Improving the Security of Session Management in Web Applications

Philippe De Ryck, Lieven Desmet, Frank Piessens, Wouter Joosen

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
Email: {firstname.lastname}@cs.kuleuven.be

Abstract. Session management is a critical component of modern web applications, allowing a server to keep track of user-specific state, such as an authentication status. Unfortunately, many applications deploy session management over an insecure HTTP channel, making them vulnerable to eavesdropping, session hijacking or session fixation attacks. On the contrary, state-of-practice guidelines advocate the deployment of session management on a secure HTTPS channel, using the HttpOnly and Secure cookie attributes, effectively eliminating these well-known session management attacks. The goal of this paper is to provide secure session management to web applications deployed over HTTP.

We propose a secure and lightweight session management mechanism, effectively improving session management security with HTTP deployments. By establishing a safely contained, shared secret between browser and server, an attacker is prevented from taking over a user's session, since the secret is never transmitted, nor accessible. We demonstrate the applicability of our solution to a common scenario involving third-party authentication, clearly indicating the gained security properties.

Our secure and lightweight session management mechanism raises the security bar for HTTP deployments, which will eventually lead to secure session management for all web applications.

1 Introduction

Session management is a critical component of modern web applications, since it allows the server to track user-specific state, such as the authenticated account, across multiple requests. Session management is an attractive, high-value target for attackers, as indicated by its third place on the OWASP top 10 of web application security risks [17]. Common attacks on session management include session hijacking [13], session fixation [16] or network-based eavesdropping attacks. Recently published work [5] identified 10 of the Alexa top 100 sites to be vulnerable to session hijacking. With the rise of publicly available networks and hotspots, stealing authenticated sessions can be reduced to a point-and-click operation, as demonstrated by tools like Firesheep [4].

To prevent such nefarious attacks on session management, current state-of-practice security guidilines advocate the use of TLS, in combination with the

Secure and *HttpOnly* cookie attributes. TLS protects the communication channel against eavesdropping or network manipulation attacks, and the cookie attributes prevent session attacks from malicious code within the browser. While these security guidelines offer adequate protection against attacks on session management, and hence should be followed, the deployment of TLS on the web remains rather limited: A 2010 study by Qualys shows that out of 119 million domains listening on port 80 or 443, only 0.72 million present a TLS certificate with the corresponding domain name [15], of which a large percentage (approx. 25%) does not even validate correctly. The culprit for the slow adoption rate has never been acurately determined, but literature covers several potential advantages and disadvantages [1, 5, 6, 12].

Since both non-TLS applications (HTTP deployments) and TLS applications (HTTPS deployments) are likely to co-exist for the forseeable future, improving the security of session management in HTTP deployments is highly advised. In this paper, we propose a secure and lightweight session management mechanism, effectively eliminating session hijacking, session fixation or network eavesdropping attacks in web applications deployed over HTTP. Instead of solely relying on the presence of a session identifier, our session management mechanism establishes a shared session secret between browser and server, and safely contains it within the browser. The shared secret is not accessible to web pages, nor is it ever transmitted over the network. Concretely, our proposal for secure session management offers (a) Protection against script- or DOM-based session hijacking or session fixation attacks, (b) Protection against network-based eavesdropping attacks, and (c) Transparent deployment to both new and legacy web applications by containment in underlying infrastructure.

We demonstrate the applicability of our session management mechanism with a common scenario involving non-TLS applications and third-party service providers. One instantiation of this scenario is an application using a third-party authentication service, such as Google, Facebook or Twitter. Since the authentication provider typically implements adequate security measures for the authentication process, relieving the actual applications of the need for additional security, such as a TLS-secured connection, leaving them vulnerable to the aforementioned attacks on their traditional session management mechanism. Our lightweight and secure session management mechanism does not suffer this problem, and is ideally suited for these kind of scenarios. Naturally, applications that deal with sensitive information or require strong security guarantees should strongly consider switching to TLS.

In the remainder of this paper, we discuss the current state-of-practice and related work (Section 2), followed by the relevant threat model (Section 3). Section 4 explains our proposal and Section 5 discusses a concrete deployment scenario, as well as infrastructural needs. Finally, we discuss secure session management in the context of the web (Section 6) and conclude the paper (Section 7).

2 Background and Related Work

This section covers the current state-of-practice in session management, followed by a discussion of related work that improves upon traditional session management techniques.

2.1 State-of-Practice Session Management

The current de facto session management mechanism uses cookies to store a session identifier, which is directly coupled to the stored server-side state. The server-side state is typically used to store information about the session, such as the user's identity and authentication status. The server is responsible for issuing the session identifier using a *Set-Cookie* header, and the browser attaches the session identifier to every request going to the target domain, using the *Cookie* header. In this scenario, the session identifier is a *bearer token* [6], meaning that whoever presents this token to the server is granted access to the server-side state.

Attacks on session management mechanisms abuse the flexibility of the session identifier as a bearer token. An eavesdropper listening in on network traffic can steal the session identifier from any request sent over an unsecured connection, as aptly demonstrated by the Firesheep [4] browser addon, which simplifies this to a single point and click operation. Additionally, an adversary that can execute scripts in the context of the target site, for example by including a malicious advertisement, can use the `document.cookie` property to read and write cookies, hence read or manipulate the session identifiers stored herein. Reading the session identifier can lead to session hijacking, and writing the session identifier to session fixation, both attacks that give an attacker full control over a user's authenticated session.

Current security guidelines regarding session management all include advice to defend against these well-known attacks. Attacks based on manipulating the session cookie through the `document.cookie` property can be thwarted by adding the *HttpOnly* flag to session cookies, making them inaccessible to JavaScript. Until recently, the adoption of the *HttpOnly* flag was rather limited [13], but nowadays several frameworks start to add *HttpOnly* to their session cookies out of the box [2, 18].

To eliminate eavesdropping attacks, all communication must be conducted over a secure channel. Web applications typically use HTTPS, which is HTTP over a TLS-secured connection. However, special attention needs to be paid to cookies, since they are shared across secure and insecure connections. Attaching the *Secure* flag to the cookie containing the session identifier guarantees its secrecy, since the browser will never send it on an insecure connection.

These security measures offer adequate protection against most common attacks on session management, but also increase the complexity of the web application, as well as the burden on the web developer. Deliberate or unintentional failure to meet any of these security guidelines immediately enables several attacks on the session management mechanism of a web application.

2.2 Related Work

Securing session management is an active research topic, which has resulted in a few different approaches to address the problem. Here, we present the four most relevant academic papers. One thing these proposals have in common is that at some point, they rely on TLS to exchange credentials or establish a shared secret. Additionally, some solutions require extensive changes to legacy applications before they can be deployed, or integrate tightly with the authentication process. These restrictions make them incompatible with several common web scenarios, such as supporting legacy applications or third-party authentication services. Nonetheless, these solutions all have their merits and provide valuable insights into secure session management.

SessionLock SessionLock [1] augments requests with an HMAC based on a shared session secret. The session secret is established over a TLS channel and stored in a secure cookie. For HTTP pages, it is stored in the fragment identifier, a part of the URL that is never sent over the network. SessionLock also supports setup over a non-TLS channel using Diffie-Hellman. At the client-side, SessionLock is implemented using a JavaScript library, making it vulnerable to injection attacks. Additionally, SessionLock requires all requests within the application to be made from JavaScript using AJAX, making it incompatible with most legacy applications.

BetterAuth BetterAuth [11] is an authentication protocol for web applications, offering protection against several attacks, including network attacks, phishing and cross-site request forgery. BetterAuth considers a user's password to be a shared secret, and uses that shared secret to agree on a session secret over an insecure channel. The session secret is subsequently used to sign requests, offering authenticity. The strength of BetterAuth is that it protects against active network attackers. A disadvantage is that the password needs to be shared with the server, requiring an initial setup phase over TLS. Additionally, because BetterAuth depends on the password, it is incompatible with current third-party authentication services.

One-Time Cookies One-Time Cookies [5] proposes to replace the static session identifier with disposable tokens per request, similar to the concept of Kerberos service tickets. Each token can only be used once, but using an initially shared secret, every token can be seperately verified and tied to an existing session. To share the initial credential, One-Time Cookies depend on the use of TLS during the authentication phase.

TLS Origin-Bound Certificates Origin-Bound Certificates (OBC) [6] is an extension for TLS, that establishes a strong authentication channel between browser and server, without falling prey to active network attacks. Within this secure channel, TLS-OBC supports the binding of cookies and third-party authentication tokens, which prevents the stealing of such bearer tokens. While TLS-OBC offers strong security guarantees, it obviously depends on TLS, making it inapplicable to web applications that do not deploy TLS.

HTTP Integrity Header The HTTP Integrity Header [7] is a draft proposing to add integrity protection to HTTP. The header depends on a key exchange, either over TLS or with a traditional Diffie-Hellman exchange, after which the integrity of the selected parts of a message is protected. A downside of the Integrity header is the key exchange, which happens either over TLS, or is only completed after the third message, leaving earlier messages unprotected.

3 Threat Model

We consider two threat models to be relevant for this paper: the forum poster and the passive network attacker. The active network attacker also poses a real threat, but we consider this model to be out of scope for this work. We cover the threat model here for completeness, and argument our decision in the discussion section (Section 6).

Forum Poster Barth et al. [3] define a forum poster as an attacker that can submit content to a dynamic website, such as a forum or a social networking site. In their case, the forum poster submits images or hyperlinks, triggering a potentially malicious request being sent from the browser.

We also consider the forum poster to be a relevant attacker model, albeit in the presence of a cross-site scripting (XSS) vulnerability in the dynamic site. An XSS vulnerability allows the forum poster to execute arbitrary JavaScript in the context of the target site within the victim's browser. Cross-site scripting is a well-known and often-exploited attack vector to conduct session hijacking [13] or session fixation [16] attacks, using the `document.cookie` property to read or manipulate session cookies in the victim's browser.

Passive Network Attacker The passive attacker, as defined by Jackson et al. [10], is capable of sniffing live network traffic, thus able to read any information sent on an unencrypted channel. Since the cookie-based session identifier is sent on every request, a passive network attacker can easily steal a session by stealing this token. With the widespread deployment of wireless hotspots and wifi-enabled devices such as smartphones, these kind of attacks are as prevalent as ever [4].

Additionally, we consider a passive network attacker to possess a second, high-grade network connection. This allows him, while eavesdropping a slower network, to craft malicious requests based on eavesdropped requests, and have them reach the target server first. A concrete example of this scenario is an attacker eavesdropping on a slow wireless network, stealing a session identifier, attaching it to his crafted request and sending it out over a fast fiber connection.

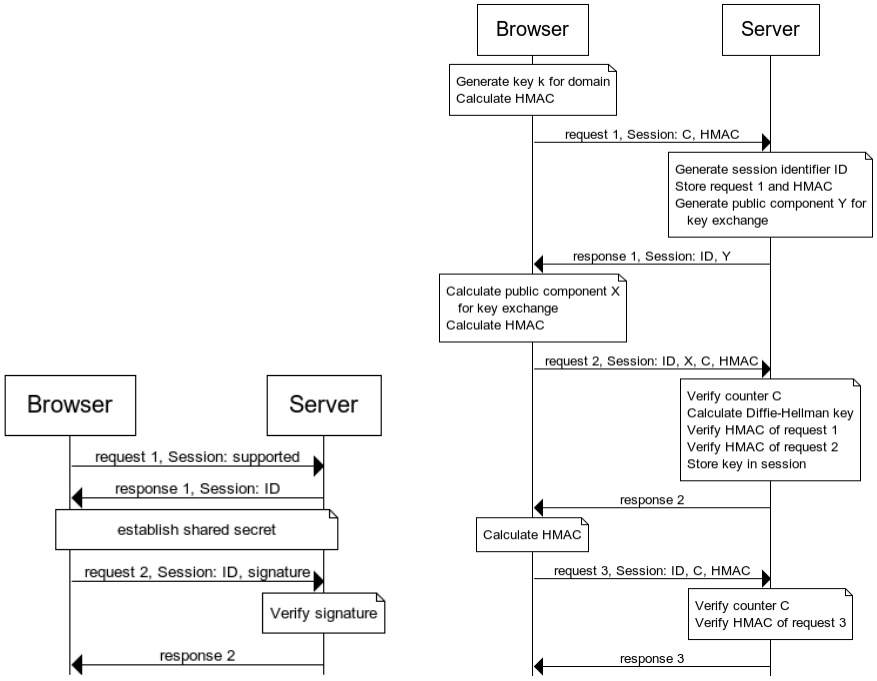
Active Network Attacker The active network attacker is commonly defined as an attacker impersonating a DNS server or controlling the user's network [10]. An active network attacker therefore has full-blown man-in-the-middle capabilities. In Section 6, we elaborate on the possibilities of secure session management in the presence of an active network attacker.

4 Secure Session Management

We present our secure session management mechanism in two stages. First, we introduce the general idea and achieved properties without getting lost in details. In a second stage, we explain how these properties are achieved by highlighting each aspect of the session management mechanism.

4.1 General Idea

The general idea of our secure session management mechanism is illustrated in Figure 1(a). All session management is done using the newly introduced *Session* header, which keeps track of a session identifier (ID), as well as the parameters needed to provide security guarantees. Our mechanism is based on a shared session secret, which is safely contained in the browser, inaccessible from any script code and never sent over the network. Using this shared session secret, we can generate request signatures, which are sent to the server in the *Session* header.



(a) High-level overview (b) Detailed explanation

Fig. 1. Our lightweight and secure session management mechanism.

Note that an eavesdropper can easily get hold of the session identifier, which is sent in the clear, but that the session identifier is no longer the bearer token for

the session, nor is it supposed to be secret. Using a simple incremental counter as an identifier is sufficient. An attacker trying to use a stolen session identifier also needs the session secret to generate valid signatures for crafted requests. Since this shared secret is safely contained within the browser, it can not be obtained by an attacker.

4.2 Detailed Explanation

In this section, we cover all aspects of our secure session management mechanism in four steps: (i) actual session management, (ii) how we establish the shared session secret, (iii) generating and verifying request signatures, and (iv) preventing replay attacks.

Session Management Associating the server-side stored state with the appropriate requests is simplified by using a simple session identifier (ID). The session identifier is provided by the server using a *Session* response header (response 1 in Figure 1(b)). The browser attaches the session identifier to each request, using the newly introduced *Session* request header. Note that while the use of a session identifier resembles traditional cookie-based session management, the session identifier is no longer considered to be a bearer token.

Shared Session Secret The shared session secret, needed to compute and verify signatures on requests, is established using the Hughes variant [8] of Diffie-Hellman, allowing one party to calculate a key and securely transfer it to the second party. After running this protocol, the server also possesses the key generated by the browser, without any information being disclosed to an eavesdropper.

In Figure 1(b), the server sends his public value (Y) after seeing the first request, in which the browser indicates support for the *Session* header. Using the server's public component Y, the browser can calculate the second public part (X) the server needs to calculate the key. In the next request, the browser sends the public value X, allowing the server to calculate the full key and verify this and any subsequent requests.

Note that the advantage of the Hughes variant of Diffie-Hellman is that the browser can compute the key before the first request is sent. This is required to attach an HMAC to the first request, so the server can verify that the sender of the first and second request are in fact the same. Omission of the first HMAC allows an eavesdropper to respond to the first response, injecting his key material into the session, which is problematic if the first request already caused some server-side state to be stored in the session.

Request Signatures When both parties possess a shared key, messages can easily be signed and verified using a hash-based message authentication code (HMAC). Since this HMAC takes the request and the shared secret as input, only the browser and the server can calculate a valid signature. A valid signature on a request indicates its authenticity, since it can only have been generated by the

browser holding the shared secret. Incoming requests with invalid signatures are simply discarded by the server.

Note that the input for the HMAC should be chosen carefully. Technically, a passive network attacker can steal the valid signature from an eavesdropped request and attach it to a crafted request, having the crafted request reach the server first. Since the crafted request requires a valid signature, the attacker can only modify the parts of the request that are not included in the signature. Including the URL of the request in the signature prevents an attacker from directing the request to a different destination, but still allows him to modify the request parameters (e.g. the destination account of a wire transfer). Therefore, the signature covers any user-dependent data, such as the URL, form inputs, attached cookies or authorization headers, etc.

Covering all sensitive inputs in the signature does not prevent an attacker from taking the valid signature and attaching it to a crafted request. However, it does ensure that the attacker can not change the sensitive data, meaning the crafted request equal to the original request, and thus harmless.

Replay Protection Signing each request prevents an attacker from modifying request data, but does not protect against replay attacks, where an attacker simply repeats a previously eavesdropped request. Replay attacks can be countered by including a unique value in the request, allowing the server to differentiate between fresh and replayed requests. The unique value (C in Figure 1(b)) is part of the *Session* header, which is in turn protected by the request signature. In case verification of the replay protection token fails, the request is discarded.

5 Deploying Secure Session Management

Until now, we have presented our proposal for secure session management in a cleanroom environment. In this section, we discuss a concrete deployment scenario involving third-party authentication services and elaborate on the infrastructure support our session management mechanism needs. Additional deployment scenarios are covered in the discussion section (Section 6).

5.1 Third-Party Authentication Services

A third-party authentication service is responsible for authenticating users, and returns an identity to the relying application. The specifics of user authentication are up to the authentication provider to determine, and are transparent to the relying application. The authentication provider can typically provide a much more secure environment, such as a TLS protected channel and multi-factor authentication. A typical example of such an authentication protocol is OpenID [14], which is designed for exactly this purpose. Recently, Google started providing OpenID authentication services [9], allowing anyone with a Google account to sign in using OpenID on any supporting site.

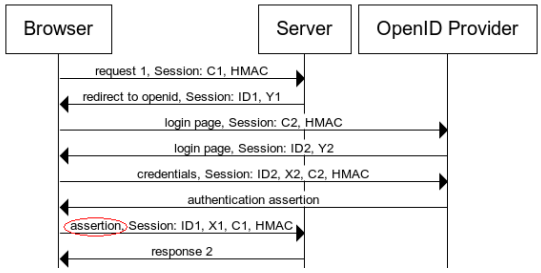


Fig. 2. The use of an OpenID authentication provider in combination with secure session management.

After completing the OpenID authentication process, the relying application typically reverts to traditional cookie management, often over an insecure connection, making itself vulnerable to eavesdropping attacks. In this scenario, our secure session management mechanism can be used to protect the session of the relying application, preventing such an eavesdropping attack. Figure 2 shows how OpenID authentication works in combination with our secure session management.

An important observation in the scenario above is that the authentication provider triggers the browser to send an authentication assertion, carrying the necessary proof about the user's claimed identity, to the relying party (marked in red on Figure 2). The relying party needs to ensure that one of the parameters given to the authentication provider to be included in the authentication assertion, is bound to the user's specific session. Otherwise, the authentication assertion becomes a bearer token, allowing a passive network attacker to attach it to a crafted request, impersonating the authenticated user. The OpenID specification counters this with the use of nonces: each assertion becomes unique, and can only be used once. However, if the attacker can steal the assertion and use it before the relying application does, the use of nonces does not suffice. Therefore, we propose to solve this problem by attaching the session identifier (ID1) to the return URL, uniquely binding each assertion to this session identifier. This prevents an attacker from using the authentication assertion in his own session, since the identifiers do not match. Similarly, in order to take over the session with the identifier used in the URI, the attacker needs to present a key (X in request 2) that leads to a valid HMAC on the first request sent by the user's browser, which is not possible.

5.2 Infrastructure Support

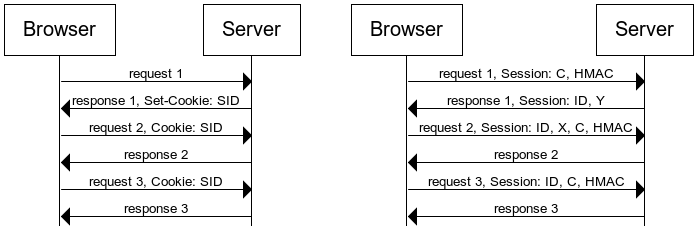
Successful deployment of our secure session management mechanism depends on both client-side and server-side support. Here we discuss the necessary support, as well as the potential impact of the newly introduced session management mechanism, compared to the current session management mechanism.

Client-side support for the newly introduced *Session* header is essential. This functionality can be added to the browser itself, using the readily available cryp-

tographic libraries. By pre-computing keys during idle time, the browser ensures the quick availability whenever a new key is needed. Note that supporting the *Session* header through a browser extension is only feasible as a proof-of-concept, since an extension has no guarantees of being the last one to inspect a request. If the signed parts of a request are changed after its signature has been computed, the server-side verification will fail.

Our preferred flavor of server-side support is extending the built-in session management of frameworks with secure session management. Instead of issuing a session cookie (e.g. PHPSESSID or JSESSIONID), the framework automatically uses the *Session* header, if supported by the client. Alternatively, a server-side proxy can translate between traditional session cookies and the new *Session* header. The proxy watches known outgoing session cookies and replaces them with the correct *Session* header. Consequently, incoming *Session* headers are mapped back to the corresponding session cookies. Both flavors of deployment provide secure session management, transparently to the application. This makes our proposal for secure management suitable for both legacy and new applications, provided the underlying infrastructure offers the necessary support. Note that infrastructure updates eventually reach almost all legacy applications through software updates (e.g. updating the PHP framework).

In case either the client or the server lacks support for the *Session* header, cookie-based session management will be used as a fallback mode. This allows gradual deployment, and gracefully deals with legacy clients and servers.



(a) Cookie-Based Session Management

(b) Secure Session Management

Fig. 3. Traffic flows of traditional cookie-based session management next to secure session management.

6 Discussion

In this section we discuss secure session management in the context of the web in general. First, we elaborate on how secure session management can benefit other deployment scenarios besides third-party authentication services. Second, we come back to the active network attacker, a threat model which we considered to be out of scope. We explain why we consider this threat model to be out of scope, and how an application can protect itself against active network attackers.

6.1 Generalizing Secure Session Management Scenarios

We have presented our proposal for secure session management in the light of third-party authentication services, mainly because they present a clear case study without complications. Here, we discuss two additional common web scenarios, where secure session management offers an advantage over the current situation.

Combining HTTP and HTTPS A large fraction of web applications offer their main content over HTTP, but switch to HTTPS for submitting sensitive information, such as authentication credentials or payment information. Often, the application reverts back to HTTP once the sensitive information has been submitted, falling prey to passive network attackers. When the application shares a session between the secure and insecure part of the site, an attacker can easily steal the authenticated session. Combining HTTP and HTTPS with traditional session management is possible, if a separate session is maintained on the HTTPS part of the site, with *Secure* cookies. Additionally, all security-sensitive operations need to be conducted over a secure connection.

Examples of this deployment scenario are Amazon and Facebook, respectively number 8 and 2 on Alexa's list of most popular sites worldwide. Amazon offers its product catalog over HTTP, but requires HTTPS for all account-specific operations, such as authentication, checking out or modifying settings. Amazon correctly maintains an HTTPS session, prompting an eavesdropper for reauthentication when trying to perform a sensitive operation. If all is configured correctly, an eavesdropper can only manipulate the products in the shopping cart. Facebook on the other hand, also uses HTTPS for sensitive operations, but does not correctly maintain an HTTPS session[1]. If an eavesdropper gets hold of a session after authentication, he can use this session to access sensitive operations over HTTPS, and freely modify the user's account settings, except for the password.

Sites using this deployment scenario can benefit from secure session management, since it will protect the session on the HTTP part of the application. In Amazon's case, an attacker would not be able to manipulate the user's shopping cart. The benefit that Facebook can reap is larger, since it would prevent an attacker from taking over an authenticated session, even with requests being sent over HTTP. If an attacker can never get hold of an active session, he can also not access the sensitive operations.

HTTP-only Applications A second, remarkably common case are web applications solely relying on HTTP. These web applications even perform user authentication over an insecure channel using HTML forms and a POST request, which makes the user extremely vulnerable to an eavesdropping attack. The attacker can not only steal the user's authenticated session, but can also get hold of the user's authentication credentials.

[1] For completeness, we'd like to acknowledge that Facebook offers an HTTPS-only experience, if the user explicitly opts-in. By default, this feature is not enabled.

Unfortunately, secure session management can not eliminate the latter problem, but can protect against the former. By deploying secure session management on an HTTP-only site, the attack vector for eavesdropping attacks is reduced from every request to a single request, holding the authentication credentials[2].

6.2 Protecting against an Active Network Attacker

An active network attacker is one of the most potent threat models, with full-blown man-in-the-middle capabilities. Traditionally, the use of TLS should protect against active network attackers, but recent incidents using fraudulent certificates have put this in different light. The use of a newly proposed extension, Origin-Bound Certificates [6], should prevent man-in-the-middle attacks.

Our proposal for secure session management does not consider the active network attacker to be in scope, since trying to protect against these attacks would result in a similar solution as provided by TLS and its extensions. However, when used in combination with TLS, our session management mechanism offers secure session management, with a safely contained shared secret. This significantly reduces the risks introduced by cookie-based session management, such as configuration errors [10] and script-based attacks [13, 16].

7 Conclusion

Session management is a critical component of modern web applications, but often suffers from network or script-based attacks. The de facto solution would be to deploy TLS, but many applications fail to do so correctly, for various reasons. We have presented a secure and lightweight session management mechanism, that does not require TLS, but that protects against eavesdropping, session hijacking and session fixation attacks. We have shown how our session management mechanism can be deployed in a common non-TLS scenario involving third-party authentication services. We have elaborated on how secure session management can be deployed to the web, and illustrated why we consider our solution to be lightweight.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, IWT and the EU-funded FP7 projects NESSoS, WebSand and STREWS.

With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission – Directorate-General Home Affairs. This publication reflects the views only of the authors, and the European Commission cannot be held responsible for any use which may be made of the information contained therein.

Images are based on diagrams drawn using *websequencediagrams.com*

[2] Note that we firmly discourage this practice, but frequently see it occurring on the web.

References

1. B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524, 2008.
2. Apache Software Foundation. Apache tomcat - migration guide - tomcat 7.0.x. *Online at http://tomcat.apache.org/migration-7.html*, 2012.
3. 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*, pages 75–88, 2008.
4. E. Butler. Firesheep. *Online at http://codebutler.com/firesheep*, 2010.
5. 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 (TOIT)*, 12(1):1, 2012.
6. M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-Bound Certificates : A Fresh Approach to Strong Client Authentication for the Web. In *Proc. 21st USENIX Security Symposium*, 2012.
7. P. Hallam-Baker. Http integrity header. *Online at http://tools.ietf.org/html/draft-hallambaker-httpintegrity-02*, 2012.
8. E. Hughes. An encrypted key transmission protocol. *rump session of CRYPTO*, 94, 1994.
9. G. Inc. Federated login for google account users. *Online at https://developers.google.com/accounts/docs/OpenID*, 2013.
10. C. Jackson and A. Barth. Forcehttps: Protecting High-SecurityWeb Sites from Network Attacks. In *Proceeding of the 17th international conference on World Wide Web*, pages 525—-534, Apr. 2008.
11. M. Johns, S. Lekies, B. Braun, and B. Flesch. BetterAuth: Web Authentication Revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 169—-178, Dec. 2012.
12. A. Langley, N. Modadugu, and W. Chang. Overclocking ssl. In *Velocity: Web Performance and Operations Conference*, 2010.
13. N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: lightweight protection against session hijacking. *Engineering Secure Software and Systems*, pages 87–100, 2011.
14. D. Recordon and B. Fitzpatrick. OpenID authentication 2.0. pages 1–35, 2007.
15. I. Ristic. Internet ssl survey 2010. *Talk at BlackHat*, 2010.
16. M. Schrank, B. Braun, M. Johns, and J. Posegga. Session fixation–the forgotten vulnerability? *Proceedings of GI Sicherheit*, 2010, 2010.
17. J. Williams and D. Wichers. Owasp top 10. *OWASP Foundation, April*, 2010.
18. Y. Zhou and D. Evans. Why aren't http-only cookies more widely deployed. *Proceedings of 4th Web*, 2, 2010.