Journal Title 0 (2018) 1–0
IOS Press

Sub-Session Hijacking on the Web: Root Causes and Prevention

Stefano Calzavara [a,*], Alvise Rabitti [a] and Michele Bugliesi [a]
[a] *Dipartimento di Scienze Ambientali, Informatica e Statistica, Università Ca' Foscari Venezia, Italy*
E-mails: stefano.calzavara@unive.it, alvise.rabitti@unive.it, michele.bugliesi@unive.it

Abstract. Since cookies act as the only proof of a user identity, web sessions are particularly vulnerable to session hijacking attacks, where the browser run by a given user sends requests associated to the identity of another user. When $n > 1$ cookies are used to implement a session, there might actually be n sub-sessions running at the same website, where each cookie is used to retrieve part of the state information related to the session. Sub-session hijacking breaks the ideal view of the existence of a unique user session by selectively hijacking m sub-sessions, with $m < n$. This may reduce the security of the session to the security of its weakest sub-session. In this paper, we take a systematic look at the root causes of sub-session hijacking attacks and we introduce *sub-session linking* as a possible defense mechanism. Out of two flavors of sub-session linking desirable for security, which we call *intra-scope* and *inter-scope* sub-session linking respectively, only the former is relatively smooth to implement. Luckily, we also identify programming practices to void the need for inter-scope sub-session linking. We finally present Warden, a server-side proxy which automatically enforces intra-scope sub-session linking on incoming HTTP(S) requests, and we evaluate it in terms of protection, performances, backward compatibility and ease of deployment.

Keywords: Web application security, web sessions, HTTP cookies

1. Introduction

Many websites need to track a user identity across multiple requests, for instance when providing access to their private area. Unfortunately, HTTP is a *stateless* protocol, in that different HTTP requests normally look unrelated to their recipient. This means that, to implement user-authenticated *sessions* over HTTP, one needs to explicitly attach to each request enough information to associate it to a user identity. The most common way to achieve this nowadays is by means of *cookies* [1]. Roughly, cookies are key-value pairs generated by a website and sent in response to browser requests; browsers then automatically attach cookies to later requests sent to the same website. If the cookies issued upon a successful login contain enough information to tell users apart, they can be used to collect different browser requests under the same user identity.

The main security risks of web sessions are well-known nowadays, as discussed in a recent survey [2]. In particular, since cookies act as the only proof of a user identity, web sessions are extremely vulnerable to *session hijacking* attacks, where the browser run by a given user sends requests associated to the identity of another user. Protection against session hijacking has been studied under different threat models, including both *web attackers*, who operate websites hosting malicious contents and are able to exploit content injection vulnerabilities on trusted websites, and *network attackers*, who can block, inspect and corrupt the HTTP traffic.

*Corresponding author. E-mail: stefano.calzavara@unive.it.

Nowadays, it is possible to implement robust countermeasures against session hijacking, even against a powerful attacker like the network attacker. Necessary (but not sufficient) ingredients include the use of HSTS [3] to enforce ubiquitous encryption and cookie security attributes like HttpOnly and Secure [1] to defend against cookie leakage. Unfortunately, web developers often ignore these recommended security practices, because the adoption of HSTS still lags behind [4, 5] and cookie security attributes are often unset [6]. Protection against session hijacking should thus not be taken for granted, especially when the attack surface is amplified by the widespread practice of using multiple cookies for session management [7, 8].

When $n > 1$ cookies are used to implement a session, there might actually be n sub-sessions running at the same website, where each cookie is used to access a different service or database and retrieve part of the state information related to the session. *Sub-session hijacking* is a subtle variation of traditional session hijacking, where the attacker breaks the ideal view of the existence of a unique user session by selectively hijacking m sub-sessions, with $m < n$. This may reduce the security of the session to the security of its weakest sub-session, e.g., the one running unencrypted and vulnerable to network attacks. Recent research identified sub-session hijacking attacks on several major websites, like Amazon, Ebay, Google and YouTube [5, 9]. Despite the significance and prevalence of sub-session hijacking attacks, however, previous research did not take a systematic look at their root causes and did not propose robust defenses against them.

1.1. Contributions

In this paper, we make the following contributions:

(1) we provide a general overview of the anatomy of sub-session hijacking attacks and we analyze their security import under different threat models, motivating their significance (Section 3);
(2) we propose *sub-session linking* as a possible defense mechanism. Out of two flavors of sub-session linking desirable for security, which we call *intra-scope* and *inter-scope* sub-session linking respectively, only the former is relatively smooth to implement. Luckily, we also identify programming practices to void the need for inter-scope sub-session linking (Section 4);
(3) we present Warden, a server-side proxy which automatically enforces intra-scope sub-session linking on incoming HTTP(S) requests. Warden is entirely based on existing web technologies and requires only minimal configuration efforts by web developers (Section 5);
(4) we positively evaluate Warden in terms of protection, performances, backward compatibility and ease of deployment (Section 6).

2. Background

2.1. Threat Model

Session hijacking traditionally considers different threat models. The *web attacker* runs a set of malicious websites hosting arbitrary attacker-controlled contents and can exploit content injection vulnerabilities on trusted websites. This conservative choice is motivated by the fact that content injections and cross-site scripting (XSS) in particular are common on the web [10]. A more powerful variant of the web attacker, known as the *related-domain attacker*, can host some of the malicious websites on a domain sharing a "sufficiently long" suffix with the domain of the target website; examples of related

domains are foo.example.com and bar.example.com[1]. By using the Domain attribute of cookies, websites hosted on related domains can set cookies which are visible to each other and indistinguishable from other cookies which are set so that this sharing across domains is not desired [11]. Finally, the *network attacker* is able to block, inspect and corrupt all the HTTP traffic exchanged on the network. He can also block outgoing HTTPS connections, but he cannot read or modify the HTTPS traffic exchanged with trusted websites, because we assume the use of signed public key certificates.

2.2. Web Sessions

Since there is no formal definition of web session, it is worth stressing two points to set the boundaries of the present paper:

(1) we only consider sessions built on top of cookies. One could use other techniques to implement web sessions [12], e.g., by exchanging JSON Web Tokens using AJAX, but cookie-based sessions still cover the large majority of the web sessions;
(2) we are only interested in user-authenticated sessions established upon a successful login. In this common type of sessions, some cookies are used to authenticate users at the server side.

The security of web sessions is thus intimately connected to *session cookies*, i.e., cookies which are used to get authenticated access to security-sensitive state information at the server.

Figure 1 shows the typical establishment of a web session. A user Alice first inputs her username and password in a page displayed by the browser B. The browser sends the credentials to the server S, which checks them against a database containing account information. If the check succeeds, S sends B a session cookie *sid* storing a random session identifier bound to Alice's identity. Since all the next requests sent by B to S automatically include the cookie, S can associate them to Alice's account.

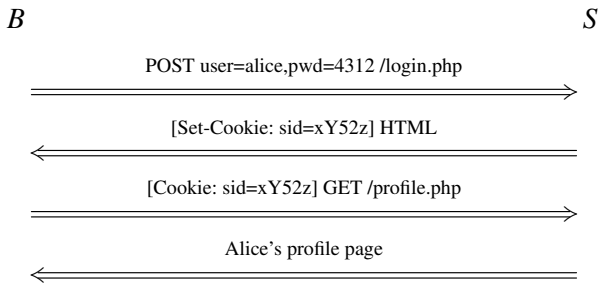

Figure 1. Example web session

2.3. Session Hijacking

Though cookie-based sessions are widespread on the web, they exhibit significant shortcomings against *session hijacking*, because session cookies act as the only proof of a user's identity. When session hijacking succeeds, the browser run by a given user sends requests associated to the identity of another user. We thus have two flavors of session hijacking:

[1]The common domain suffix must not occur in a list of public suffixes available at https://publicsuffix.org/.

(1) attacker-as-user session hijacking ($A \mapsto U$): if the cookies identifying the user's session are leaked and set into the attacker's browser, the attacker can perform security-sensitive actions on the user's behalf;
(2) user-as-attacker session hijacking ($U \mapsto A$): if the cookies identifying the attacker's session are forced into the user's browser, the attacker can fool the user into performing security-sensitive actions on the attacker's behalf.

Both scenarios may have severe practical consequences, depending on the specific website functionality. Luckily, it is possible to prevent session hijacking using existing technologies, though this requires significant work by web developers.

To prevent $A \mapsto U$, one has to implement defenses against *session fixation* attacks [13]. In these attacks, a cookie chosen by the attacker is set in the user's browser and endorsed to play the role of a session cookie after the verification of the user's password. A simple yet effective way to defend against session fixation is issuing fresh session cookies upon login. Moreover, to prevent $A \mapsto U$ it is critical that the *confidentiality* of session cookies is ensured. This can be done by preventing cookie sharing with untrusted related domains and by using the cookie attributes HttpOnly and Secure [1]. Cookies marked as HttpOnly are never exposed by browsers through channels other than HTTP or HTTPS requests: this prevents their exfiltration by malicious scripts, for instance injected via XSS. Cookies marked as Secure, instead, are never sent in HTTP requests, but only attached over HTTPS: this ensures that they are never exposed to attackers sniffing the HTTP traffic.

Protecting against $U \mapsto A$ requires the implementation of defenses against *login cross-site request forgery* attacks [14]. In these attacks, the attacker's credentials are submitted to the target website from the user's browser, thus forcing it into the attacker's session. This can be prevented by implementing standard CSRF defenses on the login form. Moreover, to prevent $U \mapsto A$ the *integrity* of session cookies must be ensured. Unfortunately, it is well-known that cookies do not provide strong integrity guarantees against related-domain attackers and network attackers [1]. One has then to carefully place web contents so as to ensure that session cookies cannot be set by untrusted related domains. Moreover, browsers must never open an HTTP connection to the website or any of its sub-domains (even non-existing ones). This can only be ensured by deploying HSTS on the base domain of the website, activating the includeSubDomains option in its configuration [5].

3. Sub-Session Hijacking

3.1. Defining Sub-Session Hijacking

The implementation of web sessions may be significantly more complex than the simple example of Figure 1. In particular, the use of multiple session cookies at the same website is surprisingly common in the wild [7, 8]. A few possible reasons why a website may use multiple session cookies are the following:

- the website needs to integrate together several services or databases, each identifying users in a different way;
- the website is developed using multiple languages and web development frameworks, each implementing session management differently;
- the website enforces different authorization policies upon presentation of different cookies.

In the end, web sessions are often implemented on top of n sub-sessions, each identified by a cookie which is used to get authenticated access to part of the state information stored at the server side; the n sub-sessions may be eventually linked together by the implementation of appropriate server-side logic, for instance before authorizing security-sensitive actions.

When a web session is built over n sub-sessions, *sub-session hijacking* attacks break the ideal view of the existence of a unique user-authenticated session by hijacking just m sub-sessions, with $m < n$. If the server-side logic for linking together sub-sessions is not robust, hijacking just m sub-sessions may fully compromise the confidentiality or the integrity of the entire session [5, 9].

3.2. Impact of Sub-Session Hijacking

Understanding how significant is sub-session hijacking in practice is an intriguing question. Given that defenses against session hijacking also prevent sub-session hijacking, it might seem that sub-session hijacking is an already solved problem. Unfortunately, real-world websites rarely implement full protection against session hijacking [5, 6, 9, 15, 16] and commonly build their sessions on top of cookies with different confidentiality or integrity guarantees, which opens the way to sub-session hijacking. Concrete examples of reasons why this might happen in practice are the following:

- mixed content websites are developed using a combination of HTTP and HTTPS. Mixed content website cannot mark all their session cookies as Secure, otherwise sessions would be disrupted when navigating their HTTP portion;
- websites may need to read the value of session cookies via JavaScript for various reasons, which prevents the adoption of the HttpOnly attribute for these cookies;
- in a multi-domain setting only a limited number of cookies may be shared among sub-domains using the Domain attribute, because other cookies may contain confidential information which must not be disclosed across domains.

A last point worth noticing is that there are also cases where traditional session hijacking is possible, yet sub-session hijacking is preferred by the attacker to act more surreptitiously. For instance, a published security analysis performed on popular websites [5] reports an attack against Gmail where the integrity of only a subset of the session cookies is compromised, so that the visual indicators in the user's browser are left unchanged, but the user is effectively forced into the attacker's session.

3.3. Sub-Session Hijacking in the Wild

Previous research already highlighted the existence of sub-session hijacking attacks on real websites, including major security-sensitive services [5, 9]. These attacks have been found by means of a thorough manual analysis of a small number of selected websites from the Alexa ranking, hence they provide only a limited insight about the severity of the problem on the web. Though a large-scale analysis of this aspect would certainly be interesting, it would also be quite complicated to carry out, because it would require the identification of the set of the session cookies issued upon login at the individual services - a task which already proved challenging to automate by itself [7, 8].

Luckily, a useful outcome of previous research is a publicly available dataset of cookies[2] marked with a binary label, used to tell apart session cookies from all the other cookies [7]. This dataset includes 2,546 cookies collected after a successful login on 215 popular websites from the Alexa ranking, including 332

[2] Available at http://bit.ly/1u8Qfiz

session cookies. The size of the dataset is relatively limited, because collecting session cookies requires the possession of personal accounts at the websites and the automation of the detection process itself is far from trivial, but this is the only publicly available dataset of session cookies we are aware of and we find it useful at least for a preliminary investigation. Notice that larger and more up-to-date datasets of cookies are also available [17], but they do not discriminate between session cookies and cookies used for other purposes, which makes them inadequate to analyze the significance of sub-session hijacking. Indeed, we are not even aware of public cookie datasets which were built after a successful authentication at the crawled websites.

We can use the aforementioned cookie dataset to assess how many of the 215 websites included therein are potentially susceptible to sub-session hijacking by counting how many websites issue more than one session cookie upon login. As it turns out, this happens on 92 out of 215 websites (42.8%), which gives an empirical evidence of the dangers of sub-session hijacking in the real world.

4. Preventing Sub-Session Hijacking

Previous research pointed out that "many web applications do not adopt mechanisms to bind sub-sessions together" [5], thus likely suggesting that linking together sub-sessions is an effective defense against sub-session hijacking. Unfortunately, a more careful analysis reveals that sub-session linking may be hard to implement, e.g., due to the presence of session cookies with different scopes, hence preventing sub-session hijacking is challenging.

4.1. Cookie Scopes

The default scope of a cookie consists of the domain and the sub-paths of the HTTP(S) endpoint setting it. For instance, a cookie set by https://www.example.com/ will be attached to any HTTP(S) request to example.com. There are however three mechanisms to change the scope of cookies [1]:

- the Secure attribute restricts the cookie scope to the HTTPS protocol;
- the Domain attribute enlarges the cookie scope to sibling domains. For example, foo.bar.com can set a cookie with the Domain attribute set to .bar.com, which is visible to any sub-domain of bar.com;
- the Path attribute restricts the cookie scope to a given path. Only requests directed to its sub-paths will include the cookie.

Browsers allow the storage of multiple cookies with the same name, provided that they differ at least in their Domain or Path attribute. Since these cookies may have an overlapping scope, the same HTTP(S) request may include multiple cookies with the same name, which might confuse the website about which cookies should be taken into account when processing it. This confusion may enable sub-session hijacking attacks via *cookie shadowing* [5], where the attacker sets a session cookie with the same name of an existing one, but with different scope. Here, we assume that all the legitimate session cookies have distinct names, which is reasonable for the very large majority of websites and simplifies the discussion, because it makes cookie shadowing easy to detect.

4.2. Sub-Session Linking and Consistency

To understand the goals and challenges of sub-session linking, it is useful to dig a bit more into the anatomy of session management and sub-session hijacking. A session starts when a login operation is

successfully performed and a set of session cookies is appointed to play the role of the user's password for authentication purposes. Additional session cookies can also be issued when the user is already authenticated: for instance, a website may grant access to its private area upon login, but only release the session cookies needed to access its mailbox service when the user later accesses it.

Assume that a website issues n_0 session cookies upon login and that its services run on top of $n \geqslant n_0$ session cookies. If u users are authenticated at some point in time, the website needs to keep track of s session cookies with $u \cdot n_0 \leqslant s \leqslant u \cdot n$, hence:

(1) the session information stored at the server-side is much more fragmented than the ideal view of u authenticated users navigating the website;
(2) the website may not always see the full set of session cookies stored in the browser sending an authenticated request, because different session cookies issued to the same user can have different scopes;
(3) the exact number of session cookies released to each user up to a given point in time may be easily overlooked.

Sub-session hijacking exploits these implementation gaps by compromising the confidentiality or the integrity of a subset of the session cookies of a target user: indeed, if only one session cookie was issued to each user, then we would have $n = n_0 = 1$ and $s = u$ and no sub-session hijacking would be possible.

To understand how sub-session linking should work to be effective, we define a notion of *consistency* for sets of session cookies. Intuitively, a set of session cookies C is consistent when each $c \in C$ identifies a sub-session which contributes to implement the same user-authenticated session. More constructively, the set of session cookies released upon login is always consistent and, if C is a consistent set of session cookies and C' is a new set of session cookies released upon presentation of C, then also $C \cup C'$ is consistent. The goal of sub-session linking is ensuring that, every time an authenticated request is processed, then *the browser sending that request stores the most updated consistent set of session cookies released to the authenticated user*, which is enough to fix (1), (2) and (3).

4.3. Implementing Sub-Session Linking

4.3.1. Intra-Scope Sub-Session Linking

If all the session cookies of a website have the same scope, each authenticated request sent to the website includes the full set of session cookies stored in the browser, which can be checked for consistency. This means that the website always receives enough information to link together all the sub-sessions and prevent sub-session hijacking. Figure 2 exemplifies how intra-scope sub-session linking can be performed: in the figure, the attacker corrupted the integrity of the session cookie *sessid* stored in the user's browser to force it in the attacker's sub-session. However, since the user's browser stores a mixture of session cookies issued to the user and session cookies issued to the attacker, the HTTP(S) requests sent from it will attach an inconsistent set of session cookies and should be rejected.

Notice that the idea of checking consistency only works as long as session cookies are not modified at the browser side, but this is a reasonable assumption for most websites, because session cookies are typically used just to authenticate users. Observe that, if session cookies need to be modified at the browser side for generic reasons, they cannot be protected using the HttpOnly attribute, which makes them potentially vulnerable to exfiltration via XSS.

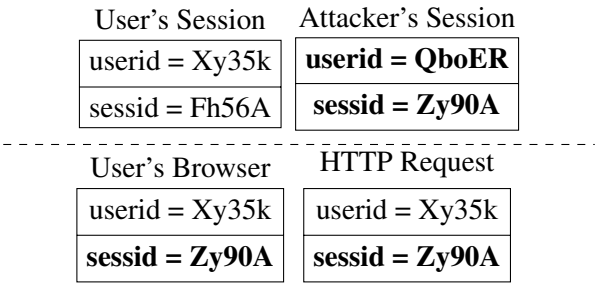

Figure 2. Intra-scope session linking

4.3.2. Inter-Scope Sub-Session Linking

If the session cookies of a website have different scopes, there might be legitimate authenticated requests to the website which do not include the full set of session cookies stored in the browser, hence the website might not have enough information to detect sub-session hijacking attacks. This is exemplified in Figure 3, where we use a light gray background to denote Secure cookies: in the figure, the attacker compromised the confidentiality of the session cookie *userid* stored in the user's browser and tries to hijack the user's sub-session by setting it in her own browser. Although the attacker's browser stores an inconsistent set of session cookies, the website cannot detect this just by inspecting the HTTP traffic, because Secure cookies are not sent in clear. Similar observations apply when a website uses multiple session cookies, but only some of them have the Domain or the Path attribute set.

Inter-scope sub-session linking is much more complicated to implement than intra-scope sub-session linking, because it calls for the implementation of a custom protocol, whose practicality is unclear. Luckily, as we discuss in the next sub-section (Section 4.4), it is typically possible to design the website so that inter-scope sub-session linking is not required for security.

4.4. Avoiding Inter-Scope Sub-Session Linking

We call *scope fragmentation* the presence of legitimate authenticated HTTP(S) requests to a service which do not include the full set of session cookies stored for it in the browser. If there is no scope fragmentation, then inter-scope session linking is not needed to prevent sub-session hijacking.

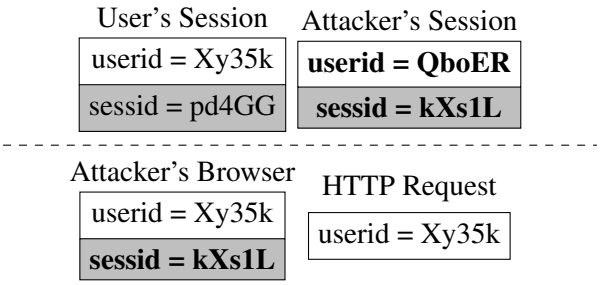

Figure 3. The problem of inter-scope session linking

4.4.1. The Secure Attribute

The use of the Secure attribute may lead to scope fragmentation only on mixed content websites, because websites which are entirely deployed over HTTPS will always receive both the Secure and the non-Secure cookies. However, the security properties of mixed content websites are dubious at the very least [9, 18] and it is likely that more and more websites will soon perform a full transition to HTTPS, due to the choice of major web browsers to mark HTTP as insecure [19].

If a website marks at least one session cookie as Secure, it should perform a full transition to HTTPS and mark all its session cookies as Secure. Even better, HSTS could be used to ensure that the website can only be contacted over HTTPS by modern web browsers, which would make the application of the Secure attribute redundant as long as the Domain attribute is not used or the includeSubDomains option of HSTS is activated [4]. Though this refactoring might require some work by web developers, it would significantly improve session security against network attackers and remove one of the causes of scope fragmentation.

4.4.2. The Path Attribute

The key insight to avoid scope fragmentation due to the use of the Path attribute is that the Path attribute is not a security feature [1, 20]. Cookies do not provide isolation by path, because web browsers can ignore path restrictions when exposing cookies to non-HTTP APIs such as document.cookie. Moreover, websites can set cookies with arbitrary paths by using the Set-Cookie header, so the use of the Path attribute does not really improve the security guarantees of cookies.

Under the assumption that all the session cookies have different names to prevent cookie shadowing, just setting the Path attribute of session cookies to the root directory would not break the website functionality and it would not weaken its security, while fixing a cause of scope fragmentation.

4.4.3. The Domain Attribute

In a multi-domain setting, only a limited set of session cookies may be shared among sub-domains via the Domain attribute and enlarging the scope of the other session cookies to the full domain to prevent scope fragmentation might be dangerous, because it may allow a related-domain attacker to disclose sensitive information or perform session hijacking. A more careful analysis is thus needed, depending on the trust assumptions between related-domain services.

If related-domain services are mutually distrusting, it is reasonable to assume that their back-ends are isolated and built on top of independent sessions, hence enlarging the scope of their session cookies is not actually needed to prevent sub-session hijacking. To exemplify, consider a scenario where a user has established a session at www.example.com, getting a cookie *rsid* intended to provide some session information to both a mail service at mail.example.com and an e-shop at shop.example.com; this cookie would have the Domain attribute set to .example.com to support this use case. The two services, in turn, release individual session cookies *m* and *s* respectively when accessed to implement their internal functionalities. Although in this setting not all the session cookies have the same scope, we observe there are actually two independent sessions running rather than a single session and no scope fragmentation occurs: both the email service and the e-shop have full visibility of their session cookies and can thwart sub-session hijacking attacks just by performing intra-scope sub-session linking.

If instead related-domains services trust each other, it is possible to uniformly enlarge the scope of their session cookies to the full domain, so that scope fragmentation is avoided and intra-scope session linking becomes again an effective countermeasure against sub-session hijacking.

4.4.4. Discussion

By using programming practices which prevent scope fragmentation, robust protection against sub-session hijacking can be entirely enforced by using intra-scope sub-session linking. We thus see protection against sub-session hijacking as a two step process, where scope fragmentation is first eliminated and then intra-scope sub-session linking is enforced.

Intra-scope sub-session linking is a systematic process to implement, which can be carried out automatically without requiring a deep understanding of the website to protect, while removing scope fragmentation is a much more delicate task, which may require a careful threat analysis and a website refactoring which may be hard to fully automate. In Section 5, we propose a server-side proxy which automatically implements intra-scope sub-session linking without requiring website changes, so as to enforce protection against sub-session hijacking as long as the website avoids scope fragmentation.

4.5. Sub-Session Linking in the Wild

We reconsidered the 92 websites which we identified as potentially vulnerable to sub-session hijacking based on the analysis in Section 3.3. For these websites, we computed the number of scopes for their session cookies, as shown in Table 1, and we observed that only the session cookies of 20 out of 92 websites span more than one scope (21.7%). The differentiation of scopes for the session cookies is only due to the use of the Secure and Domain attributes, because all the session cookies in the dataset have the Path attribute set to the root directory.

#scopes	#websites
1	72
2	18
3	2

Table 1
Session cookie scopes in the wild

By using these numbers, we can get a picture of the presence of scope fragmentation in the wild. There is only one subtlety worth noticing: if the scopes of the session cookies of a website differ only due to the use of the Secure attribute, but the website makes use of HSTS, then there is no scope fragmentation, because the session is guaranteed to entirely take place over HTTPS. We only found 2 such cases in our dataset, hence scope fragmentation affects only 18 out of 92 websites (19.6%). This suggests that scope fragmentation is present in the wild, but not pervasive, though we notice that the considered dataset only includes cookies collected upon the login process and does not account for the case of session cookies which are later added during user browsing. Though this limitation may partially affect the picture, these numbers suggest that intra-scope sub-session linking is already a robust defense against sub-session hijacking for many existing websites.

5. Warden

Warden is a server-side proxy which automatically enforces intra-scope sub-session linking on incoming HTTP(S) requests[3]. As explained in the previous section, this is sufficient to entirely prevent

[3]We make Warden publicly available at https://github.com/alviser/warden

sub-session hijacking in absence of scope fragmentation. Warden operates by creating cryptographic consistency proofs for session cookies based on HMACs, which are stored themselves into cookies. By operating as a server-side proxy, Warden can only protect session cookies which are set via HTTP headers and never modified at the client side: we discuss the import of such design choice for existing websites in Section 6.3.

5.1. Design Goals

The first design goal of Warden is *ease of deployment*. Warden aims at improving protection against sub-session hijacking without requiring changes to the server-side logic of the website to protect. This is important for the practical adoption of the tool, because even web developers who are more concerned about session security may have troubles at retrofitting new defenses into existing, complex code bases, possibly developed using multiple programming languages and frameworks.

A second design goal of Warden is *backward compatibility*. Warden should be usable off-the-shelf by existing websites without breaking their functionality and without requiring browser changes, so it should be entirely based on existing web technologies. Finally, Warden should only have a limited impact on the *performances* of the website to protect to encourage its adoption.

5.2. Challenges

The main challenge in the design of Warden is that the implementation of web sessions on existing websites is complex and variegate. On real websites, session cookies are often issued *before login* and just refreshed after a successful password-based authentication, hence the presence of session cookies in an HTTP(S) request does not necessarily mean that the request is authenticated. Detecting which session cookies do not yet identify authenticated sub-sessions is useful to make Warden more permissive and efficient, because those cookies cannot be used for sub-session hijacking. Moreover, it is possible that some session cookies are only issued *after login*, rather than upon login. For instance, a website may grant access to its private area upon login, but only release the session cookies needed to access its mailbox service when users actually access their inbox. This means that, if an HTTP(S) request does not include the full set of session cookies, it is not necessarily a sub-session hijacking attempt, but rather it is possible that some session cookies have not been released yet by the website.

In the end, Warden needs to reliably keep track of which cookies are used to identify authenticated sub-sessions and which ones are not, even though it runs as a server-side proxy unaware of the session management logic. This is the subtlest point of its design, which requires some ingenuity.

5.3. Design Description

5.3.1. Cookie Scopes

To clarify how Warden works, we need to be a bit more precise about cookie scopes[4]. We represent scopes as triples $s = (prot, host, path)$, where:

$$
\begin{aligned}
prot &::= \mathsf{http} \mid \mathsf{https} \mid \mathsf{any} \\
host &::= \mathsf{str}_1.\cdots.\mathsf{str}_n \mid .\mathsf{str}_1.\cdots.\mathsf{str}_n \\
path &::= / \mid /\mathsf{str}_1/\cdots/\mathsf{str}_n/
\end{aligned}
$$

[4]The following definitions are based on RFC 6265 [1]. For the sake of clarity, they simplify a few uninspiring, complicated details of the real web, e.g., no specific restriction is enforced on the syntax of valid hostnames.

Here, each str_i stands for an arbitrary string not including dot and slash. Notice that s can represent any combination of the use of the Secure, Domain and Path attributes: for instance, a Secure cookie set by www.example.com for its root directory would have scope $(\mathsf{https}, \mathsf{www.example.com}, /)$, while a non-Secure variant of such cookie would have scope $(\mathsf{any}, \mathsf{www.example.com}, /)$. In real browsers, no combination of attributes can set a cookie scope to $(\mathsf{http}, host, path)$ for some $host$ and $path$, but it is technically convenient to introduce http in the syntax of scopes for later use.

Cookie scopes can be ordered by defining a binary relation $\sqsubseteq$ with the following meaning: if a cookie c_1 has scope s_1 and a cookie c_2 has scope s_2 with $s_1 \sqsubseteq s_2$, then every HTTP(S) request which includes c_1 must also include c_2.

Definition 1 (Order $\sqsubseteq$). *Let $s_1 = (prot_1, host_1, path_1)$ and $s_2 = (prot_2, host_2, path_2)$, we say that s_1 is* no larger than *s_2, written $s_1 \sqsubseteq s_2$ iff all the following clauses hold:*

- *$prot_1 = prot_2$ or $prot_2 = \mathsf{any}$;*
- *$host_1 = host_2$ or $host_2$ is a suffix of $host_1$ starting with a dot;*
- *$path_1 = path_2$ or $path_2$ is a prefix of $path_1$.*

Observe that the $\sqsubseteq$ relation is a *partial order*, i.e., it is reflexive, antisymmetric and transitive.

5.3.2. Configuration

The configuration of Warden requires the following parameters:

(1) the URL u_l where user credentials are submitted upon login, i.e., the action of the login form;
(2) the names $k_1, \ldots, k_n$ and scopes $s_1, \ldots, s_n$ of all the session cookies used by the website.

It is required that the names k_i are pairwise distinct. Observe that, when the names of the session cookies are known, finding their scope is trivial by using a standard web browser. Upon installation, Warden generates a fresh symmetric key K, whose role is described below.

5.3.3. Overview and Key Ideas

We provide a high-level description of how Warden works before formalizing the full details in the next sub-section. As explained in Section 4.2, sub-session hijacking can be prevented by ensuring that, every time an authenticated request is processed, then the browser sending that request stores the most updated *consistent* set of session cookies released to the authenticated user. Recall that a set of session cookies is consistent if and only if each cookie identifies a sub-session which contributes to implement the same user-authenticated session, so consistency can be proved by showing that all the session cookies were issued to the same user. To check consistency, Warden relies on standard cryptography: when a first set of session cookies is issued together upon login, they are bound together by an HMAC, which proves their consistency and must be presented along with the subsequent authenticated requests. Warden verifies the HMAC every time an authenticated request is received by the website: if the attacker steals or corrupts only a subset of the session cookies, the HMAC verification fails and Warden ensures that the request is not processed in an authenticated context by stripping the session cookies from it. When a legitimate set of session cookies needs to be extended, e.g., because the user accesses a different service on the same website, the HMAC is updated to include the new session cookies. Warden uses increasing sequence numbers to keep track of which is the most updated set of session cookies released to each user, so that the attacker cannot force the reuse of outdated HMACs which bind together only a subset of the session cookies implementing the current session. The security policy of Warden is implemented on top of cookies as well and deserves a careful explanation.

We use the notation (k, v) to denote a cookie named k having value v. When a login operation is performed, Warden generates a symmetric key KS to protect the new authenticated session by cryptographically enforcing intra-scope sub-session linking by means of an HMAC. The key KS is sent to the browser in a *key cookie* of the form $(wkey, enc(KS, K))$, where *enc* stands for symmetric key encryption. Warden stores for KS a set of bindings between session cookie scopes s_i and sequence numbers, initially set to 0. The sequence number for scope s_i is incremented every time KS is used to create a *linking cookie* for s_i. Intuitively, a linking cookie for s_i cryptographically proves that the set of session cookies with scope $s_j \sqsupseteq s_i$ which identify authenticated sub-sessions is the most updated consistent set of session cookies which are visible at s_i. A linking cookie for s_i has the form $(lnk_{s_i}, HMAC((s_i, N, \vec{v}), KS))$, where N is the sequence number bound to s_i for KS and $\vec{v}$ is the concatenation of the values of the session cookies with scope $s_j \sqsupseteq s_i$ which already identify authenticated sub-sessions. Observe that, in absence of scope fragmentation, only one linking cookie needs to be released: in this case, since the session cookies attached to each HTTP(S) request coincide with the full set of session cookies stored in the browser, the consistency proof granted by the linking cookie is sufficient to entirely stop sub-session hijacking. This is the intended use case, where Warden provides maximum benefits. Still, the design of Warden supports the presence of multiple scopes, because sub-session hijacking can still be mitigated by its security policy: we discuss examples of such cases in Section 6.1.3.

Warden ensures that requests are processed in an authenticated context only when they include a valid linking cookie. However, observe that when Warden receives a linking cookie, it does not know which session cookies should be considered to verify the HMAC stored therein. Warden cannot simply use all the session cookies attached to the request, because not all of them necessarily identify authenticated sub-sessions at that point in time. To solve this issue, Warden relies on *shadow cookies*, which identify the session cookies which are *not* yet used for authentication: all the other session cookies must be used in the HMAC computation. Given a session cookie (k_i, v_i) with scope s_i, its shadow cookie is a cookie $(k_i^\bullet, HMAC(v_i, K))$ with scope s_i which acts as a cryptographic proof that (k_i, v_i) does not yet identify an authenticated sub-session. Shadow cookies are initially set along with their corresponding session cookies before login, when there is no authenticated sub-session: when the login is performed, the shadow cookies whose name matches the name of any of the session cookies set in the login response are deleted and a new linking cookie binding the freshly released session cookies is created for each of their scopes. The linking cookies can then be updated throughout the session whenever new session cookies are issued in response to an HTTP(S) request which includes them. In fact, the fresh session cookies can be immediately identified as bound to authenticated sub-sessions, because the linking cookies prove that the login was already performed, hence no shadow cookie is released and the linking cookies can be directly updated to track the existence of new authenticated sub-sessions.

The process is summarized in Figure 4, which shows how the interaction between a browser B and a server S is mediated by Warden (noted by W). We use the signs + and - in the Set-Cookie headers to represent that cookies are set or deleted respectively. We assume that the session cookie *sid* is set before login for user tracking, but it gets refreshed upon login and returned along with a new session cookie *sec*. For readability, the figure omits the values of the cookies and uses the notation $lnk(sid, sec)$ to refer to the linking cookie binding together *sid* and *sec*. Every time a request is sent to the server, Warden blocks it and partitions its set of session cookies into two sets of *sensitive* and *insensitive* cookies respectively. A session cookie is considered insensitive if and only if it comes with its corresponding shadow cookie, which proves that the cookie does not yet identify an authenticated sub-session. All the other (sensitive) cookies must be bound together correctly by the HMAC stored in the linking cookie retrieved from the request. The HMAC is verified with the symmetric key KS stored in the key cookie attached to the

request, which can be retrieved via a successful decryption with the key K generated upon installation of Warden; the sequence number used to compute the HMAC is the one bound to KS for the scope of the linking cookie. If the verification of the HMAC succeeds, the request is forwarded to the server, otherwise all the sensitive cookies are stripped before forwarding to ensure the request is treated as unauthenticated.

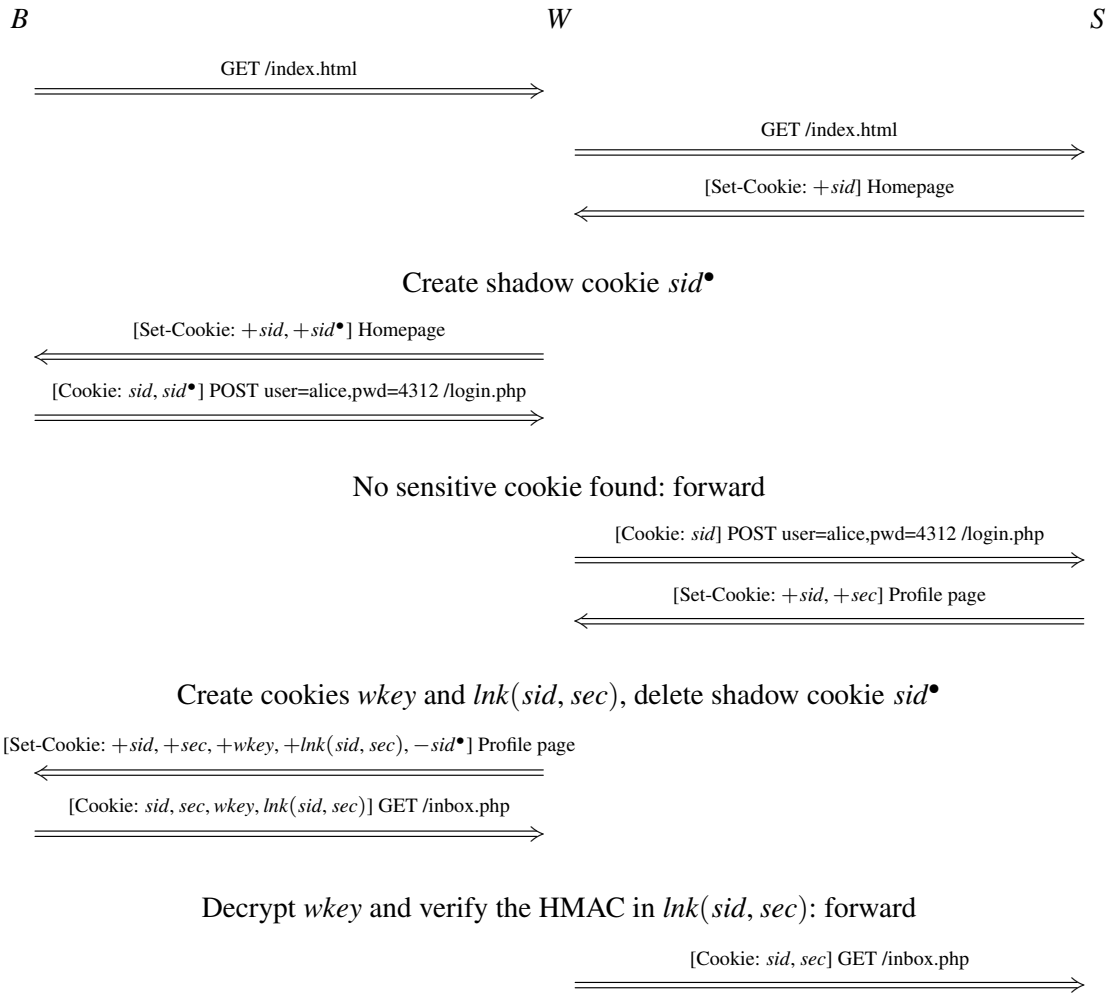

Figure 4. Warden exemplified (cookie values are elided for readability)

5.4. Formalization

We now formalize how Warden works so as to clarify the details of its security policy. We first define a convention to build the concatenation of session cookie values: let $(k_1, v_1), \ldots, (k_n, v_n)$ be a list of session cookies where all the k_i's are pairwise distinct, the concatenation of their values v_i's is built following the

lexicographic order of the k_i's. We then focus on defining the conditions used to check request *validity*, i.e., to deem when a request can be safely forwarded to the website with all its session cookies attached. Given an HTTP(S) request r, we let C_r be the list of the cookies attached to r and S_r be the list of the session cookies in C_r. We let $\overline{r}$ be the HTTP(S) response generated after processing r, $C_{\overline{r}}$ be the list of the cookies attached to $\overline{r}$ and $S_{\overline{r}}$ be the list of the session cookies in $C_{\overline{r}}$. In the following, we assume that C_r never contains two different cookies with the same name: if this does not hold, the request r is considered invalid and gets rejected by Warden. This condition is needed to ensure that Warden is not confused about which cookies must be used in the application of its security policy and, incidentally, it also thwarts potential cookie shadowing attempts by the attacker, thus providing better cookie integrity guarantees [5].

As anticipated in the previous overview, a session cookie is considered sensitive if and only if it does not come with a corresponding (valid) shadow cookie. A request is sensitive if and only if it contains at least one sensitive cookie: intuitively, insensitive requests are always safe with respect to sub-session hijacking, because none of their session cookies identifies an authenticated sub-session. Formally:

Definition 2 (Sensitive Cookie and Sensitive Request)**.** *Given a request r, a session cookie $(k_i, v_i) \in S_r$ is* sensitive *if and only if $(k_i^\bullet, HMAC(v_i, K)) \notin C_r$. We say that r is* sensitive *if and only if S_r contains at least one sensitive cookie.*

We now describe how a linking cookie is extracted from a request when multiple linking cookies are attached to it. In fact, observe that if there exist two distinct session cookie scopes s_i, s_j such that $s_i \sqsubseteq s_j$, there may exist legitimate requests including a linking cookie for both s_i and s_j (notice that these linking cookies have different names, because the scope is part of the name). The intuition is that we want to pick the linking cookie corresponding to the narrowest scope matching the request, because it binds together the largest number of sensitive session cookies: we call this cookie the *candidate* linking cookie.

To formalize this intuition, we need to introduce a few additional ingredients. Given an HTTP(S) request r directed at the URL u, we let $scope(r)$ stand for the scope matching exactly the protocol, the hostname and the path of u. We assume that Warden keeps track of the sequence numbers used for the construction of the linking cookies inside a data structure μ such that $\mu(KS, s_i) = N$ when N is the last sequence number bound to the scope s_i for the key KS.

Definition 3 (Candidate Linking Cookie)**.** *Given a request r such that $(wkey, enc(KS, K)) \in C_r$, its* candidate linking cookie *is the cookie $(lnk_{s_i}, v_i) \in C_r$ such that $scope(r) \sqsubseteq s_i$ and, for all the session cookie scopes $s_j \sqsupseteq scope(r)$ such that $\mu(KS, s_j) > 0$, we have $s_i \sqsubseteq s_j$.*

A candidate linking cookie is not guaranteed to exist, but if it exists, then it is unique by the anti-symmetry of $\sqsubseteq$. The side-condition on μ in the previous definition ensures that Warden picks a candidate linking cookie with scope s_i only if, for all the scopes s_j matching the request for which a linking cookie was previously issued, we have $s_i \sqsubseteq s_j$: this ensures that the candidate linking cookie always binds together the largest possible set of sensitive session cookies for the scope.

By using the candidate linking cookie, we can finally define request validity. Intuitively, a request is valid when it includes a candidate linking cookie with a valid HMAC. This is formalized as follows:

Definition 4 (Valid Request)**.** *A request r is* valid *iff, whenever $\vec{v}$ is the concatenation of the v_i's such that $(k_i, v_i) \in S_r$ is sensitive, there exist a key cookie $(wkey, enc(KS, K)) \in C_r$ and a candidate linking cookie $(lnk_s, v) \in C_r$ such that $v = HMAC((s, \mu(KS, s), \vec{v}), KS)$.*

Having introduced the main definitions needed to explain how Warden processes a request, we focus on the process used to create the linking cookies to be attached to a response. This is done by creating a linking cookie for each of the scopes of the session cookies included in the response, taking care that a linking cookie with scope s_i binds together all the session cookies with scope $s_j \sqsupseteq s_i$. The process is formalized by the following definition:

Definition 5 (Generated Linking Cookies)**.** *Let $S = (k_1, v_1), \ldots, (k_n, v_n)$ be a list of session cookies and let $s_1, \ldots, s_n$ be their scopes. The* generated linking cookies *for S using the key KS include, for each distinct cookie scope s_i, a cookie $(lnk_{s_i}, HMAC((s_i, \mu(KS, s_i), \vec{v}), KS))$, where $\vec{v}$ is the concatenation of the v_j's bound to a k_j such that $s_j \sqsupseteq s_i$.*

When a request r is sent to the website, Warden blocks it and checks its validity. If r is not valid, all the sensitive session cookies are stripped from it, so that the request is not processed in an authenticated context. Moreover, irrespective of the outcome of this check, all the cookies generated to implement the security policy of Warden are stripped from r, which is then forwarded to the website. Finally, Warden intercepts and extends the corresponding response $\overline{r}$ as follows:

(1) if r was directed at the login URL u_l, Warden creates a fresh symmetric key KS. For each session cookie scope s_i, Warden sets $\mu(KS, s_i) = 1$ if s_i is the scope of any of the cookies in $S_{\overline{r}}$, otherwise $\mu(KS, s_i) = 0$. The response $\overline{r}$ is extended to set the linking cookies generated for $S_{\overline{r}}$ using KS and to include the key cookie $(wkey, enc(KS, K))$. Moreover, $\overline{r}$ is extended so that all the shadow cookies whose name has the form $k_i^\bullet$ for some k_i such that $(k_i, v_i) \in S_{\overline{r}}$ are deleted;
(2) if r was not directed at the login URL u_l and it passed the validity check, Warden checks for the presence of new session cookies in $S_{\overline{r}}$. In that case, Warden extracts the key KS from the key cookie $(wkey, enc(KS, K))$ and, for each session cookie scope s_i, it increments $\mu(KS, s_i)$ by 1 iff s_i is the scope of any of the cookies in $S_{r^*}, S_{\overline{r}}$, where S_{r^*} is the sub-list of the sensitive cookies in S_r. The response $\overline{r}$ is extended to set the linking cookies generated for $S_{r^*}, S_{\overline{r}}$ using KS. Moreover, $\overline{r}$ is extended so that all the shadow cookies whose name has the form $k_i^\bullet$ for some k_i such that $(k_i, v_i) \in S_{\overline{r}}$ are deleted;
(3) if r was not directed at the login URL u_l and it did not pass the validity check, the response $\overline{r}$ is extended to set a shadow cookie $(k_i^\bullet, HMAC(v_i, K))$ for each $(k_i, v_i) \in S_{\overline{r}}$.

5.5. Implementation Details

We implemented Warden as an Apache module, so that it has direct access to both the HTTP and the HTTPS traffic towards and from the website, thus simplifying its deployment. Of course, this choice restricts the effectiveness of Warden to websites which are hosted on a single web server running Apache, but it would certainly be possible to implement Warden as an external proxy for production use at the cost of giving it access to the HTTPS certificate. As to cryptography, we used the AES-128 and HMAC implementations available in OpenSSL, choosing SHA-1 as the underlying hashing algorithm used in the HMAC construction.

We also implemented a Google Chrome extension which simulates the behavior of Warden entirely at the browser side, so that it is possible to test how well Warden fares on existing websites in terms of backward compatibility, i.e., to assess whether its security policy breaks the website functionality (Section 6.3). This browser extension is not intended to be a defense mechanism and is not part of Warden: it

is just used in the evaluation of this important aspect. Since the Google Chrome APIs for extension development provide facilities to intercept HTTP(S) requests and change the headers of incoming HTTP(S) responses, it allows for a very direct implementation of the behavior of Warden.

6. Warden Evaluation

We thoroughly evaluate the design and the implementation of Warden along four different axes: protection, performances, backward compatibility and ease of deployment.

6.1. Protection

6.1.1. Security Analysis

We consider a general threat model where the attacker is able to compromise the confidentiality and the integrity of the cookies stored in the user's browser. The lack of confidentiality or integrity of the session cookies is the key enabler of sub-session hijacking, because leaked session cookies can be used to put the attacker's browser into a user's sub-session, while tainted session cookies can force the user's browser into an attacker's sub-session. We abstract from how the leaking and the tainting of the cookies is performed: it can happen in the browser, e.g., due to the presence of XSS, or on the network, e.g., due to active tampering of the HTTP traffic.

Our security analysis proceeds in three steps:

(1) we show that all the authenticated requests forwarded to the protected website are *sensitive*, as long as the website refreshes the value of the session cookies when they are endorsed to authenticate the user upon login[5]. This ensures that stripping the sensitive cookies before forwarding a request is enough to prevent sub-session hijacking, hence the security policy applied by Warden to invalid requests is sound;
(2) we show that the set of sensitive session cookies which are attached to a valid request is always the *most updated consistent* set of session cookies issued to the authenticated user, as long as scope fragmentation is prevented;
(3) we use these results to prove that sub-session hijacking attacks are prevented by Warden under the above assumptions.

To understand why point (1) holds true, notice that cookies used for authentication purposes can only be released upon login or after processing an already authenticated request. However, shadow cookies are never released for fresh session cookies set upon login, which means that the first authenticated request to the protected website must be sensitive (see Definition 2). The corresponding response may refresh the session cookies in the request or set additional fresh cookies used for authentication, but no shadow cookie will be issued for them, because the request was sensitive, hence also the next authenticated request will be sensitive. By iterating this reasoning, we establish point (1).

Assume then that a valid request is forwarded to the protected website by Warden. This implies that the request includes a candidate linking cookie storing a valid HMAC which binds together all its sensitive cookies (see Definition 4). The absence of scope fragmentation ensures that this is the full set of session cookies which may be used for authentication, while the use of a sequence number in the HMAC ensures that this is the most updated set of such cookies. We then observe that the set of session cookies bound

[5]This is a recommended security practice, which is needed to prevent session fixation attacks [13].

by the linking cookie of a valid request must be consistent, because the first HMAC is issued for the session cookies set upon login, which are consistent. The response which sets the next HMAC, possibly with some additional session cookies, can only be produced after processing a valid request; this, in turn, implies that the request contained a consistent set of session cookies, so all the cookies used for authentication which are bound by the new HMAC must also be consistent. By iterating this reasoning, we establish point (2).

We can finally use this result to show that Warden prevents sub-session hijacking attacks under the two assumptions given above, thus establishing point (3). Assume that a request r is forwarded by Warden to the protected website. If r is invalid, Warden will strip all the sensitive cookies from it before forwarding, hence the resulting request will be unauthenticated by point (1) and it cannot be used for sub-session hijacking. If instead r is valid, we know by point (2) that the set of n sensitive session cookies which are attached to it is the most updated consistent set of session cookies issued to the authenticated user. This implies that an attacker who is able to disclose or corrupt only $m < n$ of these cookies will not be able to present a valid HMAC for them, either because no session was ever built over the m cookies or because that session is outdated (it includes less session cookies than the currently running session).

6.1.2. Red Teaming

To get further on-field evidence of the accuracy of our security analysis and the degree of protection offered by Warden, we set up a challenge with a red team of three web security experts who regularly attend international hacking competitions. The challenge was based on a toy web application developed in PHP, called Disney Playground (DP). DP has just two registered users, Mickey Mouse and Donald Duck respectively, who are authenticated by means of the following three session cookies:

- the *identity* cookie is used to retrieve the profile picture of the user, representing the corresponding Disney character;
- the *city* cookie is used to fetch a picture of the city where the user lives (Mouseton or Duckburg);
- the *partner* cookie is used to get a picture of the partner of the user (Minnie or Daisy Duck).

The first two session cookies are set before login and refreshed upon authentication, while the last one is set later on when browsing the private area of the web application: if the *partner* cookie is still missing, DP displays a question mark in place of the partner. All the session cookies are set without any security attribute and have the same scope; they store an unpredictable session identifier, which is used to index an internal data structure storing the images. In this setting, sub-session hijacking is successfully performed when DP displays a combination of images which mixes elements of two different comics.

We deployed Warden on top of DP to prevent sub-session hijacking. We configured DP so that all the communication happened over HTTP via a network controlled by the red team, thus modeling a scenario where there is no cookie confidentiality, no cookie integrity and the network traffic cannot be trusted. The red team was also instructed about the inner workings of Warden and given access to the credentials of the two users of DP. Finally, the red team was asked to perform a sub-session hijacking attack by reproducing one of the attack configurations in a web browser (of course, without modifying the response bodies to directly force these configurations). Despite the best efforts of the red team, all the sub-session hijacking attempts were prevented by Warden. Examples of attacks which were thwarted by Warden based on the server logs include the reuse of outdated HMACs, the forgery of (invalid) shadow cookies and a number of tampering attempts against the session cookies.

6.1.3. Scope Fragmentation

Both our security analysis and our on-field evaluation assumed the absence of scope fragmentation. If scope fragmentation is not prevented, sub-session hijacking may still be possible despite the security

policy applied by Warden, because Warden would not always be able to see the full set of session cookies stored in the browser sending an authenticated request. An example of such sub-session hijacking attack is shown in Figure 3.

Warden was not designed to automatically prevent scope fragmentation, because this would be very hard to do without affecting the website functionality (see Section 4.4) and we wanted to preserve backward compatibility with existing websites. Ideally, we expect web developers to manually prevent scope fragmentation by following the recipes we detailed and then close the remaining space for sub-session hijacking by installing Warden. It is worth recalling that scope fragmentation does not seem pervasive in the wild: Section 4.5 shows that out of 92 websites which are potentially vulnerable to sub-session hijacking, only 18 websites (19.6%) present scope fragmentation, hence the other 74 websites (80.4%) could entirely prevent sub-session hijacking just by installing Warden. Moreover, we empirically observe that Warden can still prevent a number of sub-session hijacking attacks even when scope fragmentation is present, because intra-scope sub-session linking is enforced anyway. We discuss next two examples of real-world attacks which can be prevented by Warden on existing websites which do not prevent scope fragmentation.

Gmail Chat Gadget Hijacking [5]. The web interface of Gmail includes a chat gadget on its left side. If an attacker hijacks the sub-session of the chat gadget without affecting the main content of the inbox, he can fake the victim's friend list and chat with her to initiate advanced phishing attempts, while significantly reducing the visibility of the attack. In particular, the attacker can inject his Google session cookies so that they shadow the victim's ones only at the chat-related URLs by using the Path attribute: this would put the attacker's chat gadget on the victim's screen without disturbing the victim.

This attack is prevented by Warden because only valid requests are forwarded to the protected website and no request containing multiple session cookies with the same name is valid. Indeed, one of the main benefits of the security policy applied by Warden is exactly the prevention of cookie shadowing attempts, which are not straightforward to detect at the web application layer, because web development frameworks typically provide access to cookies in the form of a key-value dictionary.

Purchase Hijacking on Amazon [5]. Amazon uses two long-term cookies to keep track of a user's browsing history and ongoing purchases. Unfortunately, the corresponding state information is not bound to the user's session cookie, but rather stored in a session-independent data structure. If an attacker is able to replace these two cookies with his own cookies, he can track in real-time what the victim has viewed on Amazon and inject unwanted items into the browsing history, thus affecting the Amazon recommender system.

This attack is prevented by Warden if the two cookies above are included in the list of session cookies to protect. The reason is that the user's session and the attacker's session would be built on top of different values of such cookies, hence replacing the user's cookies with the attacker's cookies would break the check of the HMAC stored in the linking cookie used by Warden.

6.1.4. Discussion

The security analysis of Warden is precise, but informal. It establishes that sub-session hijacking attacks due to the violation of cookie confidentiality or integrity are prevented, provided that session cookies are refreshed when they are endorsed for authentication and scope fragmentation is prevented. As future work, we would like to recast our analysis in a formal model and generalize it to provide better and more accurate security guarantees. It would be intriguing to identify conditions under which scope fragmentation does not enable sub-session hijacking attacks, although only intra-scope sub-session linking is enforced. The main obstacle to this kind of analysis is that, unfortunately, browser models proposed in

the literature are still somewhat underwhelming, being either too abstract or too detailed to support the development of security proofs which are both realistic and manageable in terms of complexity [21].

An interesting pragmatic observation is that the HMAC computation performed by Warden can be useful beyond the prevention of sub-session hijacking. Since HMACs provide integrity proofs, they can be used to ensure that some session cookies are never modified at the browser side. This can be useful in particular when these cookies cannot be marked with the HttpOnly attribute, because they need to be read by JavaScript for generic reasons.

6.2. Performances

6.2.1. Network Latency

The security policy implemented by Warden may have a negative impact of the perceived performances of the protected website. To assess this important aspect, we used Tsung[6], a state-of-the-art tool for the performance analysis of client/server applications. Tsung allows one to simulate the arrival of new clients at the server in terms of a Poisson process, with a configurable arrival rate λ. Clients are put into a queue and served under a FIFO policy until they complete their navigation. We analyzed the impact of Warden on network latency on a fresh Wordpress installation by measuring the mean response times and the number of pages served in presence and in absence of Warden. We configured Tsung so that each client accesses the Wordpress homepage, performs a login operation and visits a number of pages of the backend before moving back to the frontend. We tested five different arrival rates $\lambda \in \{1, 2, 3, 4, 5\}$ for clients, performing 10 runs of 20 minutes for each of them and taking the average of the results.

Table 2 shows the mean response times for page requests, as well as the number of pages served. The latency introduced by Warden is less than 10 ms in the worst case, which is negligible in practice for end users. We found that at $\lambda = 6$ our machine was not able to deal with the generated workload, both with and without Warden, which shows that Warden was not the bottleneck when serving page requests.

Arrival Rate λ $(1/s)$	Basic (ms)	Warden (ms)	Pages served
1	41.42	42.79	$\sim$ 6,000
2	42.51	43.65	$\sim$ 11,900
3	45.00	46.58	$\sim$ 18,100
4	46.67	51.96	$\sim$ 23,900
5	52.17	60.78	$\sim$ 29,800

Table 2
Mean response time for page requests

We think that the results of our performance evaluation are positive and encouraging for a practical adoption of Warden, though we also acknowledge that major real-world websites may have tight constraints on the acceptable network delays. Making a more realistic evaluation of the latency overhead introduced by Warden on such commercial services would be impossible without having access to specific information about their configuration, given that major websites are hosted on multiple machines served through a load balancer. However, we do not see any significant problem at deploying Warden at individual edge servers to support this kind of architectures.

[6] http://tsung.erlang-projects.org/

6.2.2. Traffic Overhead

In the worst case, Warden generates one linking cookie for each identified session cookie scope, one shadow cookie for each session cookie, and one key cookie. To understand the impact of this on real-world websites, we inspected the publicly available cookie dataset on which we based the analysis of Section 3.3. Websites in the dataset build their sessions on top of at most 6 session cookies spanning 3 different scopes, which means that Warden would need to generate at most 10 cookies to protect them. This is a worst case analysis, because the large majority of the websites (using multiple session cookies) in the dataset issue only 2 session cookies spanning at most 2 scopes, which means that only 5 cookies would suffice to protect them. Also, notice that these cookies will not actually be all present at the same time: if a session cookie is included in a linking cookie, no shadow cookie will be available for it anymore. Observe that non-session cookies do not affect the performance of Warden, because they do not play any role in the creation of linking cookies and shadow cookies.

Each linking cookie and shadow cookie generated by Warden contains an HMAC of 20 bytes, while key cookies contain the encryption of a 128 bit key, which requires 32 bytes. The name of each of these cookies requires 16 bytes in our implementation. This means that Warden may introduce a overhead of at most 372 bytes on the HTTP(S) requests to the websites in our dataset. The same analysis on the most common case discussed above yields a overhead of 192 bytes. These numbers are already acceptable for a practical adoption, especially because they are pessimistic, since not all the cookies generated by Warden are present at the same time and linking cookies may have different scopes, so not all the requests to the website need to attach all the linking cookies. In our fresh installation of Wordpress we observed an average overhead of just 57 bytes per HTTP(S) request, which is definitely a small price to pay. We also plan to improve our implementation of Warden to make cookie names shorter, which would make this overhead even lower.

6.3. Backward Compatibility

6.3.1. Session Cookies and JavaScript

The main potential compatibility issue coming from the design of Warden derives from the assumption that the content of session cookies is never modified at the browser side: if this is not the case, the check of the HMAC performed by Warden would fail even in absence of sub-session hijacking attacks. We believe that this assumption is reasonable for most websites, because session cookies are used to authenticate users at the server side and accessing them at the browser side is generally a bad security practice; indeed, these cookies are often protected using the HttpOnly attribute, which prevents JavaScript accesses altogether.

We performed a small-scale experiment to understand whether Warden breaks the functionality of existing websites because session cookies need to be changed via JavaScript. In particular, we selected 30 popular websites where we own personal accounts and we navigated them as deep as possible, while using the Google Chrome extension which simulates the behavior of Warden at the browser. We identified the set of the session cookies for the tested services using a previously proposed algorithm [7]. We observed that 29 out of 30 websites worked flawlessly, including Alibaba, Amazon, Facebook, Ikea and Reddit. We only encountered compatibility problems with Ebay, where a session cookie is modified at the browser side to include a random suffix in its value. The reason why this is done is obscure to us, because deleting the suffix does not seem to affect the website functionality and is enough to make Warden work correctly. Nevertheless, we were able to recover backward compatibility in this case by excluding the troublesome session cookie from the list of cookies protected by Warden, which is a general way to deal with such kind of compatibility problems at the cost of having reduced security guarantees.

6.3.2. Session Handling Operations

Since Warden operates as a server-side proxy oblivious to the session management logic, one may wonder how well it deals with common session handling operations. We discuss here two notable cases: failed login attempts and logouts.

If a user inserts wrong credentials in the login form, there are two possible implementation practices at the server side. The first possibility is that the website does not issue the expected session cookies: if this is the case, Warden will do nothing and no compatibility issue may arise. Alternatively, the website may still send the session cookies without associating them to a valid session at the server side. In this case, Warden will create its fresh cookies and use them to apply its security policy, although no session was actually established. This is useless from a security perspective, yet it will not affect the website functionality. Moreover, the user can still provide the correct credentials after the failed login attempt to get her session cookies, which would be protected by Warden.

In the case of logouts, we still distinguish two cases depending on the server-side implementation. If the logout is implemented by making some of the session cookies in the browser expire, all the subsequent requests sent by the browser will be invalid and their remaining session cookies will be stripped away by Warden. Since no session is established anymore, we think that the only functionality which may be broken by this behavior is user tracking: for instance, social network websites may leave some session cookies unexpired to track the navigation history of users [22]. These cases may be supported by not marking the tracking cookies as session cookies, though a careful scrutiny of the session management logic is recommended to ensure this does not leave space for sub-session hijacking. If instead the logout is implemented by making the session expire at the server side, Warden may still receive valid requests and apply its security policy, which is useless for security yet harmless for compatibility.

Besides these considerations, we empirically tested both failed login attempts and logouts on all the 30 websites selected to test compatibility: none of them broke due to these session management operations.

6.4. Ease of Deployment

Warden requires only limited configuration efforts by web developers. The only parameters which must be provided upon its configuration are the URL used to submit the user credentials for login, the names of the session cookies issued by the website and their corresponding scopes. Finding all the session cookies to protect can be relatively complex, because it might require a good understanding of the session management logic, but we pragmatically observed that finding sets of session cookies to protect was not too hard for us, even without any access to the source code of the services we tested. In particular, we found useful the automated techniques described in previous work [7, 8] to reliably identify the set of session cookies used by a website, which may be used to identify a candidate set of cookies to protect.

Although implemented as an Apache module for programming convenience, Warden is designed to act as a server-side proxy. This means that it can enforce protection independently from the language used to implement the server-side logic and without requiring code changes. This is a major advantage of its design, given the heterogeneity and the complexity of the web. Given all these considerations and our own experience at testing Warden on 30 existing websites (see Section 6.3), we conclude that Warden is relatively easy to deploy in practice. Since Warden is publicly available, we will collect feedback from other web developers to get further evidence about this claim.

7. Related Work

Sub-session hijacking attacks were first reported in two research papers [5, 9]. The first paper studied the practical implications of the lack of cookie integrity on popular services, while the second one analyzed the risks of session management in mixed content websites. Both papers present empirical studies which highlight the dangers of sub-session hijacking in the wild, but they do not discuss the problem of sub-session hijacking more in general, nor propose robust defense mechanisms against it.

The design of Warden bears similarities to some defenses against session hijacking proposed in the past. SessionLock [23] is a protocol which exploits a secret established over HTTPS to protect an HTTP session against hijacking. The main idea is to use the shared secret to compute an HMAC of the HTTP requests, so that an attacker cannot produce valid HTTP requests on behalf of another user. One-time cookies [24] generate a unique token per request based on a session key; each token is tied to a particular request via an HMAC to prevent its reuse in session hijacking attempts. GlassTube [25] guarantees the integrity of HTTP traffic at application level by computing an HMAC of both the HTTP requests and the HTTP responses; session hijacking is detected by a mismatch between the presented session cookies and the key used to authenticate the request including them. There are two notable differences between the papers in this research line and the present one. From the theoretical side, none of the previous papers specifically discusses sub-session hijacking, whose dangers only emerged recently. From the practical side, all the proposed solutions are much harder to deploy than Warden.

SessionLock requires the registration of appropriate event handlers for all the links and forms in the protected pages, which is particularly complex to do for existing web applications. One-time cookies are even more complex to deploy, because they require changes to both server-side code and web browsers. GlassTube requires to embed all the resources not loaded by JavaScript via data URIs, which requires a major code refactoring at the server side; this limitation can be overcome by deploying GlassTube as a browser extension, but this would make its adoption much more complicated, because all the website users would need to install the extension. A limitation of Warden with respect to these solutions is that only protection against sub-session hijacking is provided: modifications of the data stream aimed at corrupting, e.g., HTTP request parameters, are not detected. Overcoming this limitation requires browsers to produce HMACs (or other proofs of integrity) for each outgoing request, which is difficult to do without breaking compatibility or changing browsers.

Origin Bound Certificates [26] are a TLS extension designed to bind session cookies to HTTPS channels. The main idea is to generate in the browser a different certificate for each website upon an HTTPS connection, which is then used to cryptographically bind session cookies to the encrypted channel established between the browser and the website. This way, leaked session cookies cannot be reused by an attacker to authenticate as the victim at the website, because the victim's certificate is never disclosed to him. Origin Bound Certificates are an effective solution against sub-session hijacking, but they are not available in standard web browsers.

Other solutions against session hijacking improve the confidentiality guarantees of session cookies. SessionShield [15], Zan [16] and CookiExt [6] all modify browsers to automatically detect session cookies and better protect them. The effectiveness of these browser-side solutions is limited by the accuracy of the heuristic used to detect the session cookies; moreover, the integrity problems of session cookies are not addressed, which leaves space for sub-session hijacking. Origin cookies [11] improve the integrity of cookies against network attackers and web attackers controlling related domains, which is useful to prevent sub-session hijacking, but unfortunately they are not available in standard web browsers. Ses-

sion hijacking can also be prevented by holistic solutions for web session security based on browser-side monitoring and enforcement [27, 28].

Secure cookie schemes like those proposed in previous research papers [29, 30] store HMACs inside cookies, so they share some similarities with what Warden does. The goal of these schemes, however, is ensuring protection against guessing attacks on session cookie values and guarantee the integrity of browser-side state.

8. Conclusion

We performed a systematic analysis of the root causes of sub-session hijacking attacks and we explored the use of sub-session linking as an effective technique to protect web sessions against them. We showed that sub-session linking is not trivial to implement: in particular, inter-scope sub-session linking is complex and it better be avoided by using programming practices which prevent scope fragmentation. We then introduced Warden, a server-side proxy which automatically enforces intra-scope sub-session linking and requires only minimal configuration efforts by web developers. We performed a thorough evaluation of Warden and we experimentally showed that only a small price has to be paid to enjoy the additional protection offered by the tool.

There are several avenues for future work. First, we would like to study best programming practices which can be applied at the server side to prevent sub-session hijacking, even in presence of scope fragmentation. Moreover, we plan to provide a formal characterization of sub-session hijacking attacks and establish under which conditions Warden enforces a semantic security property which rules them out. Finally, we would like to carry out a large-scale analysis of the vulnerability of existing websites to sub-session hijacking attacks, to improve the scope of existing studies [5, 9] and understand this security issue more in depth.

References

[1] A. Barth, HTTP State Management Mechanism, 2011.
[2] S. Calzavara, R. Focardi, M. Squarcina and M. Tempesta, Surviving the web: a journey into web session security, *ACM Computing Surveys* (2017).
[3] J. Hodges, C. Jackson and A. Barth, HTTP Strict Transport Security (HSTS), 2012.
[4] M. Kranch and J. Bonneau, 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.
[5] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan and N. Weaver, Cookies Lack Integrity: Real-World Implications, in: *Proceedings of the 24th USENIX Security Symposium, USENIX 2015*, 2015, pp. 707–721.
[6] M. Bugliesi, S. Calzavara, R. Focardi and W. Khan, CookiExt: Patching the Browser Against Session Hijacking Attacks, *Journal of Computer Security* **23**(4) (2015), 509–537.
[7] S. Calzavara, G. Tolomei, A. Casini, M. Bugliesi and S. Orlando, A Supervised Learning Approach to Protect Client Authentication on the Web, *TWEB* **9**(3) (2015), 15–11530.
[8] Y. Mundada, N. Feamster and B. Krishnamurthy, 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*, 2016, pp. 675–685.
[9] S. Sivakorn, I. Polakis and A.D. Keromytis, 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*, 2016, pp. 724–742.
[10] OWASP, Top 10 Security Threats, 2017.
[11] A. Bortz, A. Barth and A. Czeskis, Origin Cookies: Session Integrity for Web Applications, in: *Web 2.0 Security & Privacy Workshop (W2SP 2011)*, 2011.

[12] S. Calzavara, A. Rabitti and M. Bugliesi, Dr Cookie and Mr Token - Web Session Implementations and How to Live with Them, in: *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, February 6th - to - 9th, 2018.*, 2018.
[13] M. Johns, B. Braun, M. Schrank and J. Posegga, Reliable Protection Against Session Fixation Attacks, in: *Proceedings of the 26th ACM Symposium on Applied Computing, SAC 2011*, 2011, pp. 1531–1537.
[14] A. Barth, C. Jackson and J.C. Mitchell, Robust Defenses for Cross-Site Request Forgery, in: *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*, 2008, pp. 75–88.
[15] N. Nikiforakis, W. Meert, Y. Younan, M. Johns and W. Joosen, SessionShield: Lightweight Protection against Session Hijacking, in: *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011*, 2011, pp. 87–100.
[16] S. Tang, N. Dautenhahn and S.T. King, Fortifying web-based applications automatically, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*, 2011, pp. 615–626.
[17] A. Cahn, S. Alfeld, P. Barford and S. Muthukrishnan, An Empirical Study of Web Cookies, in: *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, 2016, pp. 891–901.
[18] S. Sivakorn, A.D. Keromytis and J. Polakis, That's the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms, in: *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2016, Vienna, Austria, October 24 - 28, 2016*, 2016, pp. 71–81.
[19] Chromium security team, Marking HTTP As Non-Secure, 2016.
[20] K. Singh, A. Moshchuk, H.J. Wang and W. Lee, On the Incoherencies in Web Browser Access Control Policies, in: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 463–478.
[21] M. Bugliesi, S. Calzavara and R. Focardi, Formal methods for web security, *Journal of Logical and Algebraic Methods in Programming* (2017).
[22] F. Roesner, T. Kohno and D. Wetherall, Detecting and Defending Against Third-Party Tracking on the Web, in: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 155–168.
[23] B. Adida, Sessionlock: securing web sessions against eavesdropping, in: *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, 2008, pp. 517–524.
[24] I. Dacosta, S. Chakradeo, M. Ahamad and P. Traynor, One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens, *ACM Transactions on Internet Technology* **12**(1) (2012), 1–24.
[25] P.A. Hallgren, D.T. Mauritzson and A. Sabelfeld, GlassTube: A Lightweight Approach to Web Application Integrity, in: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013*, 2013, pp. 71–82.
[26] M. Dietz, A. Czeskis, D. Balfanz and D.S. Wallach, Origin-Bound Certificates: a Fresh Approach to Strong Client Authentication for the Web, in: *Proceedings of the 21th USENIX Security Symposium, USENIX 2012*, 2012, pp. 317–331.
[27] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan and M. Tempesta, Provably Sound Browser-Based Enforcement of Web Session Integrity, in: *Proceedings of the IEEE 27th Computer Security Foundations Symposium, CSF 2014*, 2014, pp. 366–380.
[28] S. Calzavara, R. Focardi, N. Grimm and M. Maffei, Micro-policies for Web Session Security, in: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*.
[29] K. Fu, E. Sit, K. Smith and N. Feamster, The Dos and Don'ts of Client Authentication on the Web, in: *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, 2001.
[30] A.X. Liu, J.M. Kovacs and M.G. Gouda, A secure cookie scheme, *Computer Networks* **56**(6) (2012), 1723–1730.