2017 IEEE European Symposium on Security and Privacy

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

Avinash Sudhodanan*, Roberto Carbone*, Luca Compagna†, Nicolas Dolgin†,
Alessandro Armando‡ and Umberto Morelli*
*Fondazione Bruno Kessler, Italy
Email: 6.avinash@gmail.com, {carbone, umorelli}@fbk.eu
†SAP Labs France
Email: luca.compagna@sap.com, nicolas.dolgin@gmail.com
‡University of Genova
Email: alessandro.armando@unige.it

***Abstract*— Cross-Site Request Forgery (CSRF) attacks are one of the critical threats to web applications. In this paper, we focus on CSRF attacks targeting web sites' authentication and identity management functionalities. We will refer to them collectively as *Authentication CSRF* (Auth-CSRF in short). 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.**

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 [2]. 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 [8] and has received a lot of attention from the web community. For instance, the OWASP Testing Guide [31] devotes an entire section to this vulnerability (cf. Testing for CSRF in [31]). On the contrary, pre-authentication CSRF is not mentioned in the OWASP Testing Guide although severe exploits have been reported in the literature, including:

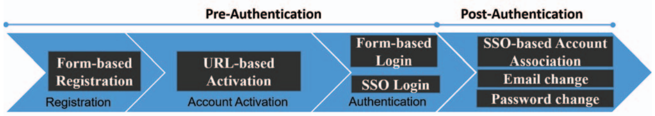

Figure 1: Most-common Auth-CSRF-vulnerable processes

- the execution of malicious JavaScript code at the victim's web browser [13],
- the association of the victim's financial details (or Google Search History) with the attacker's PayPal account (or Google account, respectively) [13], and
- the tracking of the videos watched by the victim by the attacker [37], [21].

In this paper, 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 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 1), *(ii)* rationally reconstruct 7 process-based testing

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

DOI 10.1109/EuroSP.2017.45

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., [18], [28], [20]), 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 5 for details) and 90 of these suffered from at least one vulnerability enabling Auth-CSRF (i.e. ~68% of the tested web sites are vulnerable). In total, we discovered 150 vulnerabilities in this phase and some of our most-severe findings are mentioned in Table 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 [3] that assists a tester to (semi-) automatically detect vulnerabilities enabling Auth-CSRF. 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 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 our paper is manifold:

- 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. ~70% of the web sites we tested were vulnerable to Auth-CSRF; we also report on our responsible disclosure experience.

Structure of the paper. In Section 2, we introduce some background information about CSRF attacks and defenses. We continue in Section 3 discussing Auth-CSRF over key processes of a web application. Section 4 defines precise security testing strategies to detect Auth-CSRF attacks. We discuss our first experimental evaluation of Alexa top web sites in Section 5, and some relevant case studies in Section 6. In Section 7 we present our prototype for assisting users toward testing for Auth-CSRF. Our ethics and responsible disclosure experience is reported in Section 8. In Section 9 and Section 10 we discuss some limitations of our work and a comparison with the related work. We conclude in Section 11 with some final remarks.

2. Background

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

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 [42], 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 [8]) 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

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.

TABLE 1: Excerpt of our Findings

high-impacts (e.g., transferring money from one account to another). However, in 2008, Login CSRF attack was introduced [13]. 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 [10], [12].

Pre- and Post-authentication CSRF Attacks: It is interesting to note that the victims of Login CSRF attacks (and its variants [10], [12]) 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 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* [17], [34]. 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 [16] to make the victim's browser send the state-changing HTTP request. As we will show in Section 2.2, this distinction is important in understanding the drawbacks of CSRF defenses.

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 [13] 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 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 a 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 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., [22] explains a PDF reader-based vulnerability enabling CSRF). Browser-based security vulnerabilities enabling CSRF is beyond the scope of this paper.

Custom HTTP Header Validation: A web site adopting this defense must implement all state-changing actions via *XMLHttpRequests* [7]. 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 [13], 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 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 [26], or if the victim uses a vulnerable web browser (beyond the scope of this paper, see [22] for details). In [13] 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 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 [29, §2.1.2] and [18, §3.1.1] that these frameworks have many exceptions (e.g., no protection for SSO [29, §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., [23], [25], [18], [30]).

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., [23] considers all cross-domain requests as suspicious) and then they perform certain operations on the suspicious HTTP requests (e.g., [23] removes authentication credentials from the header of suspicious requests). We will show in Section 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 on in this paper.

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 [35], [29]) and affect both pre- and post-authentication processes of web sites (cf. Figure 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.

3.1. 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 [13] 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 [21] 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 [37] 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 [29] 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 6.

3.2. Selection of Auth-CSRF-vulnerable 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 2 shows the Auth-CSRF attacks that cause the victim to be authenticated as the attacker and Table 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 2 and 3. An overview of all the processes considered in Tables 2 and 3 is illustrated in Figure 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 2 for details), the origin of the forged attack request (shown as Atk Req—meaning Attack Request—in Table 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 2) and the HTTP request containing the username, password and other registration information chosen by the attacker, similar to that of the victim (represented as $uname_A$, $pass_A$ and $info_A$ in column 5, row #1 of Table 2). A benign version of this forged request (shown as Benign Req—meaning Benign Request—in Table 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 2) which in this case is Localize. Upon receiving the request, the victim is authenticated as attacker on Localize.

URL-based Account Activation (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 (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 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

#	Reference	Referer/Origin		Credentials in Atk Req	Vulnerable Process
		Benign Req	Atk Req		
1	Localize.io's Sign up form [9]	VulnWS	AtkWS	Body[$uname_A$, $pass_A$, $info_A$]	Form-based Registration
2	openSAP's account activation URL [37, §IV.B.2]	TrustWS	AtkWS	URL[act_token_A]	URL-based Account Activation
3	Twitter's [12, §IV.E] and Google's [13, §3] Login Form	VulnWS	AtkWS	Body[$email_A$, $pass_A$]	Form-based Login
4	Facebook's Login Form [13, §4.2]	VulnWS	[ ]	Body[$email_A$, $pass_A$]	
5	Facebook's Login Form [29, §2.2.1]	VulnWS	AWPVulnWS	Body[$email_A$, $pass_A$]	
6	Two web sites implementing Mozilla's BrowserID [11, §6.2]	TrustWS	AtkWS	Body[$auth\_assert_A$]	SSO Login
7	Many web sites implementing Open ID [13, §6.1]			Body[$token_A$]	
8	Stanford's WebAuth implementation [10, §IV.E]			URL[id_token_A]	
9	Many web sites implementing OAuth protocol [37, §VI.B.3], [12, §V.C], [38, §4.4], [35, §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, a mailbox provider, etc.), (4) AWPVulnWS: Attacker-configurable Web Page on the Vulnerable Web Site, (5) [ ]: empty `Referer` Header

TABLE 2: Auth-CSRF attacks causing Victim to be Authenticated as Attacker

#	Reference	Referer/Origin		Credentials in Atk Req	Vulnerable Process
		Benign Req	Atk Req		
10	Web site implementing OAuth-based account association feature [33], [6], [41, §5.2.1(A6)]	TrustWS	AtkWS	Body[$code_A$], Hdr[$cookie_V$]	SSO-based Account Association
11	Primary Email change in MetaFilter[42, §3.3]	VulnWS	AtkWS	Body[$email_A$], Hdr[$cookie_V$]	Primary Email Change
12	Web sites having Password change forms without CSRF protection [1]	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

TABLE 3: Auth-CSRF attacks causing Attacker to be Authenticated as Victim

without the `Referer` header (see [23, §3.1], [13, §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 2. The description is as follows. The attacker creates a Facebook canvas app running on a domain with prefix *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 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 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 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 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 (P_6, 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. Preventing Auth-CSRF: Challenges

The following is our comparison of the defenses proposed for preventing CSRF attacks (explained in Section 2.2) and the Auth-CSRF attacks shown in Tables 2 and 3. The secret validation token method if carefully implemented can defeat all attacks (#1 to #12). However, it was shown in [35], [38], [33] 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 (cf. [16]) 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 [22]) is also a threat to this defense.

As we explained in Section 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., [2], [26]) show that at least the first two issues are hard to avoid.

In [29], 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 [18], [20]) that while many of the proposed techniques [19], [24], [34], [23] break normal cross-domain behavior such as SSO Login, others (e.g., [42]) suffer from drawbacks of either being too permissive or restrictive. We noticed that some of them [18], [28], [20] cannot detect attacks like #2. Similarly, stored CSRF is not supported by [25].

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., [14]) 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.

4. Manually Testing for Auth-CSRF Attacks

By carefully analyzing the attacks we discussed in Section 3, we have been able to distill testing strategies for processes P_1 to P_7 explained in Section 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 [3]), 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 AtkAcc 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 AtkAccSoc (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 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 AtkAcc
(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 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 AtkAcc?

TS$_2$: Test Strategy for URL-based Account Activation
(1) Register an account AtkAcc 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 AtkAcc?

TS$_3$: Test Strategy for Form-based Login
(1) Visit the *login page* of WUT
(2) Submit *login-credentials* for AtkAcc
(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 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 AtkAcc?

TS$_4$: Test Strategy for SSO Login
(1) *SSO login* to AtkAcc account on the WUT via AtkAccSoc
(2) Intercept the HTTP request containing the *authentication token* of AtkAccSoc
(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 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 AtkAcc?

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

TS$_5$: Test Strategy for SSO-based Account Association
(1) Login to AtkAcc on WUT
(2) Visit *SSO-based account association page* on WUT
(3) Run *SSO account association process* using AtkAccSoc
(4) Intercept the HTTP request containing the *authentication token* of AtkAccSoc
(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 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 AtkAcc on WUT
(2) Visit the *page for Email/Password-change* of WUT
(3) Submit a *new Email/Password* as AtkAcc
(4) Intercept the HTTP request containing the *new Email/Password*

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

(a) preAuthTS

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

(b) postAuthTS

Figure 2: Testing strategies

#	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 4: Alterations

(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 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 2a and 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 P as AtkAcc, intercepting the *CandidateReq* issued by the browser and corrupting the CSRF prevention mechanisms occurring in the header by applying the changes given in Table 4. In particular, A_1 is used to perform attacks like #3 of Table 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 2 where the forged request originated from a web page on the vulnerable web site. This is done by changing the `Referer` 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). A_3 is to consider attacks like #4 of Table 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 5.

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	

TABLE 5: Testing Strategy Information

Strategy postAuthTS consists in logging-in with AtkAcc 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 4.

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 7.3 that we also conducted experiments on other rank ranges but with more automation.

Result Overview. Figure 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 4 (while focusing only on reflected Auth-CSRF, i.e. applying only A_1 and A_4 of Table 4). The remaining web sites have been skipped because of language barriers (90), lack of an account creation feature (17), domain duplicates such as *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 attacks similar to the ones mentioned in Tables 2 and 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).

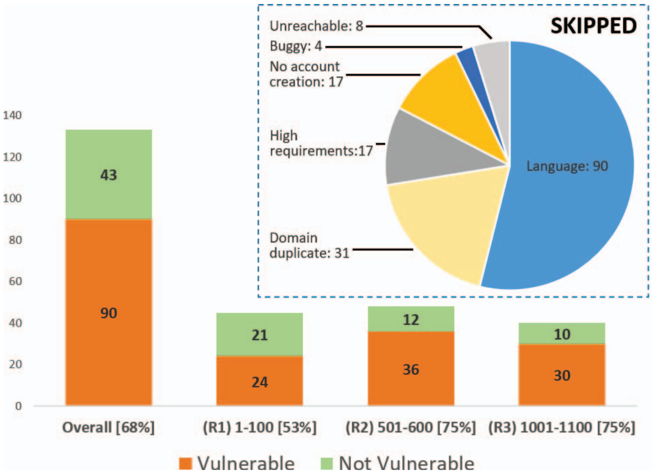

Figure 3: Result overview

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 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.

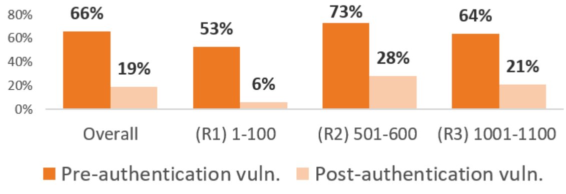

Figure 4: Result comparison

Results Per Testing Strategy. As already mentioned we applied all the security testing strategies TS_1 to TS_7 of Section 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 5 and 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.

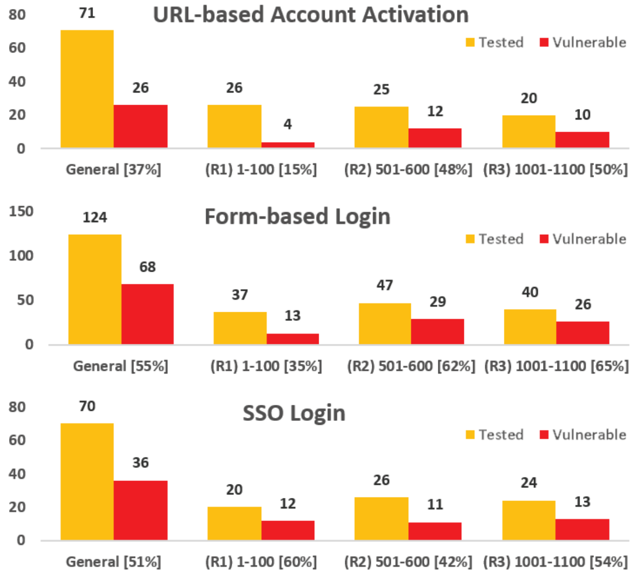

Figure 5: Incidence of pre-authentication vulnerabilities

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). 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

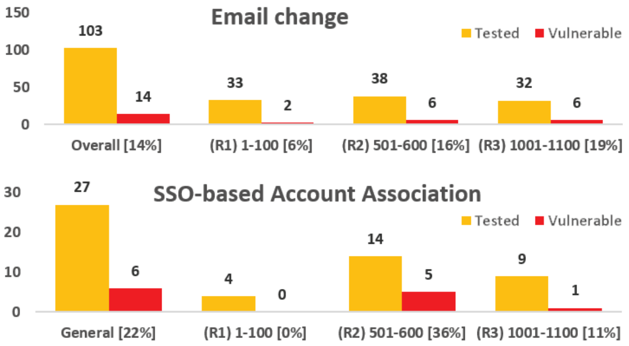

Figure 6: Incidence of post-authentication vulnerabilities

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 (R1) and increases to 62% and 65% respectively within (R2) and (R3) (no statistically significant difference between (R2) and (R3)). As explained in Section 2.2, for the custom header validation CSRF defense, in [13], 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 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 [35, §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 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 [35], 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 7.3 (see 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 4 and 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 [31], 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 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., [43], [15]) 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 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 [43, §5.1.1], depending on the web site, the impact can be serious.

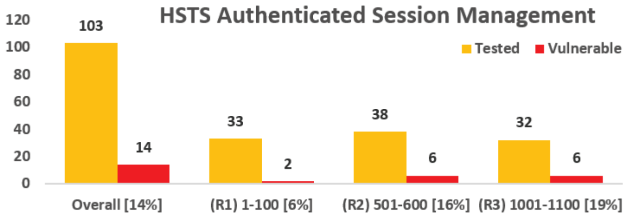

Figure 7: Incidence of HSTS

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

6. Selected Case Studies

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.

6.2. A 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.

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 [13] 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.

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.

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 on this paper's companion web site [5].

7. (Semi-)Automatic Testing for Auth-CSRF

In Section 5 we showed the results of applying each (manual) testing strategy (see Section 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 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 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 7.1 we explain the concept behind CSRF-checker. In Section 7.2 we briefly explain the implementation details of CSRF-checker. Section 7.3 presents the outcome of our experiments on Alexa top 1500 web sites with CSRF-checker.

7.1. CSRF-checker Concept

The tool implements the strategies reported in Figures 2a and 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.

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 [3] 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 4), etc. The source code, installation guide and tutorial for the tool's proof-of-concept implementation can be obtained (upon request) from the paper's companion web site [5].

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 3.2) that we found during our manual experiments (explained in Section 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 vulnerability 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 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.

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., [40], [32]) 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 [40], [32]. 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 [36] 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 on the companion web site of this paper [5].

9. Limitations

One main drawback of our approach is that most of the experimental analysis is done manually. In [44] 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 [45], 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 [45].

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 [36] that checked the feasibility of automating the process of vulnerability disclosure. But the conclusion of [36] 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 [4]) are emerging. Still, more awareness of some CSRF attacks is necessary and we provide a tool supporting the testing phase of web sites.

10. Related Work

In [35] 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 5, we identified that their approach is susceptible to a number of false positives and false negatives.

In [38], 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 5). This helped them avoid the false positives and false negatives that affected [35]. However, CSRF-checker can provide the same level of accuracy as [38] but with more automation.

In [39] 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 [7] (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 [39].

Past studies [42], [26], [27] 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 [43] and [15] 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 7.

11. Conclusions

The findings reported in this paper indicate that developers often fail to protect sensitive processes from Auth-CSRF attacks and that the default CSRF protection offered by web frameworks and automatic/semi-automatic CSRF prevention mechanisms may not protect web sites from all Auth-CSRF attack vectors. This shows the importance of security testing web sites for Auth-CSRF attacks. The security testing strategies proposed in this paper and implemented in our proof-of-concept prototype assist web developers in checking their web site against Auth-CSRF.

References

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

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

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

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

[5] Supporting Materials. https://sites.google.com/site/authcsrf/.

[6] The Most Common OAuth2 Vulnerability. http://homakov.blogspot.it/2012/07/saferweb-most-common-oauth2.html.

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

[8] Mail From Peter Watkins about CSRF. http://www.tux.org/~peterw/csrf.txt, 2001.

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

[10] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. CSF '10, pages 290–304, Washington, DC, USA, 2010. IEEE Computer Society.

[11] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th NDSS'13, San Diego, CA, USA*, 2013.

[12] 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.

[13] A. Barth, C. Jackson, and J. 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.

[14] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[15] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, pages 98–113. IEEE, 2014.

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

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

[18] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight server support for browser-based csrf protection. In *22nd international conference on World Wide Web*, pages 273–284. International World Wide Web Conferences Steering Committee, 2013.

[19] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[20] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *Computer Security–ESORICS 2011*, pages 100–116. Springer, 2011.

[21] J. Grossman. I used to know what you watched, on youtube, 2008.

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

[23] M. Johns and J. Winter. RequestRodeo: Client side protection against session riding. In *the OWASP Europe 2006 Conference*, 2006.

[24] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.

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

[26] S. Lekies, M. Johns, and W. Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web*, volume 2, 2011.

[27] S. Lekies, N. Nikiforakis, W. Tighzert, F. Piessens, and M. Johns. DEMACRO: defense against malicious cross-domain requests. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012.

[28] S. Lekies, W. Tighzert, and M. Johns. Towards stateless, client-side driven Cross-Site Request Forgery protection for Web applications. In *Sicherheit*, pages 111–121, 2012.

[29] R. Lundeen. The deputies are still confused. In *Blackhat EU*, 2013.

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

[31] M. Meucci and A. Muller. The OWASP Testing Guide 4.0, 2014.

[32] G. Pellegrino and D. Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *NDSS Symposium 2014*. Internet Society, 2014.

[33] S. Sclafani. CSRF Vulnerability in OAuth 2.0 Client Implementations. http://stephensclafani.com/2011/04/06/oauth-2-0-csrf-vulnerability/.

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

[35] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *DIMVA 2015, Milan, Italy, July 9-10, 2015*, pages 239–260, Cham, 2015. Springer International Publishing.

[36] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. 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.

[37] A. Sudhodanan, A. Armando, R. Carbone, and L. 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, CA, USA, February 21-24*, 2016.

[38] S.-T. Sun and K. 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.

[39] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen. Large-scale security analysis of the web: Challenges and findings. In *International Conference on Trust and Trustworthy Computing*, pages 110–126. Springer, 2014.

[40] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *IEEE Symposium on Security and Privacy*, pages 465–480, Washington, DC, USA, 2011. IEEE Computer Society.

[41] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Conference on Security*, pages 399–414, Berkeley, CA, USA, 2013. USENIX Association.

[42] W. Zeller and E. W. Felten. Cross-Site Request Forgeries: Exploitation and Prevention, Princeton (2008).

[43] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 707–721, Washington, D.C., Aug. 2015. USENIX Association.

[44] Y. Zhou and D. Evans. Why aren't HTTP-only cookies more widely deployed. *Proceedings of 4th Web*, 2, 2010.

[45] Y. Zhou and D. Evans. SSOScan: Automated Testing of Web Applications for Single Sign-on Vulnerabilities. In *23rd USENIX Conference on Security Symposium*, pages 495–510, CA, USA, 2014. USENIX Association.