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

Avinash Sudhodanan
University of Trento, Italy
Security & Trust, FBK, Italy
6.avinash@gmail.com

Alessandro Armando
DIBRIS, University of Genova
Security & Trust, FBK, Italy
armando@fbk.eu

Roberto Carbone
Security & Trust, FBK, Italy
carbone@fbk.eu

Luca Compagna
SAP Labs France
luca.compagna@sap.com

***Abstract*—The advent of Software-as-a-Service (SaaS) has led to the development of multi-party web applications (MPWAs). MPWAs rely on core trusted third-party systems (e.g., payment servers, identity providers) and protocols such as Cashier-as-a-Service (CaaS), Single Sign-On (SSO) to deliver business services to users. Motivated by the large number of attacks discovered against MPWAs and by the lack of a single general-purpose application-agnostic technique to support their discovery, we propose an automatic technique based on attack patterns for black-box, security testing of MPWAs. 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. In this paper, we target six different replay attacks, a login CSRF attack and a persistent XSS 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 for testing the security of MPWAs. We implemented our ideas on top of OWASP ZAP (a popular, open-source penetration testing tool), created seven attack patterns that correspond to thirteen prominent attacks from the literature and discovered twenty one 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.**

## I. INTRODUCTION

An increasing number of business critical, online 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 to obtain proof-of-payment before delivering the purchased items (e.g., Express Checkout [11] and PayPal Payment Standard [12]). We refer to this broad class of protocols as security-critical *Multi-Party Web Applications* (MPWAs). 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* [38]. 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 [23]. (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 [35], [32]. For instance, a vulnerability in osCommerce v2.3.1 that allowed an attacker to shop for free has been reported in [32]: 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 [18] which is very similar to the session fixation attack in OAuth 1.0 [10]. 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 [40].

The aforementioned attacks have been discovered through a variety of domain-specific techniques with different levels of complexity, ranging from formal verification [23], white-box analysis [35], black-box testing [32], to manual testing [18]. In this paper, 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.

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23286

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 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 [15]); 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 family of MPWAs, namely the Verificaton Via Email (VvE) scenario, which is often used by websites to send security-sensitive information to users via email. By testing the security of Alexa Top 500 websites 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 paper:

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 [39], [29]. However, ours is the first *black-box* security 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[2] 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 paper.* In Section II, 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 III. In Section IV we show how the attack patterns we defined can be used to carry out black-box testing of MPWAs. In Section V, we provide some details about our prototype implementation. We discuss the experimental evaluation in Section VI. In Section VII we discuss the related work and in Section VIII we discuss the limitations of our approach. We conclude in Section IX with some final remarks.

## II. BACKGROUND

Figure 1 provides pictorial representations of example MPWAs leveraging SSO, CaaS, and Verification via Email (VvE) solutions. 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.

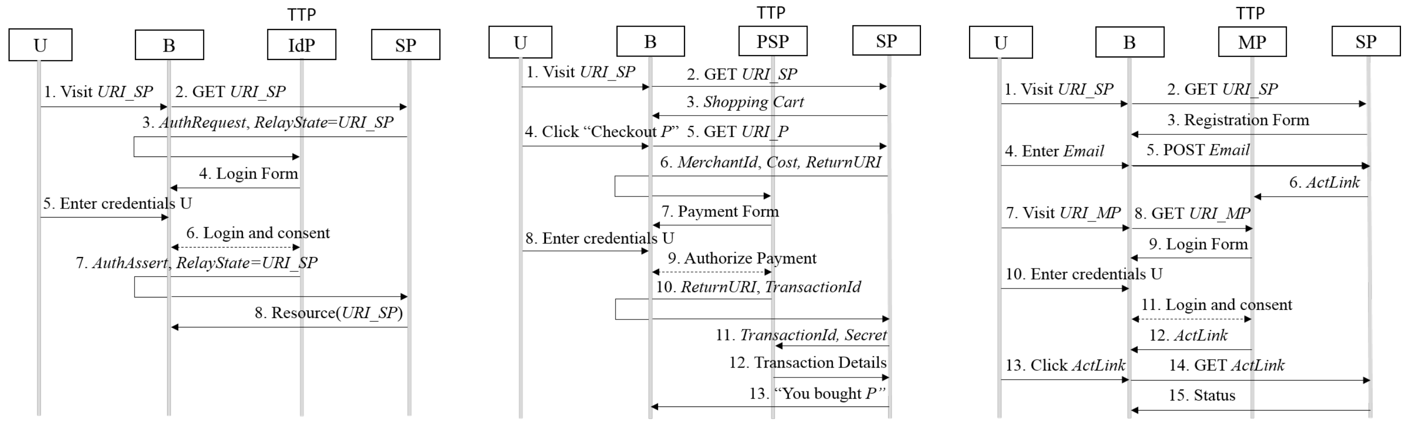
Figure 1a shows the SAML 2.0 SSO protocol [30], where *SP* relies on *TTP* (the Identity Provider, *IdP* for short) to authenticate a user *U* before granting the user access to one of its resources. The protocol starts (steps 1-2) with *U* asking *SP* for a resource located at *URI_SP*. *SP* in turn redirects *B* to *IdP* with the authentication request $AuthRequest$ (step 3). The $RelayState$ field carries *URI_SP*. *IdP* then challenges *B* to provide valid 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 [23]. 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 1b illustrates a typical MPWA running the PayPal Payments Standard CaaS protocol [12] 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

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

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

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



(a) SAML-based SSO (b) PayPal Payments Standard CaaS (c) Email notification and acknowledgment

Fig. 1: Typical MPWA scenarios

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 [32]. The attack is as follows: from a session $S_1$ of the protocol involving the *PSP* and the malicious party controlling both a user ($U_M$) and a *SP* ($SP_M$), the malicious party obtains a payee (merchant) identifier. Later, in the checkout protocol session $S_2$ between $U_M$ and the target SP ($SP_T$), the malicious agent replays the value of *PayeeId* obtained in the other session and manages to place an order for a product in $SP_T$ by paying herself (instead of $SP_T$).

While MPWAs for SSO and CaaS scenarios received a considerable attention (see, e.g., [29], [34], [35], [37], [36], [39], [32]), 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 1c. 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).

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.

## A. Attacks

Table I 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), corresponding to ***#1*** for SAML SSO, and ***#3*** for PayPal Payments Standard. We do not consider here XSS and XML rewriting attacks (see Section VII 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 [32], 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 $SP_T$ in the session between $U_V$ and $SP_M$ to obtain $AccessToken$ of $U_V$. The very same $AccessToken$ is then replayed by the attacker in a session between $SP_T$ and $U_M$ to authenticate as $U_V$ at $SP_T$. In [36], 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 $\mathrm{SP_T}$.

## B. Threat Models

The attacks shown in Table I can be discovered by considering the *Web Attacker* threat model introduced in [21] 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 [31]. We do not consider this threat scenario.

## III. From Attacks to Attack Patterns

A close inspection of the attacks in Table I 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 I and then we abstract them into general, i.e. application-independent, attack patterns.

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

- 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 $R$.

For the sake of simplicity, we present in the overall paper 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 I are given in Table II.

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 $AuthAssert$ ($AuthCode$, resp.) in a session played by $\mathrm{U_M}$ on $\mathrm{SP_T}$ is replaced on a sequence comprising a single HTTP request in which $\mathrm{U_M}$ sends a HTTP request to $\mathrm{SP_T}$ (denoted as $[\mathrm{U_M}$ SEND $req]$). Thus, the result should be the same obtained by completing a session between $\mathrm{U_M}$ and $\mathrm{SP_T}$.

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

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

[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.

TABLE I: Attacks against security-critical Multi Party Web Applications

| # | Vulnerable MPWA | Description of the Attack | Attacker's Goal |
|---|---|---|---|
| 1 | SPs implementing Google's SAML SSO [23, §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 [38, §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 [32, §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) [35, §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 [32, §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 [36, §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 [24, §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) [25, §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 $\mathrm{SP_T}$ |

TABLE II: Known Attacks Strategies against MPWAs

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

TABLE III: Attack Patterns

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

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

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 the illustrative example of Section IV.

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 $(\mathrm{U_V}, \mathrm{SP_M})$. 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 meaning is summarized in Figure 2. The preconditions in Table III 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 (inspired by attacks #7 and #8), 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 Figure 2).

In Table III, 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 [20]. 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 determine whether the attack strategy they executed was successful to detect any attack. The postconditions included in Table III serve this purpose. The idea is that each one of the four nominal sessions is associated with a *Flag* 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 a program that checks whether a certain *Flag* is captured or not while executing the strategy. A value of the form $(U, SP)$ in the column Postcondition 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.

## IV. Approach

Figure 3 outlines the two processes underlying our approach. In the first one, executable attack patterns are created, reviewed, and improved by security experts (see Section IV-A). 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 IV-B for details.

### A. 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, have clearly both. Security experts can thus read and understand attack patterns like those sketched in Table III. Improving an attack pattern, by changing few

Syntactic labels provide type information:[a]

- URL: a URL, e.g.,redirect_uri=http://google.com,
- BLOB: an alphanumeric string with (optionally) special characters, e.g., code=vrDK7rE4,
- WORD: a string comprised only of alphabetic characters, e.g., response_type=token,
- EMAIL: an email address, e.g., usrname=jdoe@example.com,
- EMPTY: an empty value, e.g., state=,
- NUMBER: a number, e.g., id=5,
- BOOL: a boolean value, e.g., new=true, and
- UNKNOWN: none of the other syntactic labels match this string, e.g., #target.

Semantic labels provide information on the role played by the element within the MPWA:[b]

- SU (Session Unique): the element is assigned different values in different sessions.
- UU (User Unique): the element is assigned the same value in the sessions of the same user.
- AU (App Unique): the element is assigned the same value in the sessions of a single *SP*.
- MAND (Mandatory): the element must occur for the protocol to complete successfully.
- RURI (Redirect URI): 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.

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.

[a]Most of the syntactic labels are borrowed from [36], [32]

[b]While the SU and UU labels are borrowed from [36], the AU and RURI labels are new. The MAND label generalizes the SEC label introduced in [36], 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 necessarily secret and SU.

Fig. 2: Syntactic, Semantic, Flow and Location Labels

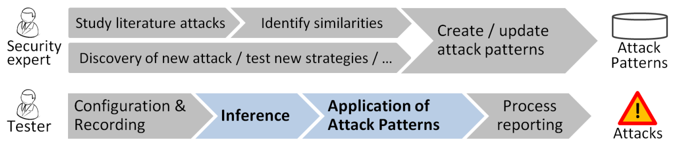

Security expert
Study literature attacks
Identify similarities
Discovery of new attack / test new strategies / ...
Create / update attack patterns
Attack Patterns
Tester
Configuration & Recording
Inference
Application of Attack Patterns
Process reporting
Attacks

Fig. 3: Approach

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 III, with the only exception of RA5, all attack patterns in Table III 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 [20, §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 VI, 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.

## B. Security Testing Framework

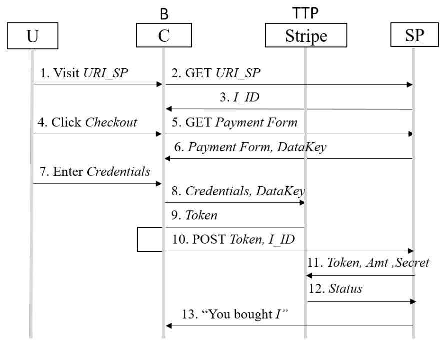
The different phases of our security testing framework are described below. Figure 4 shows how these phases concretely apply on the following illustrative example: The developer Diana has implemented the Stripe checkout solution in her web application. She is required to ensure that *(r1)* the new feature 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.

**(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., [7], [6], [9], [4]—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

The Stripe checkout protocol is illustrated in Figure 4a. It is slightly different than the PayPal Payments Standard presented in Figure 1b. 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).



(a) Stripe checkout protocol

**(P1) Configuration.** Diana uses the *SP* she implemented as $\mathrm{SP_T}$ and the official reference implementations provided by Stripe [14] as $\mathrm{SP_M}$. For each of them, she creates the two user accounts $\mathrm{U_V}$ and $\mathrm{U_M}$.

**(P2) Recording.** Table 4b 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 4a, while the *Flag* is derived from step 13 in Figure 4a ($I_1$-$I_4$ indicate four different items).

**(P3) Inference.** An excerpt of the inference results of the protocol underlying Diana's implementation of the Stripe checkout protocol is shown in Table 4c.

**(P4) Application of Attack Patterns.** The result of applying each attack pattern of Table III on this example is reported in Table 4d.

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

(b) User Actions and Flags of Stripe Checkout

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

(c) Excerpt of Inference on Stripe Checkout

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

(d) 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 4a, $\mathrm{U_V}$'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 $S$ 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 $[\mathrm{U_M}$ SEND $req]$.<br>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. |

Fig. 4: Security Testing Framework on an illustrative example

WebDriver [13] and Zest [17] 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 Figure 2. We do not exclude that in the future more information (e.g., inference of the observable workflow of the MPWA [32]) could be necessary to target more complex attacks. While we borrow the idea of inferring the syntactic and semantic properties from [36] and [32], we introduce the concept of inferring flow labels to make our approach more automatic (compared to [36]) and efficient (less no. of test cases for detecting the same attack mentioned in [32]).

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 the reported attacks.

## V. 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 5 outlines the high-level architecture of our testing engine. The *Tester* provides the necessary input to our *Testing Engine* that in turns employs *OWASP ZAP* to probe the *MPWA*.[5] 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 6. Each step is tagged by a number to simplify the presentation of the flow.

*1) Inference:* With reference to the steps of Figure 6, the following activities are performed by the inference module after the tester records (step 1) the four $\langle UAs, Flag\rangle$ corresponding to sessions $S_1$, $S_2$, $S_3$, and $S_4$ in **(P2)**.

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

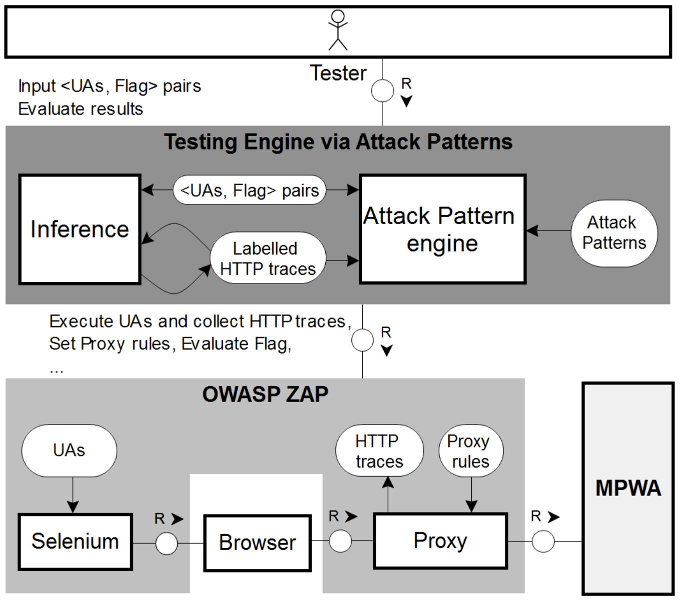


Fig. 5: Testing Engine Architecture

*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 4a) is depicted in Figure 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 7 is the $Token$ shown in step 10 of Figure 4a.

*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 Figure 2. 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-execute 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 7) is assigned the syntactic labels BLOB and the semantic labels SU and MAND.

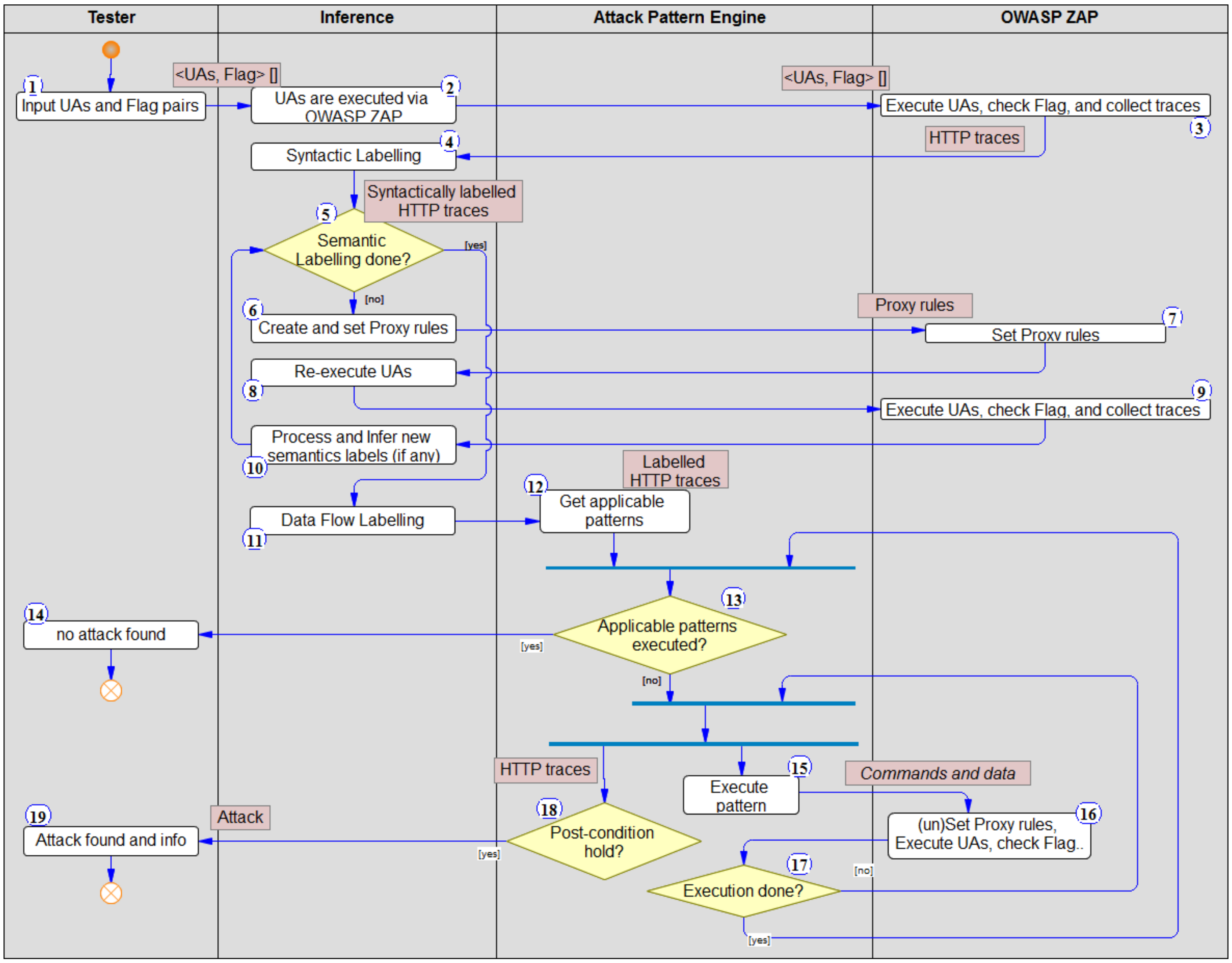


Fig. 6: Testing Engine Flow

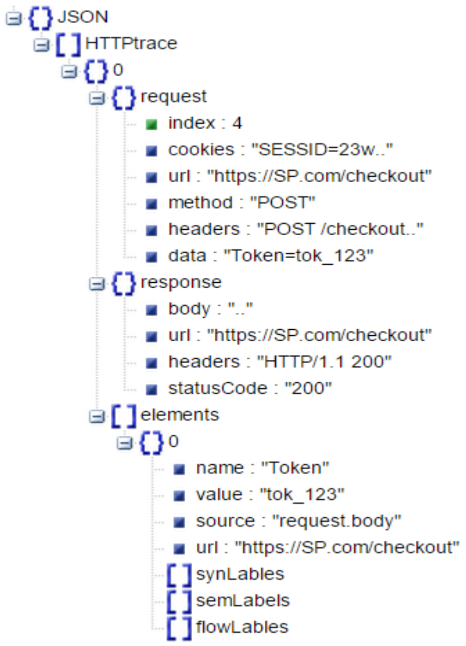
*Data Flow Labeling (step 11)* After syntactic and semantic labeling, the data flow properties of each MAND element in the trace is analyzed to identify the data flows (either TTP-SP or SP-TTP). In order to identify the protocol patterns, it is necessary to distinguish *TTP* and *SP* from the HTTP trace. 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) Attack Pattern Engine:* For the simplicity of explanation, we represent our attack patterns in the same way as the attack graph notation introduced in [33]. Each attack pattern has a `Name`, the underlying `Threat model`, `Inputs` used, the `Goal` 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 6). As soon as the pattern `Preconditions` hold, the `Actions` are executed (steps 15-17 of Figure 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 6), an attack report is generated with the configuration that caused the attack (step 19 of Figure 6).

*Example on Attack Pattern for RA1.* To illustrate, let us consider the Replay Attack pattern RA1 reported in Table III. In Listing 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



```
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
```

Fig. 7: HTTP trace with empty labels (an excerpt)

Listing 1: Attack Pattern for RA1

```
Name: RA1                                                        1
Threat Model: Web Attacker                                       2
Inputs:  UAs($U_V$, $SP_M$), LHT($U_V$, $SP_M$),                 3
         UAs($U_M$, $SP_T$), Flag($U_V$, $SP_T$)                 4
Preconditions: At least one element x in LHT($U_V$, $SP_M$)      5
  is such that (TTP-SP $\in$ x.flow AND (SU|UU) $\in$x.labels)   6
Actions:                                                         7
  For each x such that preconditions hold                        8
  e = extract(x,UAs($U_V$, $SP_M$))                              9
  HTTP_logs = replay(x,e,UAs($U_M$, $SP_T$))                     10
  Check Postconditions;                                          11
Postconditions:Check Flag($U_V$, $SP_T$) in HTTP_logs            12
               Report(e,UAs($U_M$, $SP_T$),Flag($U_V$, $SP_T$))  13
```

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($U_V$,$SP_M$)) (line 10) executes UAs($U_V$, $SP_M$), returning the value e associated with x. This value e is then used by the function replay (x, e, UAs($U_M$, $SP_T$)) (line 11) to replay the value of e while executing UAs($U_M$, $SP_T$), and generating the corresponding HTTP traffic logs (HTTP_logs). This logs are finally used in the `Postconditions` to check whether Flag($U_V$, $SP_T$) 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 and Listing 3, respectively. In Listing 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 rb in

Listing 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}
```

Listing 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}
```

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 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]

## VI. Evaluation

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

### A. Target MPWAs

We selected SSO, CaaS and VvE (see Figure 1c) scenarios as the targets of our experiments. For the SSO scenario, we adopted the Google dork strategy mentioned in [8] 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]

### B. Results

We have been able to identify several previously unknown vulnerabilities and they are reported in Table IV. We have

[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.

[7]www.alexa.com/topsites

promptly notified our findings 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 VI-D. Screencasts of the attacks and the details about our interactions with the vendors are available in the companion website. Some *SP*s have not patched the vulnerabilities yet, and thus in Table IV we have anonymized their names.

We cluster the attacks into four classes (see last column of Table IV) 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.

*1) 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).

*2) 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 IV). It must be noted that RA4 is inspired by an attack in SSO scenario (see #6 of Table I), 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 *SP* by replaying its publicly available API key ($DataKey$ in Figure 4a) to obtain a payment token ($Token$ in Figure 4a) 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 IV, this attack is applicable to all *SP*s implementing the Stripe checkout solution [14]. 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 open.sap.com and six other *SP*s (all having Alexa Global rank less than 500). One of the victim *SP* is a popular video-sharing website. The account activation link ($ActLink$ of Figure 1c) 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 account and authenticate the victim to the fake account (this can be done when victim visits attacker's website). As mentioned in [26], this enables the attacker to keep track of the videos searched by the victim and use this information to embarrass the victim.
- 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 [25] 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.

*3) 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 I) and Facebook's OAuth SSO (cf. #2 of Table I), we discovered a similar issue in the integration of the LinkedIn JS API SSO solution at INstant [7] (#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 IV, 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 I).

Our attack pattern that tampers the redirect URI (inspired by #9 of Table I) 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 IV). 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 [5] (#a11 of Table IV). However, in this case it was due to incorrect validation of the redirect URI by the *IdP* PayPal.

*4) 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 [35], 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 I). 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 IV).

## *C. Manual Findings*

In [36], 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 [36], we were also able to manually identify two attacks. We created one single attack pattern that generalizes the XSS attack strategy reported in [22, §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

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

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

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

TABLE IV: 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 | $DataKey$, $Token$ | NS |
| a4 | LCSRF | twitter.com, open.sap.com, other 6 SPs in Alexa Global Top 500 | Gmail | $ActLink$ | |
| a5 | RA1 | AlexaUS-55000 | LinkedIn JS API SSO | $Email$ | NP |
| 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 | $Code$ | |
| 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 | $AboutMe$ | |

the XSS payload was successfully executed. By doing so, we uncovered a XSS vulnerability in the INstant website [7] 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 rank less than 300 (#a13).

### *D. 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 [19]. Hence, we did not win a bounty for our report. However, in Section V.F of [25], 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.

## VII. Related Work

### *A. Attack pattern-based Black-Box Techniques.*

Wang et al. [37] conducted a detailed study of the security of Cashier-as-a-Service based web stores. Inspired from [37], Pellegrino et al. [32] proposed the idea of black-box detection of logical vulnerabilities in e-shopping applications. The proposed approach creates an abstract model of the application from the HTTP traffic, identifies the applicability of predefined behavioral patterns and generate test cases misusing these patterns. It is interesting to note that the strategy behind all the exploitable attacks discovered by [32] falls under the category of replay attacks (precisely those covered by our RA2 and RA3 attack patterns). 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 [32] could detect and we may not (even if there is no experimental evidence for this). However, it is also true that our attack on Stripe would require not-so-obvious extensions of [32]: consider malicious *SP* as we do and generate online test-cases to deal with short-lived/one-time tokens.

Somorovsky et al. [34] 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. In this paper, we do not consider the XML signature wrapping attack (XSW in short). However, we checked the feasibility of extending our approach to support XSW attacks (see Section VIII for details).

Bozic et al. [28] 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 III.

### B. *Other Black-Box Techniques.*

Wang et al. [36] 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. [27] 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. [40] 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.

### C. *Other Techniques.*

Bai et al. proposed AUTHSCAN [24] 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 [25] 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 [25] 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. [29] proposed to detect logical vulnerabilities in e-commerce applications through static analysis of the available program code. Even though the level of automation in [29] 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 [39], [29] 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.

## VIII. Limitations and future directions

Coverage is a general issue for the black-box security testing community. 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 I does present neither XML rewriting attacks [34] nor XSS attacks, e.g., [22, §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. By adding XML support, new attack patterns can be created to target also XML rewriting attacks as in [34]. This can be a straightforward extension of our approach and prototype especially considering that OWASP ZAP supports Jython [16]. Basically, all Java libraries can be run within OWASP ZAP so that Java functions performing transformations on the HTTP traffic (e.g., base64, XML parsing) can be used in the attack patterns. Our approach can also be extended to handle postMessage[3]: 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.

As mentioned in the paper, the approach 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.

## IX. Conclusions

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 the 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 identify serious drawbacks in the SSO and CaaS solutions offered by LinkedIn, PayPal and Stripe, previously unknown vulnerabilities in a number of websites leveraging the SSO solutions offered by Facebook and Instagram and automatically generate test cases that reproduce previously known attacks against vulnerable integration of the 2Checkout service.

## Acknowledgment

This work has been partly supported by the EU under grant 317387 SECENTIS (FP7-PEOPLE-2012-ITN).

## References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. CSF '10, IEEE Computer Society, pp. 290–304.

[22] Armando, A., Carbone, R., Compagna, L., Cuéllar, J., Pellegrino, G., and Sorniotti, A. From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure? vol. 354 of *IFIP Advances in Information and Communication Technology*. Springer, 2011, pp. 68–79.

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

[24] Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. Authscan: Automatic extraction of web authentication protocols from implementations. In *Proceedings of NDSS'13, San Diego, CA, USA* (2013).

[25] Bansal, C., Bhargavan, K., and Maffeis, S. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF 2012 IEEE* (June 2012), pp. 247–262.

[26] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 75–88.

[27] Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., and Venkatakrishnan, V. N. Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 607–618.

[28] Bozic, J., Simos, D. E., and Wotawa, F. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test* (New York, NY, USA, 2014), AST 2014, ACM, pp. 1–7.

[29] Chen, E., Chen, S., Qadeer, S., and Wang, R. Securing multiparty online services via certification of symbolic transactions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (May 2015), IEEE Institute of Electrical and Electronics Engineers.

[30] Consortium, O. SAML V2.0 Technical Overview. http://wiki.oasis-open.org/security/Saml2TechOverview, Mar. 2008.

[31] Mainka, C., Mladenov, V., and Schwenk, J. Do not trust me: Using malicious idps for analyzing and attacking single sign-on. *CoRR abs/1412.1623* (2014).

[32] Pellegrino, G., and Balzarotti, D. Toward black-box detection of logic flaws in web applications. In *NDSS* (2014), Internet Society.

[33] Phillips, C., and Swiler, L. P. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms* (NY, USA, 1998), NSPW '98, ACM, pp. 71–79.

[34] Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., and Jensen, M. On Breaking SAML: Be Whoever You Want to Be. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 397–412.

[35] Sun, F., Xu, L., and Su, Z. Detecting Logic Vulnerabilities in E-commerce Applications. In *NDSS 2014, California, USA, February 23-26, 2013* (2014).

[36] Wang, R., Chen, S., and Wang, X. 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* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 365–379.

[37] Wang, R., Chen, S., Wang, X., and Qadeer, S. 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* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 465–480.

[38] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 399–414.

[39] Xing, L., Chen, Y., Wang, X., and Chen, S. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *NDSS* (February 2013).

[40] Zhou, Y., and Evans, D. SSOScan: Automated Testing of Web Applications for Single Sign-on Vulnerabilities. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (CA, USA, 2014), SEC'14, USENIX Association, pp. 495–510.