Testing for Integrity Flaws in Web Sessions

Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi

Università Ca' Foscari Venezia

Abstract. Web sessions are fragile and can be attacked at many different levels. Classic attacks like session hijacking, session fixation and cross-site request forgery are particularly dangerous for web session security, because they allow the attacker to breach the integrity of honest users' sessions by forging requests which get authenticated on the victim's behalf. In this paper, we systematize current countermeasures against these attacks and the shortcomings thereof, which may completely void protection under specific assumptions on the attacker's capabilities. We then build on our security analysis to introduce black-box testing strategies to discover insecure session implementation practices on existing websites, which we implement in a browser extension called Dredd. Finally, we use Dredd to assess the security of 20 popular websites from Alexa, exposing a number of session integrity flaws.

Keywords: Web sessions · session hijacking · session fixation · CSRF

1 Introduction

Since the HTTP protocol is stateless, web applications treat each HTTP request independently from all the others by default. However, online services often need to track state information across multiple HTTP requests to provide controlled access to private data and authorize security-sensitive operations for authenticated users. HTTP *cookies* are the most common mechanism to maintain state information across HTTP requests [2], thus enabling the implementation of *authenticated web sessions*. Unfortunately, web session security is hard to get right, as shown by the huge number of attacks and defenses presented in the literature [13]. As a result, although it is possible to ensure the security of web sessions using existing technologies, web session implementations in the wild still suffer from severe security flaws [8, 35, 29, 30].

Security testing is a very popular solution to detect vulnerabilities in web applications [5]. The Open Web Application Security Project (OWASP) has long recognized the importance of testing for web application security and maintains a guide with recommendations about it - the OWASP Testing Guide [25] - which also includes a section about *session management*. Although this guide provides useful advice to improve web session security, it is not systematic and does not provide a comprehensive coverage of realistic attack vectors based on a rigorous threat model. For instance, the proposed testing strategy for cross-site request forgery vulnerabilities is insufficient to assess security against network attackers who have control of the HTTP traffic, because it does not consider their ability to compromise cookie integrity [4]. Moreover, a number of recommendations in the OWASP Testing Guide are rather generic or overly conservative, making them hard to follow for web developers. For instance, the proposed testing strategy for session hijacking recommends the use of an intercepting proxy to check that all cookies containing session identifiers are marked with the `Secure` attribute to prevent their leakage over HTTP. However, this can be hard to do in practice, because it presupposes that security testers know the role of all cookies, and it might even be unnecessary, because when multiple cookies are used for session management it might suffice to protect just one of them [15, 23].

Our main goal in the present work is to make security testing for web sessions more principled and systematic by building on precise security properties and threat models. At the same time, we advocate the adoption of testing strategies which *semantically* capture attacks, thus detecting exploitable vulnerabilities, as opposed to testing strategies based on syntactic conditions, such as the aforementioned use of the `Secure` attribute, which instead can just identify room for *potential* attacks. Our approach makes security testing more effective by eliminating false positives and enables a more precise security assessment.

Contributions. Concretely, we make the following contributions:

1. we provide an in-depth, up-to-date security analysis of web sessions that reviews attacks, current countermeasures and the shortcomings thereof. While several of such considerations have been piecemeal reported in the literature, we are not aware of any previous attempt to organize them within a comprehensive, uniform framework. In particular, our analysis is based on a clear notion of *session integrity* and a rich threat model which includes web attackers, related-domain attackers and network attackers (Section 3);
2. we design black-box testing strategies to systematically detect integrity flaws in existing web session implementations. Each testing strategy is parametric with respect to the choice of an arbitrary attacker from our threat model and targeted at detecting *exploitable* vulnerabilities, without requiring deep understanding of the web application logic (Section 4);
3. we implement our testing strategies in a browser extension, called Dredd[1], which we make available upon request. We use Dredd to assess the security of 20 popular websites from the Alexa ranking, exposing a number of session integrity flaws exploitable by different attackers (Section 5).

2 Background

2.1 Same-Origin Policy

The same-origin policy (SOP) is a security policy implemented by all browsers, which enforces a strict separation between contents provided by unrelated web-

[1] The browser extension is named after Judge Joseph Dredd, a law enforcement and judicial officer in the dystopian future created by some popular British comic books.

sites [22]. SOP allows scripts running in a page to access data of another page only if the pages have the same *origin*, i.e., the same protocol, host and port [3].

SOP mediates several operations in the browser, including DOM and cookie accesses. For instance, if a page at `https://www.example.com` embeds an iframe from `http://www.evil.com`, malicious scripts running in the iframe cannot read or write the DOM of the embedding page, because they come from a different origin. However, a few important browser operations are not subject to same-origin checks, e.g., the inclusions of scripts and the submissions of forms. This leaves room for attacks like cross-site request forgery, which is extensively discussed in the next sections.

2.2 HTTP Cookies

Roughly, a cookie is a key-value pair, which is set by the server into the browser and then automatically attached by the browser to all the subsequent requests to the server, using the `Cookie` header. Cookies can be set using the `Set-Cookie` header or by means of JavaScript through the `document.cookie` property.

Cookies rely on a relaxed definition of origin, since they are normally shared across all protocols and ports of the host setting them. For instance, scripts at `https://www.example.com` can read the cookies of `http://www.example.com`, although these two origins do not match. By default, cookies are only attached to requests sent to the same host which set them (*host-only cookies*). However, a host may also set cookies for a parent domain by means of the `Domain` attribute, as long as the parent domain does not occur in a list of public suffixes[2]: these cookies are shared across all the sub-domains of such domain (*domain cookies*).

It is sometimes desirable to improve the confidentiality guarantees of cookies. For instance, scripts at `http://www.example.com` can normally access cookies set by `https://www.example.com`, although these scripts were not sent over a secure channel. The `Secure` attribute can be used to mark cookies which must only be accessed from HTTPS pages and sent over HTTPS channels.

2.3 HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) is a security policy implemented in all modern web browsers, which allows hosts to require browsers to communicate with them only over HTTPS [18]. Specifically, HTTP requests to HSTS hosts are automatically upgraded to HTTPS by the browser before they are sent.

HSTS can be activated over HTTPS using the `Strict-Transport-Security` header, where it is possible to specify for how long HSTS should be active and whether its protection should additionally be extended to sub-domains. Alternatively, hosts may request to be included in the HSTS preload list of major web browsers, so that HSTS is activated on them by default. We refer to *full* HSTS adoption when a host activates HSTS for itself and all its sub-domains, and to *partial* HSTS adoption when a host activates HSTS just for itself. We discuss a few shortcomings of the partial HSTS adoption in the next sections.

[2] Available at `https://publicsuffix.org/`

3 Integrity Analysis of Web Sessions

We describe the typical set-up of a web session. Browser B, operated by user Alice, submits a login form including valid access credentials to website W, which replies by sending back a cookie *sid* storing a session identifier which uniquely identifies Alice; we refer to *sid* as the *session cookie*[3]. Browser B then automatically attaches *sid* to all the subsequent requests sent to W, so that W has a way to authenticate Alice and authorize operations on her behalf.

3.1 Web Session Integrity

We say that Alice's session has *integrity* when a malicious user Mallory cannot forge requests which get authenticated under Alice's identity. There are three ways available to Mallory to break session integrity:

1. *session hijacking*: if Mallory is able to steal Alice's *sid*, she can authenticate as Alice at W directly from her browser (Section 3.3);
2. *session fixation*: if Mallory can force Alice's *sid* to a known value, she can authenticate as Alice at W directly from her browser (Section 3.4);
3. *cross-site request forgery*: if Mallory cannot steal or force the choice of Alice's *sid*, she can still forge security-sensitive requests from Alice's browser B, so as to fool W into processing such requests on Alice's behalf (Section 3.5).

We only cover attacks where Mallory intrudes on Alice's session. Attacks where Alice is forced into Mallory's session, e.g., login CSRF and cookie forcing [13], are considered out of scope. The reason is that such threats are specific to a relatively narrow class of web applications and use cases.

3.2 Threat Model

The goal of Mallory is to break the integrity of sessions established between Alice and the target website $W =$ `www.target.com`. We consider three different attackers. The weakest one is the *web attacker*, who is the owner of a malicious website [1]. This attacker can reply to HTTP requests to her website with arbitrary contents and can obtain a valid HTTPS certificate for her website from a trusted certification authority. To simplify the exposition, we stipulate that the web attacker operates `https://www.attacker.com`. We assume that the user accesses this malicious website from her browser, either deliberately or because she is fooled into doing it. However, we assume that the web attacker has no scripting capabilities at the target: she has no control of active contents included by the target itself and she cannot exploit cross-site scripting vulnerabilities (XSS) therein. This is a standard assumption, motivated by the fact that most web defenses are voided when the attacker can script in the target's origin.

[3] Real services often use multiple session cookies, but the discussion abstracts from this point for simplicity. Session cookies have also been called *authentication cookies* in related work [15].

The *related-domain* attacker is a stronger variant of the web attacker, who can host her website on a related domain of the target website [6]. Two domains are related when they share a sufficiently long suffix, e.g., `accounts.example.com` and `mail.example.com`; any non-public suffix will work in current browsers. The related-domain attacker is traditionally considered in the web session security literature, due to her extended ability to compromise cookie confidentiality and integrity compared to the web attacker. Though most websites do not lease subdomains to untrusted organizations, a recent study highlighted the threats of the related-domain attacker due to the common use of a large number of subdomains, many of which at low security [12]. To improve readability, we stipulate that the related-domain attacker is hosted at `https://attacker.target.com`.

Finally, the *network attacker* extends the capabilities of the web attacker with the ability to read, modify and block the contents of all the HTTP traffic. However, we assume that this attacker cannot sniff or corrupt the HTTPS traffic, due to the adoption of trusted certificates at the target website. When reasoning about session integrity against the network attacker, we always assume that the target website is deployed on HTTPS, otherwise the website is trivially insecure.

Observe that the web attacker is weaker than both the related-domain attacker and the network attacker by definition, while the related-domain attacker and the network attacker are incomparable. In particular, although the network attacker can masquerade as any HTTP website, she cannot host any HTTPS website on a related domain of the target website.

3.3 Session Hijacking

Session hijacking happens when the attacker is able to steal the session cookies of the victim and uses them to impersonate her at the target website. Different attackers have different ways to steal session cookies: although the web attacker has no access to such cookies in absence of XSS (thanks to SOP), the related-domain attacker and the network attacker are more powerful. In particular, the related-domain attacker can potentially access all the domain cookies of the target website and intrude on the victim's session by reusing them.

The picture is more complex for the network attacker. If HSTS is not enabled at the target website, any cookie without the `Secure` attribute can be disclosed. This is true even if the target website is entirely deployed on HTTPS, because the network attacker can still force the victim's browser into trying to contact the target website over HTTP [8]. If instead HSTS is partially adopted, host-only cookies cannot be disclosed irrespective of the use of the `Secure` attribute, because the victim's browser will never contact the target website over HTTP; however, domain cookies must still be marked with the `Secure` attribute to prevent their disclosure [21]. Finally, in the case of a full HSTS adoption, the network attacker is no more powerful than the web attacker for session hijacking and the use of the `Secure` attribute is redundant.

Table 1 summarizes the conditions under which the confidentiality of cookies can be violated. To ensure protection against session hijacking, it is critical to ensure the confidentiality of the session cookies.

Table 1. Conditions for cookie leakage under different attackers

Attacker	*Condition for cookie leakage*
Web attacker	no leakage is possible (in absence of XSS)
Related-domain attacker	the `Domain` attribute is set to a parent domain
Network attacker	either of the following conditions holds true: 1. no HSTS adoption and the `Secure` attribute is not set; 2. partial HSTS adoption, the `Secure` attribute is not set and the `Domain` attribute is set to a parent domain.

3.4 Session Fixation

Session fixation is enabled by the insecure practice of preserving the same value of the session cookies before and after authentication. This typically happens when session cookies are used to store state information even before login, e.g., to add items to a shopping cart. In this case, the attacker can obtain a set of session cookies from the target website without authenticating and force them into the victim's browser, using different techniques; if the victim later authenticates at the target, she will be identified by the session cookies chosen by the attacker, who will then become able to impersonate the victim [19].

Different attackers have different ways to force session cookies into the victim's browser. Again, the web attacker lacks this capability in absence of XSS, because SOP prevents her from setting cookies for the target website, but the related-domain attacker and the network attacker are more powerful. Specifically, the related-domain attacker can set domain cookies for the target website, which are indistinguishable from host-only cookies to it [6]. Though the related-domain attacker cannot overwrite host-only cookies previously set by the target website due to SOP, she can still issue domain cookies before the host-only cookies are set, which often suffices to prevent the host-only cookies from ever being issued. Moreover, the related-domain attacker can set domain cookies with the same name of host-only cookies, which may fool the target website into choosing the domain cookies over the host-only cookies [35]. The only effective way to prevent these attacks is extending the name of the session cookies with the `__Host-` prefix, which requires cookies to be set as host-only [32]. The `__Host-` prefix also has the benefit of requiring cookies to be set over HTTPS with the `Secure` attribute, which is useful against the network attacker.

As to the network attacker, she can abuse the lack of HSTS to set cookies for the target website by forging HTTP traffic. Notice that a full HSTS adoption is needed to prevent this attack, otherwise the network attacker could still make the victim's browser contact a sub-domain of the target website over HTTP and then forge domain cookies from there. If full HSTS is not an option for some reason, an alternative protection mechanism is based on the use of the `__Secure-` prefix in the cookie name, which requires cookies to be set over HTTPS with the

Table 2. Conditions for cookie compromise under different attackers

Attacker	*Condition for cookie compromise*
Web attacker	no compromise is possible (in absence of XSS)
Related-domain attacker	lack of `__Host-` prefix in the cookie name
Network attacker	both the following conditions hold true: 1. lack of full HSTS adoption; 2. lack of `__Host-` and `__Secure-` prefixes in the cookie name.

`Secure` attribute [32]. Notice that the `Secure` attribute alone does not suffice to provide cookie integrity, because modern browsers prevent cookies with the `Secure` attribute from being set or overwritten over HTTP, yet cookies with the same name but without the `Secure` attribute can still be forged over HTTP before the legitimate cookies are actually issued [33].

Table 2 provides a summary of the conditions under which the integrity of cookies can be violated. To prevent session fixation, it is important to guarantee the integrity of the session cookies which are not freshly issued upon login.

3.5 Cross-Site Request Forgery

Cross-site request forgery (CSRF) is enabled by the standard behavior of web browsers to attach cookies to all HTTP requests by default, irrespective of the origin of the page which fired the request [4]. A CSRF attack typically works as follows: the victim first logs into the target website and gets a set of session cookies which authenticates her; the victim then visits the attacker's website, which sends a *cross-site* request to the target, e.g., using HTML or JavaScript. Since the request includes the victim's cookies, it is authenticated as coming from the victim, hence it may be abused to trigger security-sensitive operations at the victim's account. There are different defenses against CSRF.

Custom Headers. JavaScript can set HTTP headers with custom names, for instance `X-Requested-With`, to be attached to outgoing HTTP requests. Since SOP prevents the setting of custom headers for cross-origin requests, the mere presence of such headers can be used to stop CSRF attempts. Given that none of the attackers we consider controls the exact origin of the target website, custom headers are an effective protection mechanism against all the attackers. Unfortunately, custom headers can only be used by websites which implement all their security-sensitive operations via JavaScript.

Referer/Origin Header. Browsers normally attach to each outgoing request at least one between two standard HTTP headers, called `Referer` and `Origin`, which contain the URI and the origin of the page which triggered the request

Table 3. Examples of dangerous Referer/Origin headers

Attacker		*Spoofed Referer/Origin*
Web attacker		`https://www.target.com.attacker.com`
Related-domain attacker		`https://attacker.target.com`
Network attacker	no HSTS	`http://www.target.com`
	partial HSTS	`http://attacker.target.com`
	full HSTS	`https://www.target.com.attacker.com`

respectively. This can be used to filter out cross-site requests, but some care is needed to implement this check correctly. We discuss just the case of the `Referer` header, because the same reasoning applies to the `Origin` header.

Though the web attacker is hosted at `https://www.attacker.com`, she could try to fool the target via the sub-domain `www.target.com.attacker.com`, so that the `Referer` header of requests coming from this sub-domain will start with the substring `https://www.target.com`. This would be enough to fool a `Referer` check based on a liberal regular expression. The related-domain attacker, instead, could send requests from `https://attacker.target.com`, which would bypass lenient `Referer` checks allowing requests from any sub-domain. Finally, the network attacker could abuse the lack of HSTS to send requests from `http://www.target.com`, which would escape `Referer` checks that are intended to filter out cross-site requests, but do not enforce the use of the HTTPS protocol. Observe that, if HSTS is only partially adopted, the network attacker could attempt a variant of the attack from `http://attacker.target.com`. This would fool `Referer` checks which are intended to only accept requests from sub-domains, but do not check for the adoption of the HTTPS protocol. In the case of a full HSTS adoption, the network attacker loses these capabilities and becomes no more powerful than the web attacker.

Table 3 shows "canonical" examples of `Referer` values which could be abused by the different attackers based on the previous discussion. Recall that both the related-domain attacker and the network attacker subsume the web attacker, so the first choice in the table is also available to them.

Anti-CSRF Tokens. Another common approach to CSRF prevention is based on the use of random tokens to be included in legitimate authenticated requests: requests lacking a valid token do not get processed. Tokens are normally embedded in DOM elements which fire requests for security-sensitive operations, e.g., as a hidden field of payment forms. If tokens cannot be predicted or disclosed, the attacker cannot craft cross-site requests which get successfully processed by the target website. There are two traditional ways to implement tokens.

The first approach is based on *stateful tokens*. Stateful tokens are stored at the server side, typically bound to users' sessions. For example, the server could store in Alice's session the information that authenticated requests from Alice must include the token `Xf12gh68g`. The second approach is based on *stateless tokens*, which do not require such server-side state, but are harder to implement securely. Stateless tokens are stored at the browser side, often inside a cookie. For example, the server could set a cookie containing the token `G9jp3mNt` in Alice's browser and require authenticated requests to include such token inside a parameter: this pattern is called *double submit*. Variants of this approach rely on the encryption or other transformations of the token set in the cookie.

Stateless tokens can be attacked at different levels. If the confidentiality of the cookie storing the token is not guaranteed, no protection against CSRF is granted when the token validation process is performed via a parameter value which can be computed from the token alone: we refer to Table 1 for a summary of the conditions which can lead to cookie leakage. If instead the attacker can compromise the integrity of the cookie storing the token, she can acquire a valid token from the server and force it in the victim's browser, so that the session is still protected with a token known by the attacker. To avoid this issue, stateless tokens must be *session-dependent*, so that the attacker's tokens cannot be used in the victim's session [4]. We refer to Table 2 for a summary of the conditions which enable cookie compromise.

Same-Site Cookies. Cookies can be marked with the `SameSite` attribute to signal to the browser that they should not be attached to cross-site requests, thus preventing CSRF attempts directly at the client side. This approach is effective, since it fixes one of the root causes of CSRF, i.e., browsers attaching session cookies by default. Strict or lax protection can be given by the `SameSite` attribute, with the latter mode relaxing the mentioned security restriction in the case of top-level navigations with a "safe" HTTP method [34].

4 Integrity Testing of Web Sessions

We now build on our security analysis to design precise black-box testing strategies which allow human experts to detect session integrity flaws. Our main goal is finding *exploitable* vulnerabilities without requiring an in-depth understanding of the web application logic. All the testing strategies presuppose the availability of two test accounts at the website under scrutiny, called Alice and Mallory. Alice acts as the victim, while Mallory acts as the attacker; each strategy is parametric with respect to the choice of an attacker from our threat model, which defines Mallory's capabilities (see Section 3.2). If Mallory's actions affect Alice's session, we have a session integrity problem.

4.1 Testing for Session Hijacking

The intuition behind the testing strategy for session hijacking is to simulate a scenario where Mallory steals all Alice's cookies she might be exposed to (cf. Table 1). Mallory may then use these cookies to access Alice's account: if they are enough to act on Alice's behalf, session hijacking is possible. Even when this is not possible, however, security might still be at risk, because it might be that not all the cookies were disclosed to Mallory and the attempted operation failed because just a subset of the expected cookies was sent to the website. To account for this case, we also perform a fresh login to the website as Mallory to get a full set of cookies and then restore the cookies stolen from Alice before reattempting the operation, so that all the website cookies (though mixed from two different accounts) are sent as part of a new operation attempt: if the operation succeeds in Alice's account, session hijacking is possible. Specifically, the testing strategy proceeds as follows:

1. Login to `www.target.com` as Alice and reach the page under test;
2. Find the cookies which satisfy the cookie leakage conditions in Table 1 based on Mallory's capabilities and clear all the other cookies from the browser;
3. Perform the operation under test;
4. Check: has the operation been performed? If yes, report as insecure;
5. Clear the cookies from the browser;
6. Login to `www.target.com` as Mallory and reach the page under test;
7. Restore in the browser the cookies previously kept at step 2;
8. Perform again the operation under test;
9. Clear the cookies from the browser and login to `www.target.com` as Alice;
10. Check: has the operation been performed? If yes, report as insecure.

4.2 Testing for Session Fixation

To test for session fixation, we simulate a scenario where Mallory forces in Alice's browser all the cookies which are not freshly issued after login and do not have integrity against her (cf. Table 2). After Alice's login, Mallory presents the forced cookies to access Alice's account: if they are enough to act on Alice's behalf, session fixation is possible. The testing strategy for session fixation thus follows the same pattern of the testing strategy for session hijacking, with the only exception of step 2. More precisely, we replace step 2 as follows:

2. Find the cookies which satisfy the cookie compromise conditions in Table 2 based on Mallory's capabilities and were not freshly issued after the login process, then clear all the other cookies from the browser;

The testing strategy still has two exit conditions for the reasons explained above.

4.3 Testing for Cross-Site Request Forgery

The proposed testing strategy for CSRF is a variant of the one presented in [30], extended to consider related-domain attackers and network attackers, as well as to cover a few additional subtleties emerged in our security analysis. The idea is to trigger the operation under test as Mallory and intercept the corresponding HTTP request to transform it into a variant which can be forged from Alice's browser from a cross-site position. The forged request lacks both custom headers and same-site cookies, yet includes a potentially dangerous value in the `Referer` and `Origin` headers, based on Mallory's capabilities (cf. Table 3). If the request is successfully processed by the website, a CSRF attack is possible. More precisely, the testing strategy proceeds as follows:

1. Login to `www.target.com` as Mallory and reach the page under test;
2. Perform the operation under test and intercept the corresponding HTTP request before it is sent;
3. Clear the cookies from the browser;
4. Login to `www.target.com` as Alice;
5. Forge a copy of the HTTP request intercepted at step 2 from a cross-site position and let the browser attach the HTTP headers (including the `Cookie` header containing Alice's cookies, with the exception of same-site cookies);
6. Set the `Referer` and `Origin` headers of the forged request to the values in Table 3 based on Mallory's capabilities, e.g., using an intercepting proxy;
7. Check: has the operation been performed? If yes, report as insecure.

If the operation has not been performed, we can conclude security against web attackers, but the picture is more complicated in case of stronger attackers, who can break the confidentiality and the integrity of stateless anti-CSRF tokens. We identify potentially vulnerable implementations of the double submit pattern by inspecting the HTTP request intercepted at step 2. If the request contains a cookie c and a parameter p such that the value of c matches the value of p, we reason about the confidentiality and the integrity of c (cf. Table 1 and Table 2 respectively) and we revise the original testing strategy accordingly.

If c has low confidentiality, the anti-CSRF token stored in c becomes useless, because its value might be known to the attacker. However, this does not necessarily mean that it is possible to run a CSRF attack, because the tested website might implement multiple defenses against CSRF, e.g., it might also check for the presence of custom headers. To confirm the potential attack, we thus revise the original testing strategy as follows: after step 4 we read the value of the cookie c from Alice's session and at step 6 we modify the forged HTTP request so that the parameter p matches such value. This simulates a scenario where Mallory stole Alice's token and used it to forge the malicious request.

If c has low integrity, CSRF protection can be bypassed if the token stored in c is not session-dependent. To detect this, we have to test whether Mallory's token can successfully be used in Alice's session. We can do this by revising the original testing strategy as follows: after step 1 we read the value of the cookie c from Mallory's session and at step 6 we modify the forged HTTP request so that the cookie c matches such value. This simulates a scenario where Mallory's forced her own token into Alice's browser before forging the malicious request.

4.4 Discussion

Testing is a powerful tool to unveil security breaches, but clearly it is of no use in establishing security proofs. Irrespective of how carefully a testing strategy might have been designed, there is no way that black-box testing can be made complete in general: for instance, it is not possible to know whether a stateless anti-CSRF token is cryptographically bound to the value of a request parameter without having access to the web application code. A further important remark concerns *coverage*. In our testing strategies we assume that the security tester knows the security-sensitive operations to scrutinize. However, identifying all such operations in real websites might be complex: we refer to [10] for a recent research work on the topic. Having said that, the testing strategies we have just described turned out to be rather effective and useful in practice (see below).

5 Dredd: Implementation and Experiments

5.1 Security Testing with Dredd

We developed Dredd, a browser extension for Mozilla Firefox which implements and semi-automates the testing strategies presented in the previous section. The configuration of Dredd requires the specification of the access credentials of two test accounts (Alice and Mallory) for the website under test. When activated on a website, Dredd asks for the vulnerability to test for (session hijacking, session fixation or CSRF) and the attacker of interest (web attacker, related-domain attacker or network attacker). Dredd then instructs the security tester to login using one of the test accounts and perform the operation under test. Once the tester confirms that this has been done, Dredd runs the corresponding testing strategy up to completion and asks the security tester to confirm its outcome, i.e., to flag whether Mallory successfully managed to attack Alice or not. This is trivial to do for the security tester by visually checking the website, yet generally hard to automate given the variegate nature of existing web applications.

The Mozilla Firefox extension APIs support a natural and direct implementation of Dredd, because the `webRequest` API provides the ability to intercept and modify HTTP requests and responses, and extensions can ask for the `cookies` permission to access the cookie jar of all web origins. Porting Dredd to Google Chrome would be straightforward, given that Mozilla Firefox and Google Chrome essentially implement the same extension architecture.

5.2 Experimental Evaluation

We tested Dredd on 20 randomly sampled websites from the Alexa Top 1,000. We only considered websites in English or in Italian, since we were required to understand the website user interface and potential error messages. Moreover, we only considered websites which allow single sign-on access with a major identity provider (Google or Facebook), so as to avoid the manual account creation process. Finally, we only considered websites served over HTTPS, because the network attacker is part of our threat model.

For each website, we chose a single security-sensitive operation to test, based on our understanding of the web application semantics, and ran all our testing strategies under all the attackers of our threat model. Occasionally, we managed to avoid a few redundant tests: for instance, if a website is already vulnerable to CSRF attacks against the web attacker, then it also suffers from the same flaw against the related-domain attacker and the network attacker.

Table 4 provides the full breakdown of the identified vulnerabilities on the tested websites. We present the details in the rest of this section and discuss the security impact of our findings in Section 5.3.

Table 4. Overview of the vulnerabilities identified with Dredd

Website	*Session Hijacking*		*Session Fixation*		*CSRF*		
	Related	Network	Related	Network	Web	Related	Network
`www.adobe.com`	x	x					
`www.airbnb.it`	x						
`www.aol.com`	x						
`www.bitdefender.net`							
`www.coursera.org`	x						
`www.expedia.com`	x						
`www.geeksforgeeks.org`	x						
`www.genius.com`	x						
`www.glassdoor.com`		x					
`www.groupon.com`	x						
`www.imgur.com`	x	x				x	x
`www.immobiliare.it`	x	x	x	x			
`www.instacart.com`	x						
`www.kijiji.it`							
`www.medium.com`	x						
`www.mondadoristore.it`	x	x	x	x	x	x	x
`www.prezi.com`						x	x
`www.quora.com`	x	x	x	x	x	x	x
`www.scoop.it`		x					
`www.yandex.com`	x						

Session Hijacking. The first observation we make is that the related-domain attacker is significantly more powerful than the network attacker when it comes to session hijacking. In particular, we found that the related-domain attacker can hijack the sessions of 15 websites, which is motivated by the fact that large websites often span multiple sub-domains, hence the use of domain cookies for session management is widespread. The proposed testing strategy completed 13 times at step 4 and twice at step 10, which shows that having two exit conditions there is sometimes useful in practice.

The network attacker can perform session hijacking on 7 websites, all of which were deemed as vulnerable at step 4 of the testing strategy. These websites do not implement full HSTS, yet make an insufficient use of the `Secure` attribute to protect their session cookies; 5 of them do not activate HSTS at all, while the other 2 websites do it only partially, i.e., without protecting their sub-domains. As to the 13 secure cases, 9 websites adopt full HSTS and 4 websites do not, but still manage to protect their sessions thanks to an appropriate use of the `Secure` attribute and/or the adoption of host-only cookies (in case of partial HSTS).

Session Fixation. We identified room for session fixation in 3 websites. All these websites are vulnerable against both the related-domain attacker and the network attacker. None of the websites uses cookies prefixes, 2 of them do not activate HSTS at all, while the last website does it only partially, hence cookie integrity is not guaranteed against the aforementioned attackers. The proposed testing strategy completed twice at step 4 and once at step 10, hence handling two exit conditions is occasionally useful to catch real-world vulnerabilities.

Cross-Site Request Forgery. The web attacker is able to exploit CSRF vulnerabilities just in 2 websites. This suggests that the large majority of the developers of the websites we tested are aware of the dangers of CSRF, which is reassuring considered their popularity. Remarkably, however, both the related-domain attacker and the network attacker are able to exploit 2 additional CSRF vulnerabilities on the tested websites, due to incorrect implementations of the double submit pattern. We discuss these two cases in detail.

The website `www.imgur.com` performs the double submit using a domain cookie which is not marked as `Secure`, hence both the related-domain attacker and the network attacker can breach its confidentiality and circumvent the CSRF protection. The website `www.prezi.com` implements the double submit pattern by using a cookie which is not session-dependent, without deploying HSTS or cookie prefixes to protect its integrity, hence both the related-domain attacker and the network attacker can bypass the CSRF protection mechanism. Moreover, this cookie is shared with the sub-domains and lacks the `Secure` attribute, hence its confidentiality is not guaranteed against the aforementioned attackers, which enables an additional vector for CSRF.

5.3 Security Impact

Web Attacker. The web attacker is the baseline attacker to consider on the Web, since the owner of any untrusted website can potentially act as the web attacker against a high-profile service. Luckily, we only found two websites which suffer from significant security flaws exploitable by the web attacker. It is worth discussing here the case of `www.mondadoristore.it`. This e-commerce website does not implement any form of protection against CSRF, hence all the security-sensitive functionality typical of such services, e.g., shopping cart management, is left vulnerable. Even worse, the password change functionality can be abused to change the account password to a new value chosen by the attacker, which enables account takeover. This provides illegitimate access to confidential information like shipping addresses and credit card numbers.

Network Attacker. The network attacker is now a common web security threat to deal with, due to the widespread adoption of WiFi networks. Observe that all the attacks we discussed are effective even under the (optimistic) assumption that users navigate just low-profile websites over untrusted WiFi. The network attacker can perform session hijacking on 7 websites and session fixation on 3 websites, thus taking full control of the victim's session. Moreover, the network attacker can exploit CSRF vulnerabilities on 4 websites: 3 of them already suffer from a more serious security flaw like session hijacking, while `www.prezi.com` can only be targeted by means of CSRF. It is worth noticing that the network attacker can entirely bypass CSRF protection there and abuse all the website functionality, since the implementation of the defense mechanism itself is flawed.

Related-Domain Attacker. The related-domain attacker is the strongest attacker in our pool on the 20 tested websites, in particular due to her ability of performing session hijacking on 15 websites. One might argue that this was expected, given the common practice of building sessions on top of domain cookies, and that the related-domain attacker is not a realistic threat, given that most websites are not leasing sub-domains to untrusted users and organizations. Although we acknowledge that we cannot tell for sure whether the related-domain attacker is part of the threat model of the tested websites, we point out that recent work highlighted the dangers of related domains for session security [12]. In particular, the authors showed that the high number of sub-domains in popular websites amplifies the attack surface against web sessions, because it is common to identify TLS vulnerabilities in at least one of these hosts.

In our work, we further substantiate the threats posed by the related-domain attacker. By crawling the Certificate Transparency logs, we observed that 6 out of the 20 tested websites have more than 100 sub-domains and that the median number of sub-domains is 26, i.e., the attack surface coming from related domains is large. Notice that these numbers represent a lower bound of the number of existing sub-domains and that 18 out of the 20 websites use a wildcard HTTPS certificate, which means that the number of their sub-domains can grow arbitrarily large. Most importantly, we analyzed all the 20 websites with the tool developed in [12] to identify TLS vulnerabilities in any of their sub-domains, exposing 5 vulnerable websites: `www.adobe.com`, `www.aol.com`, `www.expedia.com`, `www.groupon.com` and `www.yandex.com`. Since the detected TLS vulnerabilities affect the confidentiality of domain cookies, we note that at least these 5 websites are at concrete risk of session hijacking (see Table 4).

5.4 Responsible Disclosure

We responsibly disclosed all the identified vulnerabilities exploitable by the web attacker and the network attacker to the respective website owners or security teams. We also reported all the sub-domains suffering from TLS vulnerabilities according to the tool developed in [12], which may allow an attacker with control of the network traffic to play the role of the related-domain attacker.

We also collected a few responses to our responsible disclosure. In particular, `www.imgur.com` and `www.quora.com` acknowledged the reported vulnerabilities as valid through their bug bounty programs. Though none of our reports was deemed invalid, which confirms the effectiveness of Dredd, we unfortunately noticed that developers are not always aware of the security implications of the reported vulnerabilities or do not consider security as a priority. For example, we observed that several bug bounty programs consider out of scope the following security issues: lack of cookie security attributes, CSRF, misconfiguration of TLS. However, these vulnerabilities have already been exploited multiple times, often with severe security consequences: for example, CSRF has recently been proved exploitable for purchase hijacking, payment hijacking and account takeover on existing websites [10]. We hope that the security awareness of web developers will increase in the next future as a result of recent security analyses of existing websites, including the present one.

6 Related Work

Web session security is a popular research area, which received extensive attention from the security community: we refer to [13] for a recent survey.

Session hijacking is a major threat for web sessions, yet no rigorous testing strategy for it has been proposed so far. However, several protection mechanisms have been designed, including one-time cookies [16], origin-bound certificates [17] and a variety of browser extensions which protect session cookies [24, 31, 8]. A variant of the attack called *sub-session hijacking* has also been studied [14].

Session fixation has been thoroughly studied in previous work, which also proposed a methodology to identify room for potential attacks [19]. Unfortunately, the proposed approach assumes the knowledge of the full set of the session cookies and might produce false positives, since it is based on manual code inspection and does not take cookie integrity into account.

Robust defenses against CSRF have been first presented in [4]. Although the security analysis therein is thorough and exhaustive, it is also quite outdated: when the paper was published, HSTS, cookie prefixes and same-site cookies were not available yet, and the threats posed by related-domain attackers were still unknown [6]. A testing strategy for CSRF vulnerabilities has been first proposed in [30]. Our strategy extends this work to cover related-domain attackers and network attackers, which requires one to reason about their ability to circumvent the protection granted by stateless anti-CSRF tokens.

Deemon [26] is a dynamic analysis tool which can be used to identify CSRF vulnerabilities in PHP code. Deemon is a language-based tool, which only works on PHP and requires access to the web application code, while our testing strategies are language-independent and black-box. Mitch [10] is a black-box analysis tool for CSRF vulnerabilities based on machine learning and syntactic heuristics on HTTP responses, but its threat model only considers the web attacker.

Different flavours of web session integrity have been already discussed and formalized in the literature [1, 9, 20, 11]. Notable practical papers on web session integrity include [29] and [35], which showed the dangers posed by the lack of cookie confidentiality and integrity on existing popular websites.

Finally, it is worth noticing that *model-based* testing has been proposed for web application security in a number of papers [7, 28, 27]. This approach operates on a formal model of the web application, rather than on its implementation. As such, it can leverage tools like model-checkers to perform a systematic exploration of the web application state space, but it requires the creation of formal web application models. This is typically a time-consuming and complex task, which does not match the expertise of most web developers.

7 Conclusion

We discussed session integrity as a necessary property for web session security, which is violated by classic attacks like session hijacking, session fixation and cross-site request forgery. We then proposed black-box testing strategies to detect session integrity flaws and we implemented these strategies in a browser extension called Dredd to semi-automate the testing process. Finally, we used Dredd to expose a significant number of vulnerabilities in popular websites from the Alexa ranking. Our work provides yet another proof of the security challenges of web session implementations: though all the attacks we considered are well-known, the complexity of the web platform and its threat model makes subtle security issues hard to detect. We showed that Dredd is a useful tool to support web developers in a systematic security assessment, which identifies exploitable vulnerabilities without requiring a deep understanding of the web application.

As future work, we plan to further automatize the security testing process to extend its coverage and enlarge the scope of our analysis to a larger number of websites, so as to get a better understanding of session security on the current Web. Moreover, we also would like to devise new testing strategies to cover more attacks, like login CSRF and cookie forcing [13].

Acknowledgements. We would like to thank Alessandro Busatto for contributing to an early stage of the project.

References

1. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J.C., Song, D.: Towards a Formal Foundation of Web Security. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010. pp. 290–304 (2010)
2. Barth, A.: HTTP State Management Mechanism. Available at `http://tools.ietf.org/html/rfc6265` (2011)
3. Barth, A.: The Web Origin Concept. Available at `http://tools.ietf.org/html/rfc6454` (2011)
4. Barth, A., Jackson, C., Mitchell, J.C.: Robust Defenses for Cross-Site Request Forgery. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008. pp. 75–88 (2008)
5. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.C.: State of the art: Automated black-box web application vulnerability testing. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 332–345 (2010)
6. Bortz, A., Barth, A., Czeskis, A.: Origin Cookies: Session Integrity for Web Applications. In: Web 2.0 Security & Privacy Workshop (W2SP 2011) (2011)
7. Büchler, M., Oudinet, J., Pretschner, A.: Spacite - web application testing engine. In: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012. pp. 858–859 (2012)
8. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W.: CookiExt: Patching the Browser Against Session Hijacking Attacks. Journal of Computer Security **23**(4), 509–537 (2015)

9. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W., Tempesta, M.: Provably Sound Browser-Based Enforcement of Web Session Integrity. In: Proceedings of the IEEE 27th Computer Security Foundations Symposium, CSF 2014. pp. 366–380 (2014)
10. Calzavara, S., Conti, M., Focardi, R., Rabitti, A., Tolomei, G.: Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In: IEEE European Symposium on Security and Privacy (2019)
11. Calzavara, S., Focardi, R., Grimm, N., Maffei, M.: Micro-policies for web session security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 179–193 (2016)
12. Calzavara, S., Focardi, R., Nemec, M., Rabitti, A., Squarcina, M.: Postcards from the post-HTTP world: Amplification of HTTPS vulnerabilities in the web ecosystem. In: IEEE Symposium on Security and Privacy (2019)
13. Calzavara, S., Focardi, R., Squarcina, M., Tempesta, M.: Surviving the web: a journey into web session security. ACM Computing Surveys (2017)
14. Calzavara, S., Rabitti, A., Bugliesi, M.: Sub-session hijacking on the web: Root causes and prevention. Journal of Computer Security **27**(2), 233–257 (2019)
15. Calzavara, S., Tolomei, G., Casini, A., Bugliesi, M., Orlando, S.: A supervised learning approach to protect client authentication on the web. TWEB **9**(3), 15:1–15:30 (2015)
16. Dacosta, I., Chakradeo, S., Ahamad, M., Traynor, P.: One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens. ACM Transactions on Internet Technology **12**(1), 1–24 (2012)
17. Dietz, M., Czeskis, A., Balfanz, D., Wallach, D.S.: Origin-Bound Certificates: a Fresh Approach to Strong Client Authentication for the Web. In: Proceedings of the 21th USENIX Security Symposium, USENIX 2012. pp. 317–331 (2012)
18. Hodges, J., Jackson, C., Barth, A.: HTTP Strict Transport Security (HSTS). Available at `http://tools.ietf.org/html/rfc6797` (2012)
19. Johns, M., Braun, B., Schrank, M., Posegga, J.: Reliable Protection Against Session Fixation Attacks. In: Proceedings of the 26th ACM Symposium on Applied Computing, SAC 2011. pp. 1531–1537 (2011)
20. Khan, W., Calzavara, S., Bugliesi, M., Groef, W.D., Piessens, F.: Client side web session integrity as a non-interference property. In: Proceedings of the 10th International Conference on Information Systems Security, ICISS 2014. pp. 89–108 (2014)
21. Kranch, M., Bonneau, J.: Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015 (2015)
22. Mozilla: Same-Origin Policy. `http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy` (2015)
23. Mundada, Y., Feamster, N., Krishnamurthy, B.: Half-baked cookies: Hardening cookie-based authentication for the modern web. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016. pp. 675–685 (2016)
24. Nikiforakis, N., Meert, W., Younan, Y., Johns, M., Joosen, W.: SessionShield: Lightweight Protection against Session Hijacking. In: Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011. pp. 87–100 (2011)
25. OWASP: OWASP Testing Guide. `https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents` (2016)

26. Pellegrino, G., Johns, M., Koch, S., Backes, M., Rossow, C.: Deemon: Detecting CSRF with dynamic analysis and property graphs. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1757–1771 (2017)
27. Peroli, M., Meo, F.D., Viganò, L., Guardini, D.: Mobster: A model-based security testing framework for web applications. Softw. Test., Verif. Reliab. **28**(8) (2018)
28. Rocchetto, M., Ochoa, M., Dashti, M.T.: Model-based detection of CSRF. In: ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings. pp. 30–43 (2014)
29. Sivakorn, S., Polakis, I., Keromytis, A.D.: The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 724–742 (2016)
30. Sudhodanan, A., Carbone, R., Compagna, L., Dolgin, N., Armando, A., Morelli, U.: Large-scale analysis & detection of authentication cross-site request forgeries. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 350–365 (2017)
31. Tang, S., Dautenhahn, N., King, S.T.: Fortifying web-based applications automatically. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011. pp. 615–626 (2011)
32. West, M.: Cookie prefixes. Available at `https://tools.ietf.org/html/draft-ietf-httpbis-cookie-prefixes-00` (2016)
33. West, M.: Strict secure cookies. Available at `https://tools.ietf.org/html/draft-ietf-httpbis-cookie-alone-01` (2016)
34. West, M., Goodwin, M.: Same-site cookies. Available at `https://tools.ietf.org/id/draft-ietf-httpbis-cookie-same-site-00.txt` (2016)
35. Zheng, X., Jiang, J., Liang, J., Duan, H., Chen, S., Wan, T., Weaver, N.: Cookies Lack Integrity: Real-World Implications. In: Proceedings of the 24th USENIX Security Symposium, USENIX 2015. pp. 707–721 (2015)