we found that around 37% of these web sites are vulnerable to this form of pre-authentication CSRF, indicating that the occurrence of this attack is significant. For all these web sites an attacker can trick an unaware victim into signing onto an account that seems familiar, but was created and is actually owned by the attacker (cf. attack number #2 of Section 3.3). We performed this check for all the vulnerable web sites, ascertaining that each vulnerability was exploitable. The incidence is lower (15%) for the most popular web sites (R1) and is higher for the other two ranges: 48% for (R2) and 50% for (R3). The small difference between (R2) and (R3) is not statistically significant given the sample size.

Form-based Login. We performed TS3—i.e. Login CSRF attack—against the majority of the testable web sites as most of them feature a form-based login (124 of the 133, the remaining 9 web sites only feature SSO Login to authenticate users). Login CSRF affects 55% of the tested web sites overall, making it the most prevalent vulnerability among all the Auth-CSRF attacks we tested. As usual we checked that an attacker could have indeed authenticated the victim into the attacker's account in each vulnerable web site, proving the flaw was actually exploitable. Once more the incidence of this vulnerability starts at a lower 35% for the Alexa top 100

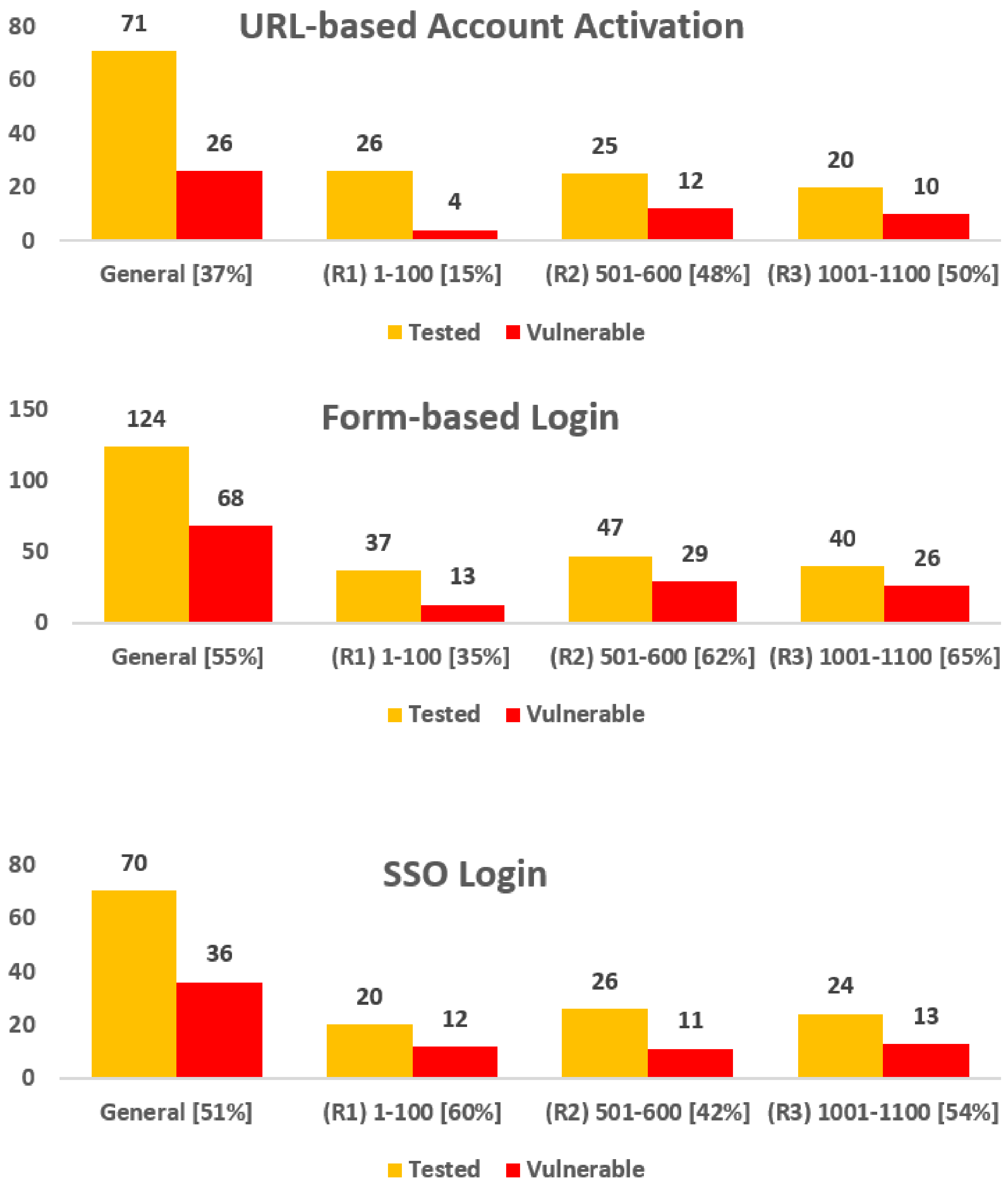
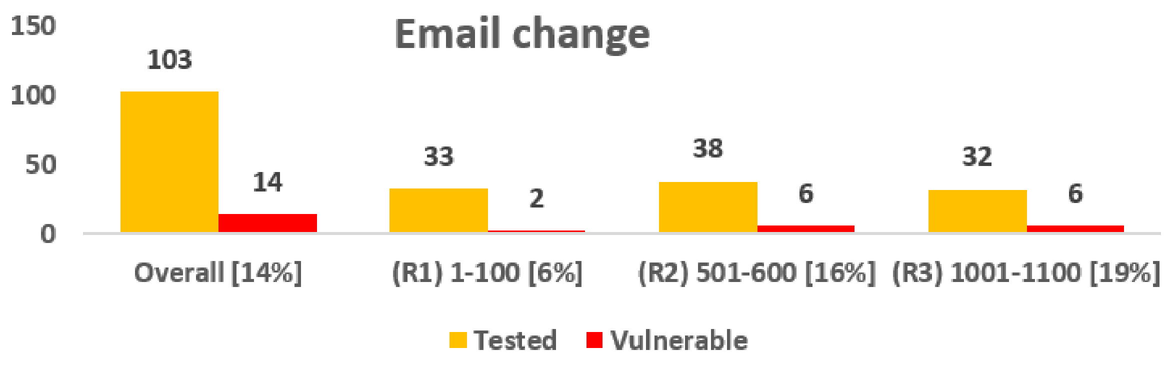


Figure 3.5: Incidence of pre-auth. vulnerabilities



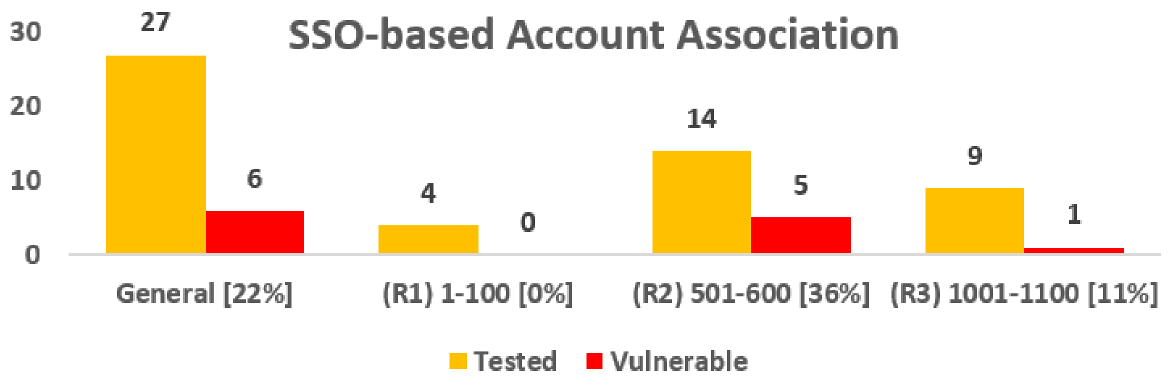


Figure 3.6: Incidence of post-auth. vulnerabilities

(R1) and increases to 62% and 65% respectively within (R2) and (R3) (no statistically significant difference between (R2) and (R3)). As explained in Section 3.2.2, for the custom header validation CSRF defense, in [51], it is suggested to implement the login request (i.e. the HTTP request generated upon submitting the login form) as a XMLHttpRequest for preventing a malicious web site from forging login requests for mounting Login CSRF attack. However, among the web sites we tested, 19 of them implement the login request via XMLHttpRequest but do not complain even if the request is sent as a standard, cross-origin HTTP request (non-XMLHttpRequest) which is allowed by the web browser. This makes these 19 web sites vulnerable to Login CSRF attacks.

SSO Login. Over our corpus of 133 testable web sites, 70 implement the SSO login feature. The overall incidence of a successful attack is about 51%. However, the incidence distribution over the three selected ranges seems to violate the classical trend. In particular the incidence of 60% in (R1) is not lower than in (R2) and (R3). The reason (interestingly enough) being the following. We observe that in (R1) there are 10 well-known service provider web sites owned by big corporations and which use proprietary SSO protocols. Namely: `google.co.in`, `youtube.com` and `blogger.com` owned by Google and associated to `accounts.google.com`; `live.com`, `msn.com`, `bing.com`, `office.com` and `microsoft.com` owned by Microsoft and associated to `login.live.com`; and two other web sites from the same vendor. These SSO protocols were designed to have a single authentication method across several services of the same company. They are not used by third parties and were much more susceptible to CSRF: all these "internal" service providers are vulnerable and Auth-CSRF attacks causing the victim to be authenticated as the attacker can be mounted (explained later in Section 3.6.3). It is interesting to observe that both Google and Microsoft use a different SSO protocol "internally" from the one provided to third-party service providers. For instance, Google provides an OAuth-based protocol for external service providers, while it uses a custom one for its own services. These cases were only encountered in (R1) since smaller-caliber companies encountered in (R2) and (R3) did not feature several services and therefore did not have a proprietary SSO protocol. By focusing on the usual third-party SSO protocols, the trend goes back to the standard one (20% incidence in (R1) versus 42% and 54% for (R2) and (R3)).

In [86, §4.2], the authors mention that 77 out of the 302 web sites implementing OAuth 2.0-based SSO (from the Alexa top 10,000) are vulnerable to Auth-CSRF. The absence of the *state* the parameter—a parameter used for implementing the secret validation token-based CSRF defense (see Section 3.2.2)—was the criterion used to classify a web site as vulnerable. We found this metric to be an unreliable approach to the issue: among the 29 web sites we tested that use an OAuth 2.0-based SSO login protocol, 20 of them use the state parameter and would have been considered safe by the approach mentioned in [86], but when we performed our test, 8 of those 20 were found to be vulnerable (due to improper validation of the state parameter by the service provider). It seems that the state parameter's presence does not imply whether a service provider validates it to prevent Auth-CSRF.

And when considering the OAuth 2.0-based web sites we tested that did not use the state parameter, only 4 of the 9 web sites were actually vulnerable to a CSRF attack. In these instances, the state parameter was replaced by a local dialogue between a child and parent window for CSRF protection.

In the end, our results imply a 40% false negative rate for their metric, and a 44% false positive rate, making it quite unreliable. If we were to combine our values (4/9 web sites that don't use the state parameter are vulnerable and 8/20 web sites that use it are vulnerable) with their findings (77/302 domains not using the state parameter) we can estimate a successful attack on 124 of the 302 web sites, giving us a 41% which exceeds their prediction (25%).

SSO-based Account Association. A few web sites (27/133) offer the possibility to link the user's form-based login account to an existing social account and thereby enabling SSO-based authentication. Forcing an association to an attacker's social account through CSRF allows the attacker to then login to and hijack the user's account. We found out that about 22% of the tested web sites (6/27) are vulnerable to this attack. Given the rarity of this functionality, it is hard to extract reliable proportions from our tests. However, we will show in Section 3.7.3 (Experiment 2) that after considering 52 additional web sites implementing the SSO-based account association process, we find that 17 of them are vulnerable (i.e. 33%).

Email Change. Most of the tested web sites, precisely 103 over 133, feature a post-authentication action for email change (or similar e.g., phone number change). Auth-CSRF attacks were successful in only 14% of these web sites, indicating a less important incidence for this. However the severity of these attacks is obviously high, given that an attacker can trick a user to change the email (or e.g., phone number) and then trigger a password reset to take control over the victim's account. As for most of the previous tests, the incidence starts lower at 6% for range (R1) and slightly increases to 16% for (R2) and to 19% for (R3). It is worth noticing that 19 web sites, outside the ones we tested, do not allow for a modification of the account's main email, i.e. by construction they do not feature the email change action and are therefore safe. Several web sites (precisely 37) featuring email change make use of an additional security mechanism: asking the user for their current account password and sending it with the email-change request. Since we aimed to use email change as an action representative for the overall category of post-authentication actions, we conducted additional experiments to be sure that this protection, which is specific to account settings, was not interfering with our general results. In this respect, we selected 25 web sites among those that had a password-based protection against email change. On these 25 web sites, we tested for CSRF against other post-authentication actions (e.g., add to cart, forum post, etc.). Only 3 web sites over 25 were detected vulnerable to the additional test for CSRF and they are all in range (R3). All together, this had a small impact (few percentage points) on the post-authentication CSRF results presented in Figures 3.4 and 3.6. Additionally, while performing the Auth-CSRF test for email-change, we also ran some tests on the password-change feature. In web application security testing guidelines such as the one provided by OWASP [80], it is explicitly mentioned to protect the password-change feature from CSRF attacks. We ran the password-change test on around 2/3 of the web sites within (R2) and (R3), those having more chances to be vulnerable and only 2 web sites were found to be vulnerable. Perhaps we can infer that since there is an explicit mention of this attack in security testing guidelines, there is a higher awareness from developers and therefore only a few web sites remain vulnerable. Though it is difficult to draw conclusive arguments from this extra experiment, it seems to speak in favor of that inference.

Form-based Registration. We expected CSRF against registration-form to have an evolution very similar to the one for Form-based Login but with a higher protection on registration to prevent mass-registration of fake accounts (e.g., by using captchas). To evaluate this hypothesis, we selected a small set of 18 web sites evenly spread across all three ranges and applied the TS_1 testing strategy explained in Section 3.4 on the registration form. As expected, there was a lower occurrence of registration CSRF (39% with 7 vulnerable web sites) and with one web site as exception, every other web site having a registration form vulnerable to Auth-CSRF also had a login form vulnerable to Auth-CSRF. Additionally, the attack on the registration form is harder to exploit than login-form CSRF: a freshly created account upon submission of the registration details is less likely to be confused with the victim's actual account. However, this is not the case for login-form CSRF as the attacker can first create a convincing forged account to ensure extended usage by the victim before being discovered. Given these information, we considered it unnecessary to perform a registration-form CSRF test on all the web sites having the registration process.

Aware of the issues reported about HTTP Strict Transport Security (HSTS for short) and their potential impact on CSRF, we decided to augment our experimental analysis with an extra test to evaluate whether or not the web site has a proper protection in this respect.

HSTS-enabled Session Management. It has been shown (e.g., [99, 54]) that it is difficult for web sites lacking proper HSTS protection and using secret token validation approach based on cookies for CSRF defense to prevent a network attacker from mounting CSRF attacks. We tested for the presence of the HSTS header with *includesubdomains* option (if the web site under test has sub-domains) against all the 133 testable web sites. The incidence of this issue is extremely high (see Figure 3.7): 75% of the tested web sites are susceptible to this attack. Percentages are a bit better for Alexa top 100 web sites, but still quite frightening (>50%). It seems this security-header is widely ignored, possibly due to the high requirements for a successful exploit compared to the relatively low payoff (i.e. victim can be authenticated as the attacker). However, as shown in [99, §5.1.1], depending on the web site, the impact can be serious.

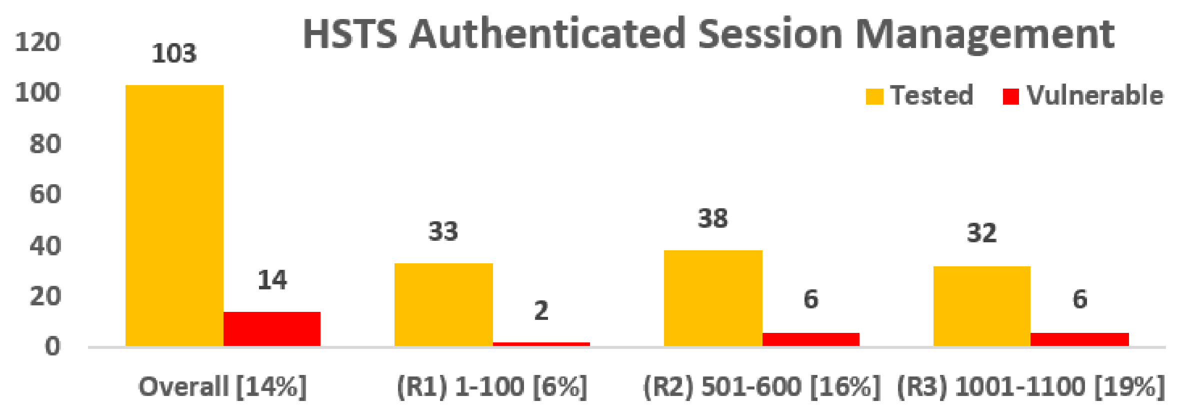


Figure 3.7: Incidence of HSTS

During our experiments, we encountered several vulnerabilities with interesting characteristics. In the following section, we explain a subset of these cases.

3.6 Selected Case Studies

3.6.1 A Very Prominent Adult Website

This vulnerable adult website has an account system that logs a watched-video history. The logged-in state is barely noticeable, so a victim would have trouble identifying the account in which he/she is logged in, especially if the attack is targeted and the attacker uses a believable username for the fake account. After a successful attack (Auth-CSRF using account activation URL), the victim will be logged in as the attacker and all content consumed by the victim will be logged in the attacker's account. The attacker can steal the victim's watch-history and given the nature of such a website, this theft could lead to a breach of privacy or even blackmail.

3.6.2 A Prominent Government Website for Tax Filing

We found a vulnerable government web site where citizens must provide sensitive personal data such as annual income, expenditure, etc. Since it is a government web site, many citizens who may not be aware of web-based attacks might use it to store their personal information. An attacker can perform a targeted attack by performing login form-based Auth-CSRF and waiting for the victim to store his/her personal details on the attacker's account.

3.6.3 Web sites of Google and Microsoft

We noticed that when a user visits `google.com` from a non-US location (e.g., France), there is a redirection to `google.x` where x is the place-holder for the country code (e.g., `google.fr` for France). Additionally, when the user logs in on `google.x`, the following happens. There is a redirection to the URL `accounts.google.x` with an authentication token having name *sidt* as one of the query parameter. This token is used for authenticating the user on `google.x`. There is no CSRF protection for this authentication request. An attacker can perform the following targeted attack against a user residing in Italy: *(i)* the attacker visits `google.it`, performs authentication and intercepts the request to `accounts.google.it` containing *sidt*, *(ii)* the attacker makes the victim visit the URL associated to the intercepted request (e.g., by tempting the victim to click on a hyperlink of the URL). *(iii)* when the victim clicks on the link, the victim is authenticated as the attacker on `google.it` and this enables the attacker to steal the victim's Google search history. This attack is more stealthy than the Login CSRF in Google mentioned in [51] because in our attack, when the victim clicks on the URL sent by the attacker, a blank page will be loaded on the victim's browser. In the meantime, the victim has been silently

authenticated as the attacker. Interestingly, if the victim is logged into all Google services (e.g., `google.it`, `gmail.com`, `youtube.com`, etc.) while clicking the link sent by the attacker, the victim will first be logged out of `google.it` (not other services) and then be logged into `google.it` as the attacker. We noticed that a similar attack is possible on YouTube as there is also a request to `accounts.youtube.com` with the *sidt* parameter having the same purpose as explained above.
Similar problems emerged on Microsoft services such as `bing.com` and `skype.com` (the authentication parameter for Microsoft services is *ANON*). The exploit on `skype.com` is particularly interesting because an attacker can trick the victim into associating the victim's credit card on the attacker's Skype account, allowing the attacker to recharge his/her Skype account using the victim's credit card.

3.6.4 twoo.com

The web site `twoo.com` (Twoo in short) is a dating web site with over 13 million monthly active users. This web site allows users to associate their social accounts. We found that an attacker can silently associate the attacker's Facebook account to the victim's Twoo account. This enables an attacker to authenticate to Twoo as the victim. The following are the steps to perform the attack: *(i)* the attacker needs to initiate the process of associating his/her Twoo account with the his/her Facebook account, *(ii)* intercept the HTTP POST request sent to the URL `https://www.twoo.com/facebook/couple` with the Facebook access token of the attacker in the POST body, *(iii)* make the victim's web browser send the intercepted POST request while the victim is logged in on Twoo (this can be done by making the victim visit an attacker-controlled web page that automatically sends the intercepted POST request). This attack can be serious because many Twoo users store their dating preferences, sexual orientation, credit card details, etc. in their Twoo account.

3.6.5 ebay.com

During our experiments we noticed that when a user requests for primary email change, eBay asks for password confirmation and other security measures such as a captcha. However, the HTTP request containing the new email address neither has CSRF protection, not has any details regarding the user who wants to change the email. This enables the attacker to make the victim's browser send the HTTP request to eBay with an email address that is under the control of the attacker. When this happens, eBay sends a confirmation link to the attacker's email address. The attacker can also send the HTTP request associated to clicking on the confirmation link from the victim's web browser (by embedding the link as the src of an image in a web page controlled by the attacker and loaded on the victim's web browser). When this happens, the primary email associated to the victim's eBay account changes and this enables the attacker to log into victim's eBay account and make purchases using the victim's credit card details stored on eBay.
Additional Details. More information on our communications with vendors and some screencasts for the case-study attacks are available at the companion web site [32] of this chapter.

3.7 (Semi-)Automatic Testing for Auth-CSRF

In Section 3.5 we showed the results of applying each (manual) testing strategy (see Section 3.4) on top web sites. During our experiments we noticed that manually performing certain steps in the testing strategies can be cumbersome and error-prone. For instance, to test a SSO Login process, the tester must intercept the HTTP request carrying the authentication token (Step 2 of TS$_4$). As shown in Section 3.6, in the SSO login implementation of Google and Microsoft, it is difficult to infer from the name of the parameters (i.e. *sidt* and *ANON*) whether they carry authentication tokens. Since there are several parameters syntactically resembling an authentication token, it takes a considerable amount of time for the tester to manually spot the relevant request to intercept. Even if the tester manages to correctly spot the request containing the authentication token, the tester must perform the subsequent steps (i.e. modifying the intercepted request based on reflected/stored criteria shown in Table 3.4 and resending the request) faster before the token expires. All these requirements points to the necessity of having an automated means to perform the challenging steps (of the testing strategies) faster. It is in this context that we introduce CSRF-checker, a tool that assists the tester in detecting vulnerabilities causing Auth-CSRF. In Section 3.7.1 we explain the concept behind CSRF-checker. In Section 3.7.2 we briefly explain the implementation details of CSRF-checker. Section 3.7.3 presents the outcome of our experiments on Alexa top 1500 web sites with CSRF-checker.

3.7.1 CSRF-checker Concept

The tool implements the strategies reported in Figures 3.2a and 3.2b. The tool detects potential *CandidateReq*s (the HTTP request containing a security token or credential) by asking simple questions to the tester. For instance, in the case of SSO Login and Account Association processes, the tool asks the tester to provide the URL of the IdP and to give an input just before authenticating into the IdP. Upon receiving the input from the user, the tool considers all subsequent requests from the IdP's domain to other domains which contain alphanumeric strings either as the value of a URL parameter, or as the value of a parameter in the request body as *CandidateReq*. The same principle is also applicable to URL-based account activation. The tester needs to provide the URL of the mailbox provider and notify the tool just before clicking the activation link. For Form-based Login, Form-based registration and Email/Password-change, to identify the *CandidateReq*, the tool requires the tester to provide the URL of the WUT and the credentials used, i.e. username and password for registration/login and new email/password for email/password change.

3.7.2 Implementation

CSRF-checker is implemented in Python 2.7.12 and uses the API of the widely-used, open-source, penetration testing tool OWASP ZAP [20] to perform standard proxy engine operations such as collecting HTTP traffic to identify the *CandidateReq*, setting proxy rules to alter the HTTP traffic (according to Table 3.4), etc. The source code, installation guide and tutorial for the tool's proof-of-concept implementation can be obtained (upon request) from the chapter's companion web site [32].

3.7.3 Additional experiments with CSRF-checker

Experiment 1. The goal of this experiment was to measure the effectiveness of CSRF-checker in finding vulnerabilities causing Auth-CSRF. In this regard, we checked whether CSRF-checker was able to rediscover 124 vulnerabilities (present in processes P_3-P_6 explained in Section 3.3.2) that we found during our manual experiments (explained in Section 3.5). The end result was that CSRF-checker was able to re-discover 88 of them (i.e. 71%). For the remaining 36 vulnerabilities, the following is what happened: *(i)* in 23 of them the vulnerability was absent during the retest as the vendor fixed the issue (the HTTP traffic of the old and the new experiments clearly indicated the presence of a fix), *(ii)* in 5 of them CSRF-checker crashed during the test (hence no result was obtained), *(iii)* in 7 of them the vul-

nerability was absent and there were no obvious indications of a fix and *(iv)* in 1 case, the vendor fixed the issue but the old vulnerable end-point was still active and hence it was still possible to mount the attack (notice that we would not have known about the existence of this vulnerable end-point if we had not performed the manual experiments explained in Section 3.5).
Experiment 2. The goal of this experiment was to estimate the incidence of Auth-CSRF in the remaining 12 ranges (of 100 web sites) of the Alexa top 1500 that we did not consider for our manual experiments (e.g., 101-200, 601-700, 1401-1500, etc.). To this end, we selected 132 web sites (11 web sites chosen from each of the 12 different ranges) and tested them using CSRF-checker. For this selection, priority was given to web sites having the SSO-based login and account association processes (as the number of web sites having these processes were relatively low in our manual experiments). In the end, CSRF-checker discovered 168 vulnerabilities in 95 of the total 132 tested web sites (i.e. 72%). The percentage of vulnerable web sites for each process is as follows: URL-based account activation 37% (37/100), Form-based Login 58% (75/129), SSO Login 28% (31/111), SSO-based Account Association 33% (17/52), Email-change 11% (8/71). This is more or less in-line with the results we obtained during our manual experiments.

3.8 Ethics & Responsible Disclosure

We ensured that our tests did not cause any harm to the web sites we tested. For instance, we neither injected any code in the HTTP requests nor tried to have unauthorized access to user accounts that are not under our control. All tests were performed using the test accounts we created on the web sites. Our tests can be seen as replaying values from one session in another. This kind of test can cause financial loss to the web site if we had tested processes such as online shopping. For instance, previous studies (e.g., [95, 82]) have shown that it is possible to shop for free from real web sites by replaying payment tokens from one session in another. Since we considered only authentication and identity management processes and replayed only credentials and authentication tokens (belonging to the user accounts we created), our test cases are different from that of [95, 82]. Additionally, when we conducted further tests with CSRF-checker, we made sure that CSRF-checker did not send too many HTTP requests in too short time interval and cause (possible) denial of service attack.

We contacted the vendors of all the vulnerable web sites through the contact information available on the corresponding web sites. A recent study [88] has shown that this procedure is hard to automate in an effective way. On web sites having well-defined communication channels to report security vulnerabilities (precisely 39 web sites, including Google, Mircosoft, Twoo, eBay, etc.), we filed vulnerability reports. For others, we contacted them through the information available on their web sites for general enquiry. We received mostly positive responses for our reports. For instance, Microsoft and Twoo patched the vulnerabilities quickly and paid us bug bounties of $1500 and $500, respectively. LiveJournal and a prominent smartphone company offered us non-monetary rewards for our findings. Google and another prominent company specialized in Internet-related services acknowledged our report. We were denied a bounty because they were already aware of the issue. However, no information regarding these vulnerabilities is publicly available. eBay appreciated our report and fixed the issue immediately. For all other vendors, we are either waiting for the acknowledgements or working closely with them to fix the issues. This is mainly due to the fact that the experiments concluded recently and it has not been long since we reported our findings to the affected vendors. We will update the details at the companion web site of the research presented in this chapter [32].

3.9 Related Work

In [86] the authors developed a crawler that automatically found 302 web sites implementing OAuth 2.0-based SSO and found out that 77 of them were missing CSRF protection parameters. In order to avoid the challenges in automatically executing the SSO login, the crawler was designed to check whether the parameter for CSRF protection was present in the SSO initialization URL. As explained in Section 3.5, we identified that their approach is susceptible to a number of false positives and false negatives.

In [92], the authors conducted a security evaluation of 96 popular web sites implementing the Facebook SSO Login. The authors also encountered the challenge of automatically executing the Facebook SSO Login and similarly preferred a mostly-manually approach (as we did for the experiments mentioned in Section 3.5). This helped them avoid the false positives and false negatives that affected [86]. However, CSRF-checker can provide the same level of accuracy as [92] but with more automation.

In [93] the authors conducted a passive security analysis of 22,000 European web sites. The criteria used to determine if a web site is vulnerable to CSRF is by checking whether the web site has a form that has a long, pseudo-random, hidden element that cannot be guessed or brute-forced by an attacker. Although it is a good criteria for a large-scale evaluation, we noticed that more than 22% of the web sites in our sample do not require a pseudo-random login form element for CSRF protection as they implement login requests via XMLHttpRequest [39] (in the absence of vulnerabilities like XSS, an attacker cannot forge a cross-site XMLHttpRequest). Hence we infer false positives in the approach used in [93].

Past studies [98, 73, 74] have shown that many web sites have an insecure cross-domain policy enabling an attacker to mount CSRF attacks. Since a large-scale evaluation has already been done in this respect, we did not focus on this specific vulnerability.

It has been shown in [99] and [54] that many web sites either lacks or incorrectly implements HSTS protection. During our experiments we also checked whether web sites are correctly implementing HSTS and our results are shown in Figure 3.7.

3.10 Limitations

One main drawback of our approach is that most of the experimental analysis is done manually. In [100] the authors faced challenges similar to ours (i.e. creating an account was necessary to check for vulnerabilities) in conducting large-scale experiments and also followed a manual approach. However, in a later study [101], the very same authors managed to completely automate the execution of Login via Facebook SSO. Since our goal was not to focus on specific protocols, we did not have other choice but to depend on manual means. To mitigate this issue we implemented CSRF-checker, allowing testers to reduce as much as possible the manual effort in conducting the tests, even if, given the generality of our approach, the automation cannot be as advanced as that in [101].

Another drawback of our study is that although we identified a lot of serious vulnerabilities in real web sites—due to the lack of good responsible disclosure plans—we had to manually contact hundreds of affected vendors. Very recently, there has been a study [88] that checked the feasibility of automating the process of vulnerability disclosure. But the conclusion of [88] is that there are no reliable vulnerability notification channels available for researchers who conduct large-scale experiments.

Lastly, we do not propose any novel techniques to tackle Auth-CSRF attacks. Indeed, we believe that currently available techniques—like the secret token validation method—can be sufficient to prevent Auth-CSRF attacks, and promising new techniques (such as same-site cookies [23]) are emerging. Still, more awareness of some CSRF attacks is necessary and we provide a tool supporting the testing phase of web sites.

Chapter 4

Migration to Industry

The research presented in Chapters 2 and 3 shows that browser-based security protocols underlying the critical processes of many prominent web applications contain security vulnerabilities. We also presented techniques for detecting these vulnerabilities. Given that this thesis is in the context of an industrial doctorate, it was important for us to evaluate how well these techniques fit the security testing needs of modern industries. In this chapter, we will discuss our experience in this regard. In particular, we will be focusing on the impacts of our research at SAP, the industrial partner of the research presented in this thesis, and other industries in general.

Structure of the chapter. In Section 4.1, we will review the main contributions of Chapters 2 and 3 and explain why they are important for SAP. In Sections 4.2 and 4.3 we will explain the impacts of the research presented in Chapters 2 and 3 at the industry.

4.1 Importance of our contributions

One of the contributions of Chapter 2 is Blast, the tool we developed for black-box security testing browser-based security protocol implementations (see Section 2.5 for details). The problem that Blast addresses is the absence of general-purpose, application-agnostic techniques for discovering security vulnerabilities present in browser-based security protocol implementations. The types of vulnerabilities that Blast can detect (e.g., logical vulnerabilities [82]) are not very well supported by currently available free and commercial web application penetration testing tools. This problem is indeed very relevant for SAP (and other industries), because browser-based security protocols underlies the MPWA-based solutions offered by SAP to their customers. For instance, the SAP HANA Cloud Platform [24] supports SSO (see [6] for details) and we showed in Section 2.7 that vulnerabilities in SSO implementations can have serious consequences such as user account getting compromised, sensitive personal information theft, etc. SAP also offers an e-commerce solution, namely SAP Hybris [25]. As we showed in Section 2.7, logical vulnerabilities in the implementation of payment checkout protocols at e-commerce solutions can enable for instance attackers to shop for free. This can cause financial loss to SAP customers and severely affect the reputation of SAP.

In Chapter 3, we presented security testing strategies (both manual and semi-automatic) that can help testers uncover Auth-CSRF vulnerabilities. The Auth-CSRF testing strategies we presented for pre-authentication processes can be of particular interest to the security validation team of SAP as widely-used CSRF testing guides (e.g., [33]) mainly focus on post-authentication actions. As we showed in Section 3.6, Auth-CSRF vulnerabilities in pre-authentication processes can cause sensitive information leakage. Vulnerabilities having such impacts can have serious consequences on SAP customers as SAP is a manufacturer of business software.

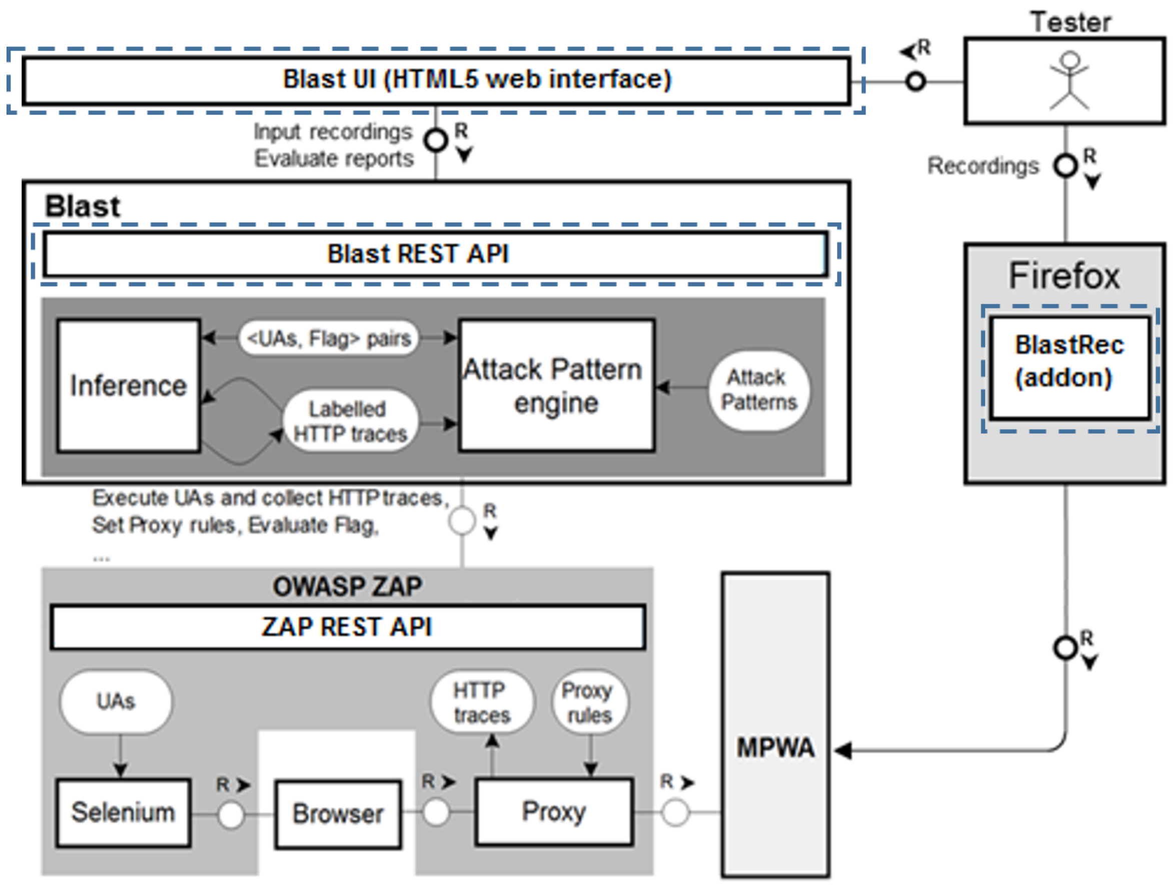


Figure 4.1: Testing Engine Architecture

4.2 Blast and Industry

Although Blast was very well received by the scientific community (e.g., Blast was selected to be presented at multiple prestigious venues [90, 89]), we had to introduce new features in Blast to make it industry-ready. For instance, it was difficult for us in the beginning to promote the usage of Blast within SAP.[1] This was mainly due to the following challenges (labelled *C1*, *C2* and *C3*) in the usage of the first version of Blast: *(C1)* the

[1]Blast was first presented to the developers of SAP when it was selected to be presented at the 2016 SAP's annual Developer Kick-Off Meeting that happened at Karlsruhe, Germany and Silicon Valley, USA.

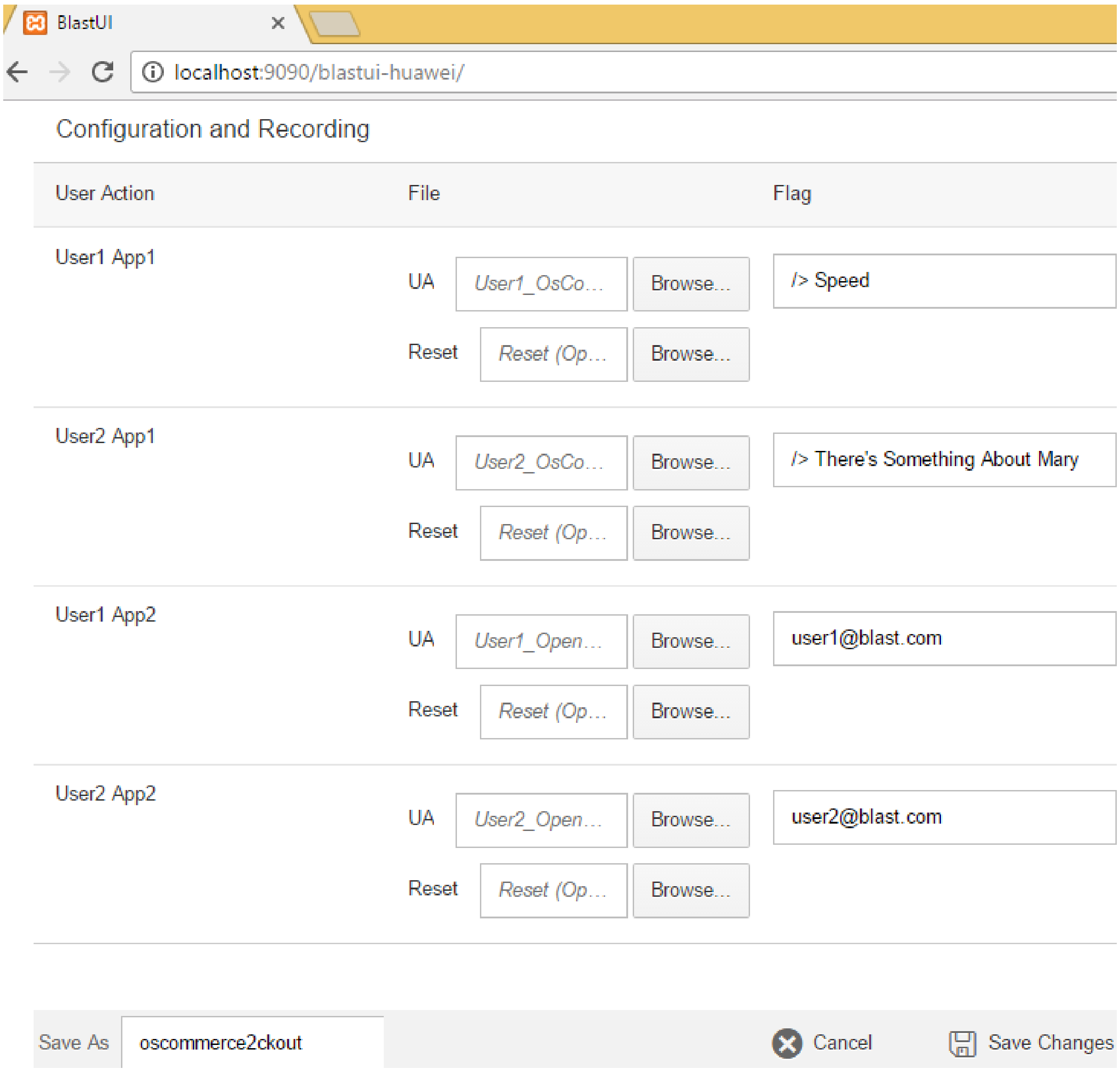
installation and interaction with Blast was not so simple (mainly because Blast was a command line-based tool in its first version), *(C2)* it was necessary to provide user actions in Zest language [35] (a relatively new and experimental scripting language) and *(C3)* the summary of the tests performed by Blast on a MPWA was provided to the tester in a very sophisticated way (in a large JSON file). After having interacted with various developers and security validation teams belonging various business units at SAP (mainly by running pilots), we have been able to find solutions to some of these challenges. In Figure 4.1, we have highlighted (in dotted lines) the components that were absent in the first version of Blast. The major differences between the first version and the latest version are the following:

- The latest version of Blast comes with a graphical user interface (what we refer to as Blast UI) that simplifies the interaction between the tester and Blast. This when combined with Docker and Monsoon Readymade-based installation (see [4, 11]) of Blast helped in solving *C1*. Figure 4.2 shows the HTML5 web interface through which the tester is prompted to provide the user actions for starting a new test. Blast UI communicated with the back-end of Blast via a REST API (shown as Blast REST API in Figure 4.1).

- The recording of the user actions can now be done automatically using BlastRec, our extension of the Selenium Firefox add-on [27]. This feature was introduced to solve *C2*. For instance, Figure 4.3 shows the tester performing certain actions in his/her web browser and Figure 4.4 shows the BlastRec-enabled Selenium Firefox add-on [27] automatically recording these actions.

- In the latest version of Blast, the tester is provided with a dashboard-style overview of the important phases of a test. For instance, Figure

4.5 shows a sample attack report indicating the presence of an RA3 attack (indicated in red color). Further details are also available to the tester such as the the complete list of elements that were replayed while applying each attack strategy. Notice that in the attack report show in Figure 4.5, the tool is reporting an RA3 attack (see 2.3 for details) that was found when the elements *key* and *order_number* were replayed.

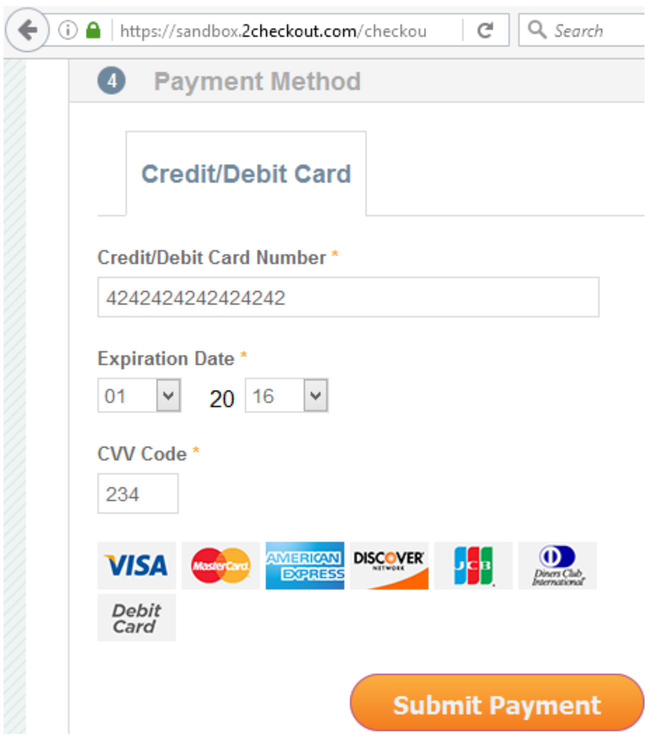


Configuration and Recording

User Action	File			Flag
User1 App1	UA	User1_OsCo...	Browse...	/> Speed
	Reset	Reset (Op...	Browse...	
User2 App1	UA	User2_OsCo...	Browse...	/> There's Something About Mary
	Reset	Reset (Op...	Browse...	
User1 App2	UA	User1_Open...	Browse...	user1@blast.com
	Reset	Reset (Op...	Browse...	
User2 App2	UA	User2_Open...	Browse...	user2@blast.com
	Reset	Reset (Op...	Browse...	

Save As oscommerce2ckout

Figure 4.2: Blast UI for creating a new test


https://sandbox.2checkout.com/checkou

4 Payment Method

Credit/Debit Card

Credit/Debit Card Number *

4242424242424242

Expiration Date *

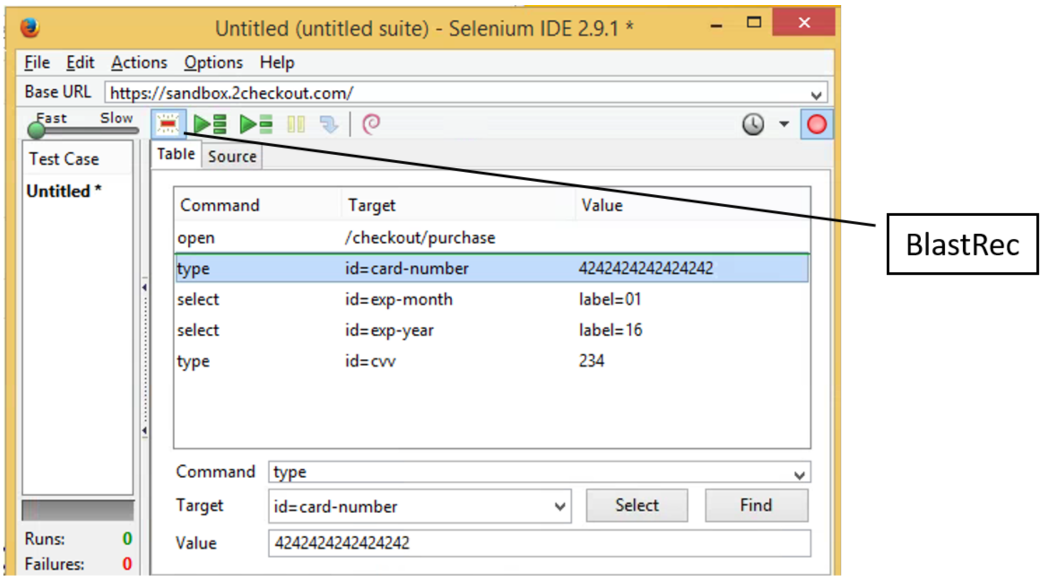
01 20 16

CVV Code *

234

Submit Payment

Figure 4.3: Payment via 2checkout


Untitled (untitled suite) - Selenium IDE 2.9.1 *

Base URL https://sandbox.2checkout.com/

Command	Target	Value
open	/checkout/purchase	
type	id=card-number	4242424242424242
select	id=exp-month	label=01
select	id=exp-year	label=16
type	id=cvv	234

Command: type

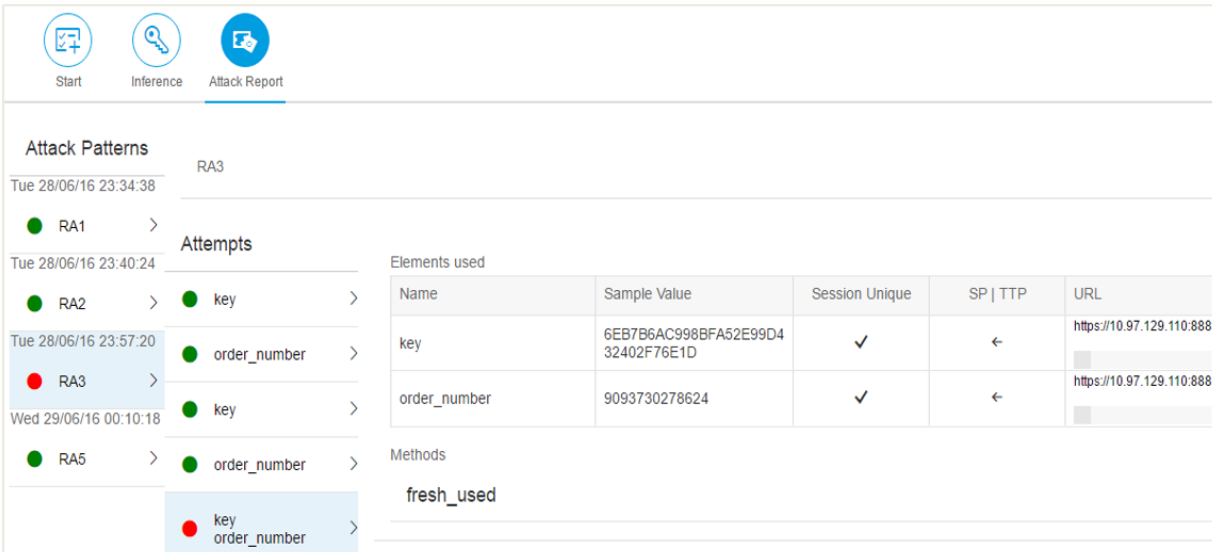
Target: id=card-number

Value: 4242424242424242

Runs: 0

Failures: 0

Figure 4.4: Recording of User Actions via BlastRec



Attack Patterns

Tue 28/06/16 23:34:38
● RA1
Tue 28/06/16 23:40:24
● RA2
Tue 28/06/16 23:57:20
● RA3
Wed 29/06/16 00:10:18
● RA5

RA3

Attempts

● key
● order_number
● key
● order_number
● key order_number

Elements used

Name	Sample Value	Session Unique	SP | TTP	URL
key	6EB7B6AC998BFA52E99D4 32402F76E1D	✓	←	https://10.97.129.110:888
order_number	9093730278624	✓	←	https://10.97.129.110:888

Methods

fresh_used

Figure 4.5: A Sample Attack Report

Our efforts to make Blast an important security testing tool at SAP was very well appreciated and supported by the management. For instance,

1. SAP filed a US patent application for the idea behind Blast (Patent Id: 14/885,001). This shows that SAP is interested in protecting the intellectual idea behind Blast.

2. SAP agreed to allocate three (paid) working students (Adrien Hubner, Nicolas Dolgin and Mathieu Molinengo) to help in the development of Blast.

3. Blast was selected to be presented at the 2016 and 2017 SAP DKOM (SAP's annual Developer Kick-Off Meeting that happened at Karlsruhe, Germany and Silicon Valley, USA).

4. multiple business units at SAP (e.g., SAP Hybris security team) and partners of SAP (e.g., `open.sap.com` from Hasso Plattner Institute) agreed to use Blast for security testing their browser-based security protocol implementations. The outcomes were well received and considered valuable to increase even further the security of their prod-

ucts. For instance, the developers of `open.sap.com` acknowledged us at their web site (see here `open.sap.com/pages/about`).

5. SAP is continuing on its own, piloting the usage of Blast.

Interestingly, the popularity of Blast was not just within SAP. For instance, the members of the security testing team of Huawei Technologies Co. Ltd., read our paper on Blast [90] and requested us to test their web site that integrated multiple browser-based security protocols. All these activities are still ongoing.

4.3 Auth-CSRF and Industry

After discovering that 181 Alexa top web sites were vulnerable to pre-authentication CSRF attacks (having serious privacy impacts), and given that widely-used CSRF testing guides followed by industries (e.g., the one provided by OWASP [33]) do not mention about this issue, we decided to spread awareness. In this regard, we made a presentation on CSRF at the 2017 SAP DKOM (Karlsruhe, Germany). This presentation was mainly targeting SAP developers and we focused on the importance of protecting both pre- and post-authentication processes from CSRF. We also discussed possible mitigations and the security testing strategies explained in Sections 3.4 and 3.7. Although in the beginning our audience were skeptical about pre-authentication CSRF attacks, the important findings of our experimental analysis (explained in Section 3.6) convinced them.

To spread awareness on Auth-CSRF outside SAP, we submitted a proposal for a talk (on Auth-CSRF) at the upcoming 2017 OWASP AppSec Europe [17]. Additionally, we contacted Matteo Meucci and Andrew Muller, the project leaders of the OWASP Testing Guide Project [18] to discuss the possibility of extending the definition of CSRF used by OWASP (see [3]) to include pre-authentication CSRF. This could possibly lead to the inclusion of our pre-authentication CSRF testing strategies (explained in Section 3.4) at the OWASP Testing Guide [80]. However, these discussions are still ongoing.

Ultimately, the research presented in this thesis has led to the discovery of serious vulnerabilities affecting the products developed by prominent vendors (e.g., Microsoft, Google, Linkedin, etc.) and consumed by millions of users. We also responsibly disclosed these vulnerabilities to the affected vendors and helped them fix the issues. So, overall the impact of our research at the industry is fairly good.

Chapter 5

Conclusion

In this thesis we presented different techniques for detecting vulnerabilities in the design and implementation of browser-based security protocols. In particular, in Chapter 2, we presented an approach for black-box security testing of MPWAs. The core of our approach is the concept of application-agnostic attack patterns. These attack patterns are inspired by the similarities in the attack strategies of previously-discovered attacks against MPWAs. The implementation of our approach is based on OWASP ZAP, a widely-used open-source legacy penetration testing tool. By using our approach, we have been able to *(i)* identify serious security drawbacks in the browser-based security protocols underlying the SSO and CaaS solutions offered by Linkedin, PayPal and Stripe, *(ii)* identify previously-unknown vulnerabilities in a number of websites leveraging the SSO solutions offered by Facebook and Instagram and *(iii)* automatically generate test cases that reproduce previously-known attacks against vulnerable integration of the 2Checkout service.

The findings of our study on CSRF presented in Chapter 3 indicate that developers often fail to protect sensitive processes from CSRF attacks and that the default CSRF protection offered by web frameworks and automatic/semi-automatic CSRF prevention mechanisms cannot protect web sites from many CSRF attacks. This shows the importance of security testing web sites for CSRF attacks. Although it is difficult to target all CSRF attacks, we focused on an important subclass of CSRF, namely Auth-CSRF. We showed that manual and semi-automatic security testing strategies (presented in Sections 3.4 and 3.7) can assist web developers in testing web site for Auth-CSRF. We showed experimental evidence in this regard by testing Alexa top 1500 web sites and discovering serious security vulnerabilities enabling Auth-CSRF. We responsibly disclosed all our findings to the affected vendors.

Last but not the least, it is worth noting that the outcomes of the research presented in this thesis helped in improving the security of the browser-based security protocol implementations of SAP and other companies in general (e.g., Microsoft, Linkedin, etc.). There is ample experimental evidence on the effectiveness of our approaches and hence they can be extended further in the future (e.g., the approach behind Blast can be extended to the mobile application scenario, CSRF-checker can be further extended to automatically detect vulnerabilities enabling Auth-CSRF etc.).

Bibliography

[1] Account hijacking by leaking authorization code. http://www.oauthsecurity.com/.

[2] Covert Redirect. http://oauth.net/advisories/2014-1-covert-redirect/.

[3] Cross-Site Request Forgery (CSRF). https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[4] Docker (software). https://en.wikipedia.org/wiki/Docker_(software).

[5] HTML5 Web Messaging. http://www.w3.org/TR/webmessaging/#posting-messages.

[6] Identity and Access Management. https://help.hana.ondemand.com/help/frameset.htm?e6b196abbb5710148c8ec6a698441b1e.html.

[7] Instagram API Console. https://apigee.com/console/instagram.

[8] Integrate Log In with PayPal. https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/.

[9] Log In with PayPal demo site. https://lipp.ebaystratus.com/loginwithpaypal-live/.

[10] LogIn to experience INstant. http://instant.linkedinlabs.com/.

[11] Monsoon Readymade. https://monsoon.mo.sap.corp/.

[12] The most common oauth2 vulnerability. http://homakov.blogspot.it/2012/07/saferweb-most-common-oauth2.html.

[13] The most common oauth2 vulnerability. http://homakov.blogspot.it/2012/07/saferweb-most-common-oauth2.html.

[14] Mozilla Identity Team. Persona. https://login.persona.org/.

[15] OAuth 2.0 Playground. https://developers.google.com/oauthplayground/.

[16] OAuth Security Advisory: 2009.1. http://oauth.net/advisories/2009-1/.

[17] OWASP AppSec Europe 2017 Belfast - 8th-12th of May. https://2017.appsec.eu/.

[18] OWASP Testing Project. https://www.owasp.org/index.php/OWASP_Testing_Project.

[19] Owasp Top Ten 2013 Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2013_Project.

[20] OWASP Zed Attack Proxy Project. https://www.owasp.org/index.php/ZAP.

[21] PayPal Express Checkout. https://www.paypal.com/webapps/mpp/referral/paypal-express-checkout.

[22] PayPal Payments Standard. https://www.paypal.com/webapps/mpp/paypal-payments-standard.

[23] Same-site Cookies draft-west-first-party-cookies-07. `https://tools.ietf.org/html/draft-west-first-party-cookies-07`.

[24] Sap hana cloud platform. `https://hcp.sap.com/index.html`.

[25] SAP Hybris. `https://www.hybris.com/en/`.

[26] SECENTIS: A European Industrial Doctorate on Security and Trust of Next Generation Enterprise Information Systems. `http://www.secentis.eu/home`.

[27] Selenium IDE. `https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/`.

[28] Selenium WebDriver. `http://docs.seleniumhq.org/projects/webdriver/`.

[29] Selenium with Python. `http://selenium-python.readthedocs.io/index.html`.

[30] Stripe Checkout. `https://stripe.com/docs/checkout`.

[31] Stripe Wiki. `http://en.wikipedia.org/wiki/Stripe_%28company%29`.

[32] Supporting materials. `https://sites.google.com/site/authcsrf/`.

[33] Testing for CSRF (OTG-SESS-005). `https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)`.

[34] The Jython Project. `http://www.jython.org/`.

[35] The ZAP Zest Add-on. `https://code.google.com/p/zap-extensions/wiki/AddOn_Zest`.

[36] Token Fixation in PayPal. http://homakov.blogspot.it/2014/01/token-fixation-in-paypal.html.

[37] UI Development Toolkit for HTML5. https://sapui5.hana.ondemand.com.

[38] Vulnerability Reawards Program Rules. https://hackerone.com/twitter.

[39] XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[40] XUL. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL.

[41] Mail from peter watkins about csrf. http://www.tux.org/~peterw/csrf.txt, 2001.

[42] OAuth 2.0 Threat Model and Security Considerations. https://tools.ietf.org/html/rfc6819#section-4.4.2.2, January 2013.

[43] Sign-up Form CSRF. https://hackerone.com/reports/7865, 2014.

[44] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. CSF '10, pages 290–304, Washington, DC, USA, 2010. IEEE Computer Society.

[45] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In V. Shmatikov, editor, *Proc. ACM FMSE*, pages 1–10. ACM Press, 2008.

[46] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The avantssar platform for the automated validation of trust and security of service-oriented architectures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282. Springer, 2012.

[47] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *International Conference on Computer Aided Verification*, pages 281–285. Springer, 2005.

[48] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti. From multiple credentials to browser-based single sign-on: Are we more secure? volume 354 of *IFIP Advances in Information and Communication Technology*, pages 68–79. Springer, 2011.

[49] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Proceedings of the 20th NDSS'13, San Diego, CA, USA*, 2013.

[50] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF, 2012 IEEE 25th*, pages 247–262, June 2012.

[51] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[52] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[53] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Ietf rfc 1945 hypertext transfer protocol (http/1.0), 2008.

[54] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.

[55] Roland Bischofberger and Emanuel Duss. Saml2 burp plugin. *Management*, 1:6, 2015.

[56] 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 '10, pages 607–618, New York, NY, USA, 2010. ACM.

[57] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. *Web 2.0 Security and Privacy (W2SP)*, 2011.

[58] Josip Bozic, Dimitris E. Simos, and Franz Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International*

Workshop on Automation of Software Test, AST 2014, pages 1–7, New York, NY, USA, 2014. ACM.

[59] Jesse Burns. Cross site request forgery. *An introduction to a common web application weakness, Information Security Partners*, 2005.

[60] Eric Chen, Shuo Chen, Shaz Qadeer, and Rui Wang. Securing multiparty online services via certification of symbolic transactions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[61] OASIS Consortium. SAML V2.0 Technical Overview. `http://wiki.oasis-open.org/security/Saml2TechOverview`, March 2008.

[62] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J Wang. Lightweight server support for browser-based csrf protection. In *Proceedings of the 22nd international conference on World Wide Web*, pages 273–284. International World Wide Web Conferences Steering Committee, 2013.

[63] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[64] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against csrf attacks. In *Computer Security–ESORICS 2011*, pages 100–116. Springer, 2011.

[65] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny cant pentest: An analysis of black-box web vulnerability scanners. In

International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 111–131. Springer, 2010.

[66] Jeremiah Grossman. I used to know what you watched, on youtube, 2008.

[67] E Hammer-Lahav. Oauth security advisory: 2009.1, 2009.

[68] Alex Infuhr. Pdf - mess with the web. In *OWASP AppSec EU*, 2015.

[69] Martin Johns and Justus Winter. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[70] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.

[71] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[72] Florian Kerschbaum. Simple cross-site attack prevention. In *SecureComm 2007*, pages 464–472. IEEE, 2007.

[73] Sebastian Lekies, Martin Johns, and Walter Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web*, volume 2, 2011.

[74] Sebastian Lekies, Nick Nikiforakis, Walter Tighzert, Frank Piessens, and Martin Johns. Demacro: defense against malicious cross-domain requests. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012.

[75] Sebastian Lekies, Walter Tighzert, and Martin Johns. Towards stateless, client-side driven cross-site request forgery protection for web applications. In *Sicherheit*, pages 111–121, 2012.

[76] Rich Lundeen. The deputies are still confused. In *Blackhat EU*, 2013.

[77] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Do not trust me: Using malicious idps for analyzing and attacking single sign-on. *CoRR*, abs/1412.1623, 2014.

[78] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration testing tool for web services security. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 163–170. IEEE, 2012.

[79] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security*, pages 238–255. Springer, 2009.

[80] Matteo Meucci and Andrew Muller. The owasp testing guide 4.0, 2014.

[81] Giancarlo Pellegrino. *Detection of logic flaws in multi-party business applications via security testing.* PhD thesis, Thesis, 11 2013.

[82] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS Symposium 2014*. Internet Society, 2014.

[83] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW ’98, pages 71–79, New York, NY, USA, 1998. ACM.

[84] Stephen Sclafani. Csrf vulnerability in oauth 2.0 client implementations. `http://stephensclafani.com/2011/04/06/oauth-2-0-csrf-vulnerability/`.

[85] Hossain Shahriar and Mohammad Zulkernine. Client-side detection of cross-site request forgery attacks. In *IEEE 21st International Symposium ISSRE, 2010*, pages 358–367. IEEE, 2010.

[86] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, pages 239–260, Cham, 2015. Springer International Publishing.

[87] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 397–412, Bellevue, WA, 2012. USENIX.

[88] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1015–1032, Austin, TX, 2016. USENIX Association.

[89] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. Attack patterns for black-box detection of logical vulnerabilities in multi-party web applications. In *OWASP AppSec Europe*, 2016.

[90] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. Attack patterns for black-box security testing of multi-party web applications. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[91] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS 2014, California, USA, February 23-26, 2013*, 2014.

[92] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[93] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In *International Conference on Trust and Trustworthy Computing*, pages 110–126. Springer, 2014.

[94] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 365–379, Washington, DC, USA, 2012. IEEE Computer Society.

[95] 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 *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 465–480, Washington, DC, USA, 2011. IEEE Computer Society.

[96] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating sdks: Uncovering assumptions underlying

secure authentication and authorization. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 399–414, Berkeley, CA, USA, 2013. USENIX Association.

[97] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. In *Network & Distributed System Security Symposium (NDSS)*, February 2013.

[98] William Zeller and Edward W Felten. Cross-site request forgeries: Exploitation and prevention, princeton (2008).

[99] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies lack integrity: Real-world implications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 707–721, Washington, D.C., August 2015. USENIX Association.

[100] Yuchen Zhou and David Evans. Why aren't http-only cookies more widely deployed. *Proceedings of 4th Web*, 2, 2010.

[101] Yuchen Zhou and David Evans. Ssoscan: Automated testing of web applications for single sign-on vulnerabilities. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 495–510, CA, USA, 2014. USENIX Association.